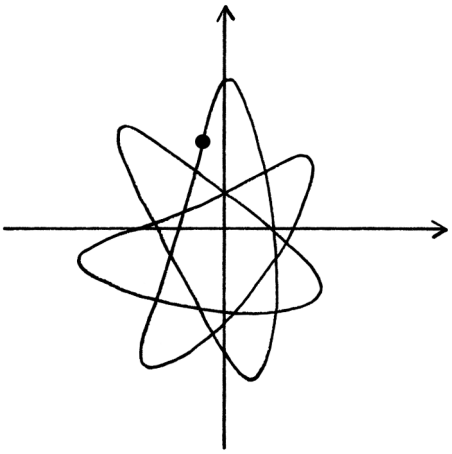
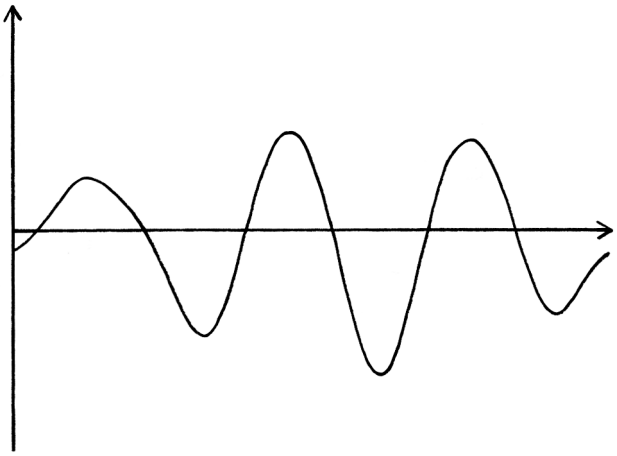
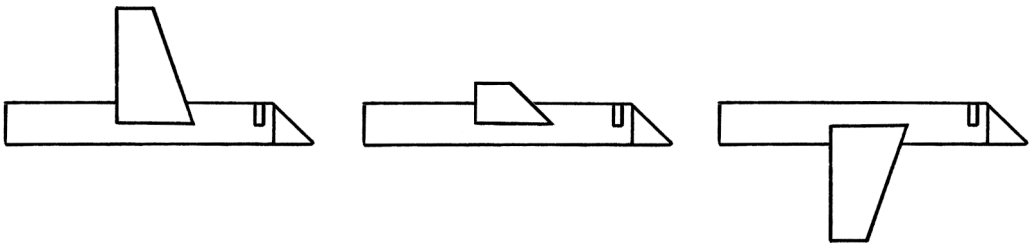




A Book About Waves

Part Two

By Tim Warriner



17th December 2022

A Book About Waves
Part 2

by Tim Warriner

www.timwarriner.com
30th November 2022
[Last edited on 2022-12-17]

To see if this is the latest version of this book, visit www.timwarriner.com

Contents

Introduction to Part 2

This is the second part of my book about waves. Whereas the first part concerned itself with theoretical waves and signals, this part starts by focusing more on waves in reality.

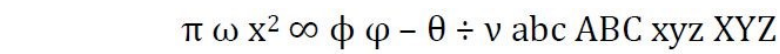
When dealing with maths, it is easy to know if ideas are correct because either the maths works or it does not work. When describing real-world waves, for which it is harder to test theories, there is much more leeway for being wrong. It is also harder to follow the standard naming of concepts because the names vary between different people, languages, regions, subjects and mathematical cultures. Waves and signals are often explained badly, and it is frequently difficult to discover exactly what someone means by a particular word. At times in this book, I have used my own naming conventions to avoid ambiguity.

To see if this is the latest version of this book, visit www.timwarriner.com

To tell if this PDF is being displayed correctly, this line of text:

$\pi\ \omega\ x^2\ \infty\ \phi\ \varphi - \theta \div \nu$ abc ABC xyz XYZ

... should appear roughly the same as in this picture:



Chapter 31: Real-world waves

So far, in this book, we have looked at waves in an abstract, theoretical way. Now, we will look at waves as they appear in nature. I will call these "real-world waves". By this, I mean the fluctuating movement of some real-world entities, the behaviour of which can be portrayed using waves.

Ideally, real-world waves would not be called "waves", and only their *portrayal* would be called "waves". However, a combination of convention, language and the effort in doing otherwise, means that real-world entities themselves are often called waves if their behaviour can be described using waves. For example, people often say that "sound is a wave". Generally, they mean that sound is a phenomenon that has a significant characteristic that can be portrayed using waves. Sound's wave-like attribute is not the entirety of what sound is. *Usually,* people who say things such as "sound is a wave" are doing so to save time in an explanation.

The distinction between the behaviour of an entity and the waves that portray it leads to the following idea. There is a way of thinking about real-world waves that will help reduce a lot of confusion while you are learning, and that is to believe that:

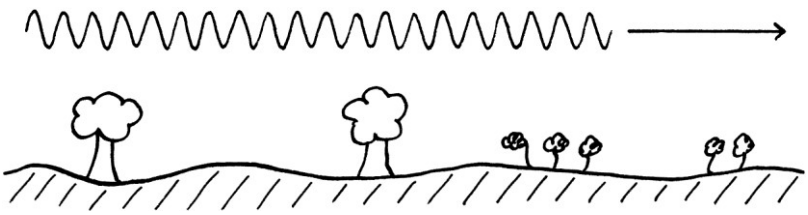
"Real-world waves do not exist on their own."

A number of people will disagree with this statement, but it is irrelevant whether it is completely true or not – it is the best way to think about real-world waves while you are learning about waves. When you have learnt enough to be completely sure of the validity of the statement, your knowledge will be superior to anything in this book.

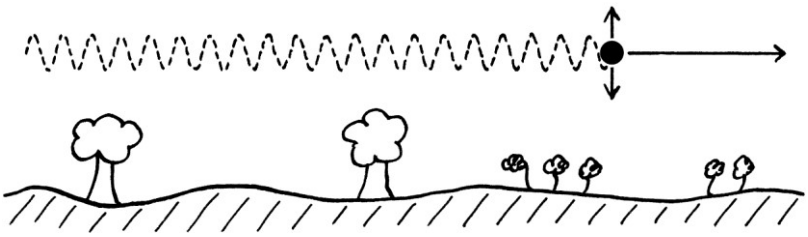
The statement means that waves do not exist as entities in their own right in nature. Instead, there are real-world entities that have properties that behave in a way that can be described using waves. To put this another way, there are entities that have characteristics that, to some extent, match some of the theoretical characteristics of circles. There are no entities that are literally waves – there are only entities that have particular properties that can be described with waves. For example, a magical flying snake that flies in the shape of the curve of a Sine wave is not a wave – it is a magical flying snake whose flight can be described using waves.

The idea is really saying that a wave is something you see only in a graph or a formula. The graph or formula portrays the fluctuations of an aspect of an entity, but those fluctuations are not, in themselves, waves. Most explanations of waves do not make this distinction, which, if you think too hard about waves, can be very confusing.

Waves describing real-world phenomena are often portrayed in illustrations as if they were the curve from the graph of a Sine wave travelling through space:



This is a simple way to portray an idea, but it is misleading if taken literally. It suggests that waves *on their own* travel through space. A better portrayal of a wave would be to have an object travelling through space leaving a wave-like trail, perhaps like this:

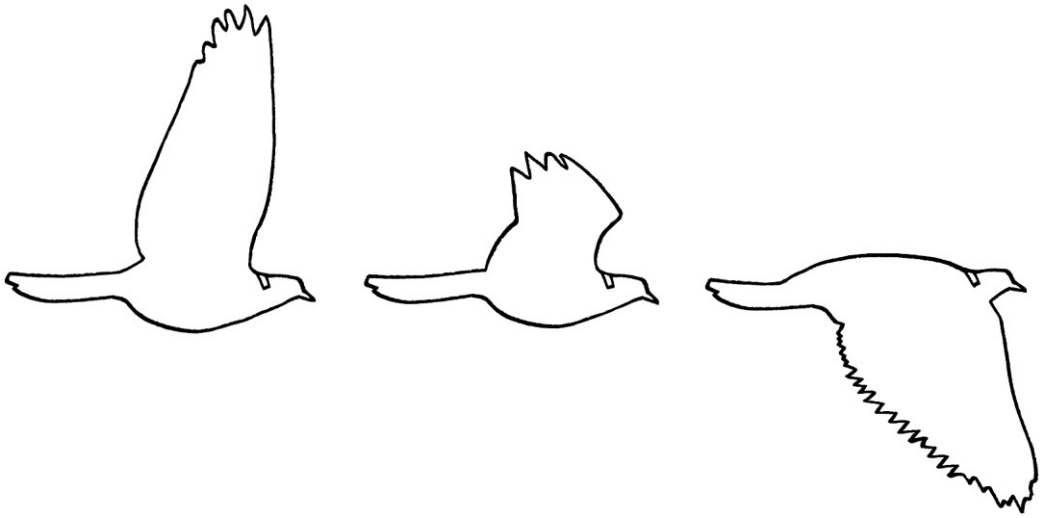


To summarise all of the above, you should think of all waves as showing the characteristics of entities, and not existing on their own. This will help you learn about waves more easily. There are times in the first four chapters of this part of this book when I will temporarily use the idea of waves existing in reality because it makes the explanation easier to write and read. [It is often much easier to discuss waves by treating them as if they did appear on their own.]

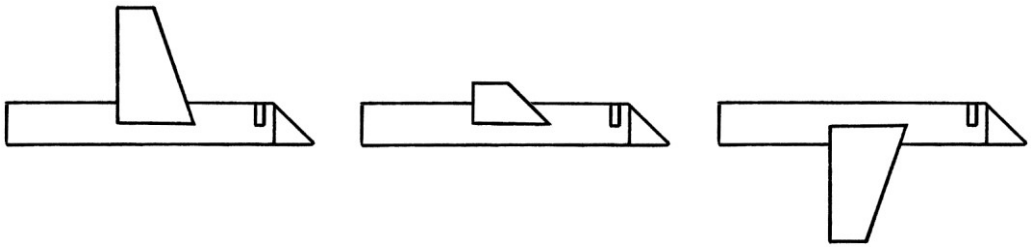
To learn about waves in the real world, we will first look at some examples to give an idea of the range of possible types. In this and the following chapters, we will be using radians in the formulas.

Pigeon wing tip waves

An easy way to introduce the idea of waves in the real world is to consider the flight of wood pigeons. A wood pigeon, when in the middle of travelling long distances through the air as part of its daily routine, raises and lowers the tips of its wings to about the same distance above and beneath the vertical centre of its body. A wood pigeon also raises its wings at a very similar rate to which it lowers them. [An adult wood pigeon can be distinguished from other types of pigeons and doves by the white band around the sides of its neck.]

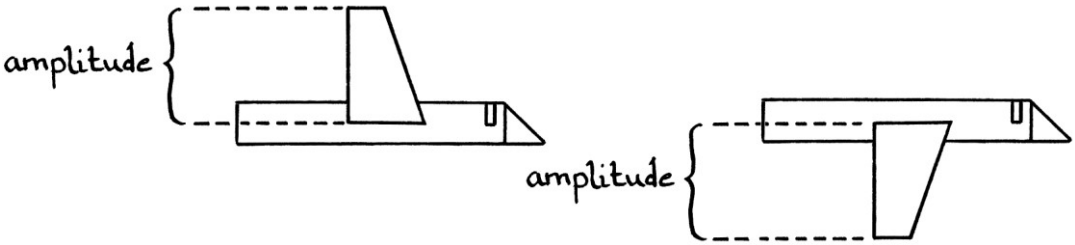


For the purposes of this example, we will work with an idealised model of a wood pigeon. In this model, the wood pigeon moves its wings above and below its body to the same distance, and it raises its wings at the same rate as it lowers them:



We will say that the pigeon flaps its wings upwards and downwards once every second, and that it moves its wings 25 centimetres above and below the centre of its body. Most importantly, we will say that if we take measurements of the wood pigeon's wing tip height in relation to the centre of its body over a set amount of time, we will end up with a Sine wave.

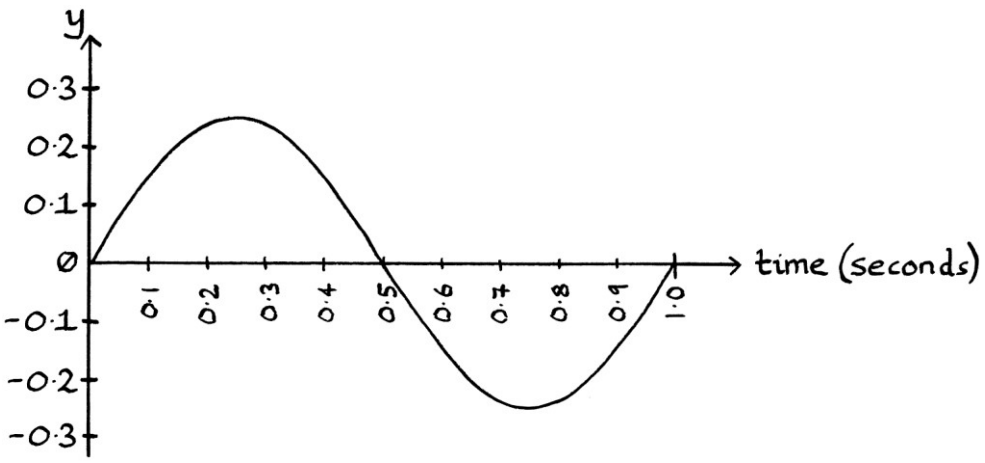
The amplitude of the Sine wave will be equal to the maximum or minimum vertical distance from the wing tips to the centre of the pigeon's body.



The amplitude of the Sine wave for our pigeon is 25 centimetres. We will give this in metres for consistency's sake, and say it is 0.25 metres. The frequency of the Sine wave is 1 cycle per second. We will say that the time in the formula is the time since we first started observing the pigeon.

The formula that gives the height in metres of our wood pigeon's wing tips above the centre of its body at any moment in time is (in radians):
"$y = 0.25 \sin 2\pi t$"

The graph for one cycle looks like this:



All of this means that we can describe the movement of a wood pigeon's wing tips using waves. The waves are unlikely to be ones that will interest electrical engineers or quantum physicists, but they are still valid waves.

The pigeon wing tip wave follows the rules for a basic Sine wave formula:
"$y = A \sin (2\pi * ft) + \phi)$"
... where:

- The amplitude of the wave is the maximum distance that the wing tips reach above or below the centre of the pigeon's body. If the pigeon raises and lowers its wings to a lesser extent, the amplitude of the wave becomes less. If the tips reach to higher and lower points, the amplitude becomes higher. A larger pigeon would have a higher amplitude; a smaller pigeon would have a lower amplitude. [Although, in reality, wood pigeons that are old enough to fly are all the same size.]

- The frequency is the number of times that the pigeon completes one raising and lowering of its wing tips in one second. If the pigeon flaps its wings more quickly, the frequency will be higher. If it flaps them more slowly, the frequency will be lower. We will call one raising and lowering of the wing, a "flap cycle".

- The phase gives the position of the wing tips when we first started observing the flying pigeon. On its own, the phase in this case does not tell us much. However, if we had two pigeons flying side by side, the phases of the pigeons might be different. For simplicity's sake, we will say that the phase of our pigeon is zero. This means that when we first started observing the pigeon, its wing tips were at the same height as the centre of its body and about to rise.

- The time is the number of seconds since we first started observing the pigeon.

- As all measurements are from the vertical centre of the pigeon's body, and its wings are attached to its body, the concept of mean level is less useful, so we will say it is always zero. If we had a wood pigeon that could raise or lower its shoulders while it flew, then that would count as a mean level, but we will ignore that for now. Note that the mean level is *not* the height of the pigeon above the ground – the y-axis shows the height of the wing tips with respect to the centre of the pigeon's body. The pigeon could be flying at a height of 1 metre, 10 metres, or 100 metres, and that fact would not be reflected in the graph or in the formula.

One thing to notice is that there is only one wave generated by the pigeon's wing tips. We could say that the wave is a Sine wave, or we could say that it is a Cosine wave. If we treat the pigeon's wing tips as being described by a Sine wave, then there is an implied corresponding Cosine wave, and if we treat the pigeon's wing tips as described by a Cosine wave, then there is an implied corresponding Sine wave. The corresponding waves in this situation are theoretical, and do not describe anything in reality. The pigeon's wing tip waves do not derive from a circle – the movement of the wings is a result of the pigeon's muscle and bone structure, and ultimately controlled by the pigeon's brain. The fact that the movement can be described by a pure wave is just a coincidence. Despite this, we can still treat the waves as being derived from a circle. Doing so can make the waves easier to visualise, and might simplify some calculations.

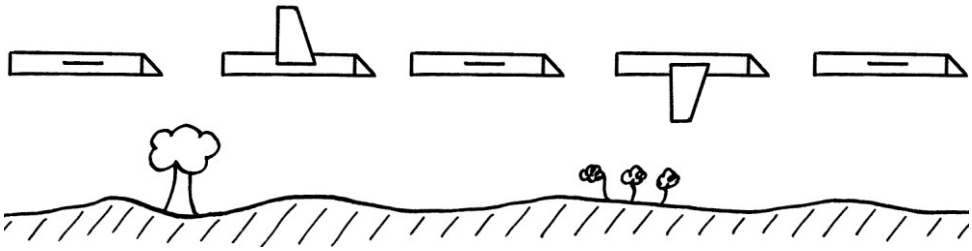
Notice how it is an arbitrary choice whether we treat the wing tip wave as being a Sine wave or a Cosine wave. It does not matter which is chosen as long as we are consistent. For the pigeon as we have described it, using a Sine wave means that we do not need to include a phase in the formula – the pigeon starts with its wing tips at the same height as the centre of its body. If we had used a Cosine wave, we would have needed a non-zero phase, so the formula would not have been as short.

The wing tip wave would be useful if we were studying certain attributes of bird flight, and for some aspects of bird flight study, the wing tip wave would be all we needed. [It is possible to distinguish between many birds by the wave or signal drawn out by their wing tips.] However, the resulting wave does not tell us everything about the pigeon – it does not tell us the speed of the pigeon, the direction, the weight, the efficiency, how feathery it is, what pigeons eat, where they nest, and so on. It only tells us the distance of the height of its wing tips above or below the centre of its body with respect to time. If a scientist spent years studying pigeon wing tip waves, they would not learn a great deal about other aspects of pigeons. Similarly, if someone created a robotic pigeon with wings that flew exactly according to the wings described in the wave graph, they would not be able to say that this meant the robot was identical to a real pigeon. The point I am trying to make here is true for all waves that describe a real-world characteristic – the wave describes only one attribute of the entity: the entity is not the wave and the wave is not the entity.

Distance-based waves

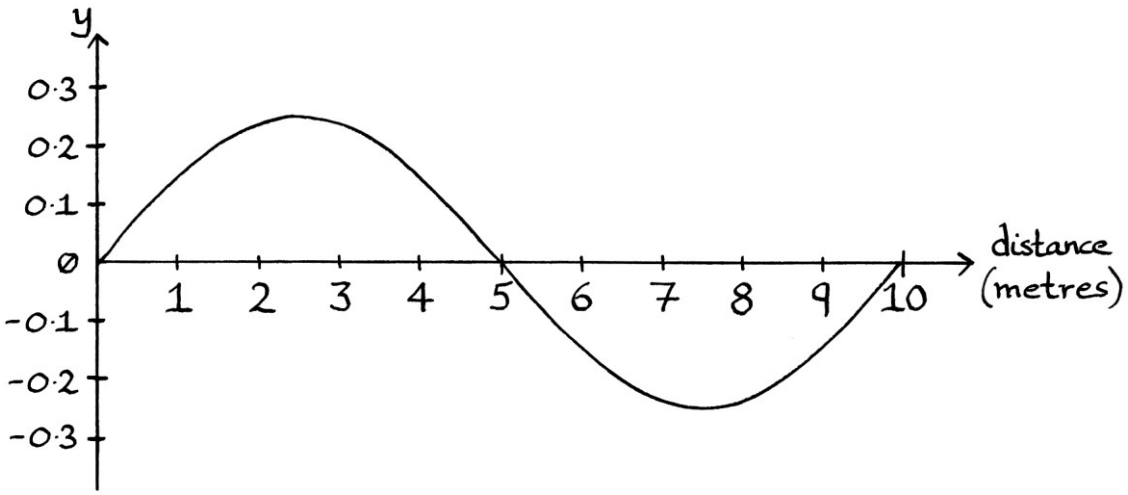
We have just seen that we can portray the height of the wing tips of our wood pigeon with respect to the centre of its body over time in terms of a Sine wave. As time passes, the wood pigeon raises and lowers its wings, and that is portrayed in the graph. If the wood pigeon completes one "flap cycle" in 1 second, then the wave has a frequency of one cycle per second, and a period of 1 cycle per second. As we know, frequency and period relate to time. If the wood pigeon flew directly into the wind and carried on flapping at the same rate, even if it were blown backwards, the above graph would be the same, and the frequency and period would be the same. If the pigeon held onto a branch and flapped its wings at the same rate, without moving forwards or backwards at all, the graph would be the same, and the frequency and period would be the same. This should all be obvious – it is the raising and lowering of its wings with respect to the centre of its body over time that is important and nothing else.

We will now complicate matters by considering the *distance* that the wood pigeon travels while it flaps its wings. In other words, we will stop looking at the *time* for each "flap cycle", and instead look at the *distance* for each "flap cycle". This concept is straightforward if we imagine seeing a wood pigeon flying past us:



The pigeon raises and lowers its wings as it travels along, and we will say that it completes one flap cycle over 10 metres.

We can portray this on a graph with the y-axis showing the height of the wing tips above or below the pigeon's centre, and with the x-axis as the *distance* of the pigeon from the starting place. Note that the y-axis and x-axis are drawn to different scales despite both referring to metres.



Spatial frequency

Because the pigeon moves at a constant speed, the above graph is a Sine wave. It is not a Sine wave that relates to time, but one that relates to distance. From looking at the graph, we can see the state of the pigeon's wing tips at any *distance* along those ten metres. For example, when the pigeon had travelled 5 metres, its wing tips were at the same height as the centre of its body. When the pigeon had travelled 7.5 metres, its wing tips were at their lowest point. The graph does not show us *when* the wing tips were at their lowest point though – it shows us only *where*. There is no mention of time in this graph at all. Given that there is no mention of time, we cannot deduce a frequency for the Sine wave – a Sine wave that does not relate to time does not have a frequency. Frequency refers to the number of cycles completed in one moment of time. This Sine wave has a distance-based frequency, or what is generally called "spatial frequency", where the word "spatial" is the adjectival form of "space", and "space", in this sense, refers to "distance".

A distance-based wave has no reference to time, although it works in a similar way to a time-based wave. In a distance-based wave, we pay attention to "cycles per *metre*", as opposed to "cycles per *second*". Therefore, we have "spatial frequency" as opposed to a time-based frequency. [Some people refer to "time-based frequency" as "temporal frequency", where "temporal" is the adjectival form of the noun "time".]

Our wood pigeon completes one cycle in 10 metres. Therefore, the *spatial frequency* of its distance-based wave is 0.1 cycles per metre.

Side note

Distance-based frequency, or spatial frequency, is sometimes called “wavenumber” [as one word] or “wave number”. Given the number of terms you need to learn, I think it is easier to remember the term “spatial frequency” than it is to remember the term “wavenumber”. “Spatial frequency” is a much more descriptive term. To complicate matters, some people say that “spatial frequency” and “wavenumber” refer to different concepts, and some people say that they are the same concept. There are also people who define wavenumber as “*angle* per metre” [as in the portion of a cycle per metre] instead of “*cycle* per metre”. Different academic fields have their own variations of the definitions, and it seems they never get together to discuss them. [All of this is just for English-language maths – I do not know if other languages have similar problems.] We will let other people argue amongst themselves about definitions, and in this book, we will use the term “spatial frequency” to mean distance-based frequency measured in cycles per metre.

Wavelength

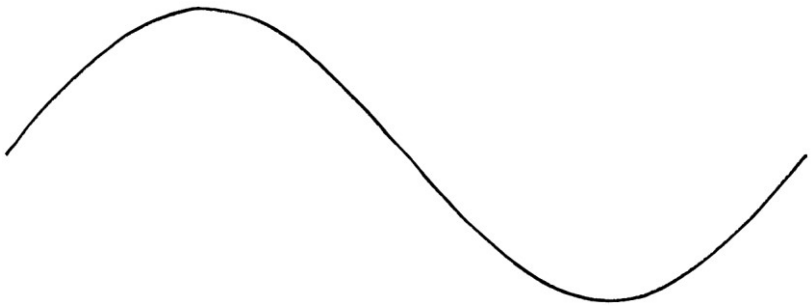
As we know, time-based frequency is the number of cycles completed per second, and period is the length in time of one cycle. Frequency is the reciprocal of period, and period is the reciprocal of frequency. If a time-based wave has a frequency of, say, 5 cycles per second, then its period is 1 ÷ 5 = 0.2 seconds per cycle. A cycle takes 0.2 seconds to complete. The distance-based equivalent to period is “wavelength”. Wavelength is the reciprocal of spatial frequency. Wavelength is the length of one cycle in metres. It is the distance from one place in the curve to the place where the shape of the curve repeats. The name is slightly misleading as it is not the length of the *wave* (which could go on forever), but the length of one cycle of the wave.

If a distance-based wave has a spatial frequency of, say, 20 cycles per metre, then its wavelength will be: 1 ÷ 20 = 0.05 metres per cycle, or 0.05 metres, depending on how we want to phrase the result.

As we have already seen, our wood pigeon completes 1 cycle in 10 metres, so its wave has a spatial frequency of 0.1 cycles per metre. The wavelength of the wave is, therefore, 1 ÷ 0.1 = 10 metres. The length of one cycle is 10 metres.

The most important thing to remember here is that wavelength refers to *distance* and period refers to *time*. Wavelength and period are often confused by people, even though the concepts refer to different ideas.

If our wood pigeon fell in some mud and flew past a wall, touching it with the tip of one of its wings as it went past, it would draw out a distance-based wave:



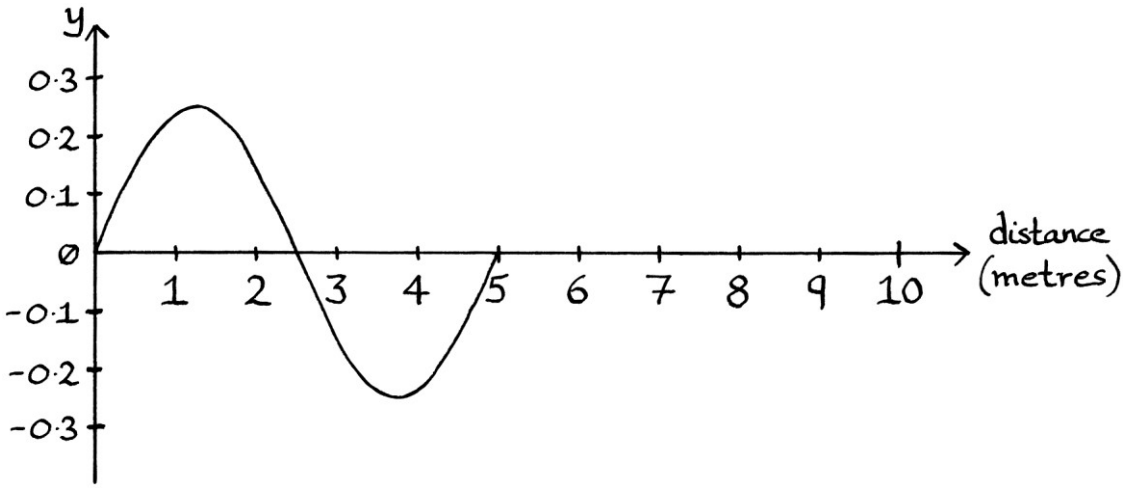
From the markings left by the pigeon, it would be possible to calculate its spatial frequency and its wavelength long after the pigeon had gone. Its wavelength would be the distance from one cycle to the next; its spatial frequency would be the reciprocal of that. It would not be possible, however, to calculate the pigeon's wing tip frequency, its period, or the pigeon's speed from the markings – this is because there is no record of anything to do with time.

The symbol for wavelength is the lower-case Greek letter "λ", named "lambda" and pronounced "lam-da", which is the Greek equivalent of the Latin letter "l" for "lima".

Note that it is much more common for people to talk about *wavelength* than it is for them to talk about *spatial frequency*. [Conversely, it is more common for people to talk about frequency that it is for them to talk about period.] One reason for this is that in many situations, it is clearer as to what the wavelength is measuring than to what a formula containing spatial frequency is describing.

Period and wavelength are unconnected

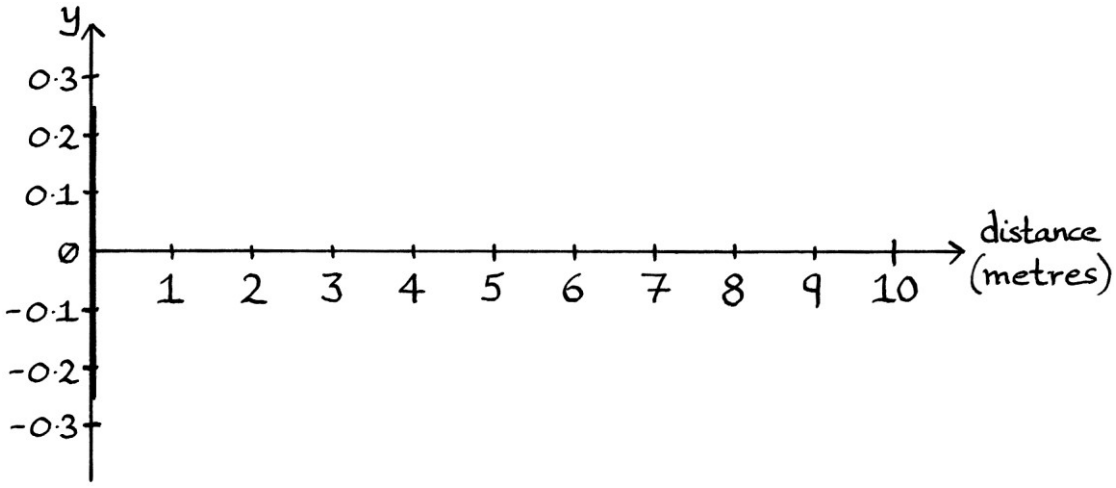
We will say that our pigeon always flies at 10 metres per second through still air. We will imagine that our pigeon flies against a 5 metre-per-second wind. If it still flaps its wings at the same rate as before, its wings will still have the same frequency (and therefore the same period). However, the wind will mean that the pigeon flies only half the distance in one flap cycle. Therefore, its wavelength will be halved – instead of one cycle being completed over 10 metres, it will be completed over 5 metres. The wavelength will be 5 metres. One cycle takes the same amount of *time*, but as the pigeon is travelling more slowly, it takes longer to travel the same distance, so the cycles repeat at a shorter distance. The spatial frequency will now be 0.2 cycles per metre – more cycles per metre are completed now because the pigeon is moving more slowly.



Perhaps counter-intuitively, if everything else remains the same, a slower speed means a faster spatial frequency – more cycles are completed in the same distance.

If the pigeon flies against a 10 metre-per-second wind, and continues to flap at the same rate as before, it will remain stationary in the air. The frequency of its time-based wave will be the same as before, as will be the period. However, as the pigeon is stationary in the air, the wavelength will be zero metres per cycle. The distance between one cycle and the next is zero metres because they all happen in the same place.

A graph showing this is as so:



The spatial frequency will be infinitely high. This is because all the cycles are happening over an infinitely small distance – zero metres. It does not matter how many cycles, or partial cycles, are completed (as long as there are more than zero cycles), as this will always be the case. The spatial frequency is the number of cycles completed per metre. Therefore, if any number of cycles is completed, the spatial frequency will be a non-zero number divided by zero, which will be infinitely high [or an undefined number, depending on how you learnt maths]. We could also find the spatial frequency by knowing it is the reciprocal of the wavelength. The number 1 divided by 0 is infinitely high.

This is a good example of how frequency and period, and spatial frequency and wavelength have similarities, but are not necessarily connected to each other.

Distance-based wave attributes

The term "wavelength" is commonly mentioned when discussing waves such as radio waves, but *formulas* for waves that relate to distance are much less common. This is partly because analysis of waves is generally done on time-based waves. Another reason is that it is often easier to think about wavelength than it is to think about the actual wave itself. In the study of radio and sound, it is rare that we would need to think about distance-based wave formulas.

A distance-based wave formula has the same basic structure as a time-based wave formula: it has amplitude, *spatial* frequency, phase and mean level.

Amplitude in a distance-based wave formula refers to the same idea as amplitude in a time-based wave formula. For our wood pigeon *time*-based wave, the amplitude referred to the height of the tips of the wings above or below the centre of the pigeon's body. In a *distance*-based wave, the amplitude refers to the same thing. In a time-based wave, we are seeing how the wing tip height varies over an amount of time. In a distance-based wave, we are seeing how the wing tip height varies over a *distance*. [If the units of amplitude for a time-based wave were, say, volts, then the units of amplitude for the corresponding distance-based wave would still be volts.]

Mean level refers to the same units in a distance-based wave as in a time-based wave, and in both cases will refer to the same units as the amplitude. As with a time-based wave, the mean level for our pigeon example is *not* the height of the pigeon above the ground – the y-axis shows the height of the wing tips with respect to the centre of the pigeon's body. The mean level still relates to the centre of the pigeon's body. If the pigeon could raise or lower its shoulders, it could alter its mean level. However, the height of the pigeon above the ground is irrelevant to the mean level for pigeons, even in a distance-based wave, and it plays no part in the formulas or graphs.

Spatial frequency in a distance-based wave is analogous to frequency in a time-based wave. Instead of being portrayed by the letter "f", spatial frequency is portrayed by the lower-case Greek letter "ν" (called "nu" and pronounced "new"). This is the Greek equivalent to the Latin letter "n" as in "number". Confusingly, "ν" looks just like the Latin letter "v" for "victor", and in some texts, it is literally impossible to know which of the two is being used, especially if the letter is in italics. It is not a good choice of symbol. Note that "ν" is also used to refer to time-based frequency in such academic subjects as optics. Spatial frequency is also sometimes portrayed with the letter "k", and it is worth noting that the letter "k" is also sometimes used to refer to *angular* spatial frequency.

Phase in a distance-based wave formula refers to the same concept as phase in a time-based wave formula. It indicates at what angle, or where in the cycle, the wave started.

A *time-based* wave formula includes a multiplication by 360 or 2π, so that the Sine and Cosine functions can operate on the time as if it were an angle, and have a default frequency of 1 cycle per second. A *distance-based* wave formula needs the same correction, so that the functions can operate on the *distance* as if it were an angle, and have a default spatial frequency of 1 cycle per metre.

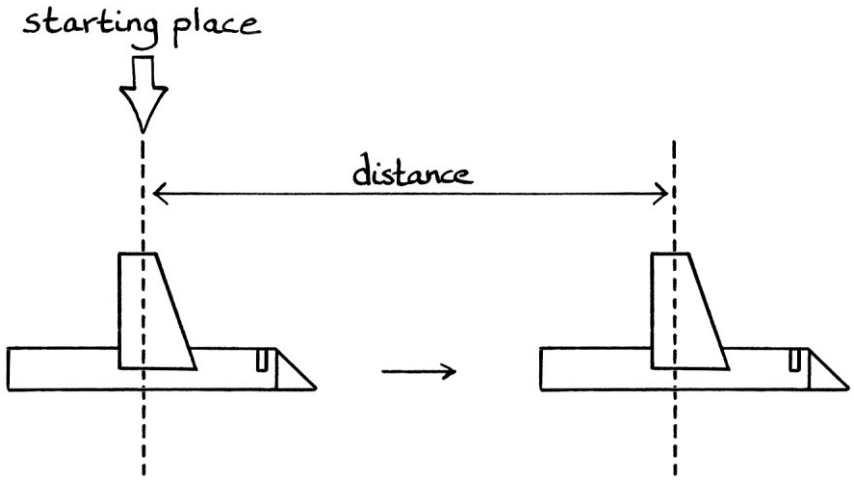
Instead of the letter “t” for time, we use the letter “x” for distance. [Occasionally, in the next chapter, we will use the letter “d” because we will be using “x” for another purpose.]

The complexities of distance

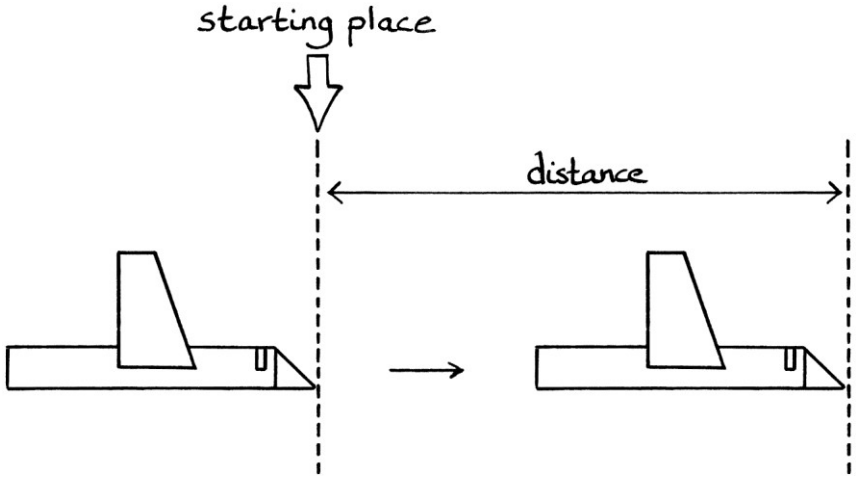
Before we look at a distance-based wave formula, we need to think about the meaning of distance.

At first glance, the idea of distance seems straightforward, but exactly which distance we are measuring can be often be difficult to discern. We have to decide *from* where, and *to* where, we are measuring. The idea of distance is often much more complicated than the idea of time. In our pigeon example, we *could* say that “x” will refer to the distance that the pigeon has travelled. Although this seems an adequate meaning for “x”, in practice, it is too vague to be usable. For one thing, the pigeon is not a single point – we have to know to where on the pigeon we are measuring. Similarly, we have to know *from* where we are measuring.

For our pigeon example, we could say that “x” refers to the distance that the point in the midpoint of the pigeon’s wing tips has travelled from a chosen starting place.

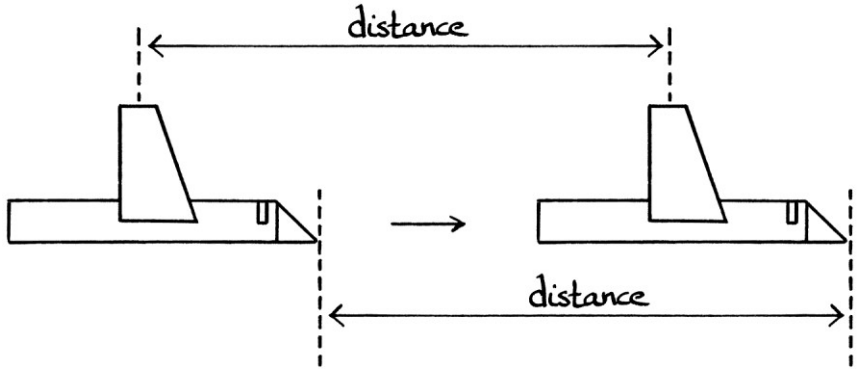


Another possible way we could define "x" is by saying it refers to the distance that the tip of the pigeon's beak has travelled from a chosen starting place.

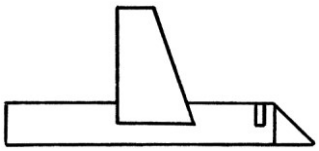


This is a different measuring point on the pigeon, and if we were literally observing a real pigeon, the wave describing the situation would not be identical – the phase would be slightly different. The tip of the beak on a pigeon is always about 15 centimetres ahead of the point in the middle of the wing tips. The pigeon's wings will be in a later state when the midpoint of the wing tips crosses the start line than when the beak crosses the start line. Similarly, after a set distance, the wings will still be in a later state for the mid-wing-tip measurement than for the beak-tip measurement.

If we were measuring from the place where we first *observed* the pigeon, then it would not matter whether we measured to the midpoint of the wings or to the tip of the beak, as long as we were consistent for all our measurements. The wings would be in the same state at the place where we first observed the pigeon, no matter at which point on the pigeon we considered.



There are countless ways to make measurements for our distance-based pigeon wing tip wave. We could measure from a fixed starting place to the tail of the pigeon, or even measure from a fixed starting place to a point that is always 30 centimetres in front of the pigeon. It is important to realise that we do not have to measure to the point where the instantaneous amplitude of the wave is being measured, and even if we said that we *were* going to do that, we would have to be more specific in identifying that exact place. In our pigeon example, the instantaneous amplitude is the height of the wing tips above or below the centre of the pigeon's body. That height could be measured at various places on the pigeon because the (idealised) pigeon's wing tips do not end in a single point.



However we measure the distance, the wavelength of the wave will be the same. This is one of the reasons that it is easier to state the wavelength of a wave, than it is to deal with distance-based wave formulas.

In our time-based wave from the start of this chapter, we had "t" refer to the time since we first started *observing* the pigeon. I did this to make the explanation simpler. If we are going to be using time-based and distance-based waves for the same object, and we want the waves to be compatible, then we must use the same idea for the "start". For example, if we say that "x" in the distance-based wave refers to the distance that the midpoint of the pigeon's wing tips has moved since we started *observing* the pigeon, then "t" must refer to the time since we started *observing* the pigeon. [As we are measuring time, we do not need to think about the midpoint of the pigeon's wing tips in the time-based wave after it has started – we only need to know about the situation at t = 0.] If we say that "x" in the distance-based wave refers to the distance between the midpoint of the pigeon's wing tips and a particular starting place, then "t" in the time-based wave needs to refer to the time since the midpoint of the wing tips were at that same starting place. If we say that "x" in the distance-based wave refers to the distance between a particular start point and the tip of the pigeon's beak, then "t" in the time-based wave needs to refer to the time since the tip of the pigeon's beak passed that particular start point.

To summarise the above, the most important idea when referring to "x" in distance-based waves is to decide what we are measuring and to be consistent with the measuring. Similarly, "t" in time-based waves that refer to the same entity should be measuring time from the same starting event.

For our pigeon example, we will say that "x" refers to the distance from a chosen starting place to the midpoint of the pigeon's wing tips. This means that we will need to change the meaning of our earlier time-based wave, so that "t" refers to the time since the midpoint of the pigeon's wing tips was first at that chosen starting place.

With other types of real-world waves, we will have similar problems in specifying exactly what is being measured as the "distance".

Distance-based formula

A general formula for a distance-based Sine wave is, in radians:
"$y = h_s + A \sin (2\pi * \nu x) + \phi)$"
... where:

- "h_s" is the mean level. In our pigeon example, this is the average height of the wing tips above or below the centre of the pigeon's body. For our pigeon, this will be zero, but for other types of waves, it might be non-zero.
- "A" is the amplitude.
- "ν" is the spatial frequency in cycles per metre.
- "x" is the distance in metres, where the meaning of "distance" will usually need to be clarified.
- "ϕ" is the phase in the angle system we are using.

The formula for a distance-based Cosine wave is:
"$y = h_c + A \cos (2\pi * \nu x) + \phi)$"
... where "h_c" is the mean level of the Cosine wave.

Our wood pigeon performs one flap cycle every 10 metres. Therefore, its spatial frequency is 0.1 cycles per metre. It raises and lowers its wing tips to 0.25 metres above and below the centre of its body. Its mean level is zero. We will measure the distance from a chosen starting place to the midpoint of the pigeon's wing tips. We will say that the phase is zero radians, which means that the wings had a height of zero metres and were about to rise when the pigeon crossed the starting place.

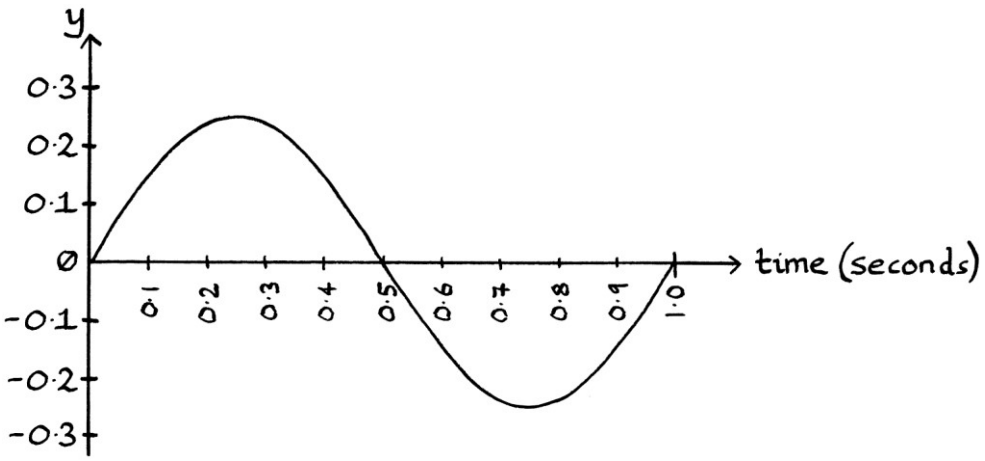
The pigeon's distance-based Sine wave formula is, therefore:
"$y = 0.25 \sin (2\pi * 0.1x)$"

If we want to know the height of the pigeon's wings at any particular distance from its starting point, we can enter the distance as "x" in the formula. For example, after flying 2 metres from the starting place, the pigeon's wing tip height will be:
0.25 sin (2π * 0.1 * 2)
= 0.25 sin (0.4π)
= 0.2378 metres, which is 23.78 centimetres above the centre of its body.

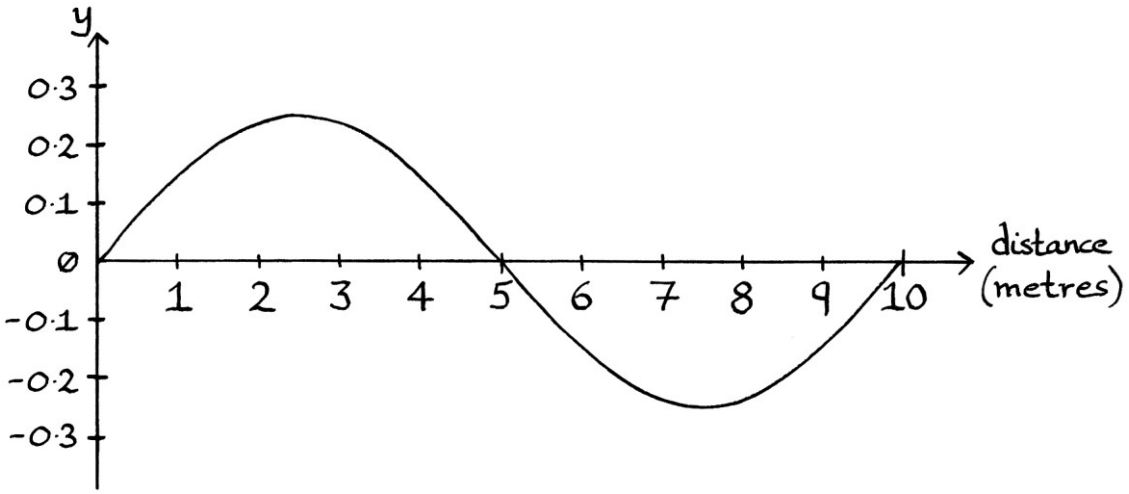
When the pigeon has flown 2.5 metres from the starting place, the height of its wing tips above the centre of its body will be:
0.25 sin (2π * 0.1 * 2.5)
= 0.25 sin (0.5π)
= 0.25 * 1
= 0.25 metres, which is 25 centimetres above the centre of its body. This is the maximum height for its wing tips.

Connecting time and distance

Our pigeon has a Sine wave that shows its wing tip height at any *time* since the midpoint of its wing tips passed the starting place:



Our pigeon also has a Sine wave that shows its wing tip height at any *distance* from the midpoint of its wing tips to the starting place:



The two waves are completely independent of each other. If we just have one wave, we do not have enough information to draw the other wave. We cannot make any inferences from one wave about the other wave. The only way we can connect the two waves is by knowing the speed of the pigeon. If we just have the time-based wave, and we know the speed of the pigeon, then we can create the distance-based wave. If we just have the distance-based wave, and we know the speed of the pigeon, we can create the time-based wave. Similarly, if we have both waves, we can calculate the speed of the pigeon. The speed cannot be deduced from either wave on its own. The speed is not mentioned in either the time-based formula or the distance-based formula.

If we have just the time-based wave, and we know the speed of the pigeon, then we can work out the distance that the pigeon travelled during one cycle of its wing tips. If one cycle takes 1 second, and the pigeon flies at 10 metres per second, then one cycle also takes 10 metres to be completed. The length of one cycle is 10 metres.

If we have just the distance-based wave, and we know the speed of the pigeon, then we can work out the time that the pigeon travelled during one cycle of its wings. If one cycle takes 10 metres to be completed, and the pigeon flies at 10 metres per second, then one cycle is completed in 1 second.

If we have both waves, we can work out the speed of the pigeon. If one cycle takes one second and 10 metres to be completed, then the pigeon must be travelling at 10 metres per second.

To know the details of speed or the missing waves, we can use the standard formula for distance, speed and time:
distance = speed * time

To make it more relevant to what we are doing, we can think of it as so:
distance of one cycle = speed * time of one cycle

That can then be thought of as:
wavelength = speed * period.

Wave speed

We can think about the speed of the pigeon, or we can think about the speed of the *wave*. The speed of a wave is called the "wave speed". The speed of a wave can be thought of as the speed that the wave moves over a distance. However, this can be a confusing idea for two reasons:

- First, ideally, we should not say that the wave exists outside of the formula or graphs. It is better to say that real-world waves do not exist on their own. Instead of the speed of the wave, we should really be thinking about the speed of the pigeon's fluctuating characteristic as that characteristic moves. If we were not dealing with pigeons, we should still be thinking about the speed of the relevant entity's fluctuating characteristic as that characteristic moves. However, it is often easier to ignore this rule and refer to the speed of the wave. Note that ignoring the rule can make explanations quicker, but at the expense of often making them much more confusing. A better, but less succinct, term than "wave speed" would be "the speed of the entity's characteristic that is being described using waves" or maybe "entity speed". At the start of this chapter, I said that you should say that real-world waves do not exist – waves appear only in graphs and formulas. The trouble with maintaining this rule is that it is not adhered to, or even thought about, in most explanations, which is why the study of real-world waves can be so confusing. We end up with terms such as "wave speed", which at first glance seems obvious, but is much more complicated on closer inspection.

- Second, if we said that the wave did exist on its own, in this particular example, the wave does not really *move* over a distance – instead, it is *extended* over a distance. Therefore, we will need to think about the speed that the wave is extended over a distance.

If we ignore the first problem, we can say that the speed of a wave is the speed of the end of the curve as it progresses over a distance.

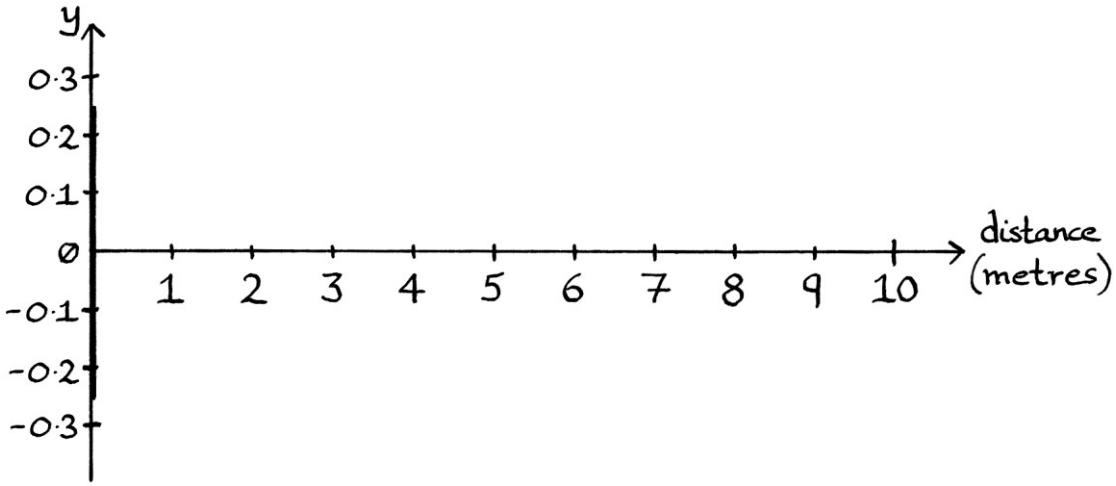
In this example, the wave speed is the speed of the pigeon. The pigeon creates the wave and the wave exists only where the pigeon is. If the pigeon moves at 10 metres per second, then the wave speed is 10 metres per second. The term "wave speed" is mainly useful as an abstract idea when dealing with theoretical distance-based waves. If we know the entity that the wave is describing, it is easier to think of the speed of the entity than it is to think of the speed of the wave. The wave speed and the speed of the entity that is portrayed by a wave will always be the same. [If we were thinking of sound, the source of the sound might be stationary, but the fluctuations in pressure that make up the sound (and would be portrayed by waves) would be moving. The wave speed would be the speed of any particular fluctuation in air pressure as it travelled.]

Distance-based and time-based waves

Not all entities that can be described with time-based waves can *sensibly* be described with distance-based waves. This is because not all entities that can be described with time-based waves are moving. If an entity is stationary, its wavelength will be zero, and its spatial frequency will be infinite. [In case this is not clear, its spatial frequency would be infinite, no matter how many cycles were completed, because it would have moved zero metres. We would divide the number of cycles completed by the distance (zero), and end up with an infinitely high number. Even if it completed only 0.00000001 cycles, the calculation would involve a division by zero.]

Technically, we *could* use distance-based waves for a stationary object, but there would be no point as the graph would not tell us anything useful.

For our wood pigeon, if it were standing stationary on a branch while flapping its wings at its usual rate, its distance-based Sine wave graph would look like this:



The formula for this graph can really only be described as so:

- If the distance is 0, then “y” is every value between −0.25 and +0.25
- Otherwise, “y” is undefined.

A common mistake that some people make is to think that the only waves in existence are radio and sound waves, and therefore, to conclude that every time-based wave has a corresponding distance-based wave. This mistake manifests itself in their thinking that every wave has a wavelength. If we were being pedantic, we could say that every wave *does* have a wavelength, and for stationary waves, the wavelength is zero. However, I would say it is better not to think of distance-based waves for stationary objects.

Whether it is possible to have an entity that can be described with a distance-based wave, yet which has zero period for its time-based wave, is a philosophical point that is above the level of this book. It would imply that an entity travelled a distance without it taking any time to do so. Such a thing might be possible in some branch of theoretical physics.

Radio antennas

You will probably experience the connection between time-based waves and distance-based waves most often when choosing antennas for radios. Antennas are usually constructed to a size related to the wavelength of the waves that we want the radio to receive or transmit. As radio waves move at a fixed speed (the speed of light), this wavelength will be directly related to the time-based frequency of the waves.

We will say that we want an antenna for a radio so that it can best receive a broadcast at 100 MHz. The frequency of 100 MHz is a time-based frequency from a time-based wave. Wavelength is the reciprocal of the *spatial* frequency. Therefore, we have to find the characteristics of the distance-based wave.

Electromagnetic radiation, of which radio waves are a type, travels at a speed of roughly 299,790,000 metres per second. [This is the speed of light]. We can also say that the *wave speed* of radio waves is 299,790,000 metres per second.

We know that for a time-based wave of 100 MHz, 1 cycle takes 1 ÷ 100,000,000 = 0.00000001 seconds (There are 7 zeroes after the decimal point). In other words, the period is 0.00000001 seconds.

To calculate the wavelength, we can use the formula that connects distance, speed and time:
distance = speed * time
... or its wave version:
wavelength = speed * period

Therefore, we calculate:
wavelength = 0.00000001 * 299,790,000 = 2.9979 metres.

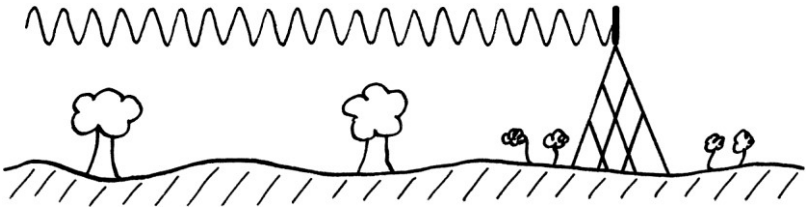
This means that the wavelength of an electromagnetic wave with a frequency of 100 MHz is 2.9979 metres.

If we wanted a very basic antenna for our radio, we could use a piece of wire that was a quarter of a wavelength long. It would be 2.9979 ÷ 4 = 0.749475 metres (74.9475 centimetres) long. [The reason it would be quarter of a wavelength long is based on the physics of antennas, and is too complicated a subject to discuss here.]

It is worth noting that a typical radio broadcast at 100 MHz would also involve waves at frequencies either side of 100 MHz, unless it were just a single wave. Therefore, an antenna would need to be able to receive a range of frequencies around that frequency, and not just at that exact frequency. A piece of wire would still be able to do this.

Radio antenna side note

Note that frequently in books on radio waves, you will see a drawing of a wave meeting or leaving an antenna, such as this:



Such a drawing is usually intended as a simple illustration. However, it has two potentially confusing ideas. First, it suggests that a wave is an entity in itself – this problem was explained at the beginning of this chapter. In the drawing, there is no object creating the wave – instead the wave is travelling on its own. The second problem with the illustration is that it makes it seem as if the significant attribute of the wave meeting or leaving the antenna is the *amplitude*, when in reality, it will be the frequency, period, spatial frequency, or the wavelength of the wave. [Because the speed of light is constant, these are all connected, so each one is proportional to the others].

Addition

As we already know, for some real-world entities that have a characteristic can be portrayed with waves, if two or more waves exist at the same time and place, they become added together. Obviously, this is not true for wood pigeons, but it is true for sound or radio waves. We can add the time-based waves together or we can add the distance-based waves together.

Three types of wave

So far, in this book, we have seen three types of wave:

- Angle-based waves
- Time-based waves
- Distance-based waves

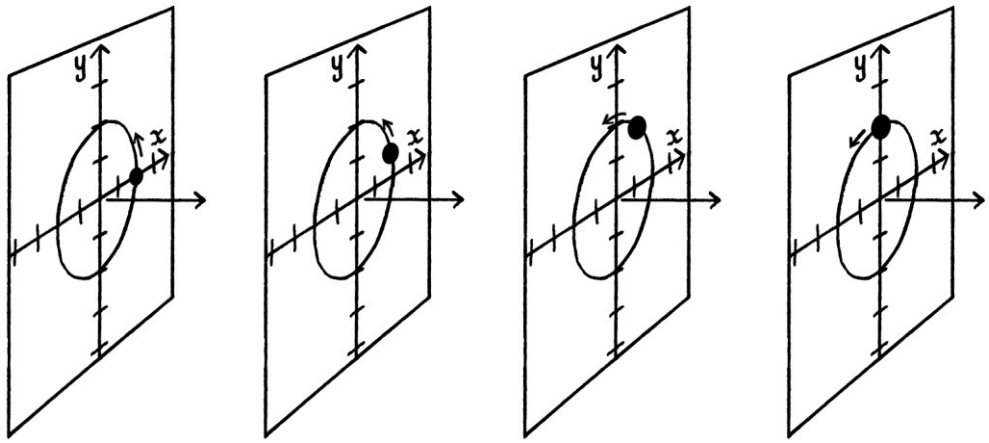
Time-based waves and distance-based waves are both essentially types of angle-based waves, where there is a fixed scaling of the angle based on time or distance.

Circles

A time-based Sine wave can be thought of as portraying the y-axis positions of an object as it rotates around a circle at any moment in time. Although a time-based Sine wave might not literally be describing an object rotating around a circle, we can think of it as doing so, because doing that helps us better understand the wave. For our pigeon's time-based Sine wave, although there are not literally any circles involved, we can still think of the wave in terms of a circle. One of the advantages in doing so is that it helps us understand the meaning of phase in the wave formula.

When it comes to a distance-based Sine wave, thinking of the circle becomes slightly more complicated. One way is to think of an object rotating around a circle at the same time as the circle *and the axes on which the circle is placed* move at a particular speed in a straight line. In this way, a distance-based Sine wave gives the y-axis value of the object rotating around the circle when the circle itself is at evenly spaced distances from the starting place. Note that that the axes must move with the circle, and the circle must keep its position on the axes.

This is easier to portray with three-dimensional pictures:

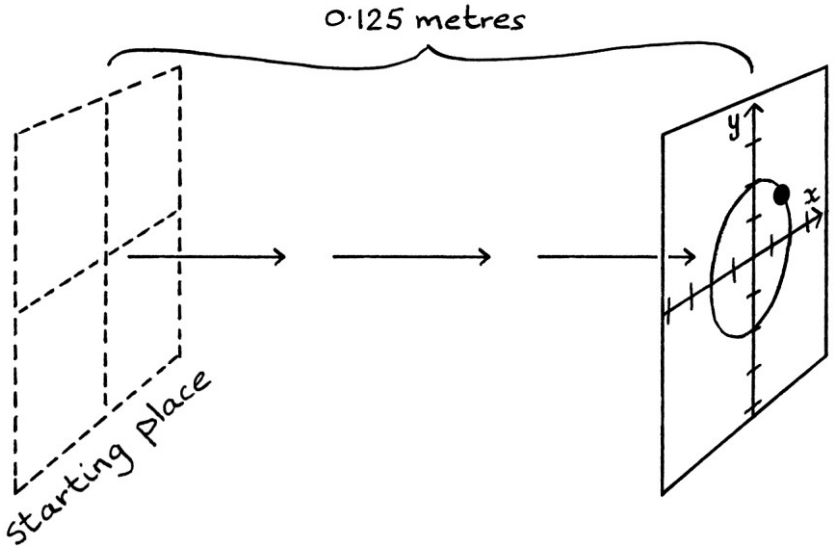


If there is zero mean level, the centre of the circle will be on the origin of the axes. If there is a non-zero mean level, the centre of the circle will be elsewhere, but it will still be fixed to the axes.

As an example, we will look at this distance-based wave (in radians):
"y = 2 sin (2πx)"

We will assume that the corresponding Cosine wave has zero mean level, and so the circle is centred on the origin of the axes.

When "x" is 0.125 metres, the origin of the moving axes, and the distance-based circle from which the wave is derived, will be 0.125 metres from their starting position. At this place, the object has a y-axis value of:
2 sin (2π * 0.125)
= 1.4142 units.



If we also knew the speed of the circle and the axes, we would be able to tell how long they had been moving, and at what time the object was at y = 0.7071 units.

For distance-based Cosine waves, we do the same thing. The only problem is that we have "x" indicating the distance of the axes from the starting place, and we also have "x" indicating the position of the object on the axes. Therefore, if we were to consider the distance-based circles from which a Cosine wave is derived, it would make sense to use a letter other than "x" in the formulas. When we have this problem in the next chapter, we will use the letter "d" for distance.

A "distance-circle" is a way of portraying an aspect of a wave from a real-world entity. Although the entity might not have any connection to circles at all, if we have a distance-based wave that shows the fluctuating characteristic of an entity, we can draw the circle from which that wave could have been derived.

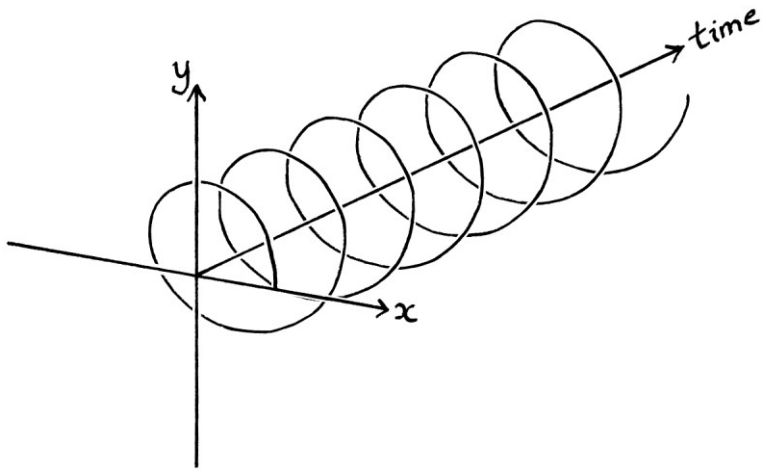
Angular spatial frequency

If we consider *time* when examining an object rotating around a circle, we can have frequency, which is the number of cycles completed per second, and we can have *angular* frequency, which is the number of angle units (radians or degrees) completed per second. These ideas translate to the time-based waves that describe the y-axis and x-axis positions of the object – we can still talk about angular frequency for a Sine wave and a Cosine wave, even though the idea of an angle is slightly more obscure on a wave than it is on a circle.

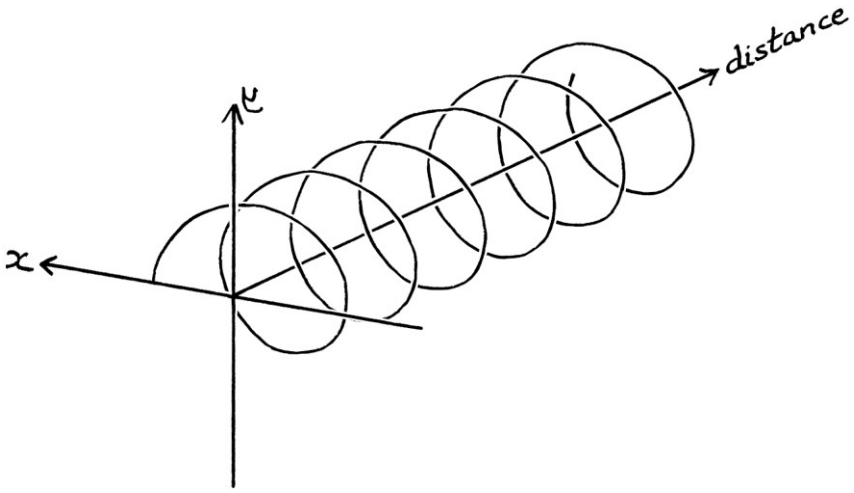
As we know, when it comes to distance-based waves, there is spatial frequency, which is the number of cycles completed per metre. As you might expect, we can also have angular spatial frequency, which is the number of angle units completed per metre. This idea is easier to contemplate on an object rotating around a *moving* circle and axes than on a distance-based wave.

Distance helices

In part one of this book, we saw time-based helix charts. In these, an object rotated around a circle or a shape while moving down the time axis.

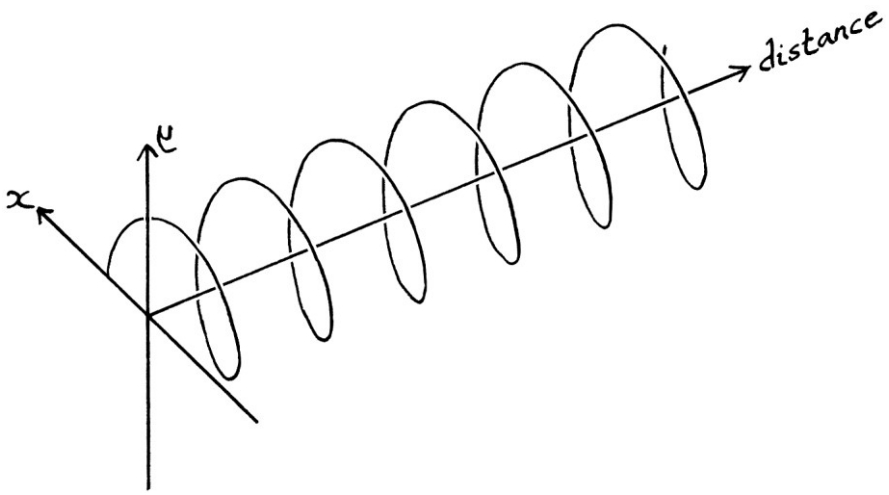


We can also have distance-based helix charts. These have x and y axes, and a third axis of distance. With the time helix charts seen in this book, the x and y axes have been facing the viewer, while the time-axis has pointed away from the viewer. With the distance helix chart, it is easiest to think of the x and y axes facing *away* from the viewer with the distance axis moving away from the viewer. [I have drawn the "y" backwards to reinforce how the axes are facing away from us.]



If the axes are drawn to the same scale as each other, the distance-based helix will portray the literal position of an object as it rotates around a circle or shape while that circle or shape moves over a distance.

If we view the distance helix chart from the side with the y-axis pointing upwards and the distance axis pointing to the right, we will see the distance-based Sine wave. If we view the distance helix chart from the *top* with the x-axis pointing upwards and the distance-axis to the right, we will see the distance-based Cosine wave. [This is different from the time helix chart, where we need to view the helix from underneath to see the time-based Cosine wave.] The viewing angles are easier to see in this drawing of the helix from a slightly different angle:



More pigeon thoughts

In our wood pigeon example, our theoretical pigeon always flies at 10 metres per second and flaps its wings at a rate of 1 cycle per second. Its time-based wave formula is:
"y = 0.25 sin (2π * 1t)"

Its distance-based wave formula is:
"y = 0.25 sin (2π * 0.1x)"

In reality, the faster a pigeon flaps its wings, the faster it will travel. If we make the (slightly false) assumption that the speed of a bird's flight is based solely on the amount of air that its wings can push through, then the distance the pigeon travels will be directly proportional to how quickly it can move its wings. We will say that our pigeon flaps its wings twice as quickly (2 cycles per second), resulting in it doubling its speed to 20 metres per second. Despite the time-based frequency and the speed doubling, the *spatial frequency* will stay the same. The pigeon will still cover the same distance with one flap cycle.

Its time-based wave will be:
"y = 0.25 sin (2π * 2t)"
... and its distance-based wave will stay the same as before:
"y = 0.25 sin (2π * 0.1x)".

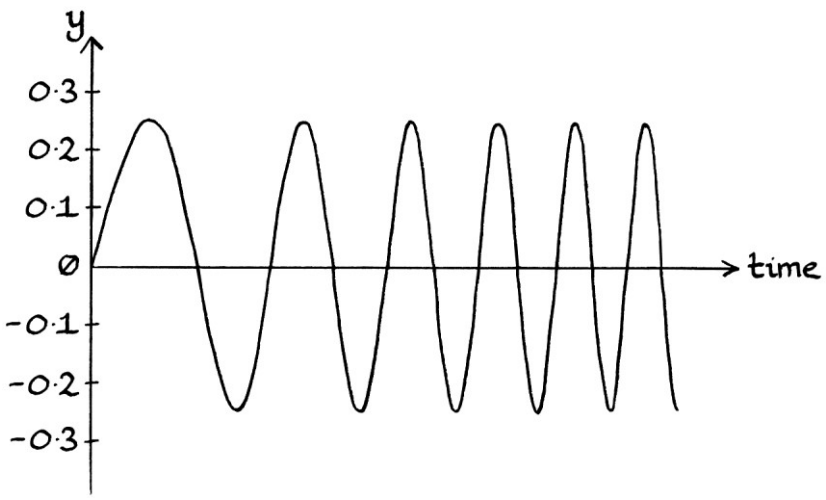
This might seem counter-intuitive, but we can show it is true by using the formula:
wavelength = speed * period

For our faster pigeon, we have:
(1 ÷ 0.1) = 20 * (1 ÷ 2)
... which is:
10 = 20 * 0.5
... which is:
10 = 10
... which shows that the idea is true.

The spatial frequency stays the same because one cycle of wing movement still covers the same amount of distance. The pigeon is moving through those cycles more quickly (a faster temporal frequency), but each cycle still only pushes the pigeon the same distance onwards. One movement of the pigeon's wings moves it forwards by a particular amount, and it does not matter how fast that wing movement is performed.

Acceleration

A similar situation occurs if our wood pigeon flies at an ever-increasing speed. For the pigeon to move faster through the air, it must be flapping at an ever-increasing frequency. Its time-based wave will have ever-faster cycles. It might look like this:



However, the distance-based wave will still be the same as if the pigeon were flying at any speed. This is because the movement of the pigeon's wings pushes it through the air by a particular amount. At any distance that the pigeon has moved, the state of its wings will be the same. It does not matter at what speed our pigeon moves, or whether that speed varies, the distance-based wave will always be the same. This idea will be clearer when we look at the toy tortoise example in the next chapter. [The toy tortoise's head moves in and out according to the movement of its wheels. If we move the tortoise more quickly, its head moves in and out more quickly. Therefore, at any distance from the starting place, its head will be in the same position – it does not matter how quickly or slowly we pushed the tortoise to get there. The pigeon's position and the extent to which it has moved its wings are similarly linked together, but the connection is slightly harder to see because it is travelling through air.]

Misguided generalisations

It is common for people to make incorrect generalisations about waves, such as "all waves add together in this way" or "all waves behave in this way". Such generalisations are usually based on the idea that there are only two types of waves: sound waves and electromagnetic radiation waves (light and radio waves). Obviously, there are countless entities other than sound and electromagnetic radiation that have characteristics that can be described using waves. By starting with an example of a wood pigeon's wings, it should be clear that it is risky to make any generalisations about real-world waves. In Chapter 32, such generalisations will seem even more misguided. Sometimes, you will read books about far more complicated ideas than those in this book, where an author implies that they do not know about other types of wave. It can be difficult to know if these are absent-minded mistakes, or if the author really does not know about what many people would say are the basics of waves. This is similar to how you can sometimes read something about the Fourier transform by an author who appears not to understand what Sine and Cosine are. Note that just because an author makes misguided generalisations about waves does not mean that the other things they say will not be useful or informative.

Conclusion

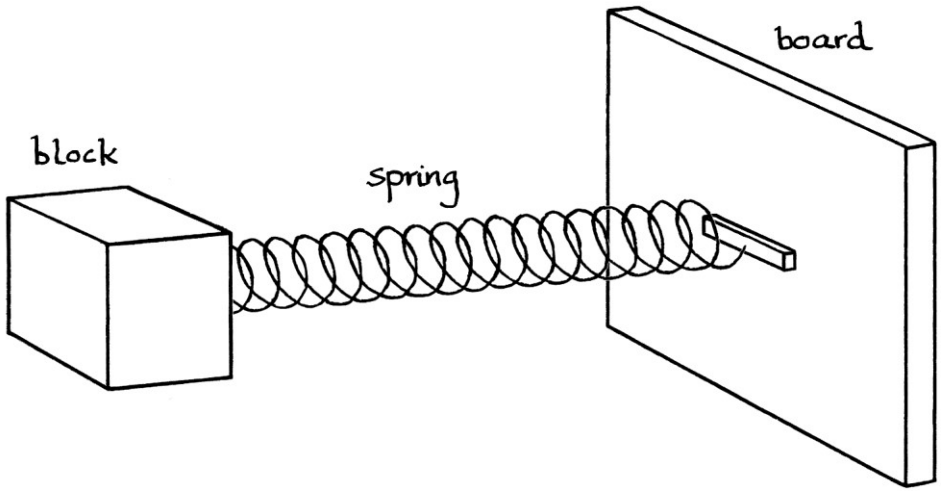
In this chapter, we have seen both time-based waves and distance-based waves. The two concepts are similar, but a time-based wave is usually easier to understand. Whenever someone mentions “wavelength” when referring to radio, light or sound waves, they are ultimately talking about distance-based waves, but often, they might not acknowledge or realise the fact. Period and wavelength are often mixed up, and this is often due to temporary absent-mindedness. It is easy to find people who understand the concept of wavelength, but who are unaware of, or do not understand, the concept of distance-based waves. This is usually because there has never been any need for them to be taught about distance-based waves.

Chapter 32: More real-world waves

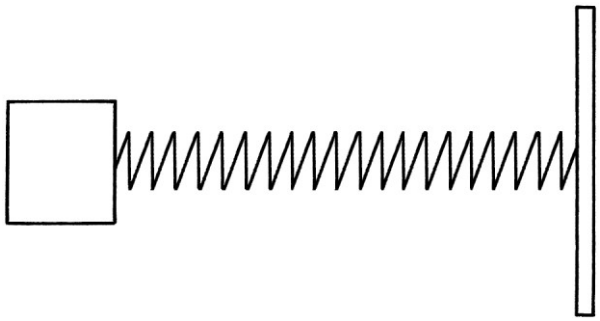
In this chapter, we will look at examples of more complicated real-world waves.

A block, a spring and a board

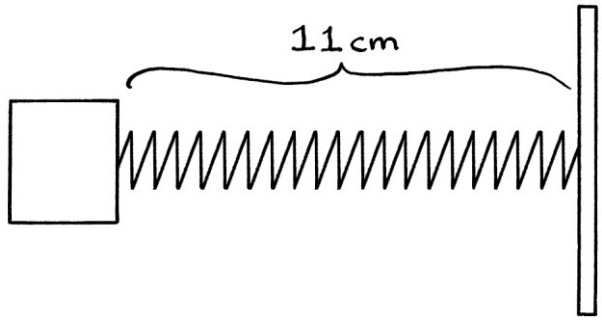
For the next real-world example, we will imagine a block of wood connected to a strong spring, which in turn, is connected to a rectangular board:



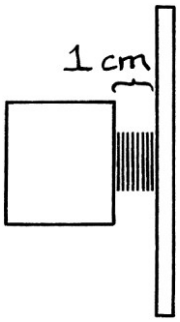
Viewed side on, it looks roughly like this:



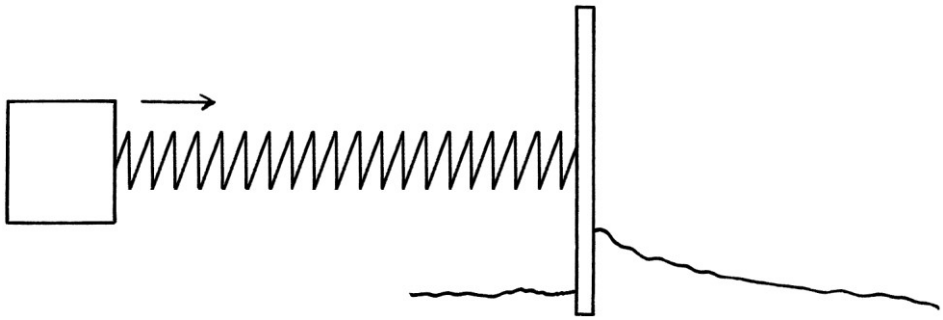
If the spring is compressed, it will push back and return to its original size. The spring is 11 centimetres long when uncompressed:



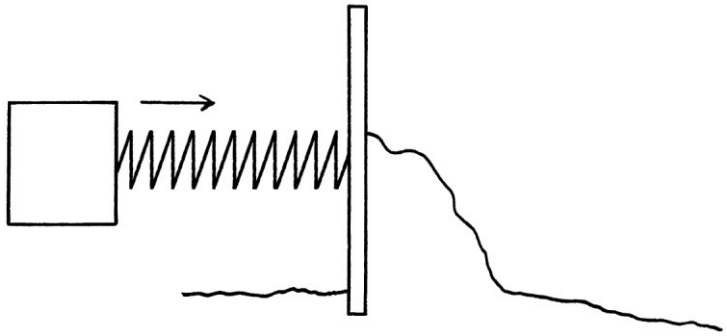
The spring is 1 centimetre long when fully compressed:



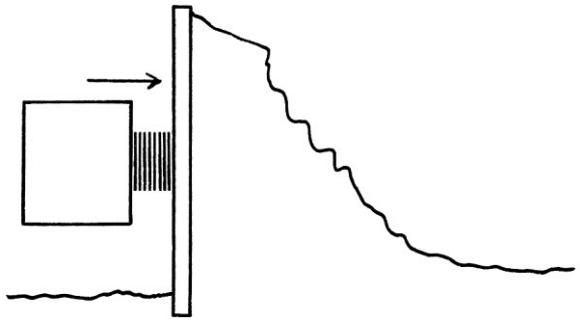
Now imagine someone pushing the block of wood through the top of a layer of sand at a constant speed, with the board digging into the sand:



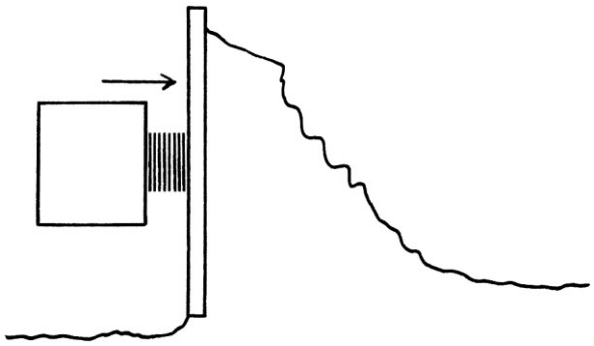
As the block is pushed through the sand, the sand builds up against the board causing the spring to be compressed:



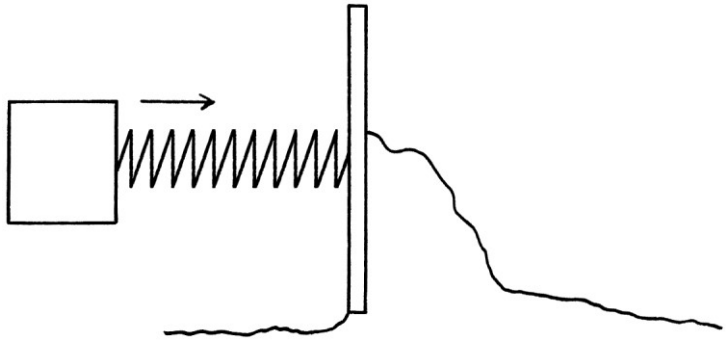
Eventually, the spring will be fully compressed:



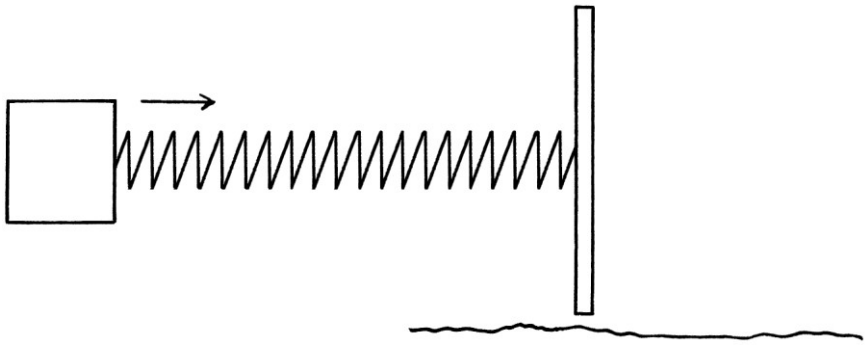
When the spring is fully compressed, the person pushing the block raises it slightly. This raises the board and lets the sand slide off underneath as the block and board continue to move forward.



As the amount of sand pushing against the block decreases, the spring expands and the board moves further away.



When all the sand has fallen off the board, the spring will be at its full length:



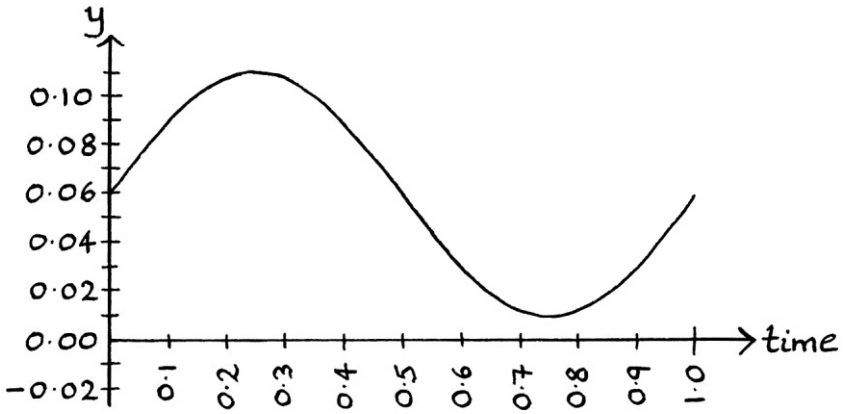
At this time, the person pushing the block lets sand build up against the board again, and the cycle repeats.

A summary of what is happening is as follows:

- The block is being pushed at a constant speed.
- The build-up of sand causes the spring to be gradually compressed.
- Once the spring is fully compressed, the block and board are raised slightly, and the sand gradually moves off the board. When this happens, the spring expands.
- Once all the sand has fallen off, the board is pushed into the sand again, and the process repeats.
- By good fortune, the length of the spring at any particular time can be calculated with the Sine function.

The spring completes one cycle of being compressed and then expanding in 10 seconds. We can portray the length of the spring over time using a time-based Sine wave. The y-axis is the length of the spring over time as the device is pushed through the sand. The y-axis units are metres.

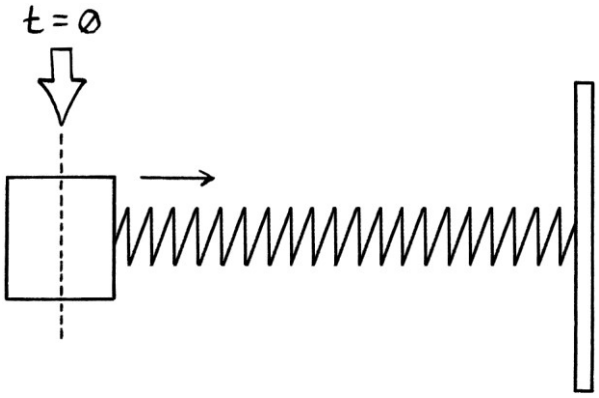
The graph looks like this:



Note that the whole wave is centred around y = 0.05 metres (5 centimetres). This is because the spring's length never becomes negative. Its minimum length is 0.01 metres (1 centimetre). It fluctuates between 0.01 metres and 0.11 metres.

In the example of a wood pigeon in the previous chapter, the pigeon's wing tip wave referred to the up and down motion of the wing tips. In this example, the spring-length wave refers to the in and out motion of the spring. For the pigeon, the y-axis of the graph rose and fell with the rise and fall of the wings. With this spring device, the y-axis of the graph rises and falls with the expansion and contraction of the spring.

The behaviour of the spring can be portrayed using all the usual attributes of waves. We will say that the time is the number of seconds since the centre of the block passed a particular starting place. Note that this is an arbitrary place on the block from which to measure. We could have chosen countless other places.



The other attributes of the wave are as follows:

- The amplitude of the wave is half the distance from the maximum length of the spring to the minimum length of the spring – in other words, it is half the distance from when it is being fully compressed to when it is not being compressed at all. The spring's maximum length is 11 centimetres; its minimum length is 1 centimetre. Therefore, its amplitude is: (11 – 1) ÷ 2 = 5 centimetres. If we had a longer spring, we would have a larger amplitude; if we had a shorter spring, we would have a smaller amplitude. We will give the amplitude in metres, so it will be 0.05 metres.

- The frequency of the wave is the number of times the spring contracts and expands in one second. The spring completes one cycle in 10 seconds. Therefore, its period is 10 seconds, and its frequency is 0.1 cycles per second.

- If we think of a cycle of the wave starting when the spring is halfway through an expansion, then we could think of the phase of the spring as how far along the cycle it was when the middle of the block passed the starting place. In this case, we will say the phase is zero, which makes things easier for this example. This means that the spring is at half its length when it starts, and it is in the process of expanding.

- The mean level of the device's wave is the average y-axis value over one cycle (which is the same as the average over all time if the device moves forever). As the maximum value is 11 centimetres and the minimum value is 1 centimetre, the mean level will be 6 centimetres, which we will phrase as 0.06 metres. Whatever the length of the fully extended spring, there will always be a positive mean level because the length of the compressed spring never becomes negative. The only way the mean level could ever be zero is if the spring had a negative length when fully compressed. If we wanted to change the mean level of the device, we would have to get either a longer spring or a spring that compressed to a smaller size.

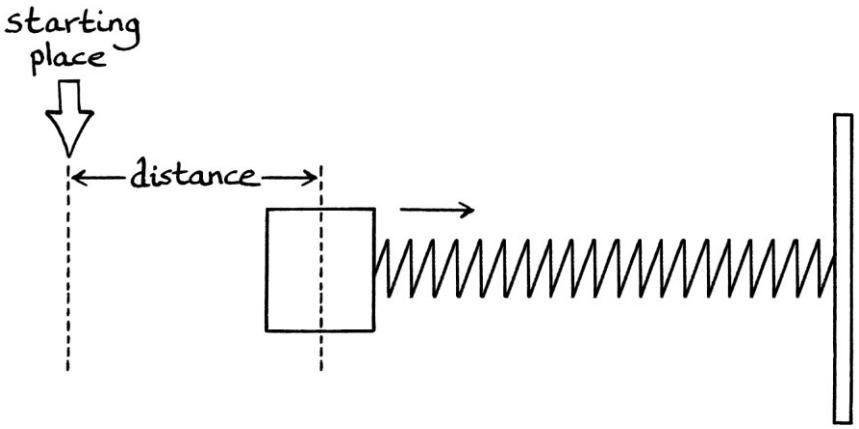
The formula of the device's time-based wave is:
"$y = 0.06 + 0.05 \sin (2\pi * 0.1t)$"
... where:

- The mean level is 0.06 metres.
- The amplitude is 0.05 metres.
- The frequency is 0.1 cycles per second.
- The phase is zero, which means that the device started with the spring at half its length and about to expand.
- "t" is the time since the middle of the block passed the starting place.

When thinking of the device's movement through the sand, it is important to notice that at no time does any part moves backwards. Everything is moving forwards, but at different rates. The block moves forwards at a constant speed, the board at the front end of the spring moves forwards at varying rates. At no time does the board move backwards. The spring's length varies over time as it expands and contracts, and the Sine wave reflects the spring's length. It would be easy to think, mistakenly, that the Sine wave meant something was moving backwards.

Distance-based wave

As the device is moving while the spring contracts and expands, we can also portray its characteristics using a distance-based Sine wave. In this way, the wave will be portraying the spring's changes in length as the device moves. The distance will be measured from the middle of the block of wood to the place at which the device started:



We could just as easily measure to where the spring is joined to the block, or the other side of the block, or in fact, any place on the device (or not on the device) that moved at the same rate as the block. However, we need to be consistent with the starting event that dictated the meaning of the time in the time-based wave. For example, if the time in the time-based wave referred to the number of seconds since the tail end of the block passed a particular point, then the distance would need to refer to the metres that the tail end had travelled since it passed that same point.

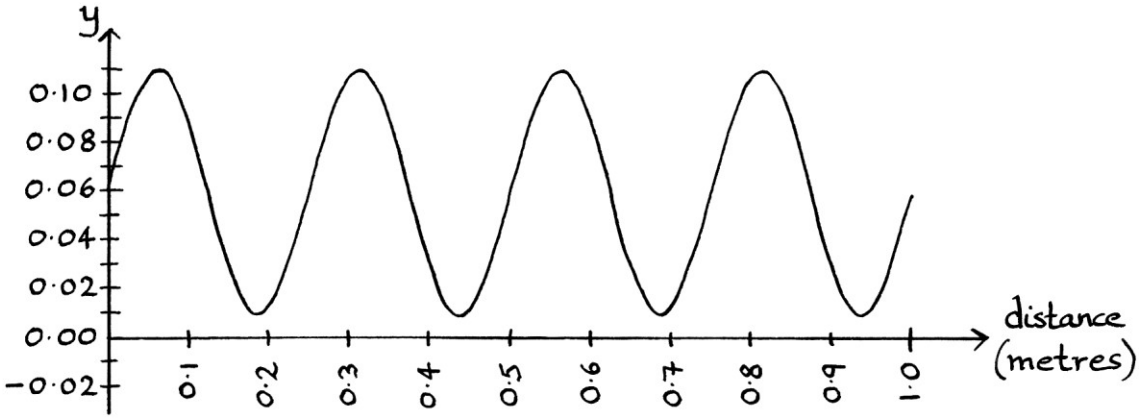
The block of wood is the only part of the device that moves at a constant speed, so all measuring points would need to be related to the block. As with the wood pigeon, it does not particularly matter to where we measure on the block, as long as we are consistent. It is also worth noting that the measurement does not have to be to the same place as where the amplitude is measured. [I intentionally chose the middle of the block instead of where the block was joined to the spring to make this idea clear.]

We will say that the block of wood is being pushed at 5 centimetres per second, which is 0.05 metres per second. Given that the period of the device's wave is 5 seconds, this means that the wavelength is: 0.05 * 5 = 0.25 metres. One complete contraction and expansion of the spring (one cycle) occurs over 0.25 metres. This means that the spatial frequency of the wave of the device is 1 ÷ 0.25 = 4 cycles per metre.

The distance-based wave formula for the device is:
"$y = 0.06 + 0.05 \sin (2\pi * 4x)$"
... where:

- The mean level is 0.06 metres, and refers to the average length of the spring.
- The amplitude is 0.05 metres
- The spatial frequency is 4 cycles per metre.
- "x" refers to the distance in metres from the middle of the block to the starting place.
- The phase is zero.

The graph for this is as so:



Calculations

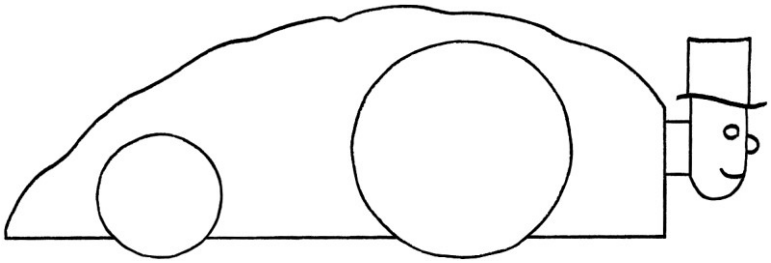
From the time-based formula and the distance-based formula, we can work out the length of the spring at any particular time, and we can work out the length of the spring for when the middle of the block of wood is at any particular distance from the starting place.

At, say, 1.1 seconds, the length of the spring will be:
0.06 + 0.05 sin (2π * 0.1 * 1.1)
= 0.09187 metres.

When the middle of the block of wood has been pushed 0.3 metres, the length of the spring will be:
0.06 + 0.05 sin (2π * 4 * 0.3)
= 0.1076 metres.

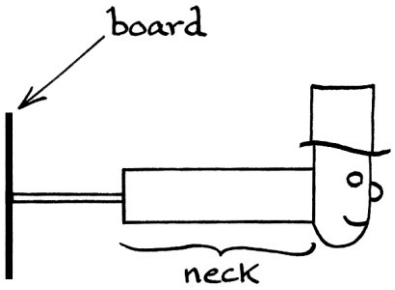
A toy tortoise

We will now imagine a child's toy that looks like this:



As the front wheels rotate, the head moves in and out. The length of the neck at any distance or at any time can be calculated with the Sine function.

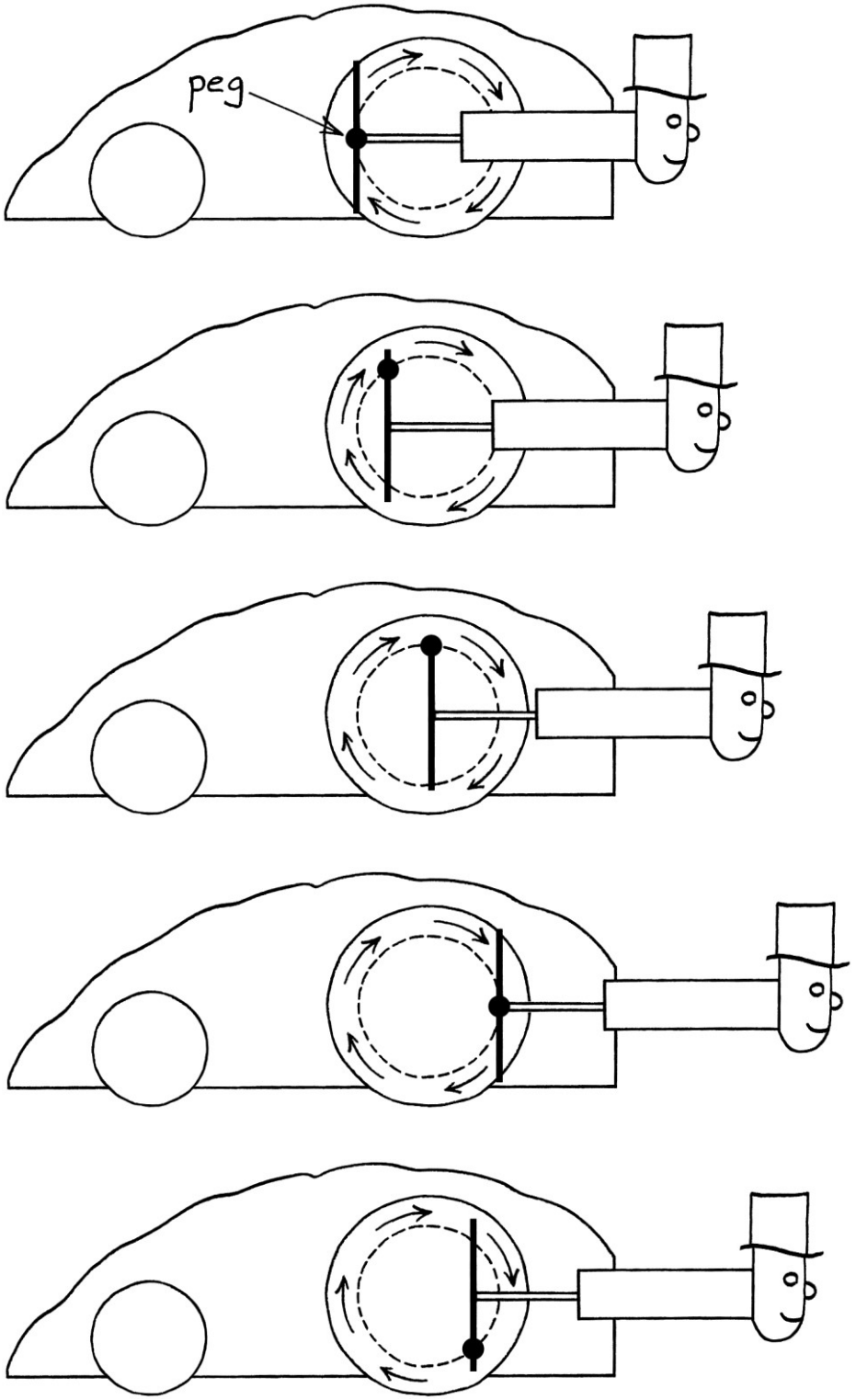
There are two significant sections to the mechanism of the toy. The first consists of a small board, connected to the neck and the head. These are all fixed together. The board has a slot down each vertical side.



The second significant section consists of the two front wheels. They each have a fixed internal peg that fits into the slot on either side of the small board. The pegs can slide up and down the slots. As the front wheels rotate, the pegs slide up and down the board, thus pushing it horizontally backwards and forwards.

For the sake of giving the toy a name, we will say it is a toy tortoise, despite how it looks.

The following pictures show the internal mechanism of the tortoise as it moves:



[Note that I made the mechanism slightly more complicated than it needs to be for a child's toy. This is so that the head moves in and out according to the Sine function. If the toy used a more simple crank mechanism instead, the neck would have a slightly different movement.]

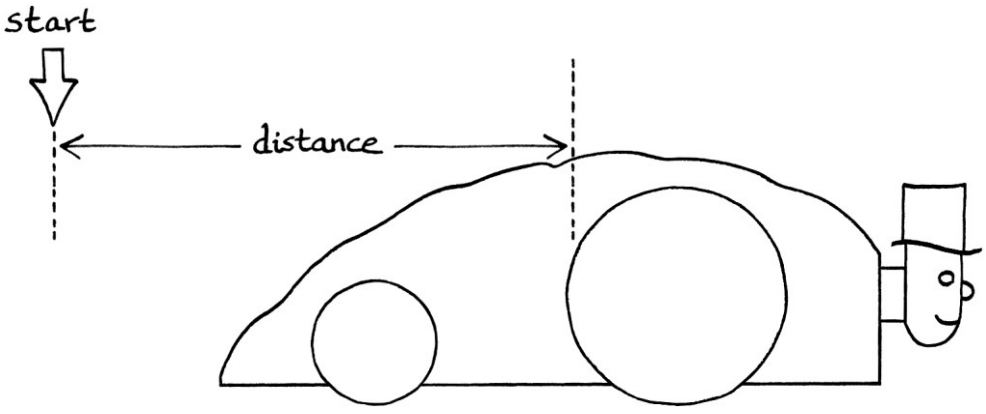
One rotation of the front wheels causes the neck of the tortoise to extend and retract once. The tortoise needs to move 20 centimetres for this to happen. Therefore, the wavelength is 20 centimetres, or 0.2 metres, and the spatial frequency is 1 ÷ 0.2 = 5 cycles per metre. We can portray the length of the exposed neck using both time-based and distance-based Sine waves. In this example, we will just focus on the distance-based wave.

The neck's length varies from being 0.7 centimetres (0.007 metres) long to 5.2 centimetres (0.052 metres) long.

The amplitude of the wave will be half the distance from the maximum to the minimum lengths:
(0.052 – 0.007) ÷ 2 = 0.0225 metres.

The mean level will be the average of the maximum and minimum:
(0.052 + 0.007) ÷ 2 = 0.0295 metres.

The distance in the formula ("x") will refer to how far the centre of the tortoise has moved from its starting position. This is an arbitrary choice, and we could just as easily have chosen a different point on the tortoise.

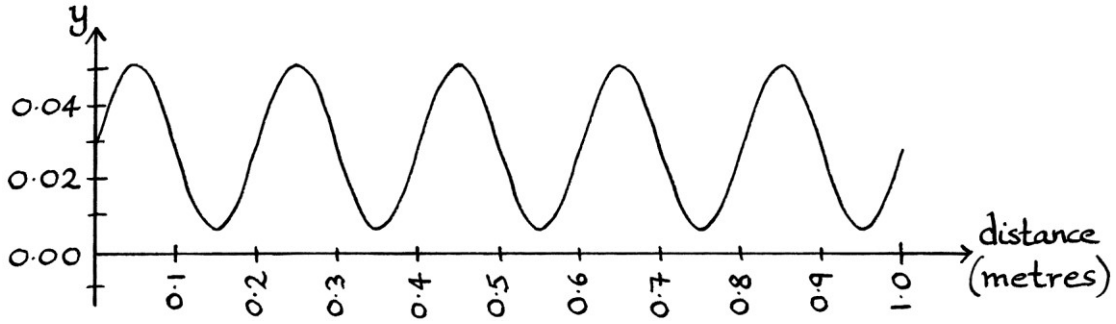


We will say that the phase is zero radians. This means that when the centre of the tortoise was at the starting place, its neck was 0.00295 metres long, and about to become longer.

The formula for the length of the neck (the distance of the head from the body) as the tortoise travels over a distance, will be:

"$y = 0.0295 + 0.0225 \sin (2\pi * 5x)$"

The graph for this is as so:



We can use the formula to calculate how far the head will extend from the tortoise after the tortoise has travelled 0.23 metres:
0.0295 + 0.0225 sin (2π * 5 * 0.23)
= 0.04770 metres.

Therefore, when the tortoise has travelled 0.23 metres, its head will be sticking out 0.04770 metres, which is 4.77 centimetres.

Note that it is irrelevant how fast the tortoise moves – its state will always be the same at 0.23 metres. If the tortoise moves faster, the pegs in the front wheels will move faster, and the in and out motion of the head will be quicker. If the tortoise moves more slowly, the pegs will move more slowly, and the head will move more slowly. The length of the neck is entirely related to the position of the front wheels, and the position of the front wheels is entirely related to the distance that they have travelled. It is irrelevant how quickly they travelled to get there. All of this means that the speed of the tortoise does not affect the spatial frequency of the neck's length.

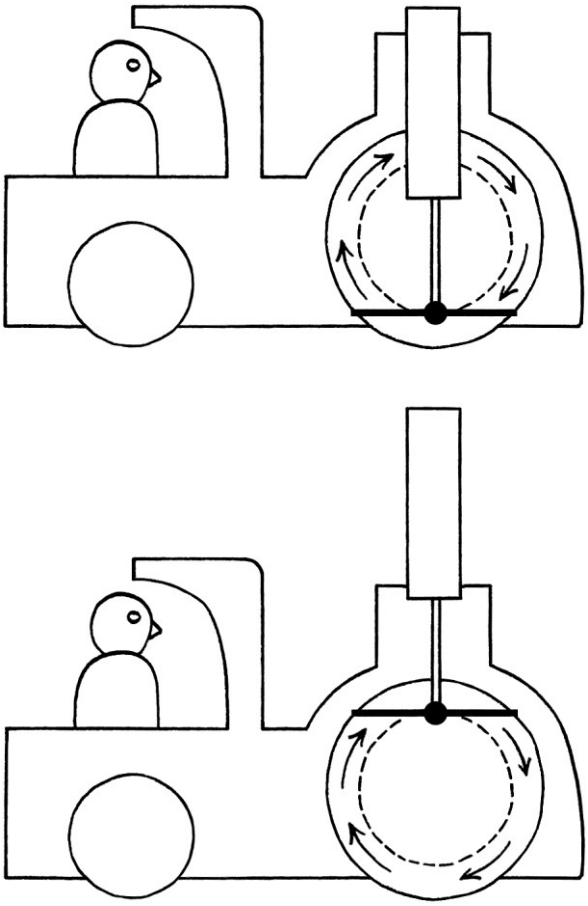
Speed is not relevant to this particular distance-based wave because the instantaneous amplitudes of the wave and the distance travelled are locked together by the nature of the mechanism. Speed's irrelevancy to this distance-based wave is more obvious in this example than in the wood pigeon example in the previous chapter. When the wood pigeon flapped twice as fast, it travelled twice as fast, but its spatial frequency remained the same. In this example, it is clearer why that should be. Note that it is possible to have a distance-based wave where the speed *_is_* relevant to the instantaneous amplitudes of the wave. For the tortoise, we could change the mechanism to one controlled by an electronic circuit and motor that moved the head in and out at a rate related to the speed of the tortoise. Therefore, it is not always the case that the speed of a wave is unrelated to its spatial frequency.

Whether speed is relevant to a *time-based* wave depends on the nature of the entity being examined. In this example, the faster the tortoise moves, the faster its head will move in and out. Therefore, speed *is* relevant to the time-based wave in this example. If the tortoise's neck were controlled by an electronic circuit and motor instead of the current mechanism, it would be possible to have the speed of the tortoise be unrelated to the tortoise's time-based wave. Whether speed is relevant to a time-based wave or not, speed itself does not appear in either a distance-based wave formula or a time-based wave formula, and it cannot be deduced by just looking at either wave formula or graph. Speed is the factor that *connects* the two types of waves, but it does not appear in either wave.

Whereas the pigeon and the block, spring and board produced pure waves, there were no circular entities producing those waves. The pure waves were really coincidences. In this example, the movement of the neck is controlled by the circular movement of the wheels. Therefore, the Sine wave is directly related to a circle. [Strictly speaking, the movement of the neck is controlled by the *horizontal* axis of the wheel, so the movement is actually portraying a circle's Cosine wave.]

A toy train

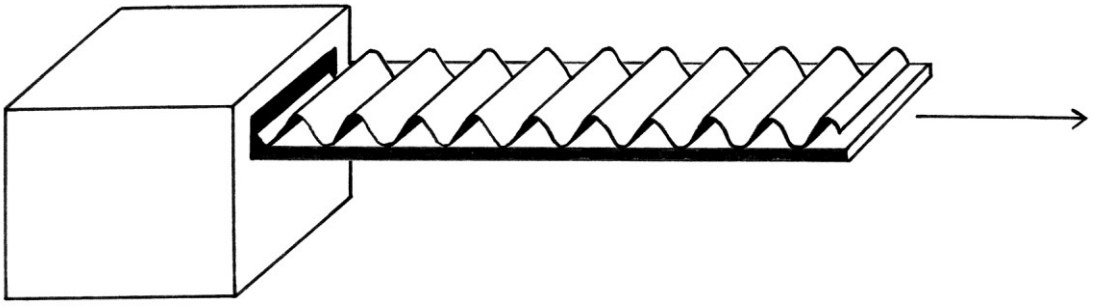
We can have a toy steam train that works in the same way as the toy tortoise. With the toy train, as the front wheels rotate, the mechanism raises and lowers the chimney.



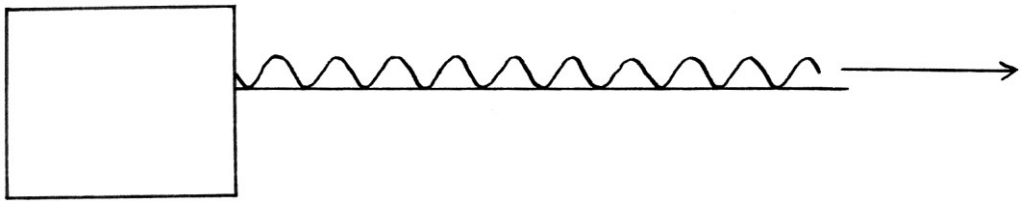
We will say that the moving parts of the toy train have the same dimensions as those of the toy tortoise, and that train moves at the same speed as the tortoise. We will say that the amplitude refers to the length of the chimney projecting from the train. We will measure time from when the centre of the train passes a particular place, and we will measure the distance from the centre of the train to that place. We will say that at the starting place at t = 0, the chimney is half way out and about to move further out, in which case, the phase will be zero radians. Given all of this, the time-based and distance-based waves describing the length of the toy train's chimney will be *identical* to the waves describing the length of the toy tortoise's neck.

A sliding corrugated metal sheet

In this example, we will imagine an eternally long piece of corrugated metal sheet coming out of the machine that makes it, and then moving across a conveyor belt in a straight line:



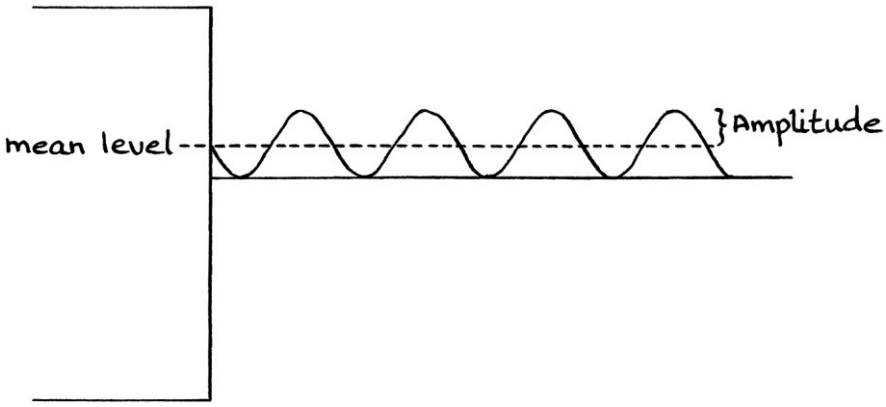
The corrugated metal sheet has a cross section that matches the shape of a Sine wave. The height of the metal sheet is 10 centimetres or 0.1 metres. The distance between the peaks is 20 centimetres or 0.2 metres.



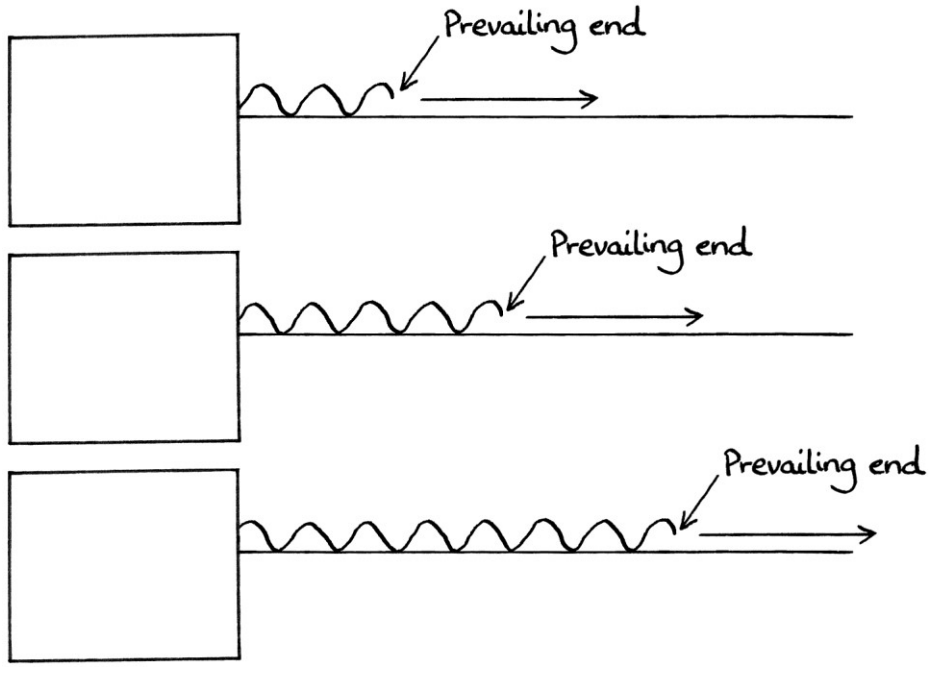
The metal sheet moves at 2 metres per second. To make the maths simpler, we will say that the metal sheet is infinitely thin. [If it had a thickness, then that would need to be taken into account in our calculations.]

We can use waves to describe the movement of the metal sheet, but doing so requires more thought than with the previous examples. Our time-based and distance-based waves will refer to the height of the cross section of the metal sheet in relation to the conveyor belt it is sitting on. In other words, the y-axis values will be the height of a particular place on the metal sheet above the conveyor belt. Without needing to know how we are going to transfer the details of the metal sheet to a time-based or distance-based wave, we can know the amplitude and the mean level of the waves.

The amplitude will be 0.05 metres. As we are measuring to the surface of the conveyor belt, the mean level will be the same as the amplitude. It will also be 0.05 metres.



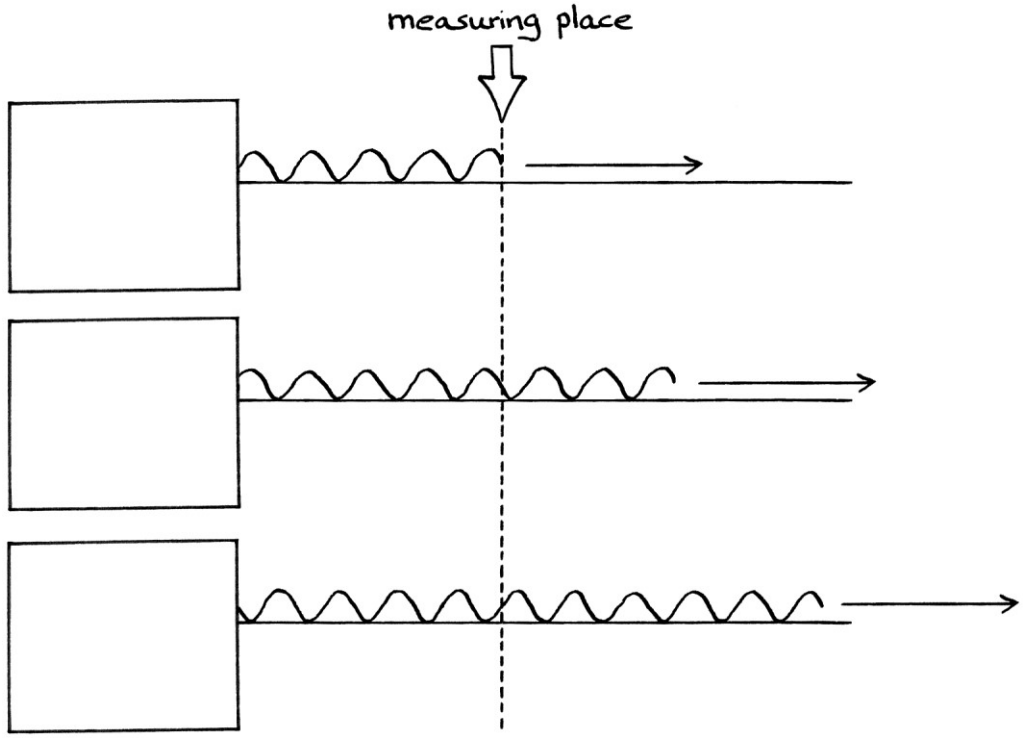
This is all straightforward, but we have to decide to which part of the metal sheet we are measuring at any time or place to obtain our y-axis values. If we choose to draw a graph showing the heights of the front of the end of the metal sheet (as in the part that precedes the rest of the metal sheet as it travels), we would find that all the y-axis values would be the same.



No matter at which time, or at which distance, we measure the height of the end of the metal sheet, it will always be the same because we are measuring the same part of the metal sheet. The end of the metal sheet has a y-axis value of 0.05 metres. The time-based wave would have a formula of “y = 0.05” because the end of the sheet is always the same at all times. Similarly, the distance-based wave

would have a formula of "y = 0.05" because the end of the sheet is always the same for all distances. If we pick any point of the metal sheet as it moves, we will have the same problem.

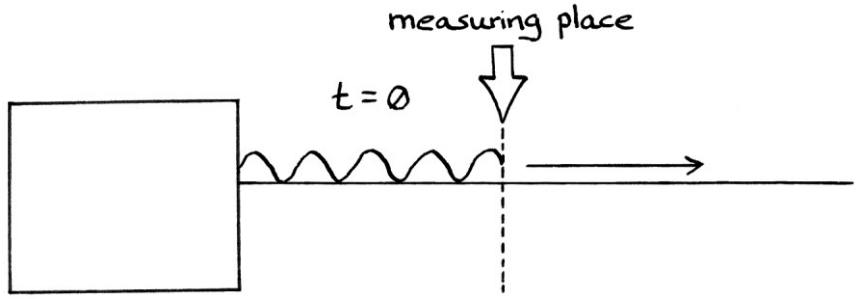
The solution is to measure the height of the metal sheet at a fixed location that is not on the metal sheet, but which the metal sheet passes. We will do this by measuring the height of the metal sheet as it passes a particular place:



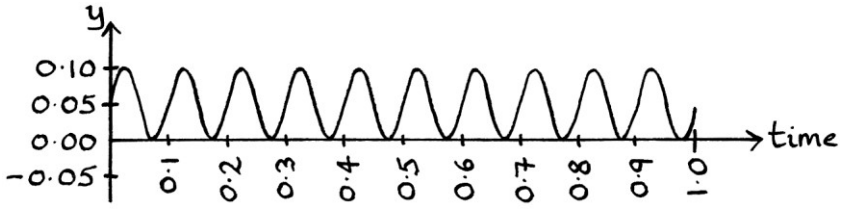
When we do this, we can create meaningful wave formulas for the moving metal sheet. We are interested in the sheet only where it passes the measuring place. Before and after that point, the nature of the sheet is irrelevant to us. [We could also measure the height of the metal sheet where it comes out of the machine, but measuring it as it passes the post makes the measuring more analogous to other types of waves such as sound waves.]

We know that the amplitude will be 0.05 metres – this is half the distance from the peaks to the dips of the corrugated metal sheet. The mean level is 0.05 metres. The wavelength of the metal sheet is 0.2 metres – this is the literal distance between each peak or each dip. The speed is 2 metres per second. Therefore, the period is: 0.2 ÷ 2 = 0.1 seconds per cycle. The frequency is: 1 ÷ 0.1 = 10 cycles per second.

In the formula, “t” will be the time since the front of the metal sheet was exactly at the measuring place:



It is worth noting that our actual measurements will result in a mirror image of the shape of the metal sheet. The measurements as drawn on a graph over time will be the reverse of the actual metal sheet. This means that our measured Sine wave will have zero phase. It will start with “y” at the height of the mean level and rising. Our time-based wave looks like this:



The time-based formula that shows the measurements at the measuring place is:
“y = 0.05 + 0.05 sin (2π * 10t)”

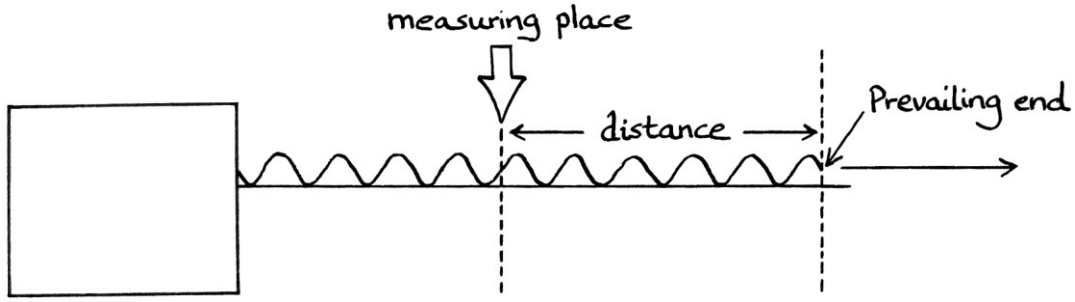
This wave formula shows the height of the metal sheet at the exact time that it passes our measuring point. It is worth noting that we knew the frequency and the period before we actually knew from when “t” was counting. [We also know the wavelength before we have decided from where “x” in the distance-based wave will be measuring.]

As an example of the time-based formula in use, we will enter the value of 0.17 seconds into the formula:
0.05 + 0.05 sin (2π * 10 * 0.17)
= 0.002447 metres.

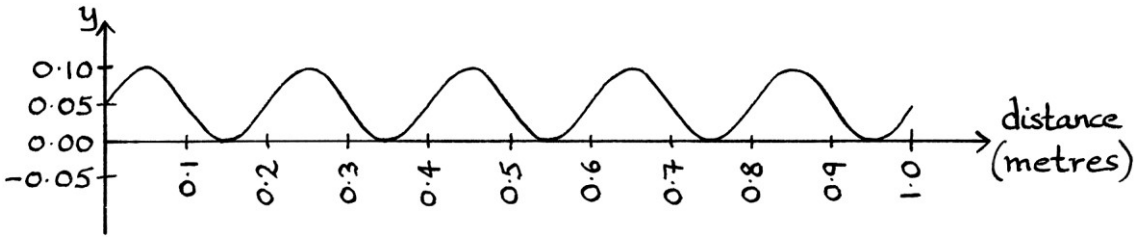
This means that 0.17 seconds after the front of the metal sheet has passed our measuring post, the height of the metal sheet at the post will be 0.002447 metres above the conveyor belt.

For our corrugated metal sheet, we would have a different amplitude if the peaks and dips of the sheet reached higher and lower from the centre. Because the sheet is lying on the conveyor belt, a different amplitude would also mean that the mean level would be different. If the peaks and dips of the sheet were spaced differently, or the sheet emerged from the machine at a different rate, the frequency would be different. If the prevailing end of the sheet had been cut differently, the phase would have been different.

The meaning behind a distance-based wave for our corrugated metal sheet requires more thought. We know that the amplitude, phase and mean level will be the same as for the time-based wave. We know that the wavelength of the metal sheet is 0.2 metres. This means that the spatial frequency is 1 ÷ 0.2 = 5 cycles per metre. From that, we can make up a formula:
"y = 0.05 + 0.05 sin (2π * 5x)"
... where "x" is the distance in metres. However, we have to decide exactly what distance "x" is measuring. The answer to this is that "x" is measuring the total length of the metal sheet that has passed the measuring place. It is measuring the distance from the prevailing end of the metal sheet to the position of the measuring place:



The distance-based wave graph looks like this:



As an example of the formula in use, we will enter 0.01 metres:
0.05 + 0.05 sin (2π * 5 * 0.01)
= 0.06545 metres.

This means that when the prevailing end of the metal sheet is 0.01 metres past the measuring place, the height of the metal sheet above the conveyor belt at the measuring place will be 0.06545 metres.

Note that, as with the time-based wave, the distance-based wave is a mirror image of the actual shape of the metal sheet.

Categories of waves

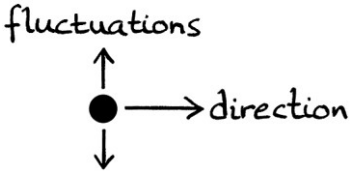
There are different ways in which we can categorise real-world entities that are described using waves. These categories refer to the characteristic of the entity that is being described using waves, and not to the wave formulas or the wave graphs.

Transverse and longitudinal waves

One way to categorise the wave-like characteristics of *moving* entities is to say that they are one of these two types:

- Transverse waves
- Longitudinal waves

A transverse wave is one where the changes in instantaneous amplitude (in other words, the changes in the measured y-axis values) are at 90 degrees to the direction of travel of the entity being described by the wave. To put this another way, the characteristic of the entity that can be described using waves fluctuates in a direction that is at 90 degrees to the direction of travel of the entity.



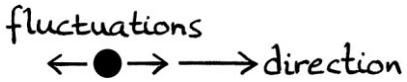
The pigeon wing tip waves are transverse waves because the fluctuations are at 90 degrees to the movement of the pigeon. The wings move up and down while the pigeon flies horizontally. If the pigeon somehow flew on its side, with one side up and one side down, the waves describing it would still be transverse waves. If the pigeon flew vertically upwards, its wings would move horizontally, and the waves would still be transverse waves. It does not matter in which direction the pigeon

flies, or from what angle we view the pigeon, its wings will always move up and down, backwards and forwards, or in and out, at 90 degrees to the direction of the pigeon's flight. Therefore, the waves that describe the movement of the wings will be transverse waves.

The movement of the toy steam train's chimney is at 90 degrees to the direction of travel. Therefore, the waves that describe its fluctuations are transverse waves.

The waves in the corrugated metal sheet example are also transverse waves – the ups and downs of the metal sheet are at 90 degrees to its direction of travel along the conveyor belt.

A longitudinal wave is one where the characteristic of the entity that can be described using waves fluctuates in the same direction as the direction of travel of the entity. [To put this in the more common (and slightly less pedantic) way to describe longitudinal waves, the changes in instantaneous amplitude (in other words, the y-axis values) are in the same direction as the direction of travel of the entity being described by the wave.]



In the block, spring and board example, the waves are longitudinal waves. The length of the spring fluctuates in the same direction as the movement of the block of wood.

The movement of the neck of the toy tortoise is in the direction of travel of the tortoise. Therefore, the waves are longitudinal waves.

Whether a wave is longitudinal or transverse makes no difference to the appearance of the resulting distance-based or time-based wave formulas or graphs. Distance-based wave formulas just show how a characteristic fluctuates over a distance, with no mention of which way it fluctuates. Time-based wave formulas just show how a characteristic fluctuates over time with no mention of which way it fluctuates. The toy tortoise and the toy train have the same time-based and distance-based waves as each other, but the tortoise produces longitudinal waves and the train produces transverse waves.

Transverse waves are usually slightly easier to visualise than longitudinal waves.

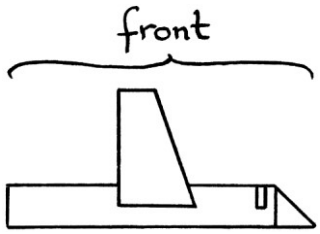
Whether a wave is transverse or longitudinal is independent of from where we view the entity producing the waves. For example, the pigeon wing tip wave will always be a transverse wave – it does not matter if we view the pigeon upside down or in the reflection of a mirror, it will still be a transverse wave.

The concepts of transverse and longitudinal waves do not apply as categories unless the entity is moving. If the pigeon flapped its wings while holding onto a branch, we would not be able to say that the wave describing it was transverse or longitudinal. The terms become meaningless in such situations. There is no movement with which the direction of the fluctuations can be compared.

Type A and Type B waves

From the pigeon wing example and the corrugated metal sheet example, we can see another way to categorise real-world waves.

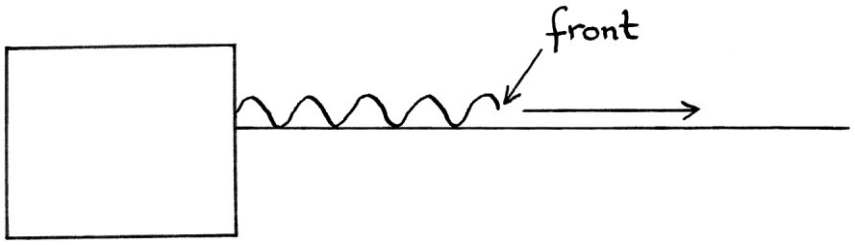
To understand these categories, we have to pay attention to what we will call the "front" of the entity. By "front", I really mean the prevailing part of the entity. The entity, from which we are formulating waves, travels, and we are interested in the outermost part. In many situations, the actual "front" can be slightly vague and arbitrary. For our wood pigeon example, the "front" can be thought of as the wood pigeon itself. The wood pigeon flies, creating the wave as it does so.



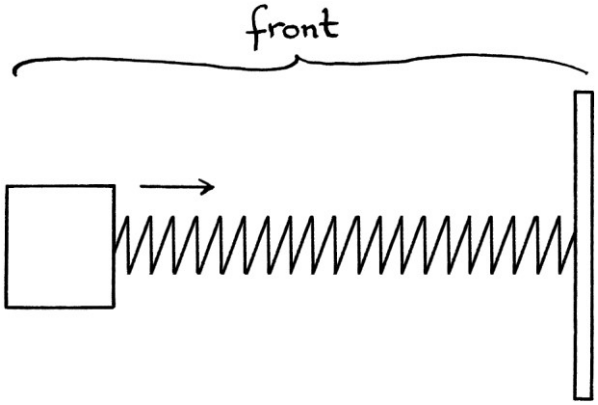
The measurements for the wave itself are made on the pigeon, but there is no single point where this has to take place. We can measure the height of the wing tips above the centre of its body at any point along the length of the wing tip. When we are dealing with distance-based waves, the distance in the formula referred to the distance from a starting place to the middle of the pigeon's wings. We could just as easily have picked a different place on the pigeon to measure the distance, and that place does not need to be where we make the wing tip height measurements. From all of this, it might be clear that there is no exact single point that we call "the place where the wave is" because, in this case, such an idea is meaningless. It is more correct to refer to "an area in which the measurements for the wave can take place". For the pigeon, we will call this area "the front", and it

can be thought of as the prevailing part of the “wave” (where the term “wave” incorrectly suggests that the wave travels on its own).

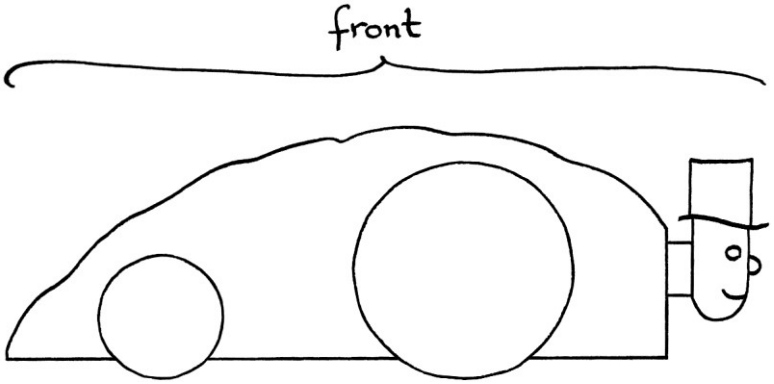
On the moving corrugated metal sheet, the “front” is the prevailing edge of the metal sheet. For the corrugated metal sheet, the “front” is one distinct point on the sheet.



On the block, spring and board, the “front” can be thought of as the whole device. As with the pigeon, there is not an individual point that can be said to be where the “wave” is, but there is an area where the outermost wave activity is taking place.



On the toy tortoise, the “front” can be thought of as the whole tortoise:



Note that the "front" does not necessarily match the place where we measure the y-axis values for the wave graphs. For example, the pigeon's y-axis values are measured on the pigeon, but the sliding corrugated metal sheet's y-axis values are measured at a single place that the metal sheet passes. For the pigeon, the y-axis value at the "front" is important for all time. For the metal sheet, the y-axis value at the "front" is relevant only for the brief moment when the "front" passes the point where the values are being measured. The "front" is the moving point or area of the fluctuations that is furthest from the source of the wave.

In this categorisation system, we will call the two types of wave, Type A and Type B. [These are not the standard names for these, but it makes everything easier to use these names.]

Type A waves are waves for which the "front" is always changing in value. Type A waves refer to entities that fluctuate at the "front". To put this another way, Type A waves are those where the only meaningful measurements are done at the "front". For transverse versions of this type of wave, as the "front" of the entity progresses over a distance, it will "draw" out a wave in the air. Wave formulas and graphs for Type A waves portray the values at the "front". Pigeon wing tip waves; block, board and spring waves; and toy tortoise waves are Type A waves.

Type B waves are waves for which the "front" always has the same value. Type B waves do not fluctuate at the "front". For Type B waves, the only meaningful measurements are those done at a fixed place that the entity passes. Type B waves act as if they have been slid as a whole from the source. A sliding corrugated metal sheet is a Type B wave.

Another way of thinking of the two types is to pay attention to whether the *source* of the wave is moving. For Type B waves, the source of the fluctuations is fixed in place. The fluctuations slide out from the source. For Type A waves, the source of the fluctuations is moving.

If you imagine that you could see a transverse wave moving, then major differences between the two types are:

- If you just watch the "front", with a Type A wave, you will see every possible y-axis value. With a Type B wave, you will see only one y-axis value.

- If you stood in one place and peeked through a slit in a wall as the wave went past, for a Type A wave, you would see only one y-axis value. For a Type B wave, you would see every y-axis value.

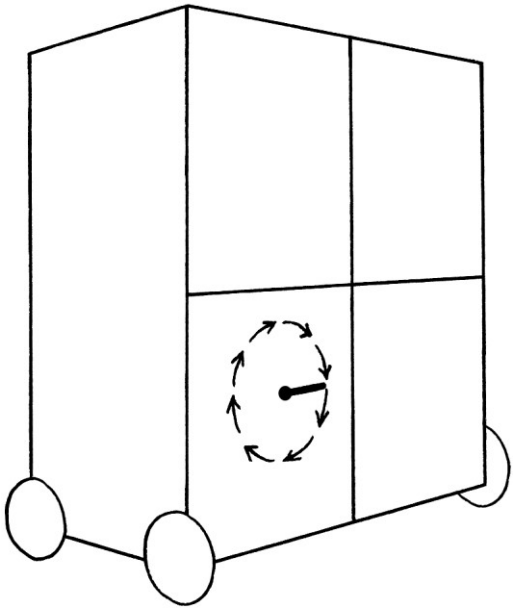
Of the two types of wave, Type A is the most straightforward for making time-based formulas and distance-based formulas. The meaning of such wave formulas is usually easiest to understand.

- The pigeon wing tip wave is a Type A wave and a transverse wave.
- The block, spring and board wave is a Type A wave and a longitudinal wave.
- The toy tortoise is a Type A wave and a longitudinal wave.
- The toy steam train is a Type A wave and a transverse wave.
- The sliding corrugated metal sheet is a Type B wave and a transverse wave.
- Sound waves, which I will discuss in the next chapter, are Type B waves and longitudinal waves.

Ripples in an otherwise still pool of water are a good *analogy* for Type B waves [although the behaviour of ripples is inconsistent with the sort of waves discussed in this book]. The “front” of the ripple has the same shape as it progresses through the water.

A mechanical device

We will imagine a large mechanical frame that is about the size of a filing cabinet. It has a metal bar on its face that rotates around a central point. The bar can be extended to different lengths, and the bar can be fixed to any place on the frame's face. The frame is on wheels, which means that as the bar rotates, the whole device can move at the same time. The device has a vertical line down its centre and a horizontal across its middle. These can be thought of as axes. We will say that when the device starts moving, the bar starts rotating. To keep the example simple, we will say that the device gets up to speed instantly – it is either moving at a set speed or it is stationary.

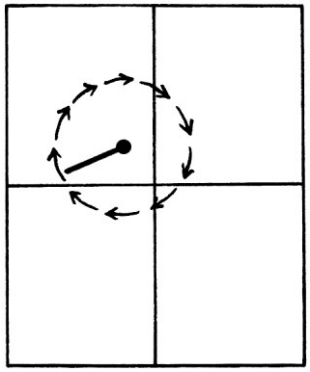


Such a device can demonstrate every aspect of time-based and distance-based waves. In this example, we will just look at the waves in a general way without any actual measurements.

Time-based waves

First, we will look at the general formulas for a time-based Sine wave and a time-based Cosine wave and examine every aspect in terms of the device. This is easiest to contemplate if we view the device face on.

Here it is shown with the bar set at a different position:



We will treat the vertical line down the centre of the device as the y-axis, and the horizontal line as the x-axis.

The general formulas for the y-axis position and x-axis position of the end of the bar at any one moment in time are as so:

$y = h_s + A \sin (2\pi ft + \phi)$
$x = h_c + A \cos (2\pi ft + \phi)$

... where:

- The mean level of the Sine wave represents the vertical position of where the bar is fixed, and around which it rotates. The mean level of the Cosine wave represents the horizontal position of where the bar is fixed. The mean levels can be changed by fixing the bar to a different place on the face of the device. We will say that the device can make the bar rotate no matter where the bar is placed.

- The amplitude refers to the length of the bar. The amplitude can be changed by extending or retracting the bar.

- The frequency is the number of revolutions per second made by the end of the bar as it rotates. If the bar rotates anticlockwise, the frequency is positive; if the bar rotates clockwise, the frequency is negative.

- The time is the number of seconds since the bar started moving. As the whole device is also moving while the bar rotates, this is also the number of seconds since the device started moving.

- The phase is the starting angle of the bar. It is the angle of the bar at the moment that the bar starts moving. As the device is also moving while the bar rotates, this is also the angle of the bar at the time that the whole device starts moving. We could alter the phase by fixing the bar so it is at a different angle before it starts.

Distance-based waves

We will now look at distance-based waves. The device is on wheels and moves at a constant speed. We can make general formulas for the y-axis position and x-axis position of the end of the rotating bar when the device as a whole is at any distance from where it started. As we are finding the x-axis and y-axis positions, we need a symbol to indicate distance that is not "x". Therefore, we will use the letter "d" as the symbol for distance. The two formulas are as so:

"$y = h_s + A \sin (2\pi * \nu d) + \phi)$"
"$x = h_c + A \cos (2\pi * \nu d) + \phi)$"

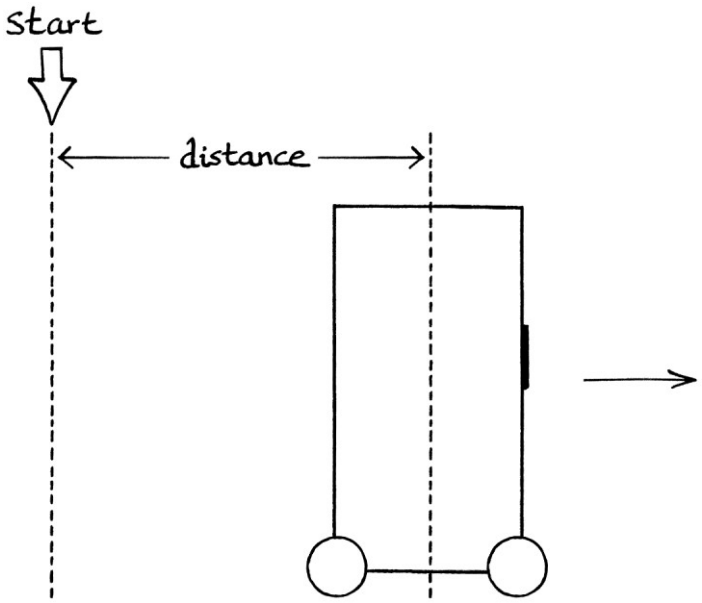
... where:

- The mean level and amplitude refer to the same ideas as in the time-based waves. The mean level is the position on the face of the device around which the bar rotates. The amplitude is the length of the bar.

- The phase refers to the same idea as phase in the time-based waves – it refers to the angle of the bar at the time that the bar started rotating. This is also the angle of the bar at the place that the entire device started moving.

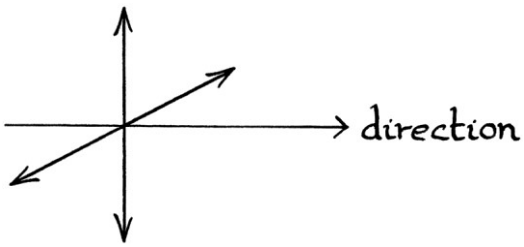
- The spatial frequency, "ν", is the number of rotations completed by the bar per metre of travel of the device. If the bar rotates anticlockwise, this will be positive; if the bar rotates clockwise, this will be negative. We can also complicate matters by realising that if the bar rotates anticlockwise *and the whole device moves backwards*, then the spatial frequency will be negative. If the bar rotates clockwise *and the whole device moves backwards*, then the spatial frequency will be positive.

- We will say that “d” is the distance of the centre of the device from the place where it started moving. We could just as easily measure to the back or front of the device, but I am measuring to the centre to emphasise how the distance does not need to be where the instantaneous amplitude measurements are made. I am using “d” because “x” is being used to represent the x-axis coordinate of the end of the bar.



Wave categories

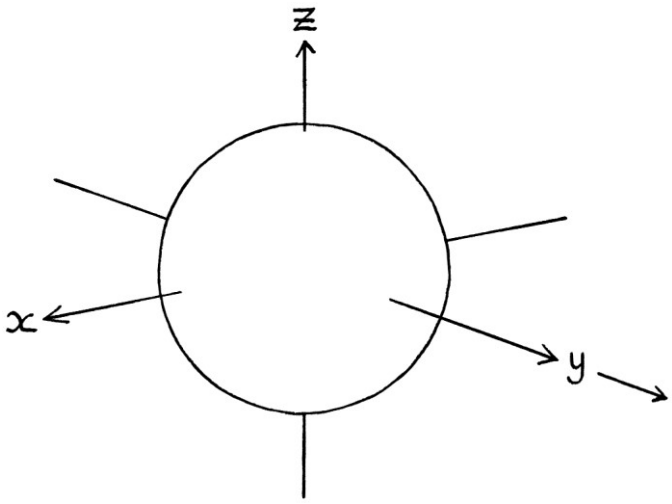
The mechanical device produces transverse waves – the waves describe the motion of the end of the bar, and the bar moves in circles at 90 degrees to the direction of travel of the device.



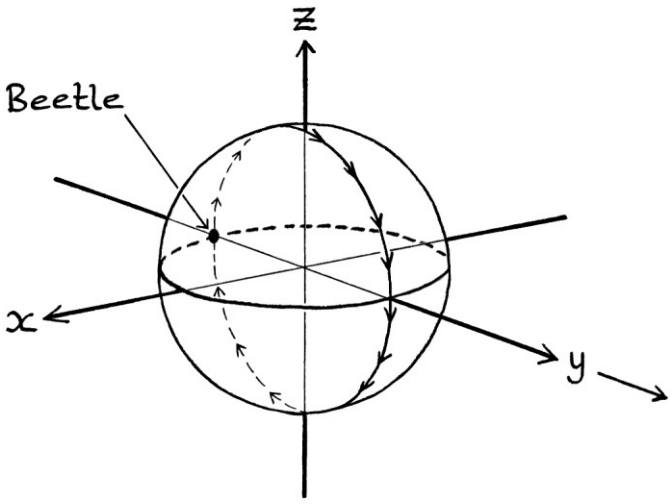
The mechanical device also produces Type A waves. The place of measurement is at the device itself, and so at the “front”.

A beach ball and a beetle

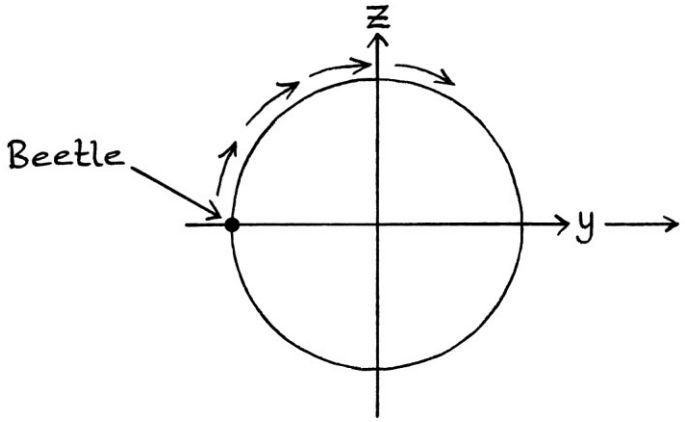
As a more complicated type of wave, we will imagine a beach ball flying through space in a straight line forever, but without rotating in any way. We will view it on three-dimensional axes. The ball travels in the direction of the y-axis:



We will say that a beetle is running around the ball at the same time as the ball flies through space. The beetle runs at a constant speed around the ball. It runs along the vertical centre of the ball, starting at the back, running forwards across the top, down across the front, and then underneath to the back again:



If we view the ball from the side, the situation is slightly clearer. We have the beetle moving around a circle:

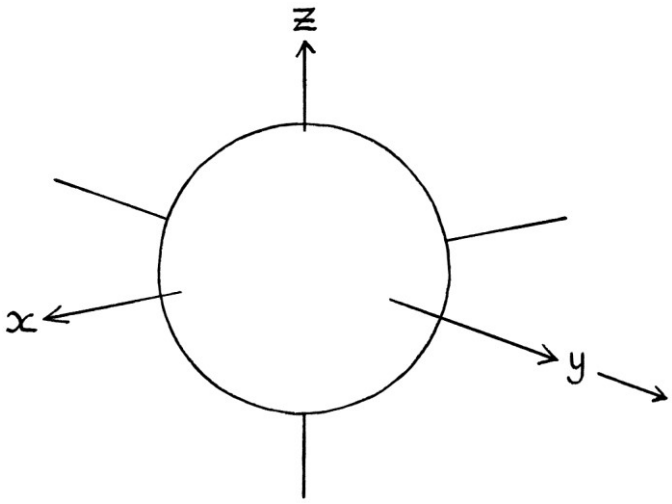


Note how the z-axis points upwards and the y-axis points to the right. The circle as a whole moves in the direction of the y-axis while the beetle rotates around it.

We will say that the ball has a diameter of 40 centimetres, which we will phrase as 0.4 metres. This means that its radius is 0.2 metres. The beetle can complete one revolution of the ball in 10 seconds.

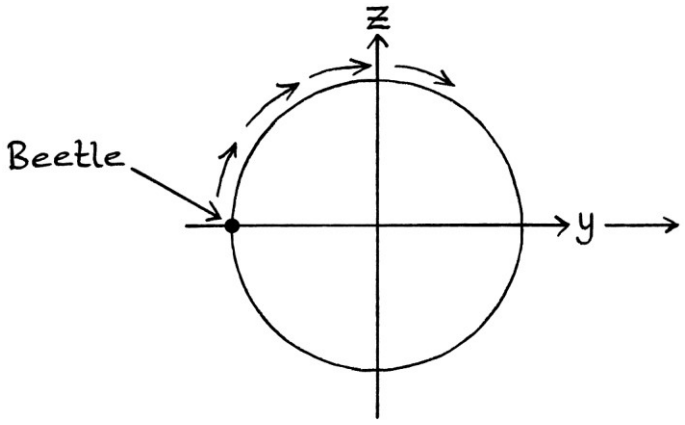
Time-based waves

The movement of the beetle can be portrayed using waves. As the beetle exists in three dimensions, it is easiest to ignore the way that Sine and Cosine usually refer to the y-axis and x-axis, and instead use Sine to calculate the position of the beetle for all three axes. For this reason, we will not call the waves the “Sine wave” and the “Cosine wave”, but the “z-axis wave”, “x-axis wave” and the “y-axis wave”.



First, we will look at the wave that shows the z-axis position of the beetle on the ball at any particular time. We will say that the time refers to the time since we first starting observing the ball and the beetle. For a time-based wave, it is irrelevant that the ball is moving.

We will say that when we first observed the ball, the beetle was at the centre back of the ball, and it was about to move upwards. It then moved to the top of the ball, down the front, and around. If we view the ball from the side with the z-axis pointing upwards and the y-axis pointing to the right, we can use the Sine of the *angle* of the beetle to give its z-axis position. In this way, the Sine function would be finding the "height" along the z-axis. As we are dealing with time, we can use the time multiplied by 2π and the frequency to find the beetle's z-axis position at any particular time. As the beetle is moving clockwise, the beetle has a negative frequency. The beetle has a phase of π radians (180 degrees) because it is starting on the left hand side. We will think about these ideas some more shortly.



The formula that describes the beetle's z-axis position on the ball in metres at any particular time is:
"$z = 0.2 \sin((2\pi * -0.1t) + \pi)$"
... where:

- The amplitude of the wave is 0.2 metres. This is the radius of the ball. If the ball were a different size, the amplitude would be a different size. If the beetle flew around the outside or inside of the ball without touching it, the amplitude would be bigger or smaller.

- The frequency of the beetle, as in how many revolutions of the ball it makes per second, is −0.1 cycles per second. If the beetle moved at a different speed, the frequency would be different. The frequency is negative because the beetle is moving clockwise.

- The phase is π radians (180 degrees). The phase indicates the position of the beetle on the ball at the time when we first started observing it. The phase is the starting point of the beetle. A phase of π radians means that the beetle started at the midpoint of the back of the ball, and continued moving from there.

- The mean level is zero. As the formula relates to the beetle's height with reference to the centre of the ball, we cannot really alter the mean level for the beetle without it seeming contrived. [A contrived way would be to have the beetle move in the same circular path around empty space while the ball was at a different height. For low mean levels, this would require the beetle pass magically through the surface of the ball.]

Note that the above formula is the same whether the ball is moving or not. This wave formula just shows the z-axis of the beetle on the ball at particular *times*.

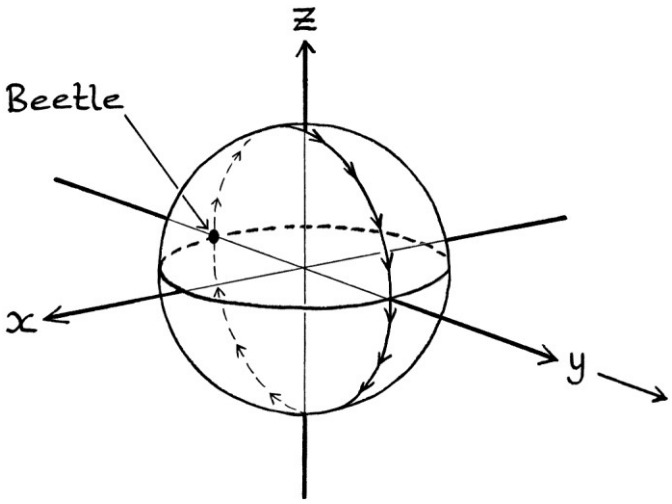
We can rephrase the formula so that it has a positive frequency. I explained how to do this in Chapter 11. The method in degrees is as follows: For a Sine wave, we see how many degrees the phase is above or below 90 degrees, and we then set it to be that same number of degrees below or above 90 degrees. We could also see how many degrees it is above or below 270 degrees and do the same. For a Sine wave *in radians*, we see how many radians the phase is above or below 0.5π radians, and then we set it to be that same number of radians below or above 0.5π radians. We could also see how many radians it is above or below 1.5π radians and do the same.

As the phase in this example is 0.5π radians above 0.5π radians, we want the angle 0.5π radians below 0.5π radians, which is 0 radians. Therefore, we set the phase to zero, and make the frequency positive:
"z = 0.2 sin (2π * 0.1t)"

This is the same wave that we would see if we viewed the ball from the other side with the z-axis pointing upwards, and the y-axis pointing to the left. If we are concentrating on just the z-axis, it does not matter which way around we view the ball, or which way around the beetle is moving because the z-axis values will be the same. In this example, whether it has a positive or negative frequency does not matter. We are interested in the beetle's z-axis values at particular times, and its z-axis values are unrelated to the place from which we observe the beetle.

Time-based x-axis wave

If we look at the beach ball and beetle with the axes drawn, we can see that the x-axis of the beetle is always zero. The beetle never moves left or right – it only moves forwards.



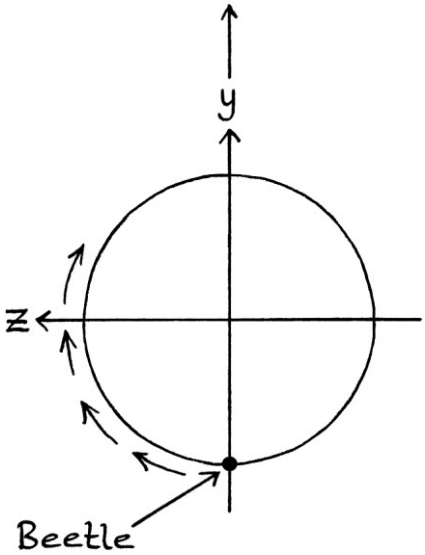
Therefore, the x-axis formula is:
“x = 0t”
... or, just simply:
“x = 0”

If the beetle moved around the ball in a way that was not directly along one of the axes, we would need to have formulas using Sine for all three axes.

Time-based y-axis wave

Now, we will look at the wave that shows the y-axis position of the beetle on the ball at any time since we first starting observing it.

We can calculate the formula for the y-axis by imagining we are viewing the ball from the side with the y-axis pointing upwards. We can then portray the ball as a circle with the y-axis uppermost and the z-axis pointing to the left.



The Sine of the beetle's *angle* would give the y-axis value of the beetle. As we are dealing with time, the Sine of the time multiplied by 2π and the frequency will give us the y-axis value of the beetle. The phase point of the beetle is at 1.5π radians (270 degrees). As the beetle is moving clockwise, it has a negative frequency.

The y-axis wave formula will be as so:
"$y = 0.2 \sin (2\pi * -0.1t) + 1.5\pi$)"

We can turn this into a positive-frequency wave. The current phase is 1.5π radians, which means that we can leave the phase the same and just change the sign of the frequency. The above formula rephrased to have a positive frequency is:
"$y = 0.2 \sin (2\pi * 0.1t) + 1.5\pi$)"

It does not matter from which direction we view the beetle, or whether the frequency is positive or negative, as the beetle will still have the same y-axis values. The formula gives the y-axis position of the beetle over time, regardless of from where we view the ball.

Distance-based z-axis wave

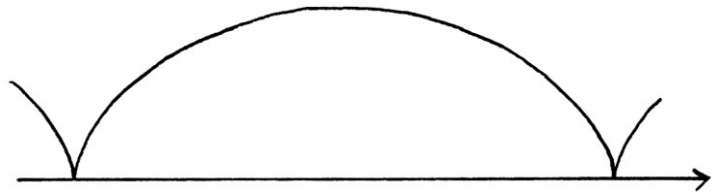
We will now look at the *distance*-based waves for the beetle.

While the beetle moves around the ball, the ball moves in a straight line through space. We will say that the ball is travelling at 10 metres per second.

We will create a formula for the distance-based z-axis wave. This will be a formula that shows the z-axis values of the beetle at any *distance* from the start. Immediately, we have the standard problem of needing to decide what "distance" in this situation means. We need to know what it is that we are measuring. The two most obvious choices we have are:

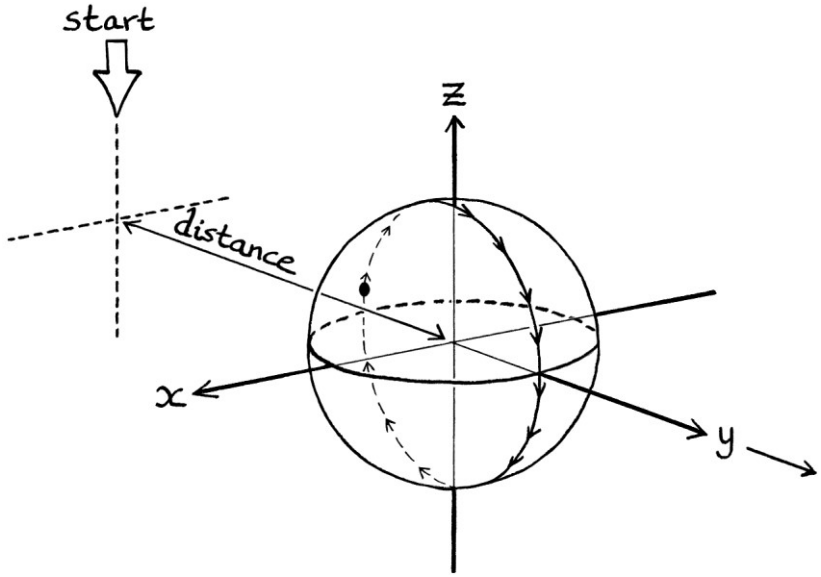
- We could measure the distance of the *beetle* from the starting place of the beetle and the ball.
- We could measure the distance of the *centre of the ball* from its starting place.

The trouble with doing the first of these is that the beetle moves forwards and backwards with reference to the centre of the ball as the ball and beetle travel through space. If we plotted the beetle's z-axis position in relation to its distance from the start, we would end up with a cycloid. [In other words, the curve would be the same as if we followed a point on a wheel as the wheel rolled along the ground.]

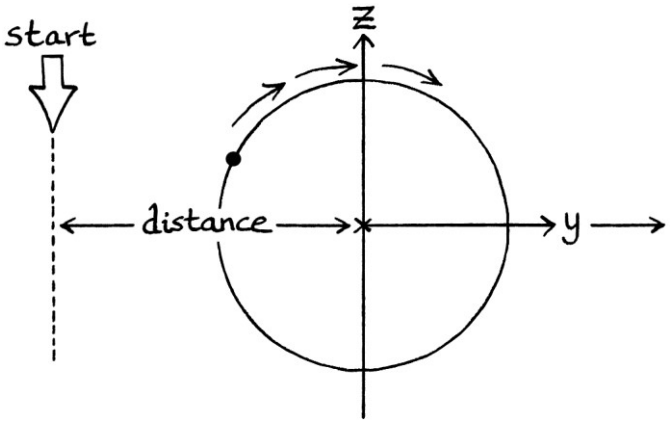


Therefore, it makes sense to use the second of these ideas. We will watch the beetle's z-axis position with reference to the centre of the ball, and consider the distance of the centre of the ball from the place that we first starting observing the ball.

Viewed in three-dimensions, the distance is measured as so:



Viewed in two-dimensions, the distance is measured as so:



The centre of the ball moves at a constant speed through space, so the wave describing the height of the beetle will be a pure wave.

[We could also have measured from the edge of the ball nearest to the place we first started observing it, or from the edge furthest away. We could measure from any place on the ball or even some fixed distance away from the ball. As long as we are consistent, it does not matter. If we are measuring from the place where we first started *observing* the ball, all of these will produce the same results. However, if we are measuring from a place that the ball passed, then the phase of the beetle will be different for each choice. We need to make sure that the time-based wave matches the criteria for the distance-based wave. Therefore, if we were measuring from the place where we first observed the ball, the time in the time-based wave would need to count from when we first observed the ball. If we were measuring

the distance of the centre of the ball from a fixed place, then the time in the time-based wave would need to count from when the centre of the ball passed that place.]

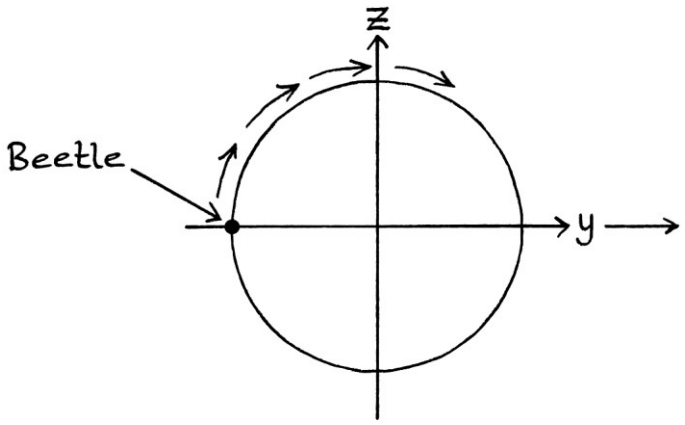
As the beetle moves at 0.1 cycles per second around the ball, its period is 10 seconds per cycle. Therefore, its wavelength will be:
10 * 10 = 100 metres per cycle.

Its spatial frequency will be:
1 ÷ 100 = 0.01 cycles per metre.

The spatial frequency is how many revolutions of the ball the beetle makes per metre covered by the centre of the ball. The wavelength is the distance that the *centre of the ball* makes while the beetle completes one cycle. The spatial frequency is based on two different but intertwined ideas, as is the wavelength. This makes things slightly more complicated, but also allows us to use pure waves to describe the situation.

Given that we will be referring to the waves as the "z-axis wave", the "x-axis wave", and the "y-axis wave", we will have to choose a symbol for distance in the formula that is not "x". For this example, we will use "d" to refer to the distance in metres.

For the z-axis wave, we will view the ball with the z-axis pointing upwards, and the y-axis pointing to the right.



This means that the beetle is moving clockwise, so its spatial frequency is negative. Its phase is π radians (180 degrees) because it is starting on the left hand side of the ball as viewed in this way.

The formula for the beetle's z-axis position on the ball for when the centre of the ball is a particular distance from the place that we first started observing it, is:
"$z = 0.2 \sin ((2\pi * -0.01d) + \pi)$"
... where:

- The amplitude of the wave is 0.2 metres. This is the radius of the ball and the maximum z-axis value of the beetle.
- The spatial frequency of the beetle, as in how many revolutions of the ball it makes per metre of the ball's centre moving, is −0.01 cycles per metre.
- "d" is the distance of the centre of the ball from where we first started observing it.
- The phase is π radians (180 degrees). When we first started observing the ball, the beetle was at the midpoint of the back of the ball, and was about to move upwards.
- The mean level is zero.

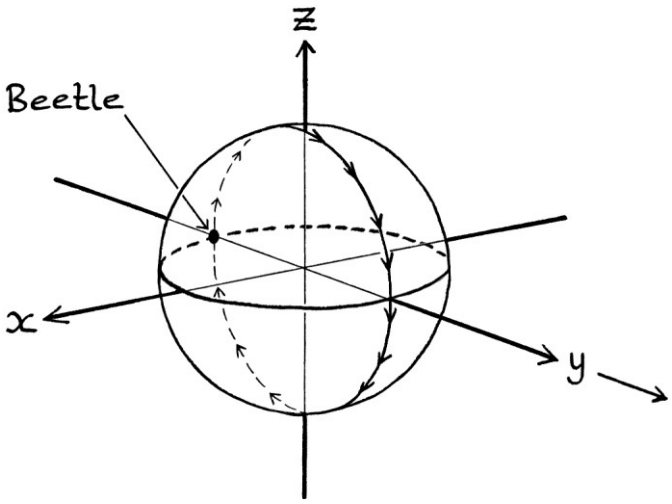
We can rephrase the formula to have a positive frequency as so:
"$z = 0.2 \sin (2\pi * 0.01d)$"

This is also the formula that we would have if we viewed the ball from the other side. As we are concentrating on only the z-axis of the beetle, it does not matter from where we view the beetle because the z-axis values will always be the same.

As an example of the formula in use, we will calculate the z-axis position of the beetle for when the centre of the ball has travelled 0.7 metres from where we first started observing it. The z-axis position will be:
$0.2 \sin (2\pi * 0.01 * 0.7)$
= 0.008794 metres

Distance-based x-axis wave

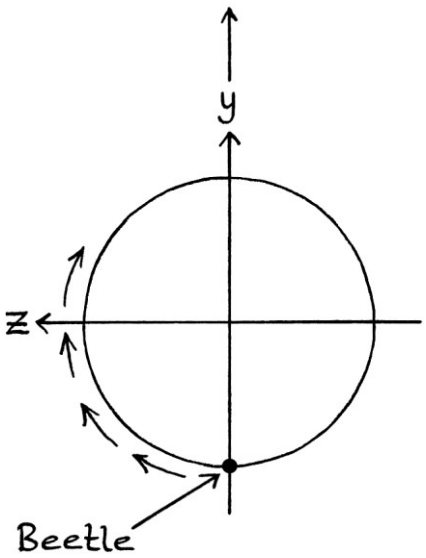
The distance-based x-axis wave is easy to calculate. The x-axis of the beetle on the ball will always be zero, no matter at what distance the centre of the ball is from where we first started observing it.



The formula will be:
“x = 0d”
... where “d” is the distance in metres of the centre of the ball from where we first started observing it. We could also give the formula as just:
“x = 0”

Distance-based y-axis wave

The distance-based y-axis wave will give the beetle's y-axis position when the centre of the ball is at particular distances from the place that we first started observing the ball. To think about this most easily, we will view the ball from the side, and imagine that the ball is a flat circle with the y-axis pointing upwards and the z-axis pointing to the left.



The beetle is moving clockwise, and therefore, it has a negative spatial frequency. Its phase point is at 1.5π radians (270 degrees). The y-axis wave formula is as so:
"y = 0.2 sin ((2π * −0.01d) + 1.5π)"

We can give the formula a positive frequency:
"y = 0.2 sin ((2π * 0.01d) + 1.5π)"

When we are concentrating on just the y-axis, it does not matter whether we consider the beetle moving clockwise or anticlockwise, as the y-axis values will be the same.

Summary so far

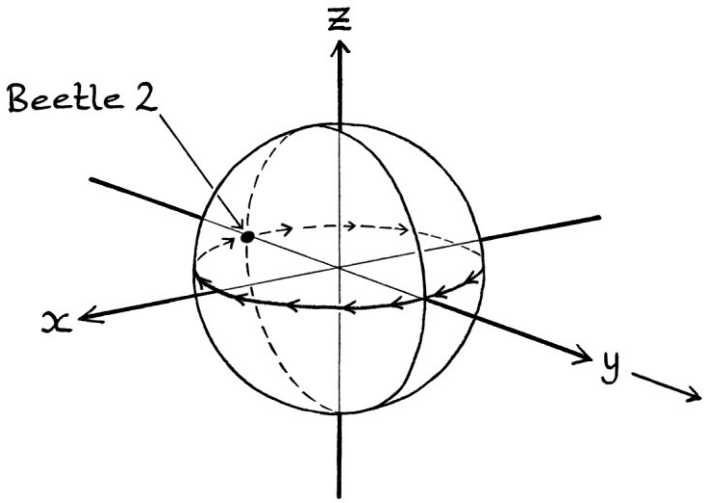
Although this beetle example seemed straightforward at first glance, there were several potential pitfalls in making up the formulas. Having three dimensions makes the use of both Sine and Cosine less suitable. It is better to keep to one or the other for all three axes. When it comes to distance, as with previous examples, we have to decide from where, and to where, we are measuring the distances. It is simplest to treat distances as measuring from the centre of the ball to where we

first started observing the ball. You might have thought that we would need to consider the angle from which we view the ball, but it turns out that this does not matter – the z-axis, x-axis, and y-axis values of the beetle will be the same regardless of the place from which we view the ball.

A second beetle

We can complicate the beetle and beach ball idea by adding a second beetle. This beetle will start at the same place as the first beetle (the back of the middle of the ball). It will then move to the far side of the ball, across the front, to the left, and then to the back again. We will call this new beetle, "Beetle 2", and we will call the original beetle, "Beetle 1".

For now, we will just look at Beetle 2 on its own. The movement of Beetle 2 is as so:



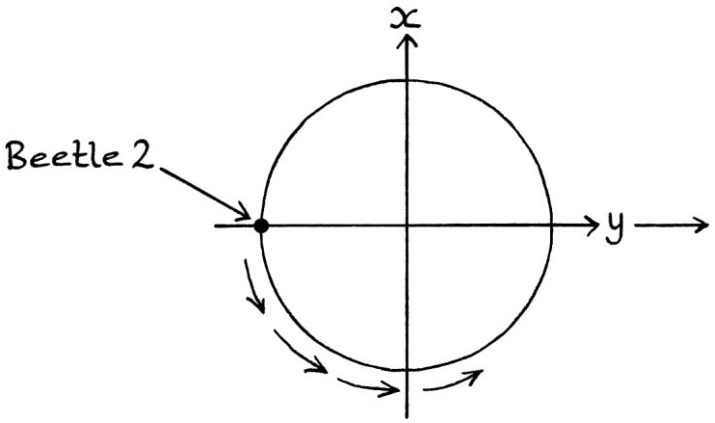
We will say that Beetle 2 moves at the same speed as Beetle 1, and starts at the same point, which is at the back of the ball. By this, I mean that when we first start observing the ball, both beetles are in the same place at the back of the ball.

Beetle 2's time-based waves

Beetle 2's z-axis values with respect to time will follow this formula:
"z = 0t"
... or:
"z = 0".

This is because it never moves upwards or downwards. Its z-axis value is always zero.

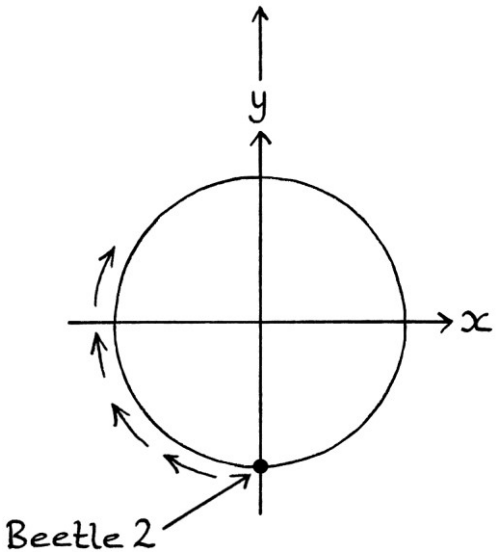
To calculate Beetle 2's x-axis formula, we can view the ball from underneath, and imagine the ball as a circle with the x-axis pointing *upwards* and the y-axis pointing to the *right*:



The phase point is at π radians (180 degrees). The frequency is positive because Beetle 2 moves anticlockwise around this circle. The x-axis values with respect to time follow this formula:
"x = 0.2 sin ((2π * 0.1t) + π)"

This choice of viewpoint is arbitrary. We could also have viewed the ball from any angle, and the results would have ended up the same.

To calculate Beetle 2's y-axis values with respect to time, we will view the ball from the top and imagine the ball as a circle with the y-axis pointing upwards and the x-axis pointing to the right:



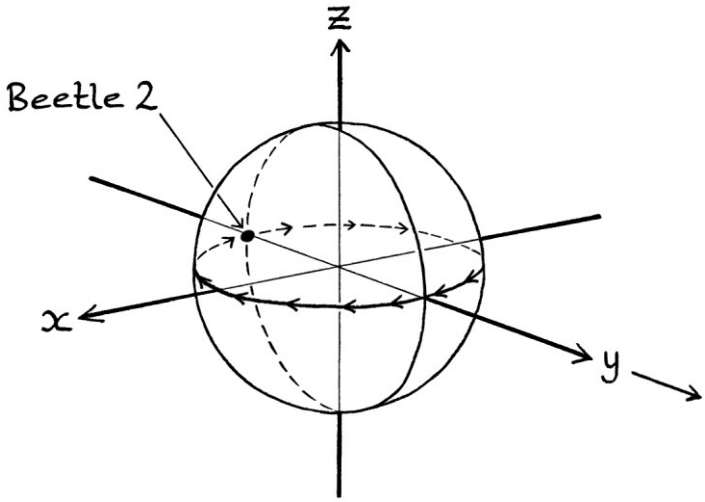
Beetle 2 starts at 1.5π radians (270 degrees). The frequency is negative. The formula is as so:
"y = 0.2 sin ((2π * −0.1t) + 1.5π)"

After converting this to a positive-frequency wave, we have:
"y = 0.2 sin ((2π * 0.1t) + 1.5π)"

Note how this is identical to the time-based y-axis formula for Beetle 1. This is because both beetles start at the same point and move at the same speed, but along different paths, to the highest y-axis point of the ball, and then back again. The y-axis positions of both beetles match for all times [and for all distances].

Beetle 2's distance-based waves

The z-axis of Beetle 2 on the ball will always be zero, no matter at what distance the centre of the ball is from where we first observed it:



Therefore, the distance-based Sine wave formula for the z-axis values will be:
"z = 0d"
... where "d" is the distance in metres from the centre of the ball to the place where we first started observing the ball. We could also give the formula as just:
"z = 0"

The distance-based x-axis wave formula for Beetle 2 will be:
"x = 0.2 sin ((2π * 0.01d) + π)"

The distance-based y-axis formula for Beetle 2 will be:
"y = 0.2 sin ((2π * −0.01d) + 1.5π)"

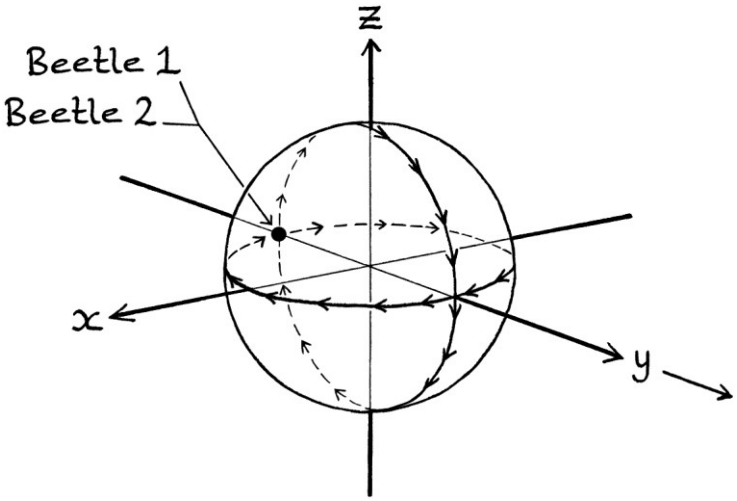
After converting this to a positive-frequency wave, we have:
"y = 0.2 sin ((2π * 0.01d) + 1.5π)"

This is identical to the distance-based y-axis formula for Beetle 1. As with the time-based y-axis waves, both beetles start at the same point and move at the same speed, but along different paths, to the highest y-axis point of the ball, and then back again.

Both beetles

Now we will look at the movement of both beetles at the same time. Both beetles start at the middle of the back of the ball, and move at the same speed. The beetles meet each other again when they have both walked half way around the ball. To keep everything simpler, we will say that the beetles do not collide with each other or have to walk over each other. We will say that they can pass through each other magically. This means that their movements can still be determined with the formulas that we have.

The movement of both beetles is as so:



The time-based formulas for Beetle 1 are:
"z = 0.2 sin (2π * 0.1t)"
"x = 0t"
"y = 0.2 sin (2π * 0.1t) + 1.5π)"

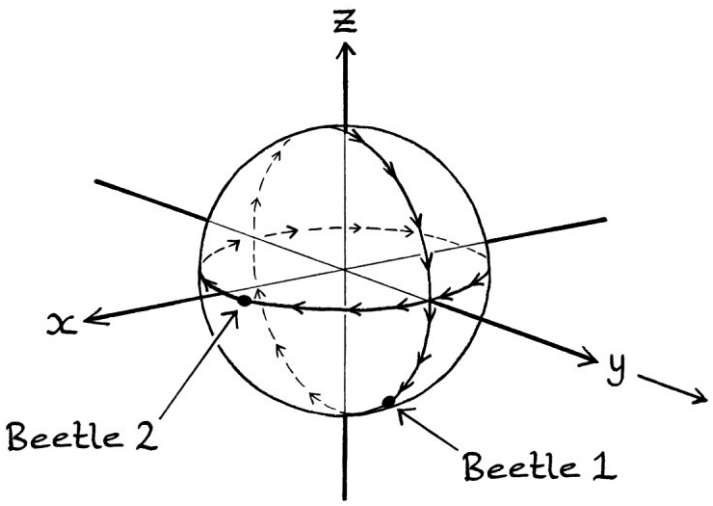
The time-based formulas for Beetle 2 are:
"z = 0t"
"x = 0.2 sin ((2π * 0.1t) + π)"
"y = 0.2 sin ((2π * 0.1t) + 1.5π)"

As an example of using these formulas, we will find the coordinates of each beetle when the ball has been travelling for 7 seconds since we first starting observing it.

Beetle 1's z-axis position on the ball will be: −0.1902 metres.
Beetle 1's x-axis position on the ball will be: 0 metres.
Beetle 1's y-axis position on the ball will be: 0.06180 metres.

Beetle 2's z-axis position on the ball will be: 0 metres.
Beetle 2's x-axis position on the ball will be: 0.1902 metres.
Beetle 2's y-axis position on the ball will be: 0.06180 metres.

Note how both beetles have the same y-axis value. The beetles are situated on the ball roughly as shown in this picture:



The distance-based formulas for Beetle 1 are:
"z = 0.2 sin (2π * 0.01d)"
"x = 0d"
"y = 0.2 sin ((2π * 0.01d) + 1.5π)"

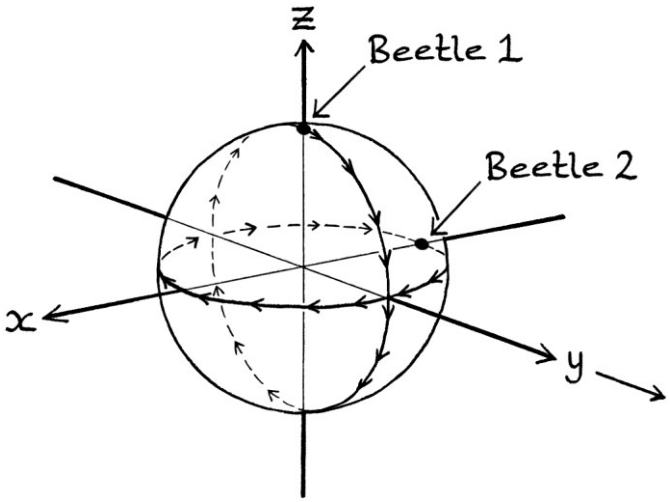
The distance-based formulas for Beetle 2 are:
"z = 0d"
"x = 0.2 sin ((2π * 0.01d) + π)"
"y = 0.2 sin ((2π * 0.01d) + 1.5π)"

We will find the coordinates of each beetle when the centre of the ball has travelled 25 metres from the place where we first starting observing it.

Beetle 1's z-axis position on the ball will be: 0.2 metres.
Beetle 1's x-axis position on the ball will be: 0 metres.
Beetle 1's y-axis position on the ball will be: 0 metres.

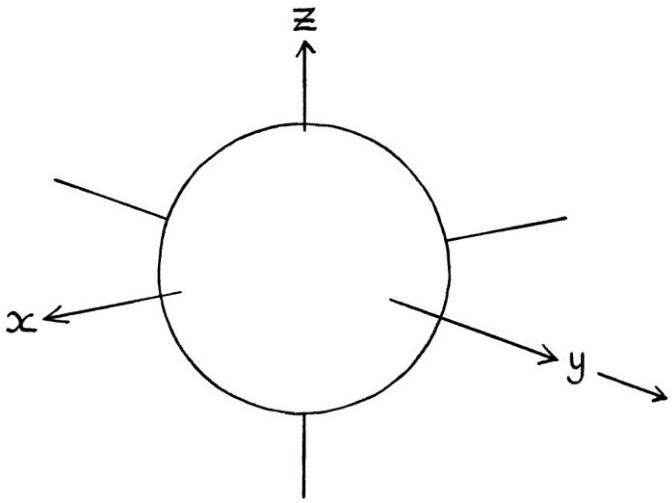
Beetle 2's z-axis position on the ball will be: 0 metres.
Beetle 2's x-axis position on the ball will be: −0.2 metres.
Beetle 2's y-axis position on the ball will be: 0 metres.

The beetles appear on the ball as shown in this picture:



Wave categories

By thinking of the basic ball, and how it travels in the direction of the y-axis, we can categorise the types of waves.



The z-axis waves for Beetle 1 are transverse waves – a z-axis wave indicates the up and down motion of the beetle, and that motion is at 90 degrees to the direction of travel of the beach ball. Beetle 2's movement does not change for the z-axis, so there is no z-axis wave to be categorised.

Beetle 1's movement does not change for the x-axis.

The x-axis waves for Beetle 2 are transverse waves – they indicate the x-axis movement of Beetle 2, which is at 90 degrees to the direction of travel. This movement is not up and down, but away and towards the sides. The movement is not in the direction of the movement of the ball, but at 90 degrees to it.

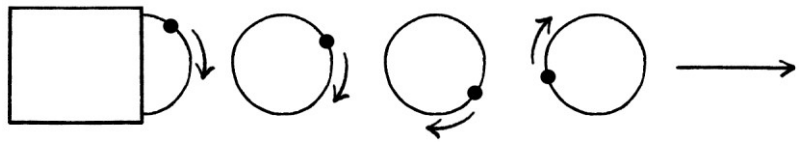
The y-axis waves for both the two beetles are longitudinal waves – these waves indicate the motion in and away from the direction of travel of the beach ball.

The movement of both beetles can be categorised as Type A waves. The instantaneous amplitude of the "front" changes as the waves move. The measurements for the instantaneous amplitudes are made at the moving beach ball.

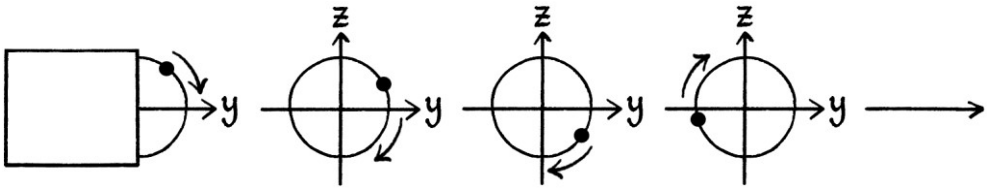
A beach ball machine

In this example, we will explore ideas that are more complicated. This example is not intended to be an analogy for anything else, although there might be situations where it could act as an analogy. The example is intended to increase your awareness of more situations involving waves.

We will imagine a machine that magically creates the beach balls from the previous example, complete with a Beetle 1 rotating around each one. To keep things simple, we will say it does not create the Beetle 2s. The machine releases each beach ball at evenly spaced intervals.



We can also portray the beach balls with their axes:



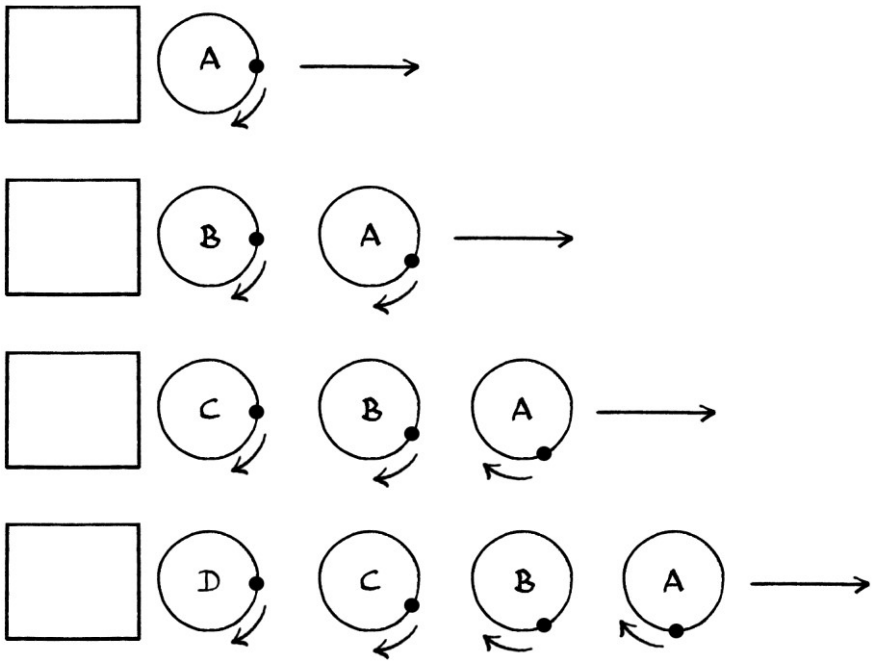
Among the countless ways in which such a machine could work, there are two interesting ones:

- It could produce each ball with the Beetle 1 in exactly the same place on the ball at the time and place of creation.
- It could produce each ball with the Beetle 1 slightly further around the ball than on the previous ball.

We will look at each of these ideas in turn.

Beetle at the same place

If the machine releases each ball with the beetle at the same place, then each beetle will rotate around its own ball, and the balls that were released earlier will have beetles that are further progressed on their journeys compared with the balls just released. This is shown in the following pictures with each line showing one moment in time later than the line before. The beach balls are labelled “A”, “B”, “C”, and “D”, with “A” being the first ball created, “B” being the second ball created, and so on.

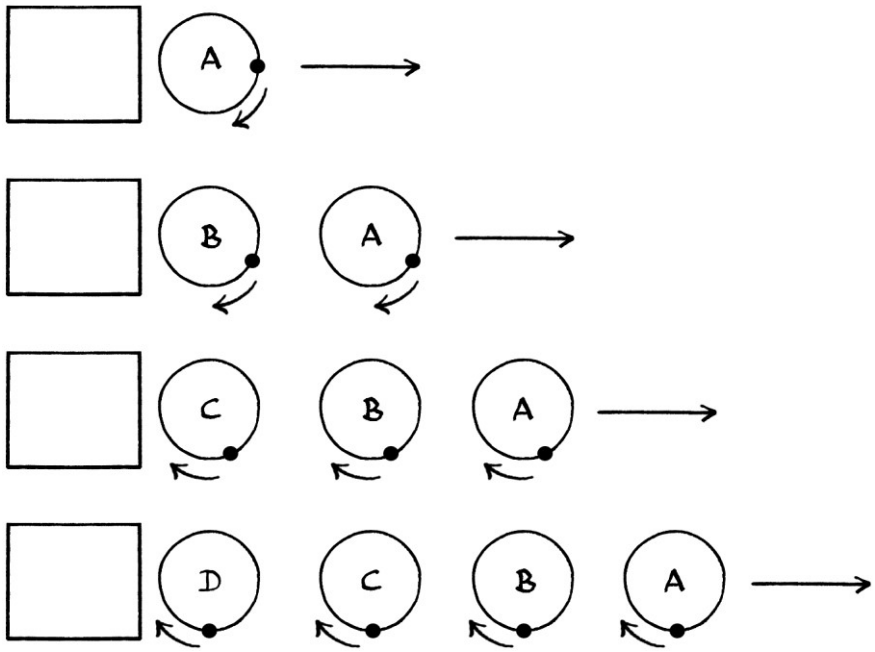


In this way, every existing ball has a beetle in a different place up until the location where the beetles have completed one full revolution, when the brand new beetles would be in the same state as the beetles at that location. There would be a single location where every passing beetle would complete its first revolution.

For beach balls with beetles yet to complete one full revolution, we would be able to tell when that ball had been released from the machine by examining the position of the beetle on the ball [as long as we knew its frequency].

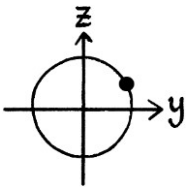
Beetle slightly further around

If the machine releases each ball with the beetle slightly further around, and the timing is carefully planned, then every beach ball that has been released will have a beetle in exactly the same place. Every beach ball in existence will be in an identical state.



Thoughts

No matter how the beach balls are released, the waves produced from the z-axis motion of the beetles will be transverse waves, and the waves produced from the y-axis motion of the beetles will be longitudinal waves.



If the balls were created with each beetle at the same position, then it would be necessary to follow an individual ball on its journey to observe the fluctuations of a beetle. In this way, the waves would be Type A waves.

If the balls were created with the beetles slightly further around on each consecutive ball, and every ball in existence was in the same state, then:

- Either we could observe one ball and its beetle, in which case, we would have a Type A wave.
- Or, we could observe the positions of beetles as the balls they were on passed a particular measuring place. In this case, as long as there were enough balls going past, we would still create the same wave, and the wave could be treated *as if it were* a Type B wave. [Ideally, there would be an infinite number of balls passing at any one moment, or else there would be gaps in the wave's curve. This would mean that the balls would overlap each other.] To record the correct phase, we would have to choose a measuring place either at the machine or at a whole number of wavelengths away from the machine. How these waves would differ from my earlier definition of Type B waves is that the "front" in this case would always be changing, unlike Type B waves where the "front" is always the same. In this way, the waves are a mixture of Type A and Type B waves, and we will call them "Type C" waves.

If we imagine that we could see the z-axis wave (a transverse wave) moving, then either we could watch the "front" or we could peek through a slit in a wall as the "wave" went past, and, as long as there were enough balls passing per second, we would see every possible z-axis value.

Water ripples

By water ripples, I really mean water *waves*, but I do not want to call them waves because they have very little in common with the sort of waves being discussed in this book. Water behaves so differently from mathematical waves that mentioning it in an explanation of waves is like mentioning seahorses in a book on equine medicine.

It pays to ignore water ripples almost entirely. However, some useful traits help in visualising mathematical waves. In this way, water ripples are best thought of as being an analogy for particular aspects of mathematical waves.

As I explained earlier in this chapter, ripples in an otherwise still pool of water help in visualising Type B waves, where the "front" is unchanging, and the whole "wave" looks as if it is being slid from the source.

Water ripples can also be used to help visualise the concept of superposition. I mentioned superposition in Chapter 13 on the addition of waves. Superposition is a phenomenon related to particular types of waves including sound and radio waves, where the instantaneous amplitudes of two passing waves become added together if they exist at the same place and at the same time. As an example, if there are two identical radio transmissions, and if you can receive them both at the same time, and if the phases of each match (they are in phase) when you receive them, then the signals of both will combine to produce a signal with a larger instantaneous amplitude. If the phases are 180 degrees apart, the instantaneous amplitudes will combine, but cancel each other out. The basic idea is that the waves are added at the exact time and place that they meet each other.

As another example, if you hear a group of people shouting, they will sound louder than if there were just one person shouting. The instantaneous amplitudes of the sounds combine to produce a louder signal. You can experience this most obviously if you ever walk near to a sports stadium.

The concept of superposition is used in noise-cancelling headphones. Such headphones try to reproduce outside sounds with a phase that is 180 degrees different, so that when the outside sound and the reproduced sound reach your eardrum, they cancel each other out. The intention is that the reproduced sound will have a phase 180 degrees different from that of the outside sound at the time when both sounds reach your eardrum. If the timing is wrong, the sounds will not cancel each other out.

Water ripples can be used to demonstrate two aspects of superposition. If you disturb an otherwise still pool of water in two different places, the ripples from each disturbance will cross over each other. The first significant idea is that where the ripples cross, the ripples will be higher than the ripples from just one disturbance. The second significant idea is that after the ripples have crossed each other, they continue as if nothing had happened. These ideas occur in sound waves and electromagnetic waves (light and radio waves), but in a more complicated way – if you happen to be at the exact time and place where waves from two sources meet, the instantaneous amplitudes of the waves will be added together. Depending on the difference in phases, this might result in the signal being stronger or weaker. Once the waves from different sources have passed each other, they continue as if nothing had happened – the waves can pass through each other unhindered.

When it comes to superposition, there are big differences between water ripples and sound and electromagnetic waves. Water ripples add to each other regardless of whether anything is there to observe them. The only way to know that electromagnetic waves add to each other is for them to be detected at that precise time and place. For electromagnetic waves in the form of light, they can only be detected if the photons hit the back of your eye or a detector, at which point they cease to exist. Therefore, light waves from two different sources carry on as normal after crossing, but only if they are not detected or seen. Sound waves add to each other and then carry on as normal, but you would never observe this unless you were in their path.

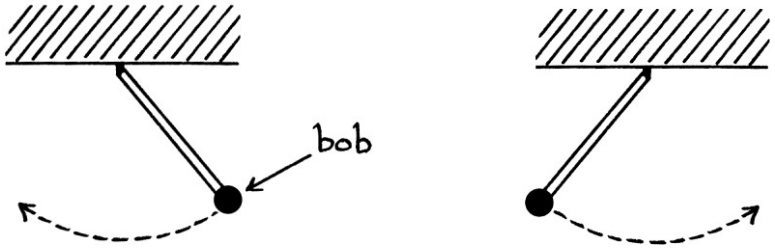
You can test that sound waves can pass through each other by talking to someone who is talking to you at the same time. You can test that light waves can pass through each other by shining a light at someone who is shining a light at you. You can test that light waves add to each other by shining two lights at a single object instead of one light.

It is easy to visualise a simplified version of superposition when thinking of ripples in a pool of water. Apart from that, water ripples are not particularly useful for learning about waves, and they can make things harder to understand. This is especially so if you do not distinguish between ripples in an otherwise still pool of water and the waves that occur in the ocean.

It is important to note that not all waves succumb to superposition. This should be obvious from the examples given so far of real-world waves.

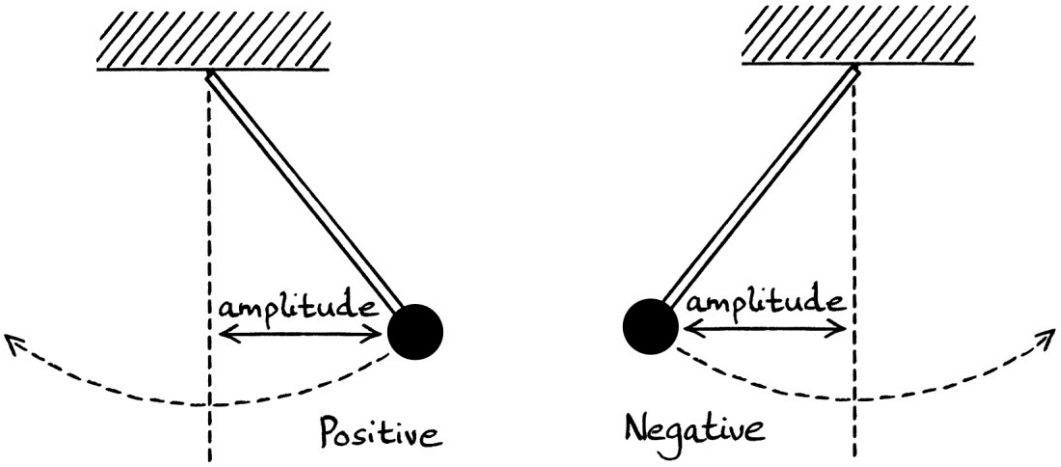
A pendulum

A pendulum is a device consisting of a weight suspended by a rope or bar, which has been attached to a fixed point above it. The proper name for the pendulum weight is a “bob”. The bob swings from side to side due to gravity and momentum.



A basic pendulum bob will travel less far on each swing, and eventually, it will stop moving altogether. The pendulum bob in a grandfather clock will swing to the same distance on each swing because it is assisted by the clock weight that is constantly pulling downwards on the clock mechanism. For this example, we will say that we have a clock pendulum that swings to the same distance left and right on each swing. We will say that it continues swinging forever.

The distance from the centre of the bob to the vertical line directly underneath where the pendulum is fixed can be portrayed using a time-based Sine wave formula. We will say that if the bob is to the right of the line, there is a positive distance; if the bob is to the left, there is a negative distance. The maximum distance from the centre line will be the amplitude.



For such a time-based Sine wave, we can create a Sine wave formula as follows:

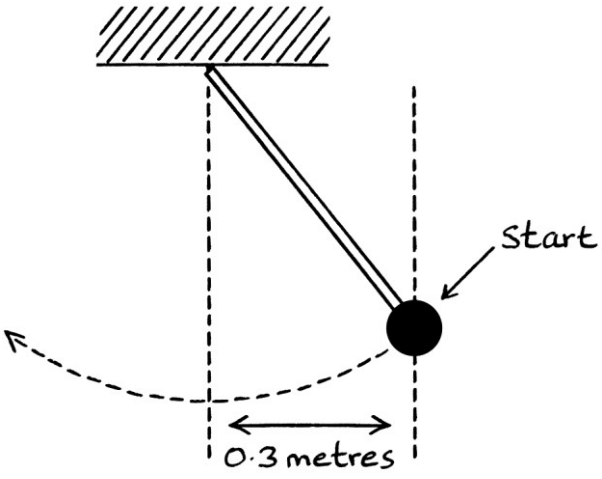
- The time will be the time since the pendulum first started moving. We will say that the pendulum was started by the bob being held to the right at its maximum positive distance, and then being released.

- The amplitude will be the distance that the centre of the pendulum bob reaches to the right or left from its centre point. We will say that this is 30 centimetres, which is 0.3 metres.

- The frequency will be the number of times the bob does one complete swing from left to right and back. Because our pendulum is part of a clock's mechanism, this will be exactly one cycle per second.

- The phase will be 0.5π radians (90 degrees). This is because we are creating the formula based on the pendulum starting at its right-most point and then falling. This means that it starts at its highest positive value and then falls. For a Sine wave, this means that it will have a phase of 0.5π radians. [If the pendulum started with a phase of 0 radians, the bob would be at its lowest point, and it would not have any momentum to move anywhere else.]

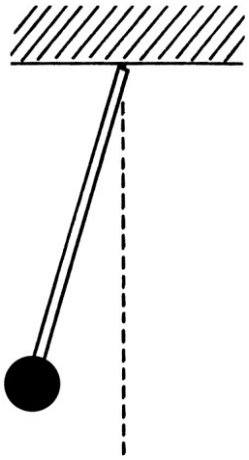
- The mean level in this example will be zero. As we are measuring to the vertical centre line underneath where the pendulum is fixed, a non-zero mean level would not make any sense in this situation.



The distance of the pendulum bob in metres from its centre point, (where positive distances mean it is to the right, and negative distances mean it is to the left) at any particular time can be calculated with this formula:
"y = 0.3 sin ((2π * 1t) + 0.5π)"

As an example of the formula in use, we will find out where the bob is at 0.675 seconds:
$0.3 \sin ((2\pi * 1 * 0.675) + 0.5\pi)$
$= 0.3 \sin (1.85\pi)$
$= -0.1362$ metres.

This means that at 0.675 seconds, the bob will be 0.1362 metres to the left of the centre point. It will be here:



Unless the clock containing the pendulum happened to be moving, there would be no point in creating a distance-based wave formula. For a stationary clock, the wavelength is zero, and the spatial frequency is infinite.

Pendulums have several characteristics other than their wave producing abilities that are of interest to physicists and physics teachers. For this reason, some explanations of waves *start* with the wave formula for a pendulum as if a pendulum were the simplest way to understand wave behaviour. In my opinion, the position of an object rotating around a circle is a far easier concept to understand than the position of a pendulum bob. Using a pendulum to introduce the subject of waves also means that there is no mention of mean level, so gives an incomplete description of waves.

Waves in electronics

We will have a trivial look at waves in electronics. Having a vague knowledge of waves in wires and cables helps in understanding some aspects of sending signals using electromagnetic radiation, and vice versa.

In electronics, we can create waves by consistently altering the current passing through a wire or cable. The result is called “alternating current” or “AC” for short. Such waves have amplitude, frequency and phase. They can also have a mean level, which, in electronics, can be referred to as the “direct current component” or “DC component”.

A zero-mean-level wave will fluctuate between having a positive and negative current. A negative current means that the current is moving in the opposite direction to that of a positive current. When a wave has a mean level higher than the amplitude, the current will always be travelling in one direction.

We can create time-based and distance-based wave formulas for current, but of the two, time-based waves would be the most useful. The speed of electrical current depends on the medium through which it is travelling. Different wires and cables allow current to flow at different speeds, and the speed will always be slower than the speed of light.

There are some similarities between electrical waves and electromagnetic waves, but also some major differences. [We will look at electromagnetic waves in Chapter 34.] Electrical waves can have frequencies from zero cycles per second upwards, where a frequency of zero cycles per second is just a constant current or voltage. Conversely, for electromagnetic waves, there is a minimum frequency at which they can be easily created, and the concept of mean level does not apply.

Electrical waves of different frequencies can co-exist in the same wire or cable at the same time without interfering with each other. In this way, they are consistent with the idea that different frequencies do not mix.

Conclusion

In this and the previous chapter, we have seen several different entities that have characteristics that can be portrayed by waves. Every example had a different type of characteristic. Although some of the examples might seem contrived, they should open your mind up to the possible forms that wave behaviour can take. They should also make sound waves and electromagnetic waves slightly easier to visualise.

Chapter 33: Sound

In this chapter, we will briefly look at the phenomenon of sound. One could easily write several books about sound, so this chapter will just be a quick introduction.

Sound is not an entity in its own right. Instead, it is the repeated variations of pressure caused by, and emanating from, the vibrations of a source. These pressure variations can travel through many materials, but are most commonly experienced through air. Mammals and birds, among other animals, primarily detect the pressure variations in the air with their eardrums. From the point of view of these animals, the air and the sound are inseparable. Although pressure variations adhering to the characteristics of sound can travel through media other than air, we only detect the sound unaided with our ears if the last step is through the air. We can be aware of vibrations without using our ears, but we are incapable of distinguishing them to anywhere near the same level.

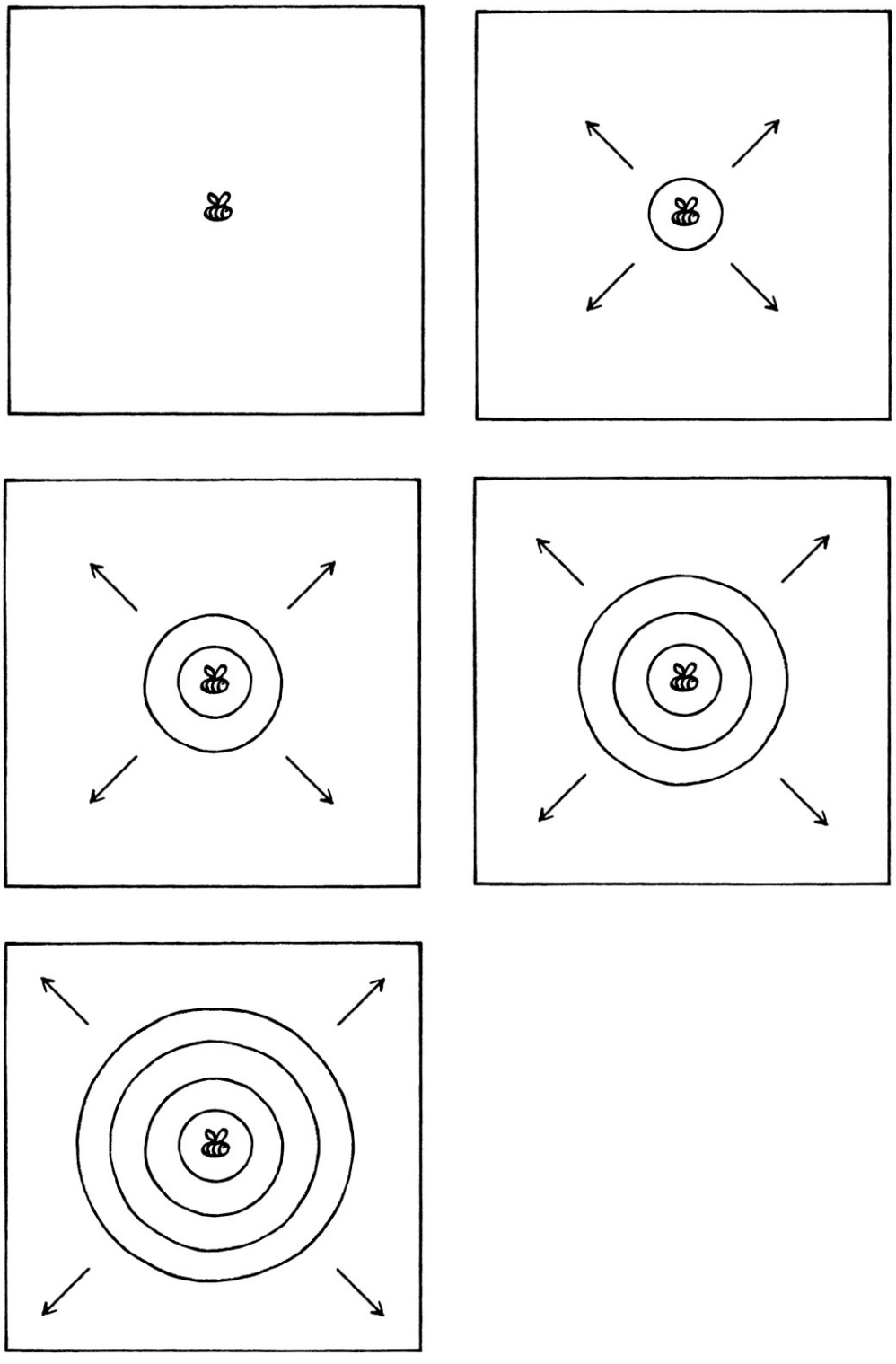
Mammals and birds, among other animals, are capable of distinguishing sounds by the frequency and amplitude of the vibration.

Sound can travel only through a medium [in other words, a gas, a liquid or a solid, and not a vacuum] because it relies on the changes in pressure. To be more pedantic, the phenomenon of sound *is* the changes in pressure that travel through a medium. Given that the phenomenon of sound consists of the changes in pressure within a substance, there must be a substance that can be subjected to changes in pressure for the phenomenon to exist.

In this chapter, we will mainly focus on sound in air.

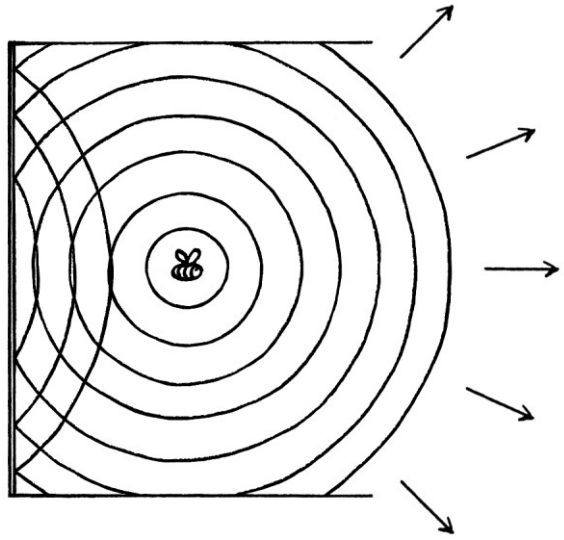
A sound example

Imagine that a flying bee buzzes briefly in an open room. The sound it makes is from the vibrations of its muscles and wings producing a barrage of constantly increasing and decreasing air pressure. These alterations in air pressure emanate from its wings as expanding spheres or “shells” that disperse in every direction.



The energy from the vibrations transfers outwards through the air in a similar way to ripples expanding outwards in a pool of water. However, the vibrations travel in every direction, not just in a two-dimensional plane as water ripples do. As the vibrations travel outwards, the energy caused by the initial bee wing vibration becomes weaker as it is spread out over a much larger volume. Eventually, the energy left becomes too weak for an average person to detect with their ears.

The ripples of air pressure from the buzzing may bump into objects other than air, for example walls, doors or windows. In which case, if there is enough energy, the vibrations will travel through the objects, and those objects will in turn alter the air pressure in other places. The sound might be absorbed into the object, or it might bounce off and back into the room. If the sound is loud enough and the room is big enough, these may be heard as echoes. In these ways, the original sound will become muddled with the new sources.



There are two important ideas to know about sound:

- First, the air itself does not move. Instead, the changes in pressure move through the air. The air returns to normal after the changes in pressure have travelled through it. Of course, if the sound occurs within moving air, then the air moves at the same time as the sound travels through it. However, the sound does not cause the air to move. The changes in pressure move through the air in a way that is analogous to someone pushing the back of a spoon down across the top of a large dish of jelly [jello] – the jelly becomes compressed, but stays where it is. After the spoon has passed, the jelly returns to its original state.

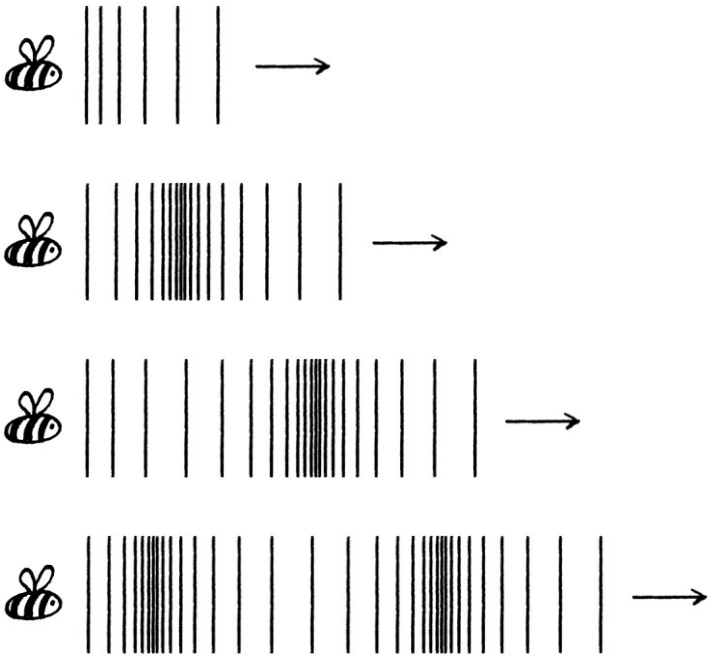
- Second, at no point do the changes in pressure move backwards [unless they become reflected]. The “wall of air pressure changes” moves ever outwards at a set speed from the source.

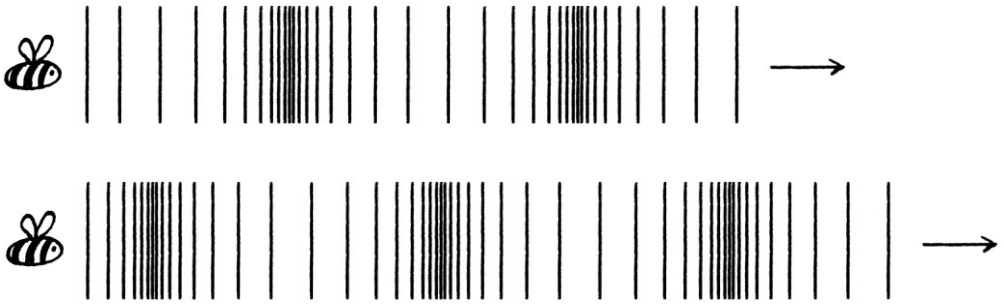
Details of sound

The meaning of the wave derived from a sound is not as obvious as the wave derived from, say, a pigeon’s wing tips. With sound, the fluctuations in air pressure are in the same direction as the movement of the sound. The waves from sound through air are longitudinal waves. [Sound travels as a longitudinal wave in air, but as either a longitudinal wave or a transverse wave through solids. In this chapter, we will focus on sound travelling through air.]

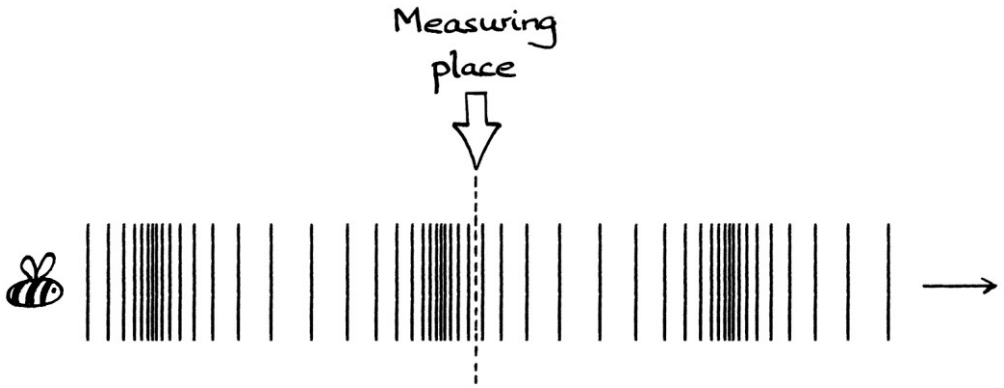
The movement of sound is analogous to certain aspects of the block, spring and board example combined with aspects of the corrugated metal sheet example. To explain what I mean, we will look at the bee from the side to see the radiating variations in sound pressure. To make things easier, we will imagine the variations in pressure as being in straight lines instead of curved lines, and we will concentrate on one section of the air pressure fluctuations.

In the following pictures, closer lines indicate an area of higher pressure; lines further apart indicate an area of lower pressure. The lines are just a guide to how the pressure might look. As the sound is emitted from the bee, the areas of pressure maintain their order and move away from the bee:



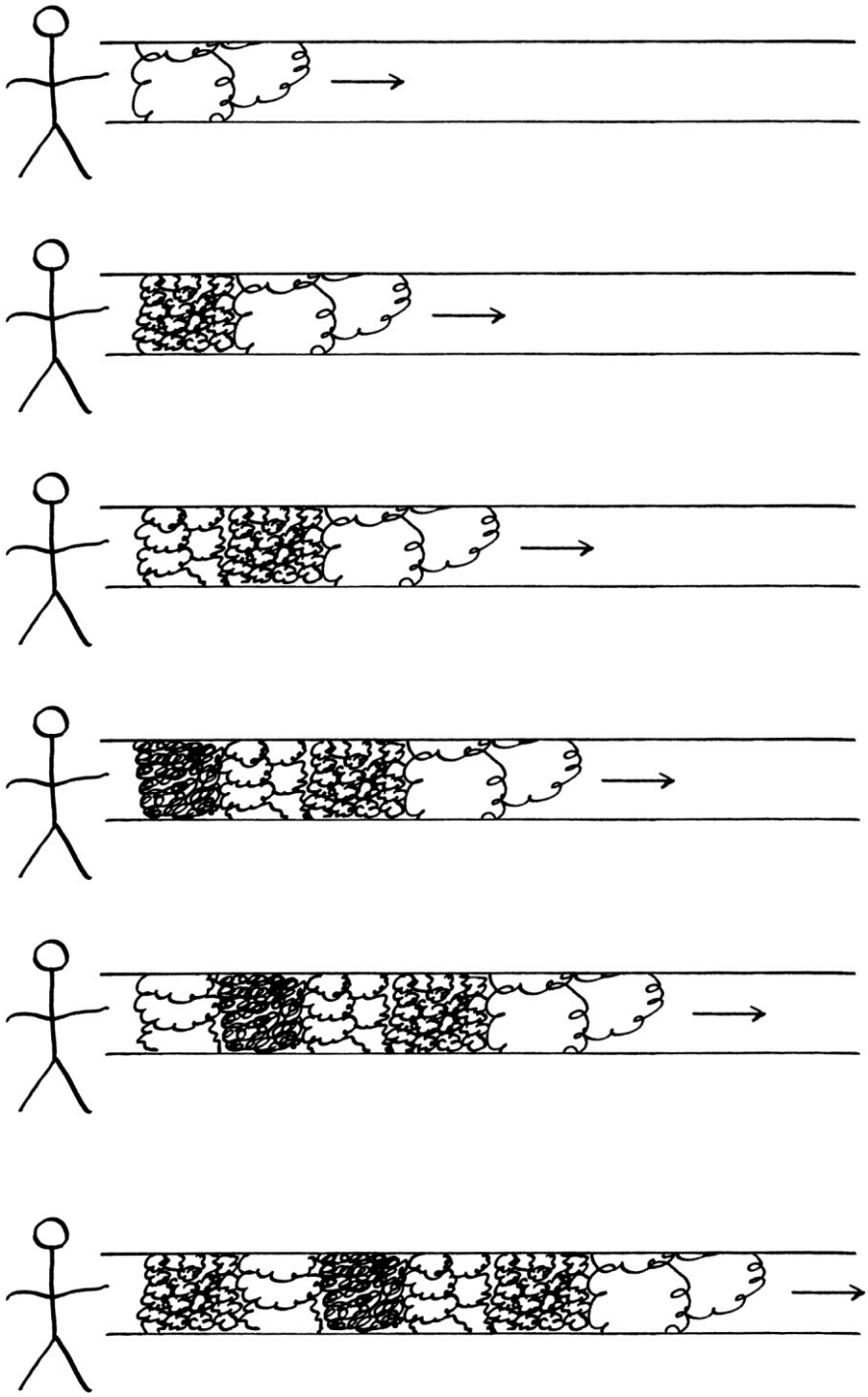


The bee is the source of the sound. From the bee come variations in pressure. The variations in pressure remain in the same pattern once they have left the bee. The variations in pressure travel through the stationary air, essentially moving their energy from one "piece" of air to the next. This means that a pattern of sound pressure that was originally at the source (the bee) will, after some time, be further away from the bee. In this way, sound is a Type B wave – it is as if the fluctuating characteristics are being slid away from the source. If we just measured the pressure at the "front" of the wave (as in the very first pressure change to move out from the source), we would see the same value for all time [although in practice, the sound would fade as it moved further away from the source.] The only way to collect sensible measurements from the ever-expanding shells of pressure changes would be to measure them at one particular place as they pass by.

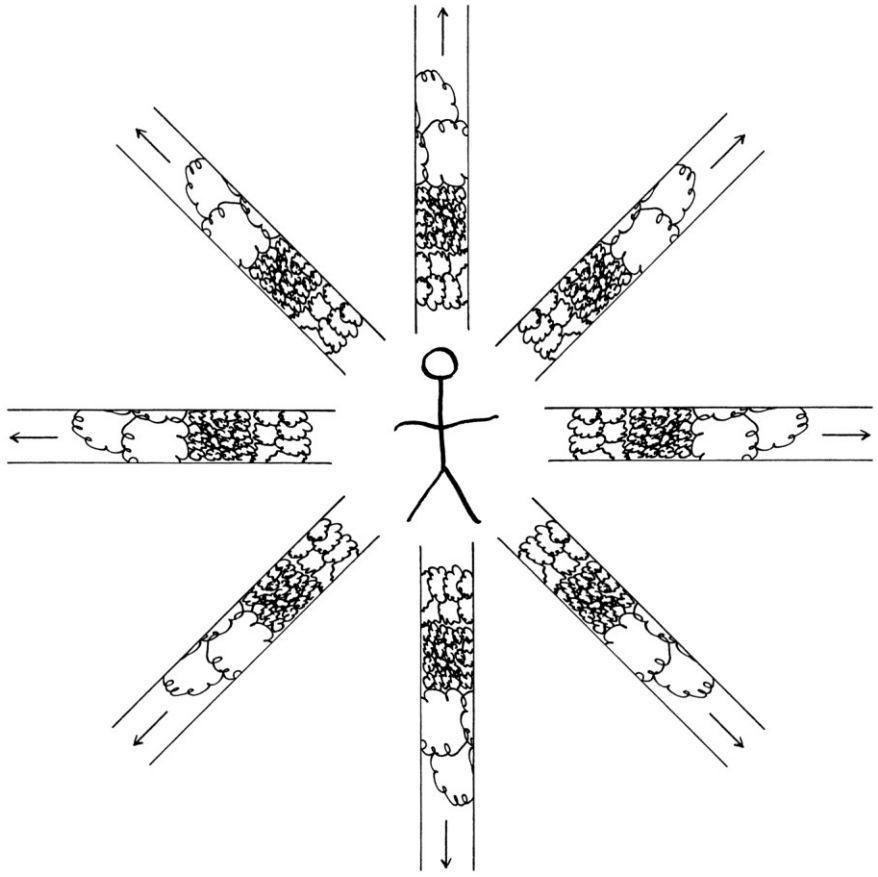


The earlier example of a block, a spring and a board is analogous to the changes in air pressure *at the time they are created* from the source. After that moment, the sound more closely resembles the corrugated sheet of metal.

Another, slightly better, analogy is someone pushing various densities of cotton wool into a pipe. Once the cotton wool has been put into the pipe, the variations in density will remain the same as the long sausage of cotton wool progresses down the pipe. The density of the cotton wool at the far end of the sausage of cotton wool will remain the same. However, the density of cotton wool as the sausage of cotton wool passes a particular point will vary.

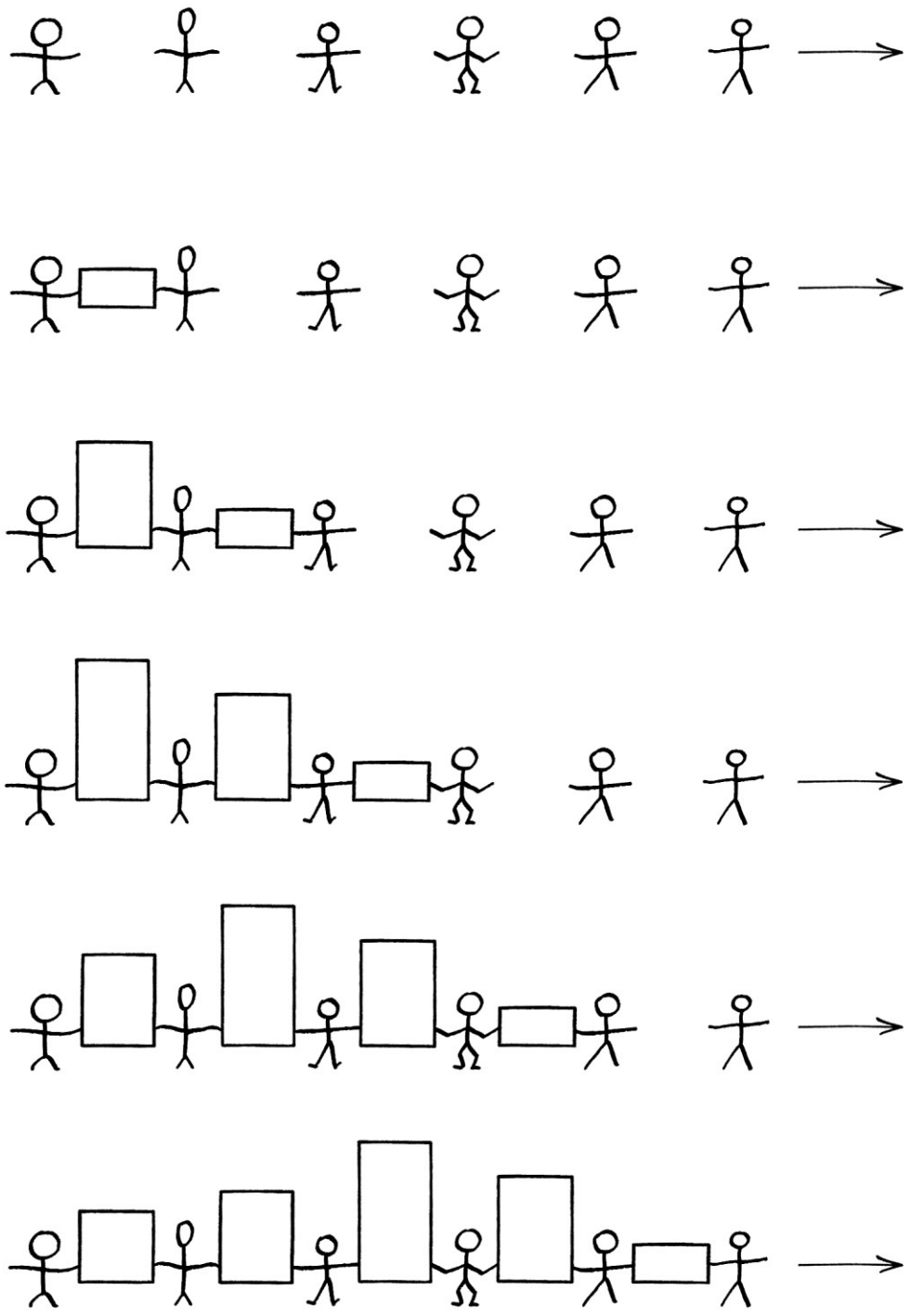


To make the analogy slightly more accurate, we can imagine that the person is standing in a place surrounded by pipes heading off in every direction away from them, and every pipe has the same types of cotton wool put into it at the same time:



In such a case, any measuring devices that are the same distance away from the person would detect the same density of the cotton wool.

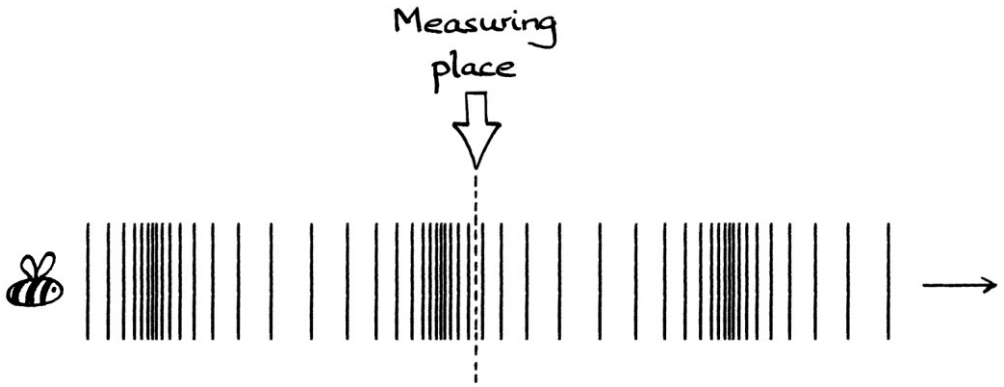
Another analogy, which emphasises how the sound requires a medium through which to travel, is a long line of people passing differently sized boxes to each other. The order of boxes remains the same once the boxes are in the line of people. The box at the furthest end will always be the same size. To measure the differences in box size, it would be necessary to stand in one place and observe the boxes going past you. You could not measure the differences in box size by just observing the outermost box.



In this way, we can think of the size of the box held by any one person at any moment in time as being analogous to the air pressure at any place and time. In the same way that the people do not move, only the boxes, the actual air does not move, only the air pressure. At no point do the boxes move backwards along the line – this matches how the changes in air pressure do not move backwards down the line. The boxes vary in size in the same way that the air pressure varies from place to place. The sizes of the boxes represent the instantaneous amplitudes of the wave formulas or graphs that describe the situation.

Graphs and formulas

We can draw a time-based sound wave graph showing the fluctuations in sound pressure for our bee as they might appear in reality. These air pressure fluctuations are observed at a place that the sound passes over a length of time. This place could be at the very source of the sound or at any place that the sound passes, but must be at a fixed position.



To make things simpler, we will imagine that the bee makes a sound that is a pure wave of 200 hertz. In reality, the strength of the sound pressure fluctuations would reduce as the sound travelled away from the bee. For this example, we will say that they stay the same.

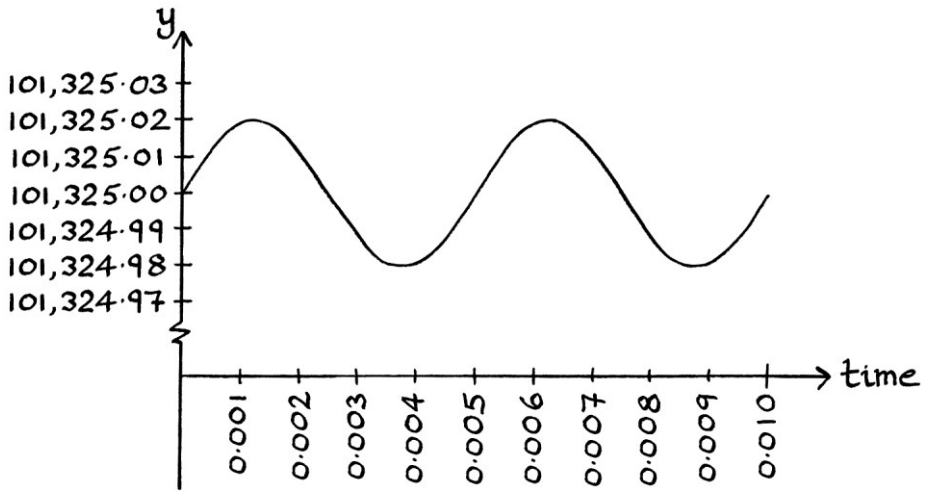
Time-based wave

The air pressure originating from our (hypothetical) bee, as measured at a particular measuring place over time, can be portrayed with this formula:
"$y = 101{,}325 + 0.02 \sin (2\pi * 200t)$"
... where:

- "y" is the total air pressure in pascals at any moment in time as measured at the measurement place. [Pascals are the standard unit of air pressure. One pascal is one newton per square metre. The unit "pascal" has a lowercase "p", but its abbreviation, "Pa", has an uppercase "P".]
- 101,325 is 101,325 pascals, which is the typical atmospheric air pressure at sea level. [To keep things simple, we will say that everything is happening at sea level.] This is the mean level of the formula. The air pressure will fluctuate around this level.
- 0.02 pascals is the amplitude of the formula. It is the maximum and minimum air pressure created by the bee. This would be a fairly loud bee.
- 200 cycles per second is the frequency of the tone created by the bee.

- We will say that the phase is zero. The phase refers to the pressure at the place of measurement at t = 0. A phase of zero radians means that the instantaneous amplitude was zero at that time and about to rise.

The graph for the formula is as follows. So that the graph can fit on the page, the y-axis starts at 101,324.97 pascals.

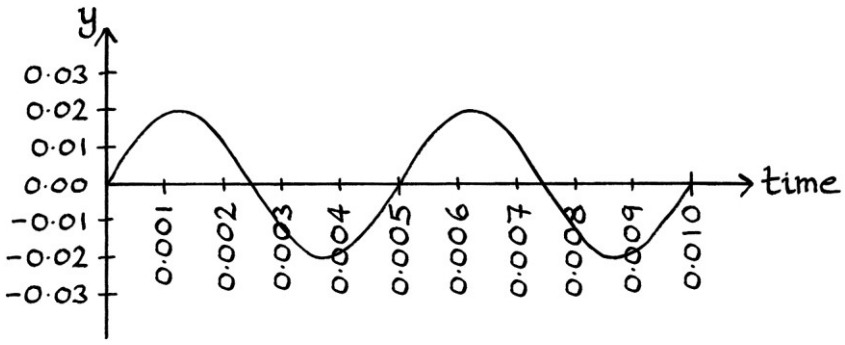


In practice, the pressure fluctuations of sound are usually measured in terms of how much the pressure differs from the surrounding air pressure. Sound is usually thought of as "sound pressure" (as opposed to just "pressure"). Another term for this is "acoustic pressure". Given that, the formula for the time-based wave would no longer mention the surrounding atmospheric air pressure. It would just concentrate on the deviation from the atmospheric air pressure.

The time-based wave formula would, therefore, become:
"$y = 0.02 \sin (2\pi * 200t)$"
... where:

- "y" is the *sound pressure* in pascals at any moment in time as created by the bee, and as measured as the sound passes a particular place.
- The mean level is zero. We are measuring how much the pressure differs from the surrounding air pressure, so the mean level will be zero. If we somehow altered the pressure of the air through which a sound travelled – perhaps by having the source of a sound and the detector contained in a large pressurised container – then the mean level would still be zero because sound pressure is the difference in pressure from the *surrounding* pressure.
- The amplitude is 0.02 pascals.
- The frequency is 200 cycles per second.
- The phase is zero.

The graph for this is:



In this graph, the mean level is zero. The y-axis values fluctuate around zero pascals. A negative y-axis value means that the air pressure at that time is less than the surrounding atmospheric air pressure; a positive y-axis value means that the air pressure is higher than the surrounding atmospheric air pressure.

It is common to see the amplitude, or in other words, the sound pressure, given in terms of decibels. [I explained decibels in Chapter 15]. This means that the sound pressure is given as the base ten logarithm of the ratio between the square of the measured sound and the square of the standard value with which it is being compared. Sound pressure is a "root power" value – therefore, it and the value with which it is being compared must be squared before one is divided by the other. [If this does not seem reasonably straightforward, then re-read the relevant section of Chapter 15.] The standard value with which sound pressure is compared is 20 micropascals. This is considered the lowest sound pressure that humans can hear. The abbreviation "dB SPL" refers to decibels comparing with a sound pressure of 20 micropascals. A sound pressure of 20 micropascals is equal to 0 dB SPL. A sound pressure of 0.02 pascals is equal to 6 dB SPL.

Distance-based wave

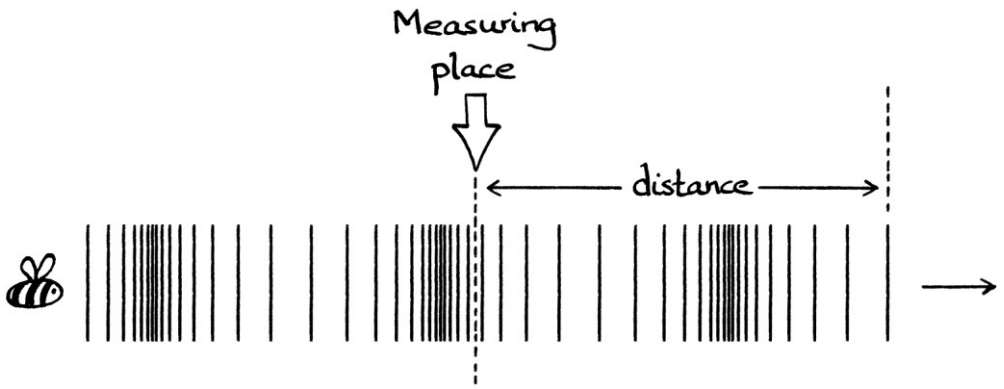
We can also formulate a distance-based wave for the sound from our bee.

The speed of sound in air at 20 degrees Celsius is about 343 metres per second. To keep things straightforward, we will say that the air temperature in our example is 20 degrees Celsius. We can calculate the wavelength of the sound using:
distance = speed * time
... or:
wavelength = speed * period
... which, because frequency is "1 ÷ period", we can phrase as:
wavelength = speed ÷ frequency

With our knowledge that the bee makes a 200 Hz sound, we have:
wavelength = 343 ÷ 200
... which is 1.715 metres.

The spatial frequency is:
1 ÷ 1.715 = 0.5831 cycles per metre.

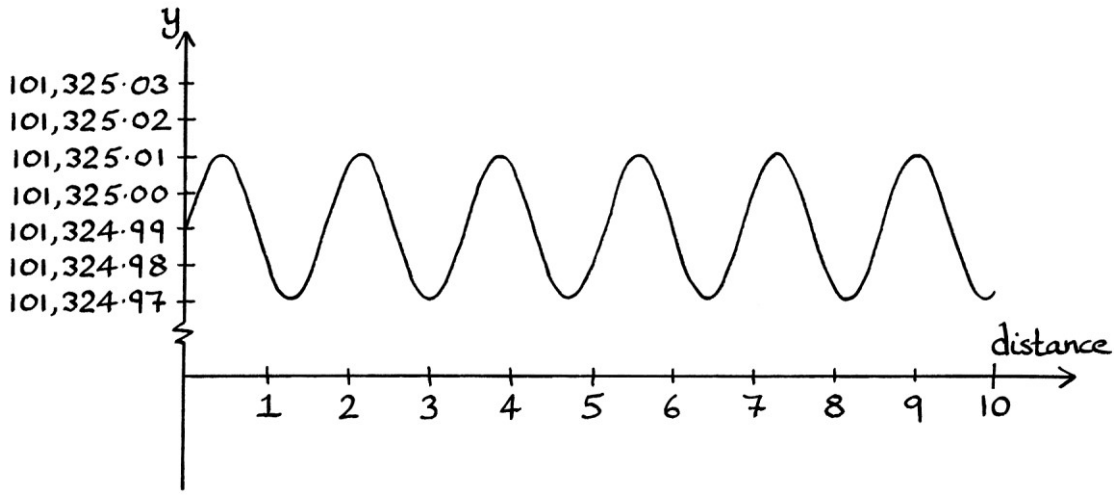
We have calculated the wavelength of the wave, but we are yet to create the distance-based formula. Creating a distance-based formula for sound requires a definition of the distance. We need to decide what the distance will be measuring. For this example, we will say that the distance in the formula will be the distance from the place where the measurements are taking place to the prevailing edge of the sound. In other words, we will measures the "length" of sound that has passed the place of measurement. This measurement idea is the same as that of the corrugated metal sheet example in the previous chapter. [We will think more about this idea later.]



For our first formula, we will look at the *total* air pressure. The formula for our distance-based wave is:
"y = 101,325 + 0.02 sin (2π * 0.5831x)"
... where:

- "y" is the total air pressure in pascals at the measurement place when the prevailing "front" of the sound has travelled "x" metres past the measurement place.
- 101,325 pascals is the typical atmospheric air pressure at sea level. This is the mean level of the formula.
- The amplitude is 0.02 pascals.
- The spatial frequency of the sound is 0.5831cycles per metre.
- "x" is the number of metres that the "front" of the sound has travelled past the measurement place.
- The phase is zero.

The wave looks like this:

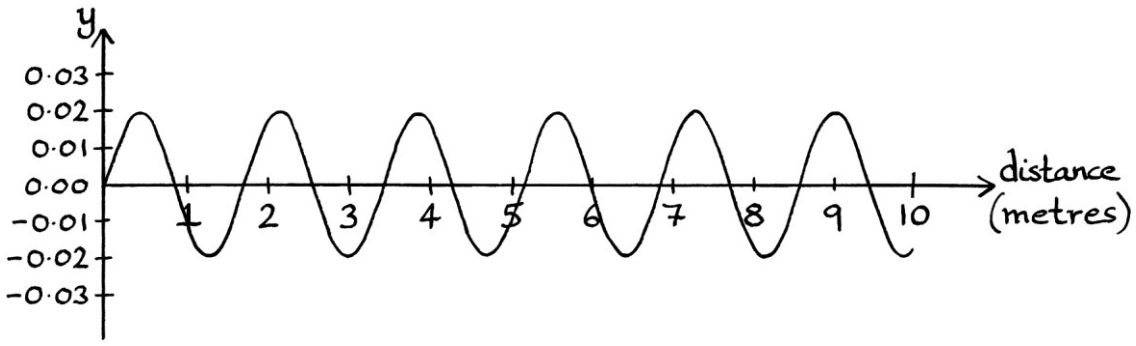


Now, we will look at the formula for the *sound pressure* (acoustic pressure) fluctuations. This means that we concentrate only on how much the sound from the bee makes the air pressure differ from the surrounding atmospheric pressure. The distance-based wave is:
"y = 0.02 sin (2π * 0.5831x)"
... where:

- "y" is the *sound pressure* (or *acoustic pressure*) in pascals when the prevailing "front" of the sound has travelled "x" metres past the measurement place.
- The mean level is zero – the fluctuations are around zero.
- The amplitude is 0.02 pascals.
- The spatial frequency of the sound is 0.5831cycles per metre.

- "x" is the number of metres that the "front" of the sound has travelled past the measurement place.
- The phase is zero.

The new distance-based graph looks like this:



Note how the formula and the graph refer to the sound pressure at a particular place when the prevailing "front" of the sound is so many metres past that place. The result of the formula is a sound pressure. A negative result from the formula means that the measured air pressure is lower than the surrounding atmospheric pressure. [If someone has not been paying attention, a possible source of confusion surrounding waves derived from sound is to believe, mistakenly, that the sound is moving backwards. From the examples in the previous chapter, it should be clear why this is wrong.]

One significant difficulty in our formulation of a distance-based wave is that it relies on knowing the distance that the prevailing "front" of the sound has travelled past our measurement point. When we had a corrugated metal sheet, we would have been able to measure the length of the sheet that had passed the measuring place. In our sound example, we cannot see the prevailing "front" of the sound, and it would be very difficult to measure consistently how far away it is. One way would be to have a series of air pressure sensors laid out in a line after the measuring point. They would be able to detect where the "front" of the sound was, but it would be difficult to have a high level of accuracy or to measure large distances. Alternatively, we could approximate where that "front" is at any time by assuming that the "front" travels at a constant and known speed. This would not work if it were windy or if the temperature were not constant. For this reason, a distance-based wave formula for sound is more likely to be a theoretical formula to demonstrate a point, than one literally created by measurements. This is another example of how it is easier to talk about wavelength than it is to talk about distance-based wave formulas. [It is worth noting that we calculated the wavelength in this example by *assuming* that the sound travels at a particular and

constant speed. Therefore, even the wavelength in this example might not be completely representative of reality.]

Measured waves

The above time-based and distance-based waves refer to the air pressure and sound pressure as measured. They rely on an accurate device that can measure in pascals. In practice, a sound is more likely to be "measured" with a microphone. The measurements will not be in pascals, but in the current or voltage that fluctuate according to the mechanism of the microphone and the electronics in, or connected to, that microphone.

The easiest way to measure the fluctuations of *sound pressure* (as opposed to *air pressure*) from a sound is with a microphone connected to an electrical circuit. [Remember to pay attention to when we are talking about "air pressure" and "sound pressure" – the first is the literal atmospheric pressure, while the second is the difference in the atmospheric pressure caused by the sound]. A very basic microphone consists of a diaphragm that moves slightly in and out according to the changes of air pressure that meet it. The diaphragm is connected to a magnet that moves with it, and the movement of the magnet induces an electric current in a coil of wire wrapped around the magnet. The fluctuations in the created current are *roughly* proportional to the fluctuations in the sound pressure. The current will fluctuate in accordance with the fluctuations of air pressure, but it might not fluctuate to the same extent as the fluctuations of air pressure. Therefore, we will have an adequate representation of the *sound* for many purposes, but it might not necessarily be an adequate representation of the *sound pressure*. The representation of reality in graphs and formulas is only as good as the accuracy and appropriateness of our measuring device.

Whereas the previous waves concerned the changes in *air pressure* and *sound pressure* over time and distance, the microphone's detection of the sound will produce a wave based on the changes in *electrical current* over time and distance.

For a time-based wave measured with a simple microphone, the formula might be something such as:
"$y = 0.005 \sin (2\pi * 200t)$"
... where:

- "y" is the current in amps as produced by the microphone at any particular time.
- The amplitude is 0.005 amps (5 milliamps). The current produced by the microphone will reach up to +0.005 amps and down to −0.005 amps.
- 200 cycles per second is the frequency of this electrical wave, which is also the frequency of the sound of the bee.
- To keep things simple, we will say that the phase is zero.
- The mean level is zero. The microphone in this example produces current centred around zero. Other microphones, or more specifically the circuits in, or connected to, other microphones might not necessarily produce waves with zero mean levels.

Note how when we were dealing in sound pressure, the amplitude and result of the formula at any particular time were measured in pascals. Previously, the amplitude was 0.02 pascals. Now the amplitude is 0.005 amps. This shows that we are at one step of separation from reality. We have ended up with a portrayal of the sound pressure. Depending on the circuit in, or connected to, a microphone, the wave formula might refer to volts instead of amps. If the formula is showing a digital recording of a wave (in which case, it will be stored as a series of y-axis values taken from equally spaced moments of time, all scaled by a particular amount), the amplitude will be measured in generic units. Generally, the actual units of the captured wave will be irrelevant as long as the *shape* of the wave matches reality, or is proportional to reality.

Amplitude as measured in amps can also be given in decibels. In such a case, the standard reference value is 1 microamp. The abbreviation "dBμA" refers to decibels comparing against 1 microamp. Amps are a "root power" value, so the value of interest and the value of 1 microamp must be squared before the ratio is calculated.

As we have seen, the amplitude might appear in many different forms. However, the frequency will always match that of the original sound. [Or at least, the frequency will always match if the measuring device is capable of keeping up with the frequency of the sound.]

The distance-based current wave will be:
"y = 0.005 sin (2π * 0.5831x)"
... where:

- "y" is the current in amps as produced by the microphone when the prevailing "front" of the sound is "x" metres away from where the microphone recorded the sound.
- The amplitude is 0.005 amps.
- The spatial frequency is 0.5831 cycles per metre.
- The phase is zero.
- The mean level is zero.

Note, again, that this distance-based wave relies on being able to calculate "x", which is the distance from the measuring point to the prevailing "front" of the sound as that "front" moves away from the measuring point. This might not be possible, so this wave is more of a theoretical one than a useable formula.

To convert the time-based electrical wave back into a sound wave, the current can be passed through a loudspeaker, which for the purposes of our example, will be one that works in the same way as our microphone. The current passes through a coil around a magnet attached to a diaphragm, and the variations in current cause the magnet to move backwards and forwards accordingly. The movement of the diaphragm causes air to be compressed at a rate matching the rises and falls of the current, and those compressions are sound.

More about sound

Reality

Although the bee in our example created a 200 Hz pure wave, in reality, its sound would be made up of a range of various and *varying* frequencies and amplitudes. It would produce a signal, and not a pure wave. The same is true for most sounds.

Sound is not a wave

Although it will seem pedantic, it is important to remember that sound is not a wave. We can use waves to portray the changes in air pressure that make up sound, but there is more to the phenomenon of sound than that described by waves. This idea is the same as how a pigeon is not a wave, or how a beach ball and

a beetle are not a wave. We can use waves to describe particular attributes of sound, pigeons, or beetles moving around beach balls, but waves do not portray everything about them. Although people often say, "sound is a wave," they generally do this as shorthand for saying that sound has a characteristic that can be portrayed using waves.

Hearing amplitude, frequency and phase in real life

As we have seen, the characteristics of a sound can be portrayed using time-based and distance-based waves. When mammals and other animals hear sounds, their mental interpretation of a received sound is more or less consistent with the attributes of a time-based wave with this formula:
"$y = A \sin ((2\pi f * t) + \phi)$"

[We will ignore mean level in the formula.]

To examine how humans interpret sound, we will think about the notes on an electronic piano. On the majority of musical instruments, the produced note consists of multiple waves of various and varying frequencies and amplitudes, all starting and ending at various times. For the following examples, we will assume that we are dealing with an electronic piano that produces a pure, single Sine wave tone for each note. In reality, such a piano would sound terrible, but it simplifies the explanation.

When the "A" key on our electric piano is pressed, it produces a tone of 440 hertz.

If the key to the right of "A", which is "A#", is pressed, the keyboard will produce a tone of a higher frequency. It would have a frequency of 466.16 hertz. We would say that the note has a higher pitch. We might say it is "a higher note" which means the same thing.

Pressing the key to the left of "A", which is "G#", would produce a sound with the lower frequency of 415.30 hertz. We would say that it has "a lower pitch", or it is "a deeper note".

The frequency of a sound is related to the perceived pitch of a sound. Faster frequencies are higher notes; slower frequencies are deeper notes. In practice, our interpretation of the pitch of a note is slightly influenced by other factors such as the loudness of the note, and whether we hear a different note just beforehand with which it can be compared.

Pressing the next “A” key to the right would produce a note at twice the frequency of the first “A” key. This would have a frequency of 880 hertz. The way that the human mind works means that we would perceive it as being the same note but higher. We can tell if a note has a frequency that is an integer multiple of another note.

Going back to the original 440 hertz “A” key – if we press it harder, it would still make the same 440 hertz “A” sound, but it would be louder – in other words, the frequency would be the same, but the amplitude would be higher. If we pressed that key more gently, it would make the same sound but more quietly. The tone would have the same frequency, but the amplitude would be lower. Our perception of “loudness” is *mostly* related to the amplitude of a sound, but there are many other factors that play a part. Perhaps obviously, our perception of loudness depends on how far away we are from the source of the sound. Less obviously, our perception of loudness works in a logarithmic scale – a sound is generally perceived as being twice as loud if its amplitude is 10 times as high.

If we press the “A” key and the “A#” key at the same time, the sound from each will be heard at the same time without affecting each other. The sound from one key does not alter the amplitude or frequency of the other. The two notes coexist independently of each other. If we had ten keyboards and pressed the “A” key on each, all at the same time, the sound would be many times louder than the sound from just one keyboard. This would be an example of superposition.

If we had two keyboards side by side, and we pressed the “A” on one, and a thousandth of a second later pressed the same key on the other, we could say that the waves representing the sounds had a difference in phase, although one might struggle to know if it were true or not.

Perception

The brains of mammals, and presumably the brains of other animals, do not necessarily relay or interpret a sound exactly as it is received. You can test this for yourself by how you can choose to focus on a particular instrument in a piece of music, and you will hear it more clearly than the others. You can then choose to focus on a different instrument, and you will hear that more clearly. Nothing has changed about the sound you are hearing, but you can adjust the relative loudness of the individual instruments with your mind. There are countless factors in play when we perceive a sound, so the relationship between the characteristics of sound waves and our perception is fairly complicated.

When our ears detect a sound, it is processed by the brain before we are consciously or subconsciously made aware of it. What we believe we hear and the exact sound that meets our ears are not necessarily the same. In a way, the brain is the gatekeeper of sounds, in that it decides what we actually "hear". A simple example of this is how your brain might ignore a persistent tone in the environment, to the extent that you might end up not being able to hear it. Similarly, if you are intensely focused on something, you might not notice other sounds.

All of this means that the connections between the characteristics of a sound wave and what we would describe as pitch or loudness are not as strongly connected as they could be.

Musical scales

The frequencies of the keys on a piano are arranged in a logarithmic scale. It is a base 2 logarithmic scale. The frequencies of the notes do not increase at a set rate, but at an ever-increasing rate. Every 12 notes, the frequency doubles. For example, the frequency of a particular "A" note will be twice the frequency of the "A" note preceding it, and half the frequency of the "A" note after it. The reason for this relates to how we perceive notes of different frequencies, and ultimately to how we perceive notes of twice the frequency as sounding similar. The idea is easier to see on a guitar fretboard where sharps and flats are treated the same as the other notes. A note on an open guitar string has half the frequency of the note at the twelfth fret (which lies exactly half way along the string). If a fretboard has 24 or more frets, the note at the 24^{th} fret will have a frequency that is twice that of the twelfth fret.

Wavelength versus frequency

Often in explanations of sound, mention is made of wavelength instead of, or as well as, frequency. In other words, someone might say that the wavelength of notes on a piano decrease as we play keys going towards the right, as opposed to saying that the frequencies get higher.

The speed of sound is about 343 metres per second through air at 20 degrees Celsius. As long as the speed of sound is constant, it does not really matter if we think of frequency, period, spatial frequency or wavelength. They are all connected to each other through the formula: wavelength = 343 * period. We can say that a

particular event is dependent on a particular frequency, or we can say it is dependent on a particular wavelength, and both will be correct as long as the sound is moving at 343 metres per second.

Difficulties arise when there is some event that specifically depends on the wavelength or specifically depends on the frequency. This is because the speed of sound is constant only through air at a particular temperature. At higher temperatures, it is faster; at lower temperatures, it is slower. [The temperature of the air within your ears will be higher than 20 degrees.] The wavelength will not have the same relationship with the frequency if the temperature changes. If the speed of sound changes, the spatial frequency and wavelength will change, but the frequency and period will remain the same.

If someone says that a particular wavelength of sound triggers a particular event, then it often needs to be clarified that they mean specifically the wavelength, and not the frequency [or the wavelength when the air temperature is 20 degrees Celsius]. If they do not explain which they mean, then the meaning might be ambiguous or incorrect. Similar ambiguities occur when discussing electromagnetic waves. One might hope that when people say wavelength or frequency in such a situation, they do mean specifically the wavelength or specifically the frequency, but often this is not the case. *Many* people confuse the concepts of wavelength and period, and *many* people confuse the ideas of time-based waves and distance-based waves. It pays to be aware of the potential for other people's mistakes.

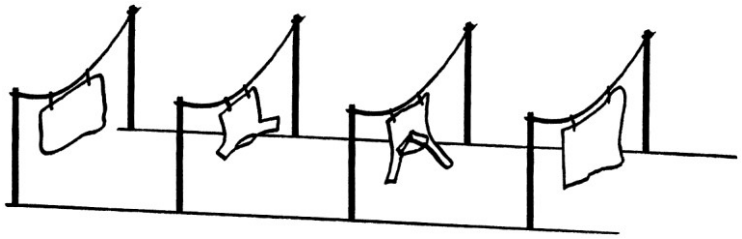
The Doppler effect

If you are travelling at sufficient speed towards the source of a sound, the pitch (that is to say, the frequency) of the sound will seem to increase. If you are travelling at sufficient speed away from the source of a sound, the pitch (the frequency) of the sound will seem to decrease. The same thing happens if the source of the sound is moving towards you or away from you. This phenomenon is called the Doppler effect after the physicist Christian Doppler.

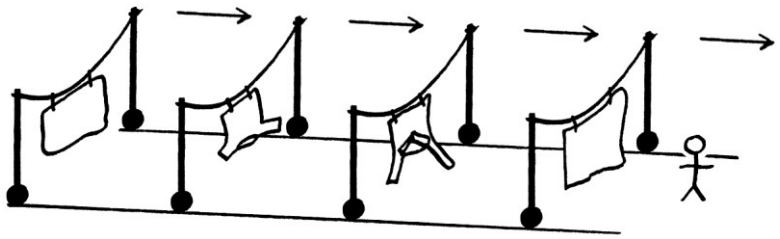
You might experience the Doppler effect when a jet plane flies overhead, or when a fire engine with its siren drives past you. The approaching sound is higher pitched than the departing sound.

With the Doppler effect, when the sound is moving towards you, the *perceived* frequency becomes higher. This is because your ears are receiving the cycles of the wave at a higher rate than they are actually being produced. As the sound moves away from you, the *perceived* frequency becomes lower. This is because your ears are receiving the cycles of the wave at a lower rate than they are being produced. In reality, the actual frequency remains the same – it is only the receiver of the waves who perceives a change. The Doppler effect also changes the perceived spatial frequency of the distance-based wave (and, therefore, the wavelength). It changes the perceived *time-based frequency* because our ears are counting more cycles for every second than before. It changes the perceived *spatial frequency* because the perceived distance between each cycle is reduced.

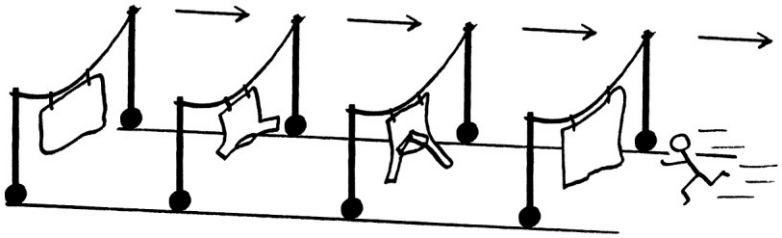
As a reasonable analogy of the Doppler effect, imagine a series of evenly spaced washing lines of drying clothes placed on posts:



Now, imagine that the posts are moving along tracks at a fixed speed towards you. If you stand still between the tracks, the washing on the washing lines will hit you in the face at regular intervals.



The concept of the Doppler effect is analogous to if you suddenly started running through the washing lines instead of standing still:



When running, you would be hit with washing at a faster rate. The washing lines are still the same distance apart, and are still moving at the same speed, but they will seem closer when you run through them [perceived lower wavelength], and they will hit you with less interval between them [perceived higher frequency].

[A significant aspect of the Doppler effect is that it is relevant only to the receiver of the waves. Someone observing you from a distance while you ran through the washing lines would not perceive any change in how far away the lines were from each other or how fast they moved. They would be able to tell that the washing was hitting you in the face more quickly, but to them, the nature of the washing lines would be the same whether you ran through them or not.]

The perceived frequency caused by the Doppler effect for a passing sound can be calculated by:

$$actual\ frequency\ *\ \frac{speed\ of\ sound}{speed\ of\ sound - speed\ of\ source}$$

... which, given that we know the speed of sound through air at 20 degrees Celsius, becomes:

$$actual\ frequency\ *\ \frac{343}{343 - speed\ of\ source}$$

... or:

$$\frac{actual\ frequency * 343}{343 - speed\ of\ source}$$

Supposing someone playing a very loud electronic piano went past us at 19.25 metres per second (69.3 kilometres per hour), playing the note of “A”, which is 440 hertz, then during the whole of the approach, the note would sound to us like the next key (“A#”) on the piano, which has a frequency of 466.16 hertz.

Theoretically, you could play simple tunes by travelling at varying speeds towards the source of a very loud tone, In practice, the noise from the air passing your ears would probably stop you from hearing the sound, and the dramatic changes from one very high speed to another might not be good for you.

The Doppler effect can also occur with electromagnetic radiation, although the subject can become quite complicated. With radio waves, the Doppler effect is noticeable when receiving a signal from a satellite moving across the sky – the perceived frequency will be slightly higher as the satellite moves nearer, and slightly lower as the satellite moves further away. On earth, mobile phones [cell phones] and mobile phone transmitters on masts have to take the Doppler effect into account so that people in moving vehicles can still use their phones.

Chapter 34: Electromagnetic Radiation

Electromagnetic Radiation

[Be aware that parts of this chapter might be incorrect.]

Electromagnetic radiation (often abbreviated to "EMR") is the phenomenon that includes light and radio transmissions. To put this another way, light and radio are the names we give to particular ranges of frequencies within the electromagnetic spectrum.

The subject of electromagnetic radiation is at a completely different level of complexity from that of the waves we have seen so far. There are different levels at which one can think about the subject that vary from being reasonably straightforward to being extremely complicated. Coupled with that, scientists are still working out the finer details of its nature.

Although we can see light, our vision does not particularly help us to understand what it or electromagnetic radiation is. However, we can deduce much about light, and therefore, electromagnetic radiation, from observing its behaviour. We cannot see radio transmissions, but we can detect them using equipment.

Electromagnetic radiation is an entity in its own right, in the sense that it does not need a medium through which to travel, unlike sound.

For a long time, scientists debated whether electromagnetic radiation consisted of particles or waves [in a way that goes against my view that "nothing is just a wave"]. Nowadays, it is generally agreed that electromagnetic radiation exhibits either wave-like characteristics or particle-like characteristics, depending on how it is observed. To put this very unscientifically, sometimes it is observed behaving as if it were a group of bouncing tennis balls; at other times, it is observed behaving as if it were a group of "object-less" theoretical waves.

When it comes to electromagnetic radiation, it is almost standard for people to say that "it is a wave", as if it were just a wave and nothing else. Whatever the nature of electromagnetic radiation, I think it is easiest to understand at a basic level if you think of it as countless particles that exhibit behaviour that can be described using waves. When you have learnt enough to know its true nature, this book will be a distant memory, so if my advice is wrong, it will not matter.

Categories of EMR

Electromagnetic radiation is usually categorised, arbitrarily, according to the frequencies being considered. For example, the radio broadcast frequencies are treated as a separate group to the visible light frequencies, and those are treated as a separate group to the ultraviolet light frequencies, and so on. In reality, there is an uninterrupted scale of electromagnetic radiation frequencies, but the categorisations help distinguish between the areas that have different uses to society.

Some people prefer to categorise electromagnetic radiation in terms of the wavelength of the wave, instead of the frequency. The speed of EMR is the speed of light (visible light being a range of frequencies of EMR). The speed of light is about 299,792,458 metres per second through a vacuum. If we know the frequency of an electromagnetic wave, we can calculate the wavelength by dividing the speed of light by that frequency. Doing this is a variation of the standard distance, speed and time formula:

distance = speed * time

... which, when altered to work with wavelength and period, becomes:

wavelength = speed * period

... which can be re-arranged to be:

period = wavelength ÷ speed

... which means:

1 ÷ period = speed ÷ wavelength

... and therefore:

frequency = speed ÷ wavelength

... and:

wavelength = speed ÷ frequency

As an example, an electromagnetic wave with a frequency of 100 MHz has a wavelength of: 299,792,458 ÷ 100,000,000 = 2.9979 metres.

Given that the speed of light is constant through the same medium, the wavelength and frequency of an electromagnetic wave have a fixed relationship while the wave is travelling through one particular medium. Therefore, for most situations it does not really matter if we use wavelength or frequency. Using wavelength can make things simpler when dealing with high frequencies – it might be easier for some people to say and visualise "a 1 mm wavelength" than "a 300 GHz frequency". However, wavelength is a less useful measure if the wave passes from one medium to another while it travels – the frequency will remain the same, but the wavelength will change, even if ever so slightly.

When it comes to categorising the *radio broadcast* part of the electromagnetic spectrum, it is common to see it divided into regions or bands based on one of several criteria. Some of these are as follows:

Low or high frequency

One system of categorisation is based on arbitrarily chosen adjectives describing the extent to which the frequencies can be considered "high" or "low". These categories are specified by the International Telecommunication Union. This is an organisation that oversees the rules for radio transmissions that all countries in the world have agreed to follow. The basic categories are as follows:

- VLF or "very low frequency", which is 3 kHz to 30 kHz
- LF or "low frequency", which is 30 kHz to 300 kHz
- MF or "medium frequency", which is 300 kHz to 3 MHz
- HF or "high frequency", which is 3 MHz to 30 MHz
- VHF or "very high frequency", which is 30 MHz to 300 MHz
- UHF or "ultra high frequency", which is 300 MHz to 3 GHz
- SHF or "super high frequency", which is 3 GHz to 30 GHz
- EHF or "extremely high frequency", which is 30 GHz to 300 GHz

From a linguistic point of view, the adjectives "very", "super", "extremely" and so on, are not ones that have an implied order, so the only way to remember the order of this list is to learn it. In some languages other than English, the abbreviations remain the same, but the spelling out of the abbreviations is translated. In other languages, the abbreviations are translated too.

Wavelength

Another category system for radio broadcast bands is based on the relative size of the wavelength:

- Long wave
- Medium wave
- Short wave

The frequencies in the “Long wave” band have longer wavelengths than those in the “Medium wave” band, which in turn, have longer wavelengths than those in the “Short wave” band.

The centre wavelength

A system used by radio amateurs is to name a range of frequencies on a rounded-up wavelength from within, or close to, that range. For example, the “20 metre band” refers to frequencies from 14 MHz to 14.35 MHz. The wavelength for the frequency of 14 MHz is 21.4137 metres. The wavelength for the frequency of 14.35 MHz is 20.8915 metres. The frequency with a wavelength of 20 metres is actually 14.9896 MHz, which is above the range of frequencies. However, it is easier to call the whole band, the “20 metre band” than to call it the “21 metre band”, or to remember wavelengths with decimal points.

Remember that as the frequency increases, the wavelength decreases. Sometimes, tables such are this are ordered with the smallest wavelength first, which puts it in reverse frequency order.

The amateur radio bands within the UK, at the time of writing [2022], are:

160 metres, which is 1.81 MHz to 2 MHz
80 metres, which is 3.5 MHz to 3.8 MHz
60 metres, which is 5.1 MHz to 5.405 MHz
40 metres, which is 7.0 MHz to 7.2 MHz
30 metres, which is 10.1 MHz to 10.15 MHz
20 metres, which is 14.0 MHz to 14.35 MHz
16.5 metres [or 17 metres], which is 18.068 MHz to 18.168 MHz
15 metres, which is 21.0 MHz to 21.45 MHz
12 metres, which is 24.89 MHz to 24.99 MHz
10 metres, which is 28.0 MHz to 29.7 MHz

According to the anonymous author of "The Radio Spectrum – UK Allocations" list, the bands were originally 1.8 MHz, 3.6 MHz, 7 MHz, 14 MHz, 21 MHz, and 28 MHz. All were either multiples of 7 or divisors of 7.2 (so roughly divisors of 7). To use their terminology, they were all "harmonically related".

Modulation type

Often a range of frequencies is categorised according to whether the radio signals within that range are generally modulated with amplitude modulation or frequency modulation. [I discussed amplitude modulation in Chapter 16. I will discuss more modulation in the next few chapters.] This categorisation is really only used by "non-radio" people to refer to broadcast radio station bands: AM is often used to refer to the long wave, medium wave, and sometimes short wave bands, while FM is often used to refer to the 87.5 MHz to 108 MHz band. As amplitude modulation and frequency modulation can be used at any frequency, this categorisation is not an accurate one to use if you are being pedantic, but usually people will know what is meant.

Letters

Another system identifies higher frequency ranges by letters. There are variations of this system, so the following list is that of the United States Institute of Electrical and Electronics Engineers (IEEE). This list has been copied from Wikipedia:

- HF or "high frequency", which is 3 MHz to 30 MHz
- VHF or "very high frequency", which is 30 MHz to 300 MHz
- UHF, or "ultra high frequency", which is 300 MHz to 1 GHz
- L for "long wave", which is 1 GHz to 2 GHz
- S for "short wave", which is 2 GHz to 4 GHz
- C, which is 4 GHz to 8 GHz
- X, which is 8 GHz to 12 GHz
- Ku or K_u, which is 12 GHz to 18 GHz [The "u" indicates the range *under* "K"]
- K, which is 18 GHz to 27 GHz
- Ka or K_a, which is 27 GHz to 40 GHz [The "a" indicates the range *above* "K"]
- V, which is 40 GHz to 75 GHz
- W, which is 75 GHz to 110 GHz
- G or "millimetre wave", which is 110 GHz to 300 GHz

[Sources]

The above categories section was compiled with information from:

- "The Radio Spectrum – UK Allocations" list from 2012, the author of which wanted to remain anonymous, as far as I can tell. At the time of writing, it is available here: https://ukspec.tripod.com/spectrum.html
- Wikipedia
- Countless other documents

Interesting EMR frequencies

What follows is a simple list of some of the more interesting frequencies of electromagnetic radiation. This is not a complete list, but just a list of the frequencies I think are interesting. This has been compiled with information from:

- "The Radio Spectrum – UK Allocations" list from 2012, available at: https://ukspec.tripod.com/spectrum.html
- General observations
- ODTR (the Irish equivalent to Ofcom)
- Ofcom (the UK equivalent to ODTR)
- Wikipedia
- Countless other documents

Values in brackets after some of the frequencies refer to the wavelength for that frequency. Note that the boundaries of some frequency ranges are a matter of opinion, such as the frequencies for colours. Any frequencies that are assigned by a government are likely to change, so some of this list will be out of date.

0 Hz: All electromagnetic waves have a frequency higher than this, or else they would not be waves.

60 kHz: The MSF time signal in the UK is transmitted at this frequency. This consists of a short tone played at the start of every second in time. The length of the tone varies to indicate other information such as the time and date. Some clocks use the signal to set the correct time automatically.

77.5 kHz: The DCF77 time signal in Germany. This is similar to the MSF time signal.

148.5 kHz to 283.5 kHz: The long wave band. There are three internationally agreed regions for radio transmissions, which have been specified by the International Telecommunication Union. This means that radio broadcasts from one country within a region will not generally interfere with those of neighbouring countries within that region. The spectrum allocated to particular radio frequency usage varies slightly between the regions. Region 1 covers Africa, Europe, and northern Asia. Region 2 covers North and South America. Region 3 covers southern Asia and Australasia. The range of frequencies for the long wave band given here are those at which it is legal for a licensed radio station in Region 1 to broadcast according to the international agreement. Individual countries within a region sometimes ignore aspects of the agreement, and sometimes they restrict the broadcast range further. The difference in regions means that, for example, an FM radio being sold in the United Kingdom might have a slightly different range of reception from one being sold in a country in other regions. Countries outside Region 1 do not generally use the long wave band. In Ireland and the UK, the long wave broadcast band is from 153 kHz to 279 kHz.

526.5 kHz to 1606.5 kHz: The medium wave band in Region 1.

3.5 MHz to 3.8 MHz: The 80-metre amateur radio band. This is called the 80-metre band because the wavelength of a frequency within the band (about 3.75 MHz) is 80 metres. Thinking of it as the 80-metre band is easier than thinking of it as the 85.65-metre band.

5.9 MHz to 6.2 MHz: One of the short wave radio bands.

27.60125 MHz to 27.99125 MHz: The CB radio band in the UK. The band in the European Union countries is 26.965 MHz to 27.405 MHz.

87.5 MHz to 108 MHz: The "FM" radio broadcast band in Region 1 (Africa, Europe and northern Asia). For Region 2 (the Americas), it is 88 MHz to 108 MHz. For Region 3 (southern Asia and Australasia), it is 87 MHz to 108 MHz. Not all countries within a region adhere strictly to these frequencies. For example, the FM broadcast band in Japan used to be 76 MHz to 95 MHz.

174 MHz to 230 MHz: DAB digital radio in Ireland and the UK.

433.075 MHz to 434.775 MHz (a wavelength of about 69 cm): A frequency assigned to low power devices such as wireless garden thermometers and wireless boiler controllers. These frequencies are generally just used in Europe.

470 MHz to 862 MHz: Digital television in Ireland and the UK.

2.4 GHz (a wavelength of about 12 cm): Wi-Fi routers, Bluetooth, and microwave ovens. Microwave ovens transmit with a much higher power than Wi-Fi routers. For example, a typical Wi-Fi router might transmit 0.1 watts as a signal that is dispersed in every direction over dozens of metres, while a typical microwave oven might transmit 1000 watts, nearly all of which remains within the oven's small insides. That is why your microwave oven cooks food, but your Wi-Fi router does not.

10.7 GHz (a wavelength of 2.8 cm): Satellite television signals. These are transmitted from geostationary satellites floating 36,000 km above the equator of the earth. The low-noise block, which is the device sitting within the receiving dish, converts the radio signal from a frequency of 10.7 GHz into an electrical current with a frequency of about 1 GHz. This means that the signal can travel down a coaxial cable to the decoder. At high frequencies, signals in cables require much more complicated handling than those at lower frequencies. Therefore, it makes sense to convert the signal to a lower frequency as soon as possible in the decoding process.

20 THz (15 μm) to 37.5 THz (8 μm): Infrared radiation that can be detected with a typical thermal camera. A terahertz, abbreviated to "THz", is a million million hertz. A micrometre, abbreviated to "μm" is a millionth of a metre or a thousandth of a millimetre. "Infrared radiation" is the name given to electromagnetic radiation with a frequency above "radio waves" and below visible red light. The prefix "infra", in this case, is the Latin preposition meaning "below" or "lower than". Therefore, "infrared" refers to the group of frequencies immediately below that of visible red light. [Red light has the lowest frequency of the visible light frequencies.] Generally, the frequencies within the entire infrared radiation band are given as being from roughly 300 GHz to 441 THz. I am dividing the range of infrared radiation into two arbitrary sections.

The first section is the range of frequencies that can be detected with a typical thermal camera, such as is used by the military, police helicopters, or people tracking animals at night. Thermal cameras detect *thermal radiation*, which is another name for radiated heat. Apart from some complicated exceptions, every object in the universe radiates heat if it has a temperature above absolute zero. *Radiated* heat or "thermal radiation" is just electromagnetic radiation. The frequency of electromagnetic radiation emitted from an object as thermal radiation is related to the temperature of that object – the higher the temperature, the higher the frequency. An example of this is a red-hot poker in a fire – the poker is so hot

that it is emitting electromagnetic radiation at a frequency in the visible light band. It glows and becomes a visible light source. As the poker cools, the electromagnetic radiation reduces in frequency, and the heat radiation stops being in the visible light spectrum. In daylight, you can still see the poker, but that is because it is reflecting light from other sources. In the dark, you will not be able to see it at all. However, for a while as it cools, you will still be able to feel the electromagnetic radiation on your skin, which you will interpret as heat. The cooler your skin is, the lower the temperature of the poker that you will be able to detect. Thermal cameras can detect thermal radiation at much lower frequencies and at much lower power than human skin. All mammals, perhaps all animals, and perhaps all plants, can sense thermal radiation below the frequency of visible light to some extent, as it is an important factor in survival.

A common misconception is that thermal radiation occurs only in the infrared band of frequencies. However, thermal radiation can have any frequency. The term "infrared" is often misunderstood as implying "heat".

Humans and animals can *see* thermal radiation with their eyes if it is at a frequency in the visible light spectrum. If it is at a frequency above or below the visible light spectrum, then by the definition of *visible* light, it cannot be seen. However, it might still be *detected*. In humans, the detection of radiated heat through skin should be considered one of the senses, but it is generally forgotten about, along with sensing vibration and gravity. Although thermal radiation occurs at any frequency, the general range that a typical modern thermal camera can detect is between 20 THz and 37.5 THz. Hence, I am treating this range as an arbitrary band.

316 THz (950 nm) to 371 THz (808 nm). This is the second section of infrared radiation as I am categorising it, which consists of frequencies just below visible light. ["nm" is the abbreviation for nanometres. There are 1000 nanometres in a micrometre, a million nanometres in a millimetre, and a billion (9 zeroes) nanometres in a metre]. In this range are the frequencies of radiation emitted by infrared LEDs. Infrared LEDs produce the signal in television remote controls. They are also used to illuminate areas at night, in a way that the illumination is visible only to infrared-detecting CCTV cameras. These cameras detect the infrared light reflected back from objects that have been illuminated by infrared LEDs. Infrared light in this range is undetectable by humans without equipment, but it is thought that some animals can see the higher range. Some infrared LEDs glow a faint red colour, which gives an idea of the maximum frequency of the electromagnetic radiation they are emitting. This band of infrared radiation has little to do with the "thermal camera" infrared radiation band, except that very hot things might emit thermal radiation in this band if they are at a high temperature.

As a red-hot poker cools, it will glow in this band briefly as its thermal radiation drops in frequency. Of course, the glow would be visible only to a camera that can detect infrared light. Humans and animals will not sense heat from infrared LEDs, and a thermal camera will not see the LED as warmer than its surroundings (unless the electronics behind the LED is producing its own heat). The "thermal camera" infrared range and the "infrared LED" infrared range tend to be treated as one big range of infrared radiation, which can be confusing. Infrared cameras, in the generally accepted sense of their definition, only detect light of frequencies starting at the high end of the infrared part of the electromagnetic spectrum.

441 THz (680 nm) to 714 THz (420 nm): Between these frequencies is light that is visible to humans – in other words, frequencies of electromagnetic radiation that the average human can detect and distinguish with their *eyes*. [The exact maximum and minimum vary from person to person, so you will see other values in other documents. I have read that some people can see down to 374 THz (800 nm). The values given here are taken from "The Radio Spectrum - UK Allocations" list.] This range starts with the frequency for the colour red, continues with yellow, green, and blue, and ends with violet. The boundaries for what we would describe as being a particular colour are completely arbitrary and socially constructed. In reality, there is a continuous scale of frequencies, and the names of the colours that we assign to particular frequencies are a matter of opinion. [For example, many people might think of violet as being blue. Some people think of pink as being a type of red, while others do not. Some languages treat light blue and dark blue as being different colours.] The frequency range of other animals' vision varies. Although many animals have better night vision than humans, this is generally because they are better adapted to seeing in low light as opposed to being able to see frequencies of electromagnetic radiation that we cannot. [Even if they could see other frequencies, once the sun has gone down and there are no other sources of light to reflect off objects, that ability has no use.] Although human eyes see visible light, our brains interpret it before the information is passed to our conscious or subconscious. [The same thing is true of other mammals and birds.] This means that, ultimately, the brain controls our perception. What our eyes detect might not match with what we think we see. The most obvious example of the separation between our perception and reality is how, in certain circumstances, it is possible to "see" hallucinations. A less extreme example is how we are generally unaware of the blind spot in the vision of each eye. The brain ignores the gaps. Similarly, we are usually unaware of when we are blinking.

714 THz (420 nm): The start of "near-ultraviolet" light. The "ultra" part of the word "ultraviolet" comes from the Latin preposition "ultra" meaning "beyond" in its various senses. Therefore, "ultraviolet" refers to the frequencies above that of the visible colour of violet. The term "near-ultraviolet" means those frequencies above visible violet, but nearer to it than those in the "far-ultraviolet" range. Humans cannot see ultraviolet light, but if near-ultraviolet light is shone on certain materials such as white cotton, the ultraviolet light is reflected back at a lower frequency, which we *can* see. This makes the material seem bright compared with everything around it. This phenomenon is called "fluorescence". The most common example of fluorescence is when a yellow fluorescent safety jacket appears brighter than the surrounding environment. The light hitting it is reflected back at a lower frequency, and therefore, the jacket stands out from the surrounding objects. A similar phenomenon is "phosphorescence", which is when a portion of the light hitting a material is reflected back over a period of time instead of instantly. We see this happening with so-called luminous paint. Luminous paint glows for a while after light has been shone at it. [Strictly speaking, modern luminous paint should be called "phosphorescent paint" because literally luminous paint would glow without needing exposure to light to charge it up.]

30,000 THz (10 nm): This is the frequency of X-rays.

30,000,000 THz (10 pm): Gamma rays. ["pm" is the abbreviation for picometres. There are 1000 picometres in a nanometre, a million picometres in a micrometre, a billion (9 zeroes) picometres in a millimetre, and a thousand billion (12 zeroes) picometres in a metre].

Superposition

All the different frequencies in the electromagnetic radiation spectrum co-exist at the same time. As I have said before, when a radio receiver receives a chosen broadcast, it is actually receiving frequencies from the entire electromagnetic spectrum passing that place, all added together as one gigantic signal. As I have also said before, different frequencies do not mix. We can add waves of different frequencies together, and it will always be possible to separate them (or calculate which ones were added together). This is why Fourier series analysis works. A non-mathematical example of this "unmixability" can be seen when a light is shone at a prism. Despite the frequencies that make up the light having been added together, the prism separates them out in order of frequency. A rainbow does the same thing.

When an FM radio, for example, decodes a broadcast, it has to filter out every frequency of electromagnetic radiation outside the narrow band in which it is interested. Fortunately, most of the electromagnetic spectrum is incompatible with the nature of an FM radio – an FM radio would not (normally) have to make an effort to filter out light or x-rays, for example. The radio's antenna filters out most of the unwanted frequencies while amplifying the wanted frequencies. The electronics within the radio does the rest of the filtering.

Bandwidth

The extent of the electromagnetic radiation spectrum that can be used by humans for communication is limited by both physics and the abilities of our existing technology. Different frequencies have different properties. This is most obvious when comparing light with radio broadcast frequencies.

When it comes to radio communication, there is a limited range of frequencies that can be used, or to put this another way, there is a limited bandwidth. Generally, it is easier to make radio equipment that works at lower frequencies. Higher frequency radio equipment needs to be much more exact in its construction and is generally more expensive. Higher radio frequencies tend to require a clear, straight path between the transmitter and the receiver. For example, a terrestrial satellite receiver dish needs to be able to "see" the satellite in space – communication will not work if there is a hill or a tree in the way. This limitation makes higher frequencies less easy to use than lower frequencies. All of this means that there is really a finite range of frequencies – a finite bandwidth – that can be used for radio communication.

A common theme in part one of this book was that a signal that is not a pure wave is actually made up of two or more pure waves of different frequencies. This means that any broadcast that is not a pure wave will use up more than one frequency of the radio spectrum. As most useful communication uses modulated signals, as opposed to just single pure waves, most useful communication will use up multiple frequencies at a time. In Europe, for example, a medium wave broadcast from a radio station generally uses up 6.3 kHz of the frequency spectrum. A broadcast from a stereo FM radio station will use over 100 kHz. From this, we can see that there is only so much space available for radio broadcasts. This is why governments and international organisations control who is allowed to broadcast, and where in the spectrum they are allowed to broadcast. Nowadays, the radio spectrum is essentially fully used, although a lot of the spectrum is kept back by governments for their own purposes. The only way to increase capacity is either to

use higher frequencies (which are less versatile and harder to use) or to find more efficient ways of using the spectrum that we are using now. Digital television and digital radio are a good example of the latter. By converting sound and vision to a digital form, and by using more advanced modulation systems, digital television and radio use up less bandwidth than their analogue alternatives would have done.

[A similar limit occurs in waves sent down electrical cables. For any piece of cable of a particular design and of a particular length, there is a maximum frequency of wave that can travel down it without being corrupted or being lost due to its amplitude falling too much. Therefore, there is a limited number of frequencies that can travel down any particular cable, and so there is a limited bandwidth. The higher quality the cable and the shorter the cable, the higher the frequencies that can travel down it, and so the larger the available bandwidth. A higher bandwidth means that more information can be sent at the same time.]

EMR in more detail

What follows is a disorganised list of facts about electromagnetic radiation that can help in understanding it.

Electromagnetic radiation consists of particles called “photons”. An infinitesimally short burst of light from the tiniest of sources would consist of a densely packed spherical shell of photons all travelling away from the source in much the same way that fluctuations in sound pressure leave the source of a sound.

Electromagnetic radiation’s name is due to the radiation having two attributes:

- An electrical attribute
- A magnetic attribute

These attributes are inseparable and make up the two wave aspects of a photon. The frequency of a photon relates to the frequency of the electrical attribute and the frequency of the magnetic attribute. These attributes share the same frequency. The two attributes fluctuate at 90 degrees to each other, and also at 90 degrees to the direction in which the photon travels. [By this, I do not mean that they are 90 degrees out of phase, but that the fluctuations are literally in different directions.]

In practice, a source of light might exist for seconds of time or longer. In which case, there would not be a single spherical *shell* of photons, but an ever expanding, densely packed, *sphere* of photons, which is constantly filled from the source.

As the sphere of photons expands ever outwards, there will be a point some distance away after which the wall of photons would appear to be nearly flat, as opposed to curved.

A simple real-world example of a source of light is a burning candle on a table in an otherwise dark and empty room. The light travels away from the flame in the form of a densely packed sphere of photons of many different frequencies. Some of the photons will quickly hit the top of the wax part of the candle; some will hit the table underneath the candle. Some will hit the walls and ceiling of the room.

When a photon reaches an object in the room:

- It might go straight through, in which case, it will continue on the other side, as if the object were not there.
- It might go straight through, but have its direction altered. This is called refraction.
- It might be reflected, in which case, it will bounce off in a different direction.
- It might react with the object's surface, in which case, it is essentially absorbed into it. The reaction will provoke further behaviour in the object's surface, perhaps creating heat or inducing an electrical current, among other things.

For an individual photon, only one of these events can happen at a time. However, for a group of photons, a portion might go straight through, a portion might have their direction altered, a portion might be reflected, and a portion might be absorbed by a material, all at the same time. Which of these happens, and the extent to which it happens, depends on the properties of the object, the properties of the *surface* of the object, the angle of approach of the photons, the frequency of each photon, the density of the photons at that point, and other factors.

Glass is a material through which visible photons can travel unhindered. Prisms are objects that can refract visible photons. Mirrors are made of materials that cause visible photons to bounce off them. Any object that is not transparent or mirror-like is one that absorbs an amount of visible photons, while letting others bounce away. These objects are essentially filtering photons by frequency. The photons of particular frequencies leaving these objects are what give an object its colour.

If some or all of the photons are reflected off an object, that object can be treated almost as if it were a source of electromagnetic radiation itself. The photons reflecting off an object will travel outwards until they meet other objects, in which case, they will pass, be refracted, be reflected or be absorbed.

If photons within a particular range of frequencies reach your eye, they react with the surface of the back of your eye, which sends a signal to your brain acknowledging their existence. This range of frequencies is the range of what we would call "visible light". Our brains can categorise the different frequencies as colours. Frequencies outside the range of visible light frequencies will also hit the back of your eye, but either they will not react with the back of your eye, or your eye will not be capable of acknowledging them.

A photon is destroyed when it is detected (either through a detecting device or by being seen with eyes). For it to be detected, it must react with the detector. The act of detection causes it to cease to exist. When a photon (of a frequency that will react with your retina) reaches your eye, it is destroyed. Photons travel in such huge numbers that this is mostly irrelevant from a detection point of view, but it means that it would be impossible to follow every point in the journey of one photon – once it is observed, it ceases to exist.

Photons are undetectable until such point as they hit a detector of some kind. You cannot observe photons going past a detector. The only way to observe a photon in any way is to "catch" it – in other words, for it to hit the detector. To put this in terms of light, you cannot see light unless it enters your eyes. You cannot see light going past you. You can only observe light that is moving perpendicular to your eyes. [If you shine a torch (flash light) into the sky at night, you might see the light illuminating dust or moisture in the air, in which case it is bouncing off the dust or moisture into your eyes, but you cannot see the light otherwise.

From a superficial point of view, two people standing next to each other and looking in the same direction will see the same sights. However, from an electromagnetic radiation point of view, the two people will be receiving different photons. They cannot receive the same photons as the photons are destroyed when they are seen.

Humans and other animals can see photons within a set range of frequencies directly from a source, and also those that have been reflected from an object. To put this another way, we can see direct light and reflected light.

For say, a yellow object in the room, the photons of frequencies other than those that appear yellow to us will be absorbed into the object, and the photons of frequencies that *do* appear yellow to us will be reflected. Therefore, the object will appear yellow in colour. To put this another way, some objects will absorb all the photons apart from those that appear yellow to us, and therefore, we will see the yellow frequency photons reflected from that object, and so we will describe the colour of that object as being yellow.

Objects that do not reflect much or any light appear black to us. They absorb a large number of photons of many frequencies. A truly black object would reflect no light at all. In everyday life, you would only be aware of its existence because the objects surrounding it *would* reflect light, and therefore, it would stand out. Objects that reflect most, or all light, appear silvery or mirror-like to us. Objects that reflect slightly less light appear white to us. Objects through which light can travel unhindered appear transparent to us.

When we look at an object that does not emit its own light, we are seeing light that has been reflected from something that does, such as the sun or a lamp. If we look at a printed photograph of that object, we will be looking at reflected light from off the printed photograph. If we look at a photograph of that object on a computer screen, we will see *emitted* light from the screen. In this way, there is a difference between a printed photograph, a photograph on a screen, and the actual object that was photographed.

The quantity of photons being detected in one place is what gives electromagnetic radiation its power. With light, the higher the density of the photons received, the brighter the light appears; the lower the density, the dimmer the light appears.

Photons become sparser as they move outwards from the source because the same number of photons fills a larger area. Therefore, the further away from the source that you view a light, the fewer photons will reach you, and so the dimmer the light will appear. Given that, one can understand that the number of photons emanating from even the tiniest of light sources must be huge. Similarly, the number emitted from a star must be unimaginable, especially when you consider that it would be possible to see the light from a star at the same distance in any direction around it.

The more densely packed the photons are, the less the electromagnetic radiation as a whole is blocked by objects. An example of this is how you cannot normally see through your hands, but if you shine a very bright light against your hand, your hands will glow red and you might be able to make out the shape of your finger bones.

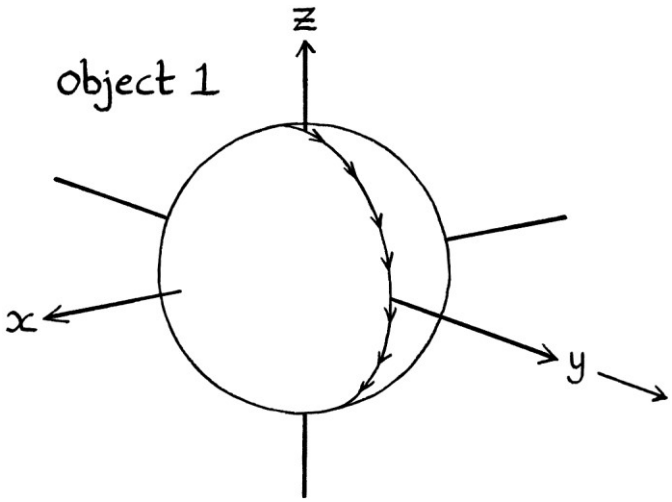
An analogy

The physicist Richard Feynman said that he had no way to visualise electromagnetic radiation. He was not admitting to a flaw in his abilities, but making a statement about how it is ultimately impossible to visualise something as complicated as electromagnetic radiation in an accurate way. Many properties of electromagnetic radiation can be known through maths and experiment, but it is too vast and complicated a subject to be reduced to a simple analogy or visualisation for humans. Any simple way of imagining how electromagnetic radiation works will be inherently incorrect. We saw the difficulties of making good analogies for sound in the previous chapter, and sound is a far simpler concept than electromagnetic radiation. Despite all of this, in this section, I will give an analogy for the basic behaviour of electromagnetic radiation. This analogy is complicated enough that it might be misconstrued as an explanation. However, it should really be thought of in the same way that we would think about how a line of people passing boxes to each other is similar to one attribute of sound. The line of people is analogous to one aspect of the phenomenon of sound, and, with other information about sound, it helps us visualise sound more easily. The following analogy is not representative of electromagnetic radiation, but might help you visualise aspects of it more easily.

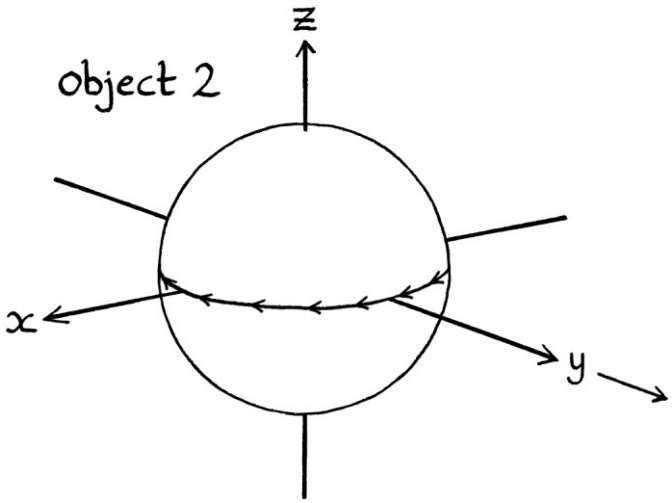
The analogy will not help you with maths or physics, but it might be helpful if you want to visualise electromagnetic radiation without drowning in maths. As you learn more, you can decide in which ways it is lacking. If you wrote this in a physics exam, you would probably fail.

A photon can be thought of as being analogous to the beach ball and beetles example from Chapter 32. We will start by imagining the photon as the beach ball, and imagine objects rotating around it in the same way as the beetles. In this way, we are starting to have a visualisation that incorporates the two wave aspects of a photon as well as the particle attribute. The beech ball will be treated as a sphere, Beetle 1 will be called Object 1, and Beetle 2 will be called Object 2. Instead of thinking of three waves for each object, we will only consider the z-axis wave of Object 1 and the x-axis wave of Object 2. Our beetles both started at the same place on the back of the ball, and so will Object 1 and Object 2.

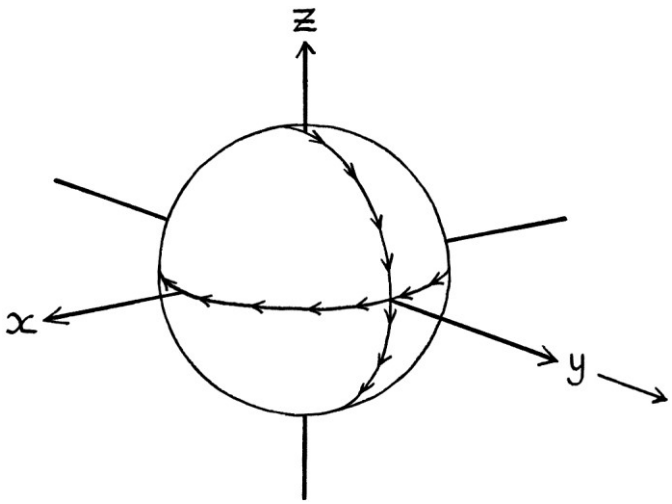
The path of Object 1 around the sphere is as so:



The path of Object 2 around the sphere is as so:



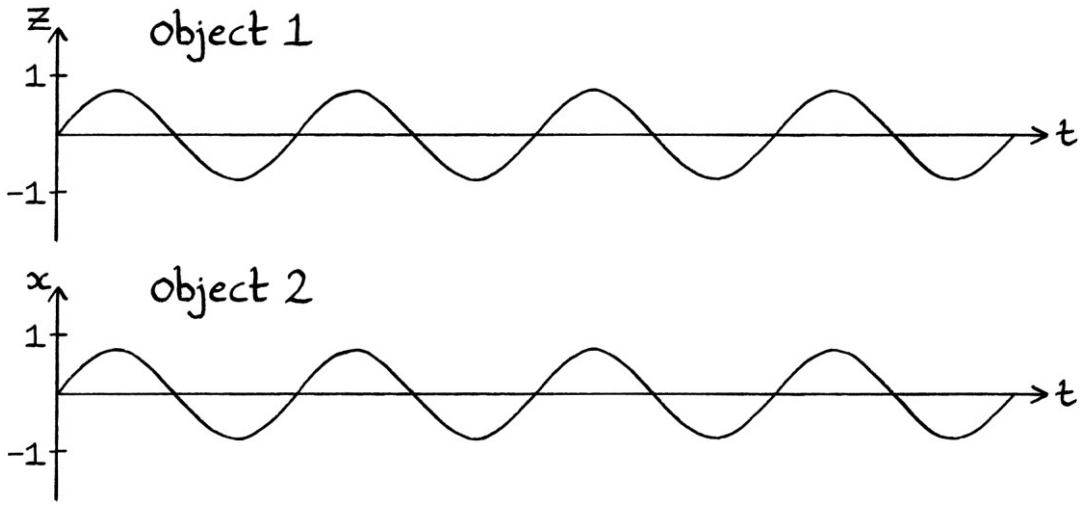
The paths of each object together are as so:



We will say that Object 1's z-axis wave is the electric wave and the Object 2's x-axis wave is the magnetic wave. In this way, we have a situation where we can have an electric wave and a magnetic wave in one contained entity.

If Object 1 and Object 2 were to rotate around the sphere at 100 million cycles per second, while the sphere moved away from the source at the speed of light, then the situation would be analogous to a 100 MHz photon.

We will say that both Objects start at the back centre of the sphere. This means that Object 1 will start by rising up the z-axis, and Object 2 will start by rising along the x-axis. The Sine waves that portray their heights along their respective axes will have the same phase and dimensions as each other, and will appear as in the following pictures. To maintain how this is just an analogy, the vertical units are nameless units. The time axis is unlabelled to allow the same graph to apply to any frequency.



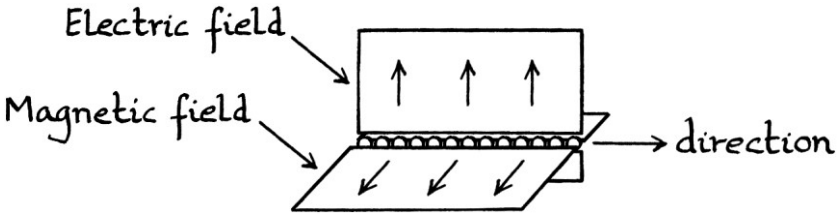
In this analogy, the waves from Object 1 and Object 2 are transverse waves, in that they are at 90 degrees to the direction of travel of the spheres. They are also Type A waves, in that measurements of the y-axis of Object 1 and the x-axis of Object 2 would need to be made at the moving spheres. [They might be Type C waves, but to keep things simple, we will try not to think about this.]

Continuing with the analogy, we will say that there are countless spheres being emitted from a source at any one time. They travel outwards away from the source at the speed of light. The spheres can overlap each other, and even occupy the same space as each other.

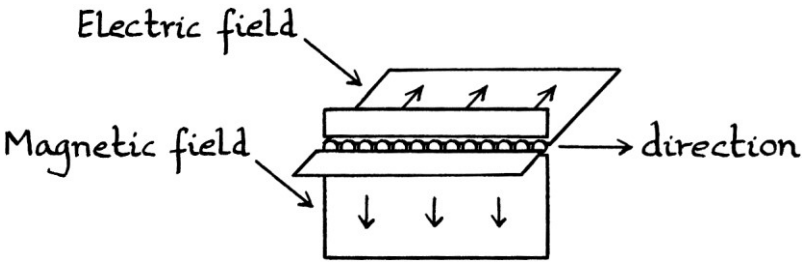
Electric and magnetic fields

If we say that the z-axis movement of Object 1 is analogous to being an electric attribute, and the x-axis movement of Object 2 is analogous to being a magnetic attribute, then one sphere will have an electric attribute and a magnetic attribute. We will say that a group of spheres combine to produce an electric *field*, and a magnetic *field*. The electric field will fluctuate upwards and downwards in the direction of the z-axes of the spheres, and the magnetic field will fluctuate in and out in the direction of the x-axes of the spheres. The fluctuations of the electric and magnetic fields will have the same frequency as the Objects that create it.

In each of the next two pictures, there is a row of spheres. The collection of Object 1s creates an electric field that fluctuates up and down (along the z-axis). The collection of Object 2s creates a magnetic field that fluctuates in and out (along the x-axis). When the electric field is high, the magnetic field is high:

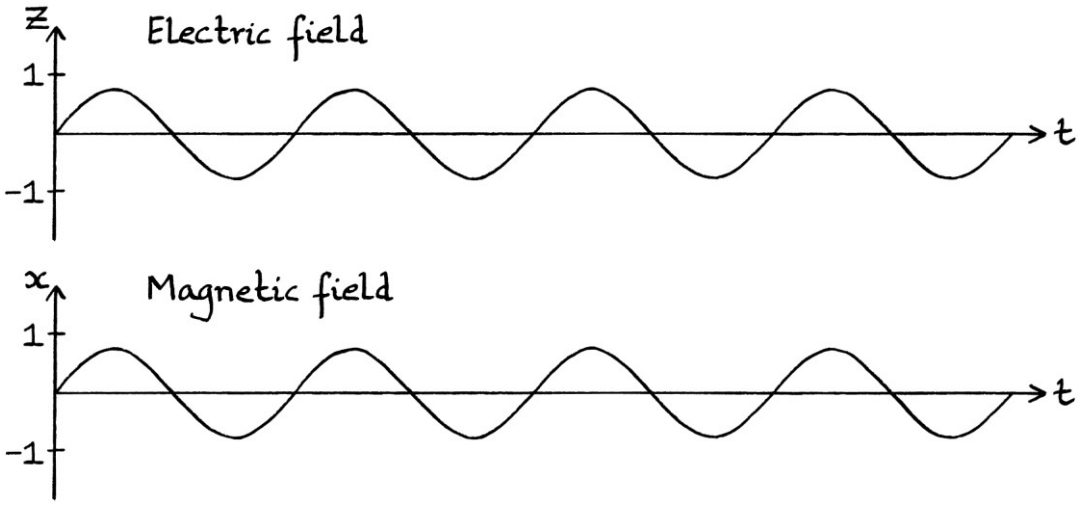


When the electric field is low, the magnetic field is low:

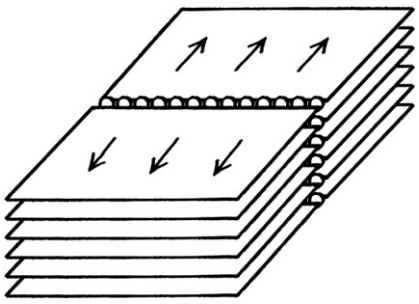


The electric field and the magnetic field fluctuate according to the z-axis positions of the Object 1s, and the x-axis positions of the Object 2s.

The waves that indicate the fluctuations of the two fields will match the waves for each Object 1 and each Object 2, but with the amplitudes measured in different units.

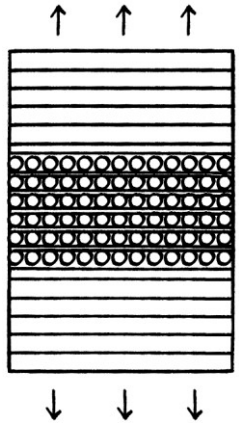


The analogy and reality become much more complicated when we acknowledge that the spheres would seldom exist in a neat row such as this. In practice, there would be a dense volume of spheres moving out from a source. Therefore, the nature of the electric and magnetic fields would not be in neat planes as in the picture, but in a form that is conceptually harder to visualise. The electric and magnetic fields would be *everywhere*, and there would be no way to distinguish one area of a field from any other. However, there would still be fluctuations at right angles to the direction of motion, and these would still be detectable. As the first step to demonstrating how complicated the idea becomes, supposing we look at just the magnetic fields, and we have a one-sphere-thick grid of spheres, we would have the following situation:



There are several horizontal planes of magnetic fields.

For the electric field for this situation, we would have overlapping planes:



In practice, given how small the spheres are, there would be countless indistinguishable planes for both the electric and magnetic fields. If we had more than a two-dimensional array of spheres, the magnetic field planes would all overlap each other, and the electric field planes would all overlap each other. There would really be a continuous electric field and a continuous magnetic field throughout the entire area where the spheres existed.

One way to simplify the idea in one's mind is to say that the fields only appear when they are detected. By thinking in this way, there will be fewer layers to worry about. Whether this makes the analogy better or worse is a different matter.

Units

Outside of analogies, electric fields are measured in volts per metre or in newtons per coulomb. Magnetic fields are measured in tesla or in gauss.

Sometimes, in graphs that you might see in books or online, the magnetic field is portrayed as having a smaller amplitude than the electric field. However, because they are different types of phenomena and are measured with different units, it is difficult to say that one is weaker, stronger or equal to the other. Any comparison would be like comparing the height of an orange to the hardness of an apple. They are different, so although they could be compared, the comparison would not be useful.

Waves

In our analogy, we could create time-based and distance-based wave formulas that described the z-axis position of an Object 1 around a sphere, and the x-axis position of an Object 2 around a sphere. We could also create wave formulas based on the strength of the electric field and the magnetic field created by a collection of Object 1s and Object 2s. [Although, as the analogy is intentionally vague, we do not have enough information to create formulas with actual values.]

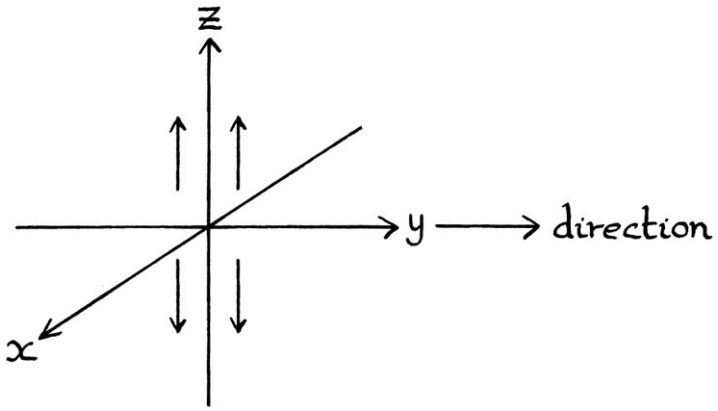
Hitting other objects

In our analogy, if a sphere hits the surface of a material, it might be reflected, in which case, it will bounce off in the same way that a ball would bounce off a wall. Alternatively, it might travel through the material or react with it in some way. This is still analogous to the behaviour of a photon.

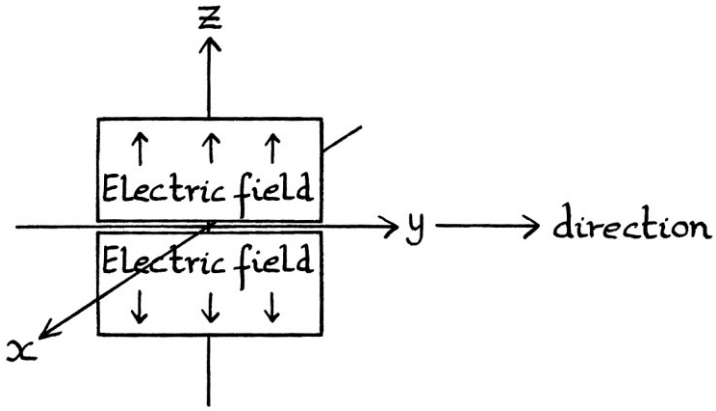
Polarisation

The analogy lends itself to understanding the concept of "polarisation". Polarisation is the direction in which the *electric* field of an electromagnetic wave fluctuates. If the electric field fluctuates vertically, the wave describing the electric field is said to have a "vertical polarisation". This idea is easiest to imagine with an Object 1 moving around a sphere and so fluctuating up and down the z-axis:

The z-axis movement of an Object 1 is as so:

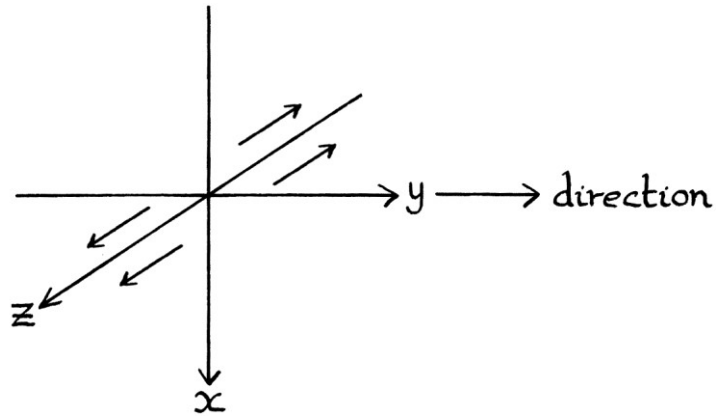


If there are a group of spheres, then the electric field created from them will also fluctuate up and down in this way.

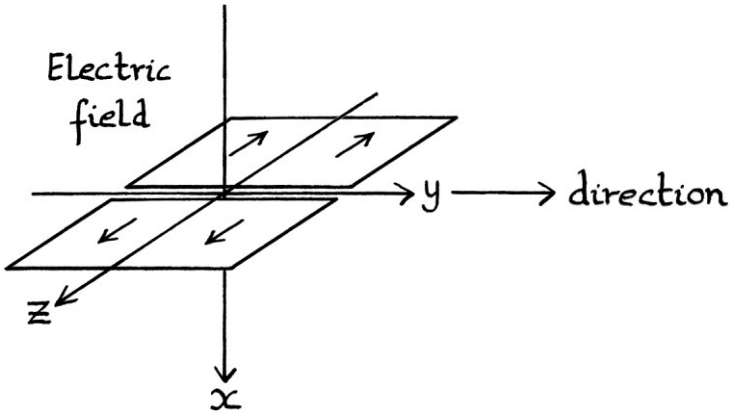


As the electric field fluctuates vertically, the situation is called “vertical polarisation”.

If the electric field fluctuates horizontally, the wave describing the electric field is said to have a “horizontal polarisation”. This is easiest to visualise with an Object 1 moving around a sphere and fluctuating up and down the z-axis, but with the axes rotated, so that the z-axis is now on its side. The z-axis points sideways instead of upwards. The x-axis points downwards. All the axes have been rotated around y = 0:



If there are a group of spheres, then the electric field created from them will also fluctuate horizontally, and we can say that there is “horizontal polarisation”.

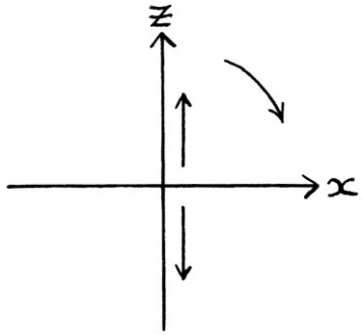


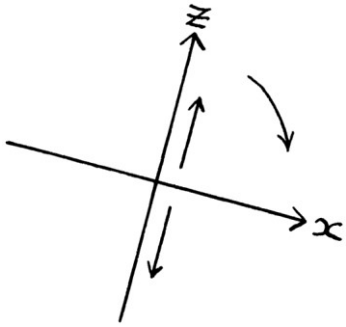
[Note that the rotation of the axes is just a way to make the idea of polarisation easier to understand. We could just as easily have pictures with the axes remaining the same, but with the Object 1s and the electric field fluctuating along the x-axis. It is the state of fluctuating horizontally or vertically that is important, and not which way up the axes are.]

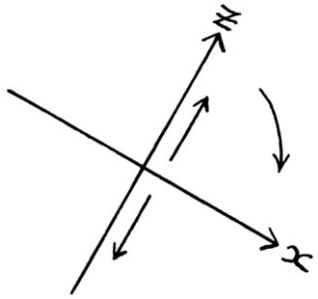
Polarisation does not have to be just horizontal or vertical. It can be at any angle. Polarisation can be used to describe the fluctuations of all transverse waves. The pigeon wing tip wave has a vertical polarisation. If the pigeon flew on its side, it would have a horizontal polarisation. If the pigeon flew upside down, it would still have a vertical polarisation – its wing fluctuations would still be upwards and downwards. If the pairs of Objects in our analogy were upside down, they would still have a vertical polarisation. In our analogy, there is no way to know if the Objects are upside down or not – they will still fluctuate up and down and in and out with reference to the centre of the sphere.

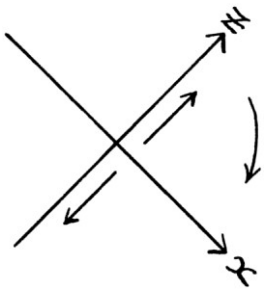
Polarisation as a concept does not apply to longitudinal waves because the nature of longitudinal waves means that the fluctuations are always in the same direction as the movement of the wave.

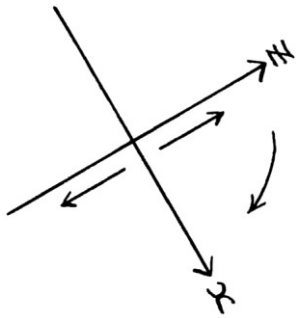
We can also have a situation where the phenomenon being described by waves rotates as it moves. In our analogy, the sphere would rotate as it travelled, with the objects still rotating around it but maintaining their path around the same parts of the sphere. The idea can be easier to visualise if we consider the axes rotating at the same time. A portrayal of the idea, with just the axes drawn, and with the y-axis pointing into the paper away from us is as follows. We are interested only in the movement of the Object 1 up and down the z-axis. This is more complicated than usual as the z-axis (and all the axes) are rotating as the Object 1 moves.

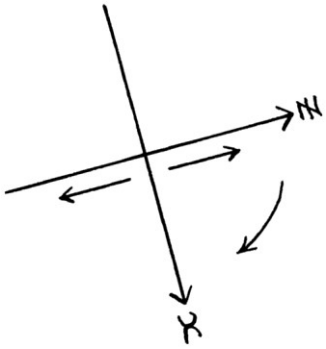

N
x


N
x


N
x


N
x


N
x



In such a case, we would say that the wave describing the Object 1 has a "circular polarisation", and that the wave describing the fluctuations in the electric field has a "circular polarisation". In our analogy, we can distinguish between the directions of rotation:

- If the sphere is rotating clockwise from the point of view of someone observing it moving away from them, then there is a "right-handed circular polarisation" or "RHCP" for short. In reality, we would be more interested in the electric *field*. Therefore, we can say that right-handed circular polarisation is when the electric field rotates clockwise as seen from the point of view of somebody seeing the fields moving away from them.

- If the spheres are rotating anticlockwise from the point of view of someone observing the spheres moving away from them, then there is a "left -handed circular polarisation" or "LHCP" for short. Again, in reality, we would be more interested in the electric field. Therefore, we will say that left-handed circular polarisation is when the electric field rotates anticlockwise as seen from the point of view of somebody seeing the fields moving away from them.

[If the terms "left handed" and "right handed" are not clear, then imagine something rotating with its *top* moving to the left for "left handed" and its *top* moving to the right for "right handed".]

To complicate everything, the concept of circular polarisation is sometimes defined in the opposite way – in other words, some people think of right-handed polarisation as rotating clockwise from the point of view of the entity moving *towards* them, and of left-handed polarisation as rotating anti-clockwise from the point of view of the entity moving *towards* them. The "moving away" definition is used in engineering among other subjects, while the "moving towards" definition is used in the academic subject of optics. The conflict in definitions means that whenever you use the terms, you should clarify which definition you have in mind.

For our pigeon, circular polarisation would mean that the pigeon would still fly with its head first, but it would rotate as it flew.

The different polarisations are most apparent in everyday life when using radio antennas. For vertical and horizontal polarisations, a receiving radio antenna might need to be placed either upright or at 90 degrees, depending on which of the two common polarisations the transmitter is using. An RHCP or LHCP antenna also has to match the circular polarisation direction of the transmitter. Some circular polarisation antennas are distinctive because they have a helical shape.

The different polarisations of visible light become apparent with the use of polarising filters. Sources such as the sun produce light of many different polarisations. Such light is called “unpolarised light”. Polarising filters block light that is not of a particular polarity.

Philosophical ideas

We could extend the analogy in ways that are not necessarily correct in terms of electromagnetic radiation, but that create philosophical ideas about how other theoretical wave-related particles could exist. In this way, the analogy stops being an analogy of a real-world entity, and instead becomes an analogy of a fictional entity. Doing this can help broaden our ideas of how waves could exist elsewhere. For example, we could say that maybe when an object hits a surface, it is instantly attracted to the objects within one wavelength’s distance around it. Maybe each object repels those next to it, but this is only apparent when some objects are blocked and there is a gap to be filled.

Summary of this section

Remember that this section has been an analogy of electromagnetic radiation. It might be helpful in some situations, but conversely, it might interfere with your learning in other situations.

Conclusion

Electromagnetic waves are the most complicated form of waves that we have looked at in this book. In this chapter, we have seen only the most trivial of introductions to the subject. Note that there might have been things in this chapter that are not completely correct, and some of what you have read might interfere with your academic learning. To understand electromagnetic radiation well requires a higher level of maths than that taught in this book.

Chapter 35: Simple modulation

If we can control one or more of the attributes of a wave, we can use the wave to encode information. Radio and sound waves are two such types of wave. Radio waves can travel over long distances taking the information with them, which makes them ideal for communication.

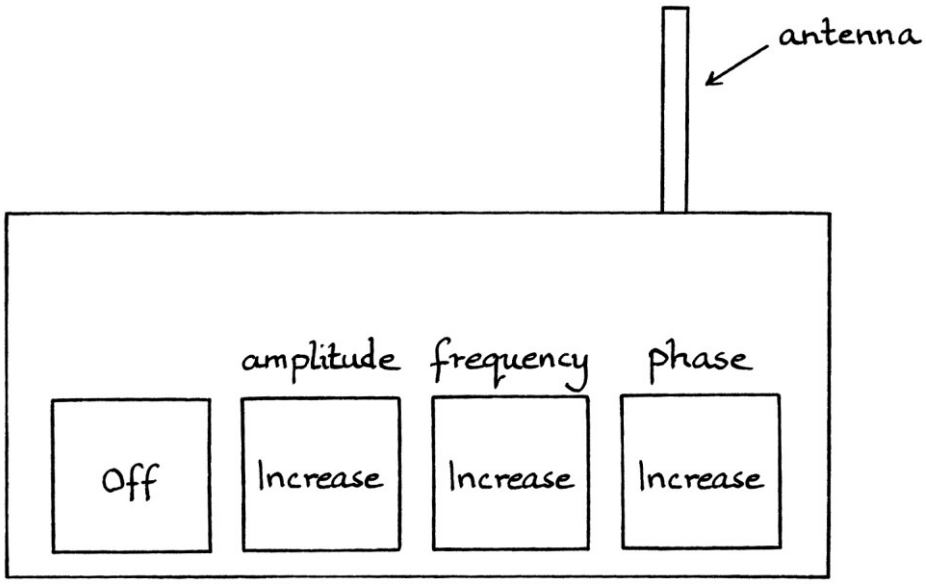
Information is generally encoded by altering the amplitude, frequency, or phase of the wave in a pattern that is related to the information being encoded. This alteration of the wave is called "modulation". Modulation means modification, influencing, or alteration. When used with waves, the word "modulation" usually implies an alteration *for the purposes of conveying a message*. It is rarer to see the term used to mean any alteration at all, but strictly speaking, it is just as valid to do so.

We looked at amplitude modulation in Chapter 16. The basic idea of modulation is that an existing pure wave with a particular amplitude, frequency or phase, has one or more of its attributes altered to carry a message. This wave is called the "carrier wave". The change in attributes is usually done in accordance with the instantaneous amplitude of a second wave or signal, which is essentially the message. This second wave or signal is called the "modulating wave" or the "modulating signal".

In this chapter, we will look at "shift keying", which is the easiest form of modulation to understand. For nearly all of this chapter, we will work in degrees to make the explanation clearer.

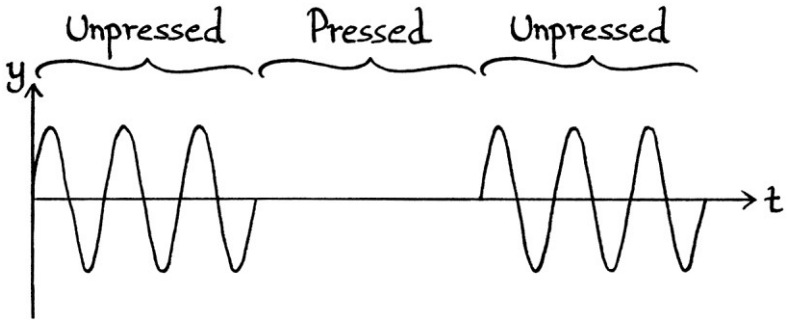
We will imagine a theoretical radio transmitter that constantly transmits a wave at a particular frequency. The actual frequency is not important, and to keep this example simple, we will say it is a very low frequency. [We will use different frequencies in different examples to make the examples clearer.] In practice, a low frequency wave would be difficult to transmit, but low frequencies make everything clearer and easier to understand. We will say that the amplitude is 1 unspecified unit.

The only way to control our radio transmitter is via four push buttons: an "off" button, an "increase amplitude" button, an "increase frequency" button, and an "increase phase" button.

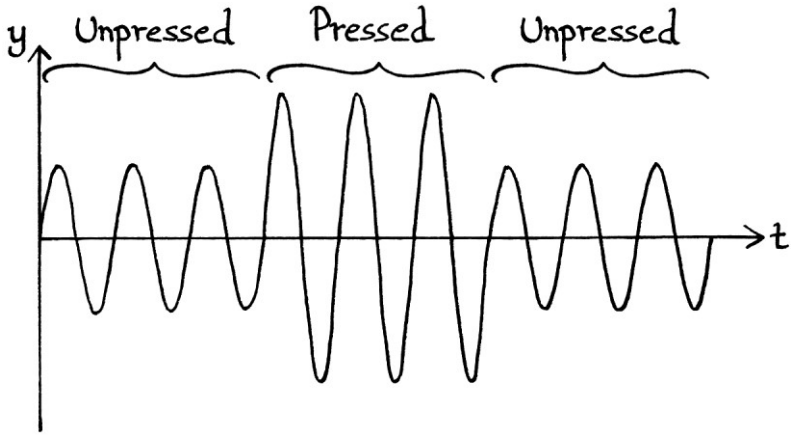


- The “off” button instantly switches the transmitter off, and keeps it off while pressed. When the button is released, the transmitter starts transmitting again.
- The “increase amplitude” button instantly doubles the amplitude for the time that it is pressed. When the button is released, the amplitude returns to normal.
- The “increase frequency” button instantly doubles the frequency for the time that it is pressed. When it is released, the frequency returns back to how it was before.
- The “increase phase” button increases the overall phase of the wave by 90 degrees (0.5π radians) for the time that it is pressed. When it is released, the overall phase returns back to zero. I will explain what this means in more detail later in this chapter.

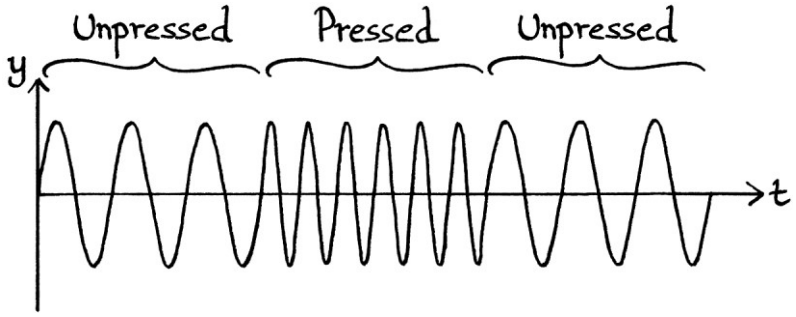
The effects of the four buttons can be seen in the following four wave graphs. While the “off” button is pressed down, the wave is switched off completely:



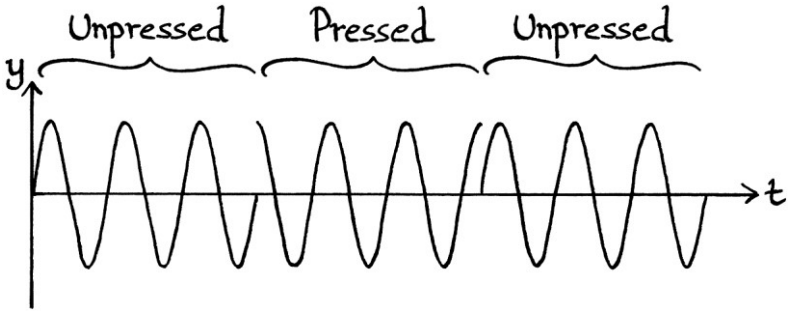
While the “increase amplitude” button is pressed down, the amplitude is doubled:



While the “increase frequency” button is pressed down, the frequency is doubled.



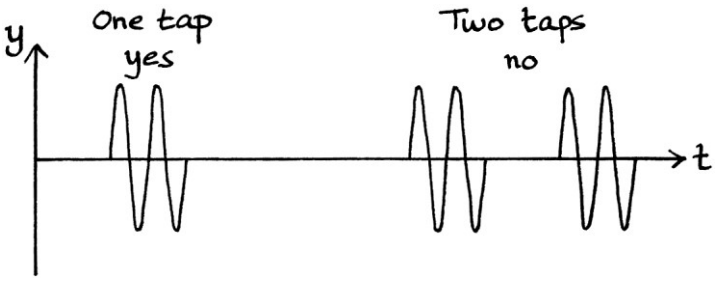
The “increase phase button” increases the instantaneous phase of the wave by 90 degrees while the button is pressed. This results in the wave curve being shifted 90 degrees to the left. When the button is released, the phase returns to normal. The effect is the same as if we had two different waves with starting phases of 0 degrees and 90 degrees, and pressing the button switched from one to the other.



Despite having only four controls, we will use this transmitter to send messages to the outside world.

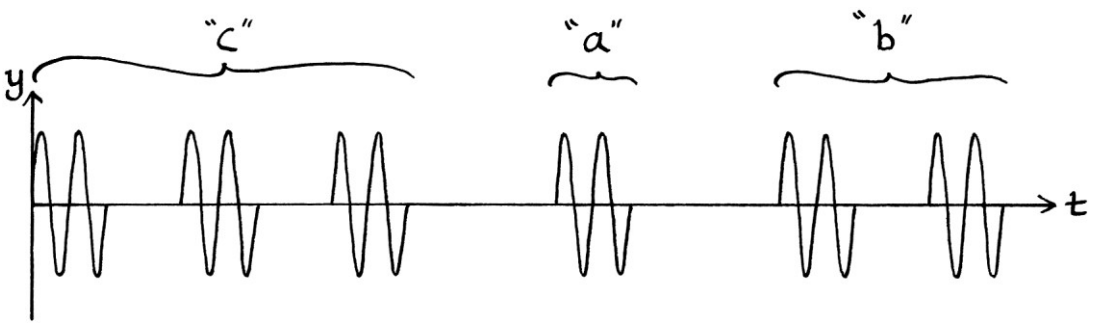
On-off keying

The simplest way to send a message with our transmitter would be the equivalent of “one tap for yes; two taps for no”. In other words, we would use the “off” button so that the transmitter is on for one short burst to indicate a “yes”, and on for two short bursts to indicate a “no”. This is better than nothing, but it relies on our being able to hear someone asking us questions, and that they ask relevant questions.



In the above picture, we are using two cycles for each “tap”. This is an arbitrary amount that makes the pictures clearer. We could just as easily use one cycle, half a cycle, twenty cycles, or any number of cycles. The number of cycles we need to use for each “tap” depends on how easily the message can be decoded. If our wave had a frequency of, for example, 10 million cycles per second, then it would be difficult for someone to distinguish one cycle, and we would need many more cycles for each “tap” for the message to be interpreted correctly.

Expanding on the “one tap for yes” idea, we could switch the transmitter on and off to count out letters of the alphabet where “a” is 1, “b” is 2, “c” is 3, “d” is 4, and so on. For example, the transmitter could be off for a while, on for a short moment and then off again to indicate the letter “a”. It could be off for a while, then on for two short moments, then off again to indicate the letter “b”. It could be on for three short moments to indicate the letter “c”; and so on. This would take a lot of time to send a message of any length. The word “cab” could be portrayed as so:



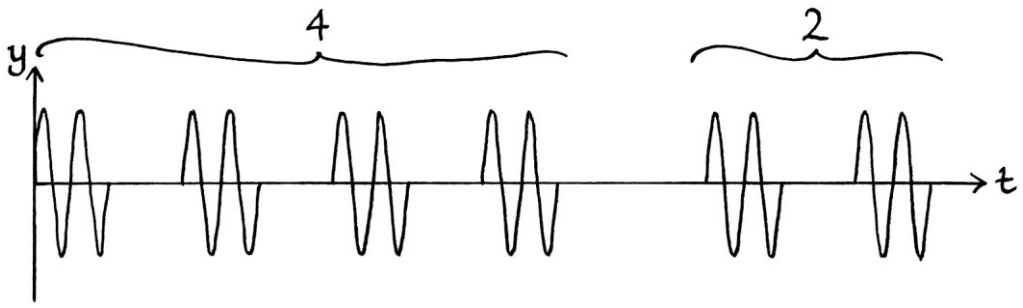
A slightly better method is based on a tapping system often used by prisoners of war. This uses a five-by-five grid of 25 letters of the Latin alphabet. By convention, the letter “k” is left out – if we need to use a “k”, we can use “c” instead.

	1	**2**	**3**	**4**	**5**
1	a	b	c	d	e
2	f	g	h	i	j
3	l	m	n	o	p
4	q	r	s	t	u
5	v	w	x	y	z

Each letter is portrayed by tapping out its coordinates in the grid. Using our transmitter, we can do the same thing by switching the machine on and off the relevant number of times.

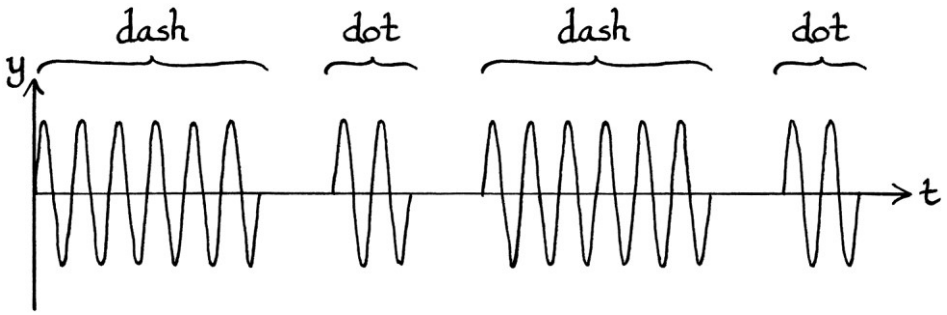
For example, the coordinates of the letter “s” are (3, 4). Therefore, we have the transmitter on 3 times, pause with it off briefly, and then switch it on 4 more times. This method is slightly faster than the previous one. In this method, the letter “z” needs 10 switches on and off, whereas it would have needed 26 switches on and off with the previous method. This method still requires the listener to understand the system (or to figure it out), and it still requires a lot of switching on and off.

The letter “i” (4 followed by 2) would appear as so:

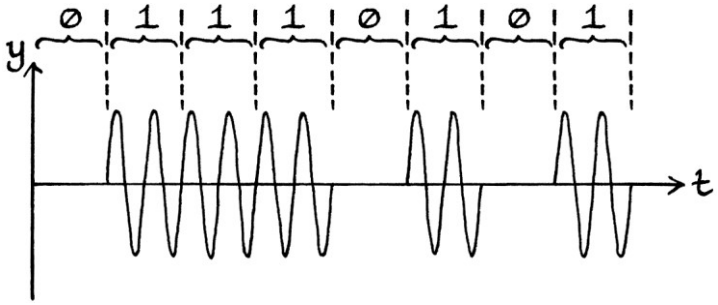


[The layout of letters in the grid is not the best one possible. For English messages, it would be better to keep the letter “k” and instead leave out “z”, which can be replaced in any words with “s”. Doing that would leave the order of the alphabet undisturbed. A faster system would have the most frequently used letters of the alphabet at the start of the grid, but that would make the system harder to remember.]

A more advanced way of sending a message by switching on and off the transmitter would be to use Morse code, and to switch the signal on and off according to the dots and dashes of the Morse code alphabet. This is probably the best method for sending letters of the alphabet. Depending on how many cycles we use for each dot or dash, the letter "c" [dash dot dash dot] might appear as so:



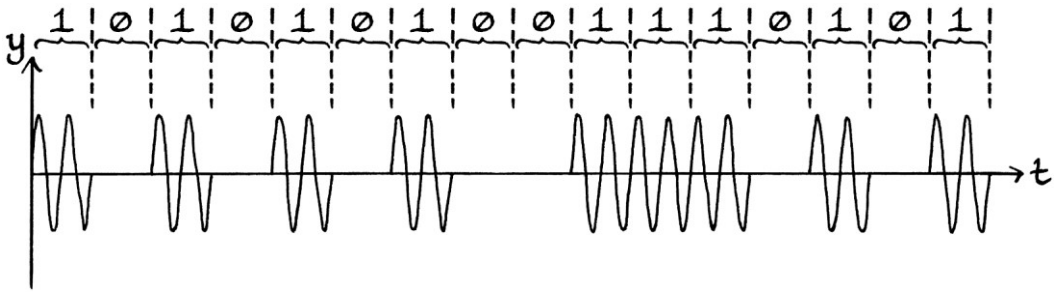
A more complicated method for sending a message involves switching the signal on and off according to the ones and zeroes of a binary encoded message. For example, if we wanted to send the letter "u", we would use the ASCII binary encoding for that letter: 01110101. We would then send each digit of this number in turn, from left to right as 0, 1, 1, 1, 0, 1, 0, 1. We would do this by using a carefully spaced series of on switches and off switches with an "on" representing a 1, and an "off" representing a zero. Our signal would look like this:



[I will explain binary in detail in Chapter 40. For now, it is enough to know that it is a counting system where every digit is either zero or one. Each digit is called a "bit". The "American Standard Code For Information Interchange" (ASCII) is a system that assigns every letter of the alphabet a number, so that computers can have a consistent way of dealing with, and storing, text. The numbers can be treated as being decimal numbers, hexadecimal numbers or binary numbers. Treating them as binary numbers is useful in situations such as this. When treating them as binary numbers, each letter is represented by 8 binary digits (which we could also refer to as 8 *bits*). As an example, in the ASCII system, the letter "A" is 0x41 in hexadecimal, which is 01000001 in binary and 65 in decimal. The letter "B" is 0x42 in hexadecimal, which is 01000010 in binary and 66 in decimal. The

numbers from 0x01 to 0x7f in hexadecimal (1 to 127 in decimal, or 00000001 to 01111111 in binary) are used for a subset of the Latin alphabet, some punctuation, and some mostly obsolete printer control codes. The numbers from 0x80 to 0xff in hexadecimal (128 to 255 in decimal, or 10000000 to 11111111 in binary) are used to portray other letters and special characters. Which characters this half of the ASCII set represents varies depending on the language in which the ASCII characters are being used. For example, if you use an English version of Microsoft Windows, they will appear as accented Latin letters. If you use the Russian version of Microsoft Windows, they will appear as Cyrillic letters.]

The above method of sending a message has both advantages and disadvantages over the previous methods. The first *disadvantage* is that this method requires very precise timing – if one of the on or off switches is too long or too short, the message ceases to make sense. We could get around this by making every digit last one second and looking at a watch while we sent the message. Another disadvantage is that is harder to tell when the message starts or stops – if we switch the machine off at night to go to sleep, someone might think we are sending a very long series of zeroes. Conversely, if we did want to send a very long series of zeroes (not that there are any ASCII characters consisting entirely of zeroes), someone might think the message had ended. This problem can be solved by sending a set number of ones and zeroes just before the actual message starts. For example, we could send alternating 1s and 0s eight times before our message begins. This alerts the listener that a message is about to start, and indicates at what point the message starts. Such an introduction is called a "preamble". A preamble might be 8 pairs of ones and zeroes, 16 pairs, or even more. The length and pattern of the preamble of 8 pairs or more mean that it is unlikely to be mistaken for the content of an actual message. So that the picture fits on the page, the following drawing uses just 4 pairs of ones and zeroes before the message starts. [In the real world, this would not be enough to avoid confusion with an actual ASCII character.]

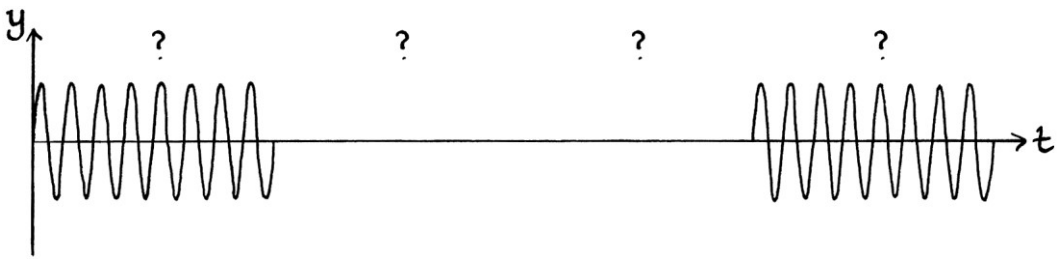


Note that the actual message has to start immediately after the preamble because any moments when there is no wave will be interpreted as zeroes.

The end of the message could similarly be marked by a set number of alternating ones and zeroes, or the context of the message alone might be enough to indicate when it had ended.

The preamble is also useful to get the attention of someone waiting for the message that the message is about to start.

The preamble solves yet another problem with this communication method. Without there being an agreement between the sender and receiver over how much time, or how many wave cycles, represent one individual binary digit, it can be difficult to tell how many digits have been sent. For example, the binary number 1001 could just as easily be misinterpreted as 11000011 or 111000000111 or 1111000000001111 and so on, depending on how we divide the time up.



The timing of the bits in the preamble solves this problem because it can be used as a reference for the rest of the message.

A disadvantage of the binary encoding method from the point of view of our transmitter is that it requires more switching on and off to send a letter of the alphabet than, for example, Morse code. However, this and other disadvantages are offset by one *huge* advantage – an advantage that makes this method so important in modern times – by sending binary, we can send *any* type of information. We do not have to send text – we can send any information that can be encoded as binary. In other words, we can send images, documents, music, videos, encrypted messages, and so on. It is unlikely that we would want to use our transmitter to send someone an MP3 file, for example, by hand-keying the binary data, but we could if we wanted to do so.

When a message is conveyed by switching the signal on and off as in all of the above examples, the method is called “on-off keying”. This is often abbreviated to “OOK”. On-off keying was used in early radio communications because it was simple to do – a message could be sent by a person manually operating a lever that switched the signal on and off. That lever was called a “key”, which is where the term “keying” comes from. [Use of the word “key” to mean a lever-like switch (as

opposed to an object that opens locks), can also be seen when we discuss keys on a piano and keys on a typewriter.] As on-off keying was used in the early days of radio communications, the concept can seem a bit simplistic compared with some other forms of modulation, but there are still situations when it is useful. For example, the simplicity of Morse code means it is possible to understand a received message when other types of communication are too faint to be heard. This means that we can communicate with someone much further away or in much worse conditions than when using other methods. [With Morse code, on-off keying is often called “continuous wave” or “CW”.] On-off keying is used to control the light source in some fibre optic cable systems. Television remote controls use on-off keying with an infrared LED.

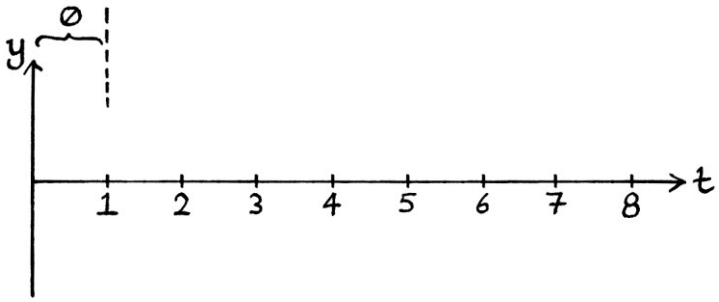
A potential source of confusion

[This section will be irrelevant to many people, and most people might never even need to think about it.]

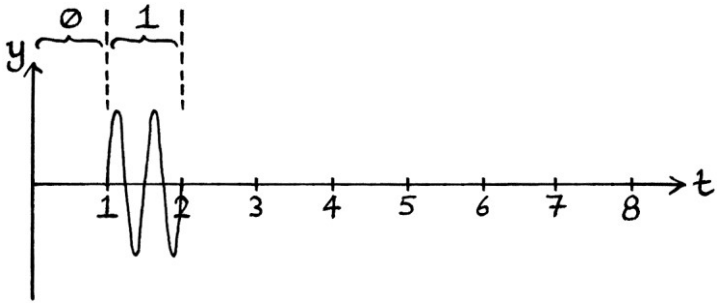
If you think too much or too little about waves, it is easy to believe, mistakenly, that the time-based wave graphs should show the binary digits in reverse order. This comes from forgetting what the wave graphs are showing. The graphs show the instantaneous amplitude at particular times since the start (or the starting event). At t = 0 on the graph, we see the instantaneous amplitude at the moment when we started recording the information shown on the graph. No matter how much time passes, the shape of the graph at t = 0 will stay the same. Similarly, once the time has passed 0.1 seconds, the shape of the graph at t = 0.1 seconds will always stay the same. The graph curve is *extended* down the time axis, as opposed to *sliding* down the time axis. The time refers to the time since we started and the events that happened at those times. The time does not refer to “the number of seconds since now”. Although this might be obvious, it is a misunderstanding that is easy to make, especially when dealing with aperiodic signals as we do in this chapter. The mistake can arise from viewing time-based frequency domain graphs, in which the contents of the graph might be in forwards or reverse time order, and *can* slide up or down the graph as time progresses, depending on how they are laid out.

If you are suffering from any doubt about the layout of the wave graphs, this section will help remove any confusion. We will look at a wave graph while we send the binary digits 01110101 digit by digit. This time, we will number the time axis, and we will send one binary digit every second.

First, we send the digit “0”. Our graph looks like this:

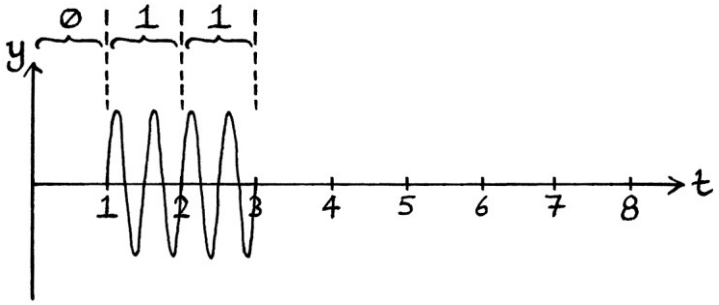


There is nothing for the time from t = 0 to t = 1. Next, we send the digit “1”, which we do by transmitting one second of a two-cycle-per-second wave. As we are doing this between one second and two seconds since we started, this appears between the times t = 1 and t = 2 on the graph:

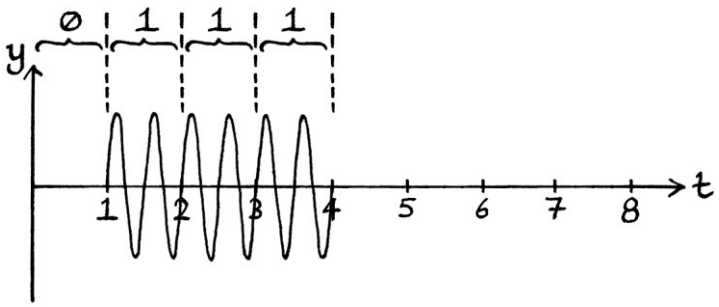


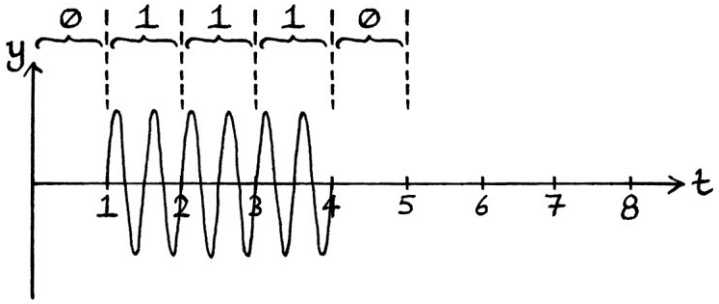
The gap we “sent” between t = 0 and t = 1 will stay forever in the same place on the graph because that part of the graph refers to the time up to one second from when we started sending the message. The one second of wave that we sent between the times of t = 1 and t = 2 will remain forever at that place on the graph because the time refers to the time since we started, and that cannot change.

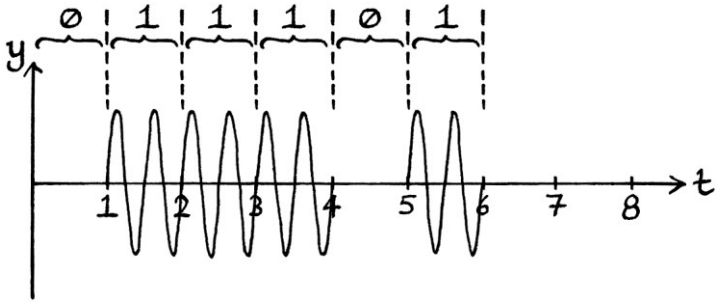
Next, we send another “1” digit, which we do by sending one second of a two-cycle-per-second wave. Our graph now looks like this:

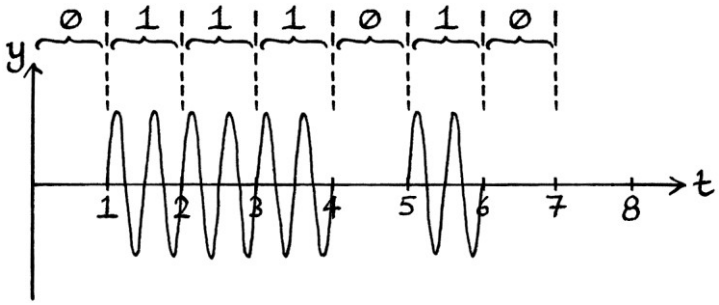


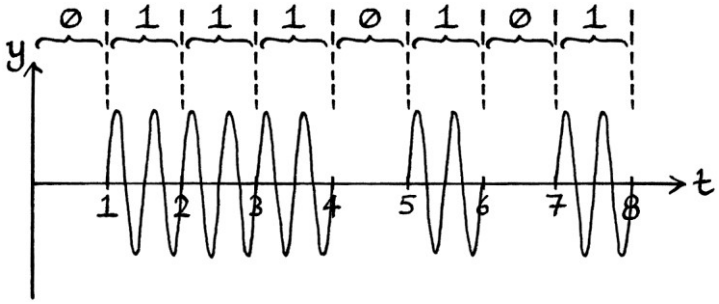
Everything that has happened at a time since the start remains fixed time-wise because the time at which an event has happened relative to the starting time cannot change. The rest of the signal is sent as shown in the following pictures:











Although this all might seem very obvious, it is easy for the situation to stop being obvious, in which case, it pays to think back to what the time axis and the graph are really showing.

The speed of light

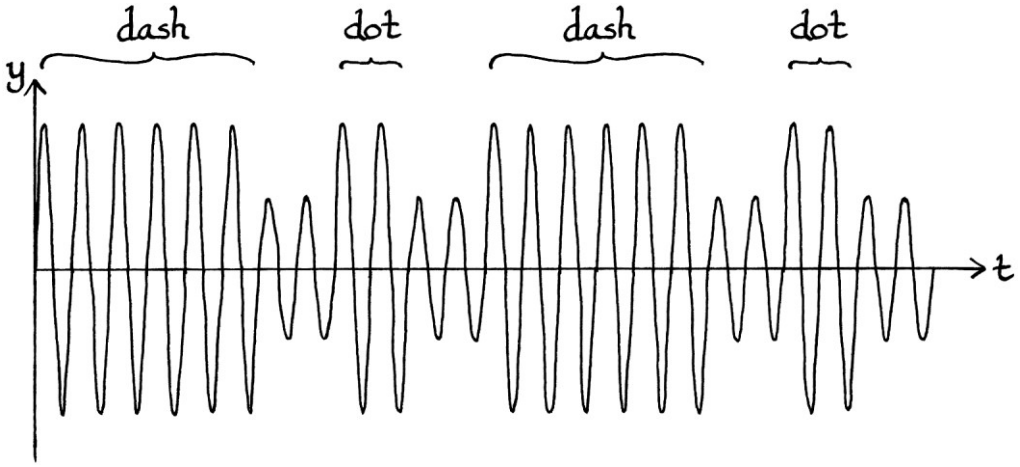
An interesting point about communication is made more obvious when thinking about on-off keying. It is common to hear people say that a radio signal travels at the speed of light, and from that they deduce that the message being conveyed travels at the speed of light. However, while the electromagnetic radiation in radio or light might travel at the speed of light, the message *as a whole* does not. In on-off keying, the speed of the message is dependent on the speed that a person or a machine can switch the signal on and off. If we can only switch the machine on and off once every second, then obviously, the message as a whole is not going to be sent particularly quickly. Even with the other forms of modulation discussed later in this chapter, the speed of the message is limited by how quickly the switching of an attribute can be done. It is true that each change in a state of the radio wave (such as whether it is changing from on to off, or from off to on) can be received after a delay based on the distance of the receiver and the speed of light. However, the *whole message* requires more than one change of state. This concept becomes even more obvious when you imagine someone using semaphore to pass a message at a distance using flags – the *change* in the letter of the alphabet being shown travels at the speed of light, but how quickly the changes occur depends on how quickly someone can move their arms.

Shift keying

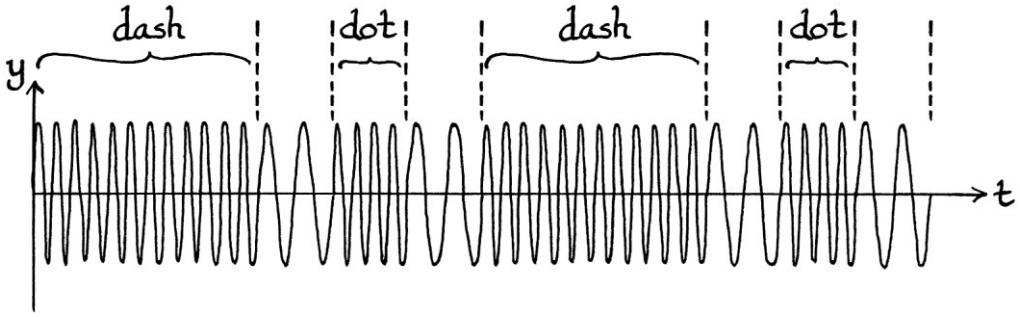
On-off keying used the “off” button of the transmitter. There are also buttons for amplitude, frequency and phase. These three buttons alter three of the four main attributes of a wave. [We will ignore mean level for now.] By changing one of the three attributes in an organised way, we can encode information into the wave.

It is possible to use the amplitude, frequency and phase buttons in the same way that we used the off button. For example, we could say that the amplitude being its normal value is analogous to the transmitter being off, and the amplitude being a higher value (when the button is pressed) is analogous to the transmitter being on. Therefore, the “one tap for yes; two taps for no” system would be executed as one press of the higher amplitude button for “yes” and two presses for “no”. We could also do one press of the higher frequency button or the different phase button to mean “yes” and two presses to mean “no”. Each press of a button, or each change in the amplitude, frequency or phase makes a noticeable change to the state of the wave, and is, therefore, one that can be interpreted as having a meaning.

We could also do the “a is 1 press; b is 2 presses; c is 3 presses” method, the letter grid method, and even the Morse code method using the amplitude, frequency or phase buttons. Using the amplitude button to send the letter “c” in Morse code would look like this:



Using the frequency button to spell out the letter “c” in Morse code would look like this:



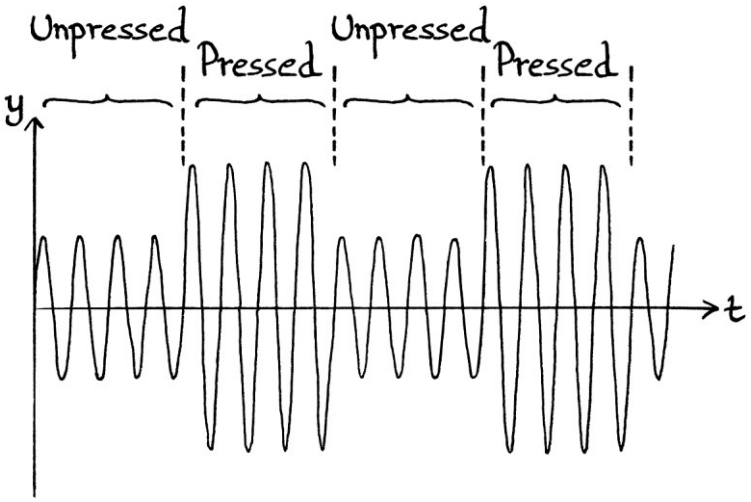
In the above picture, the dashes and dots are distinguishable from the gaps between them by the way the frequency is twice as fast during a dot or a dash.

All of these methods are known as “shift keying” because an attribute of the wave (amplitude, frequency or phase) is *shifted* to indicate a change. The “keying” part of the term comes from the “key” (as in lever) that was originally used in manual on-off keying. Usually, when people refer to shift keying, they are thinking of using the shifts to encode and transmit binary digits. The amplitude, frequency and phase buttons on our transmitter are just as suited to transmitting binary digits as the off button. We will ignore the above methods of shift keying, and we will focus on using shift keying just for transmitting binary numbers.

Amplitude shift keying

Amplitude shift keying on our transmitter

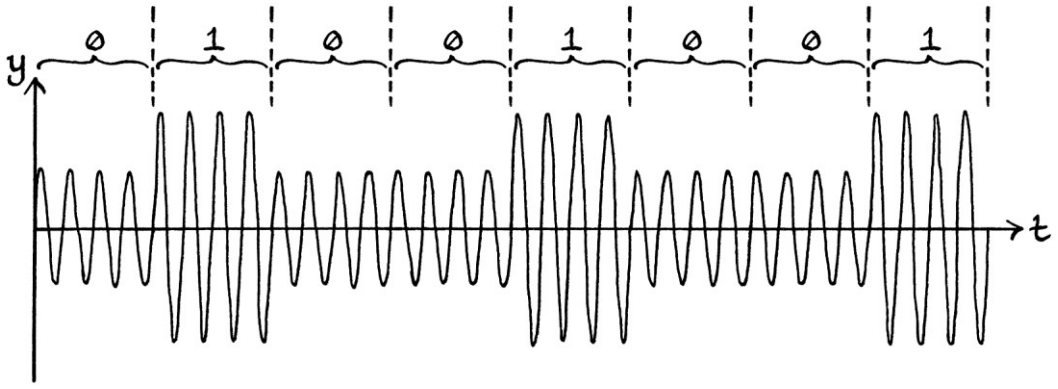
For our transmitter, we will use the button that doubles the amplitude to send binary messages using shift keying. While the button is not pressed and the wave’s amplitude is at its normal amount (1 unit) for one second, we will be indicating a zero. When the button is pressed for one second, thus raising the amplitude to 2 units, we will be indicating a one. This system is called “amplitude shift keying” or “ASK” for short. Amplitude shift keying does not need specifically one second for each one or zero – it can use any length of time or any number of cycles, as long as it is consistent for the whole message being sent, and as long as the signal can be easily decoded.



Amplitude shift keying is analogous to someone repeatedly pressing one key on a piano, but pressing it harder to make a louder sound when they want to indicate a 1, and pressing it more softly to indicate a 0.

To indicate that we are going to start a message, and to indicate the timing, we could send a 101010... preamble, but to make things simpler, we will ignore the preamble for now.

If we want to send the uppercase letter "I" using amplitude shift keying, we would first look up the ASCII encoding for that letter in binary: 01001001. We would then send each digit of this number in turn, from left to right, by pressing the "raise amplitude" button for one second to indicate a 1, and leaving the "raise amplitude" button unpressed for one second to indicate a 0. This has to be done with perfect timing or the message will be corrupted.



Sidebands

The frequency and phase of the wave *formula* remain untouched during the amplitude shift keying. However, it is worth noting that the signal as a whole is not a pure wave or a periodic wave. There is no pure wave that looks like this entire signal. As I explained in part one of this book, when a signal is not a pure wave, it is, instead, the sum of two or more pure waves of various amplitudes, frequencies and phases. Therefore, the above signal as *a whole* is really the sum of waves of different amplitudes, frequencies and phases. Furthermore, as this signal is aperiodic, we could also say that it is the sum of *pieces* of pure waves of various amplitudes, frequencies, and phases.

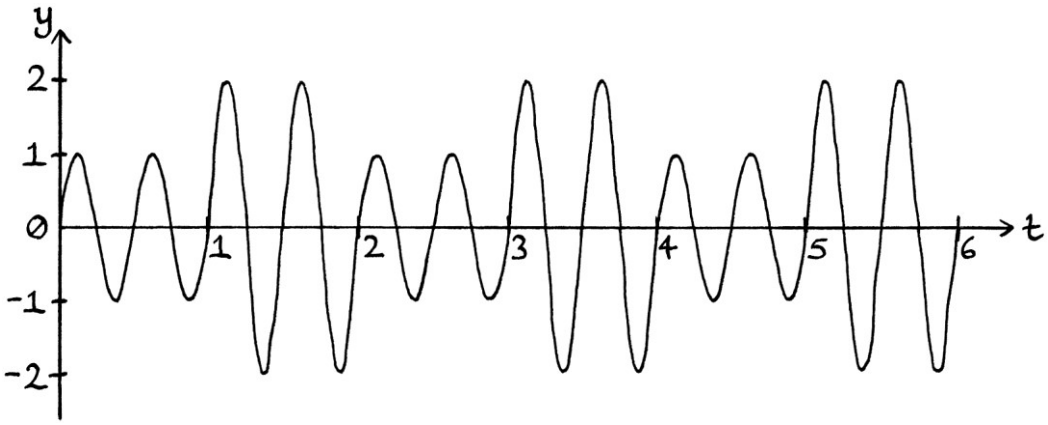
The fact that the signal is the sum of pieces of waves of various amplitudes, frequencies and phases means that the signal will take up more of the frequency spectrum than if it were just a pure wave. An unavoidable consequence of all shift keying, and in fact, of all modulation, is that when a pure wave is altered to carry information, it stops being a pure wave. Therefore, it takes up more of the frequency spectrum. Careless use of modulation can cause a signal to interfere with broadcasts on nearby frequencies. The extra frequencies existing from modulating a pure wave are called “sidebands” [because when viewed on a frequency domain graph, they appear as bands of frequencies to either side of the main frequency]. Sidebands are unavoidable with any modulation because any temporary alteration of an existing pure wave stops it being a single pure wave. Strictly speaking, the sidebands are the message that is being sent. If there were no sidebands, there would just be the pure wave and, therefore, no message. The extra frequencies will be visible on a time-based frequency domain graph at the time surrounding the change.

[From a theoretical and pedantic point of view, unless a wave continues forever, and has continued forever, it is not a pure wave because no pure wave ever stops or starts. In this way of thinking, there can be no real-world pure waves, and every real-world wave is actually a signal consisting of more than one frequency. In nearly all of the theoretical study of waves, and in practice, this idea can be ignored.]

We can analyse a signal to see the sidebands. Because a typically modulated signal would not be a periodic signal, we would not be able to analyse it using Fourier series analysis (as discussed in Chapter 18). Instead, we would need the more complicated Fourier Transform. If there were sufficient cycles per binary digit, Fourier series analysis would still give us a reasonable idea of the message if we analysed the signal section by section. Fourier series analysis would also work

perfectly if we had a message that, by chance, happened to be a periodic signal. For example, if we send the binary digits: "1010101010... ", the signal would be periodic.

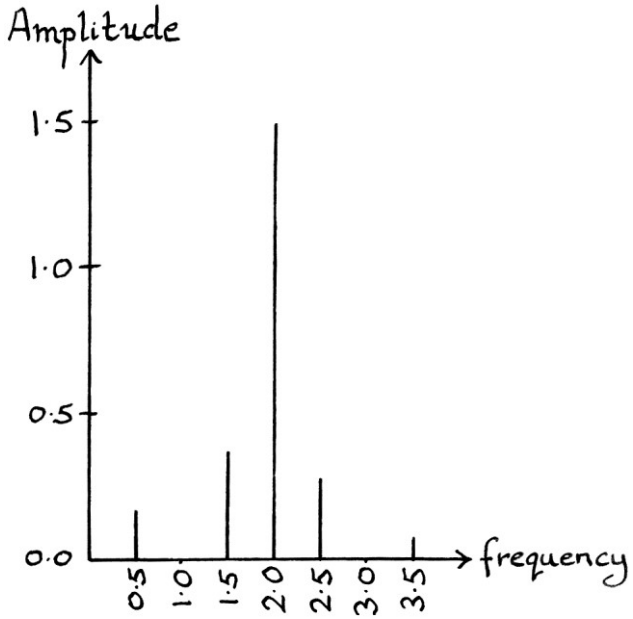
As a simple example of sending "1010101010...", we will look at the following signal, which changes from being:
"y = 1 sin (360 * 2t)"
... to:
"y = 2 sin (360 * 2t)"
... and back again.



The signal as a whole has a frequency of 0.5 cycles per second. Therefore, the constituent frequencies will be multiples of 0.5 cycles per second.

Fourier series analysis would reveal that the constituent waves are as follows (ignoring amplitudes less than 0.05 units):
"y = 0.16977 sin ((360 * 0.5t) + 270)"
"y = 0.36378 sin ((360 * 1.5t) + 270)"
"y = 1.5 sin (360 * 2t)"
"y = 0.28294 sin ((360 * 2.5t) + 90)"
"y = 0.07717 sin ((360 * 3.5t) + 90)"

The frequency domain graph, with the y-axis as amplitude, looks like this:



This frequency domain graph shows the waves that when added would create the amplitude-shifted signal that represents 0101010101... . This is a simple example of the sidebands resulting from amplitude shift keying. In practice, the signal would seldom be periodic, and the sidebands would vary in frequency and amplitude over time.

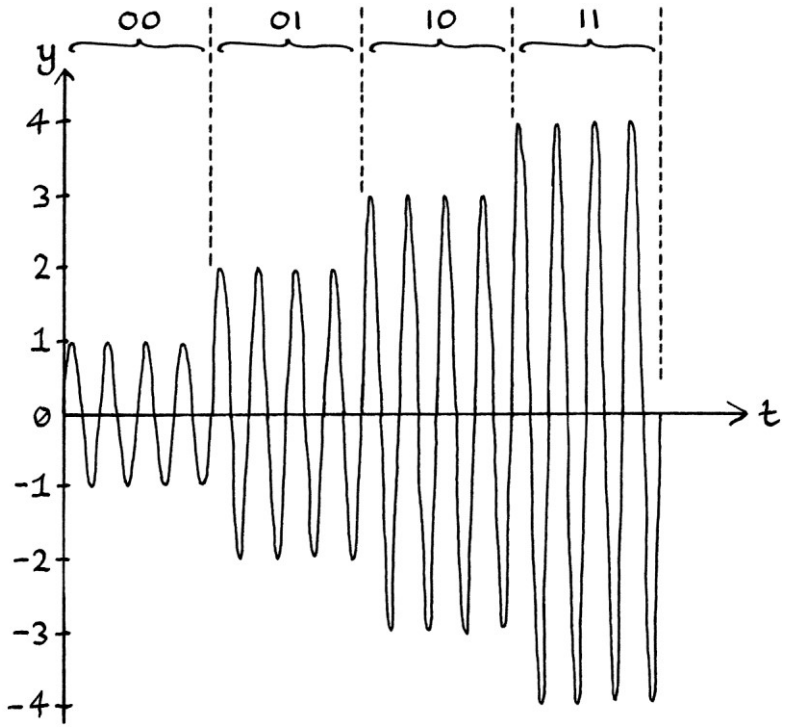
Amplitude shift keying in practice

Amplitude shift keying has similarities to on-off keying, and strictly speaking, on-off keying is a form of amplitude shift keying where the amplitude switches between a zero amplitude and a non-zero amplitude.

The particular form of amplitude shift keying used in the transmitter example is called “binary amplitude shift keying” because there are two states of amplitude used. “Binary” in this case refers to how there are two states, and not to how we are encoding digits from the base 2 counting system called “binary”. We happen to be sending binary digits, but the “binary” in “binary amplitude shift keying” refers to how there are two states. The term “binary amplitude shift keying” is abbreviated in several different ways, among which are “BASK”, “2-ASK”, “2ASK” and “ASK2”, where the 2 refers to the number of states used. In this book, I will use the abbreviation “2-ASK”.

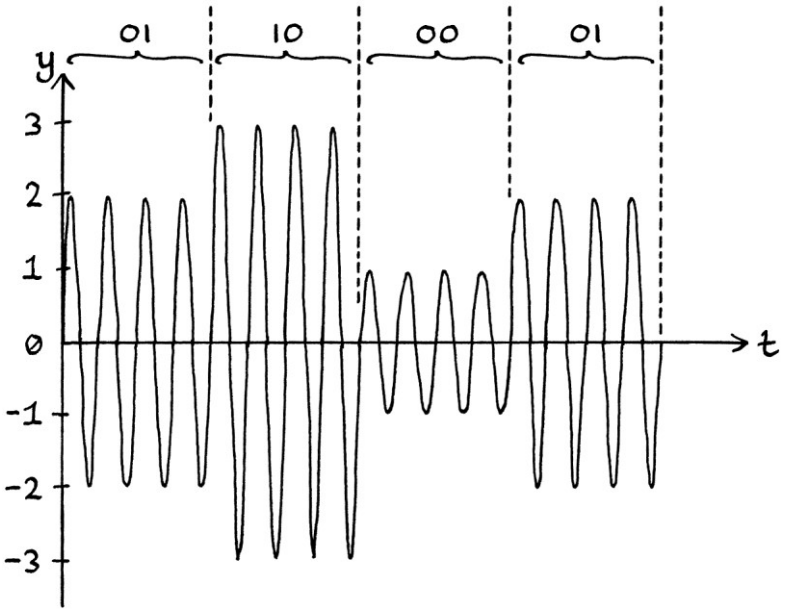
Amplitude shift keying is used in real-life radio communications in much the same form as in our transmitter example.

It is possible to have more complicated amplitude shift keying involving more than two different amplitudes, and therefore representing more than two different states. For example, instead of having 2 amplitudes representing 0 and 1, we could have 4 different amplitudes representing 00, 01, 10 and 11. We could have the default amplitude of 1 unit representing the binary digits “00”, an amplitude of 2 units representing the binary digits “01”, an amplitude of 3 units representing the binary digits “10”, and an amplitude of 4 units representing the binary digits “11”



Such a system can be called “4-state amplitude shift keying” or “quaternary amplitude shift keying”, where “quaternary” is an unnecessarily complicated way of saying “four”. [The word “quaternary” is the “four” equivalent of the word “binary”.] The abbreviations that you will see for this include “4-ASK”, “ASK4” and “QASK”. Sometimes, the “Q” is considered as standing for “quad” instead of “quaternary”. Personally, I think it is better and easier to use numbers instead of adjectives to distinguish the types of shifting. Therefore, I will refer to this type of ASK as “4-ASK”.

4-ASK allows us to send a binary message in half the time of normal ASK because each different amplitude level represents two digits. For example, the letter “a”, which is the binary number 01100001 in ASCII, would be sent as “01”, then “10”, then “00”, then “01”. Sending the whole 8 digit binary number would require only 4 changes in amplitude instead of 8.



[If our binary number did not contain an even number of digits, we would have to add an extra zero at the end, and hope that the person receiving the message knew that the extra zero was just irrelevant “padding”].

With 2-ASK, we would have had to send the message as “0”, “1”, “0”, “1”, “0”, “0”, “0”, “1”, which obviously takes twice as long.

We could increase the number of amplitudes used to 8, 16, 32 or even more. These would be called 8-ASK, 16-ASK, 32-ASK and so on.

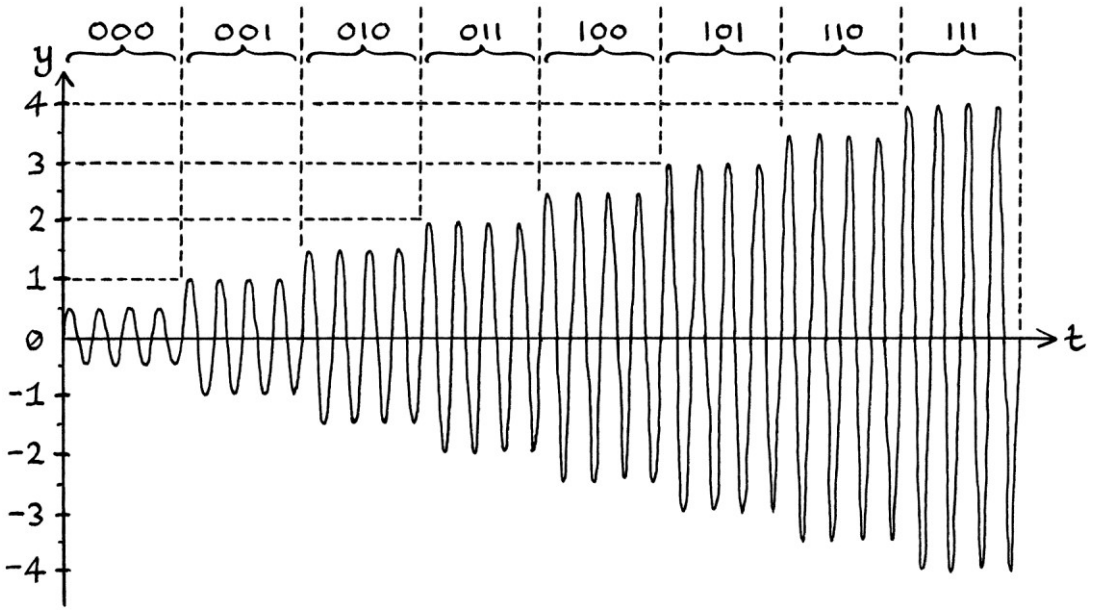
8-ASK would let us send three digits at a time: 000, 001, 010, 011, 100, 101, 110, 111.

We could assign meanings to each amplitude according to this table:

Amplitude	**Bits represented by this amplitude existing for 1 second**
0.5 units	000
1.0 units	001
1.5 units	010
2.0 units	011
2.5 units	100
3.0 units	101
3.5 units	110
4.0 units	111

The choice of which amplitude is used to represent which three bits is arbitrary. I am choosing these particular amplitudes because they fit nicely in a drawing. We could use any amplitudes in any order as long as they are different and distinguishable, and the receiver knows what each one represents.

In this system of 8-ASK, the binary numbers from 000 to 111 appear as so in the form of a signal:



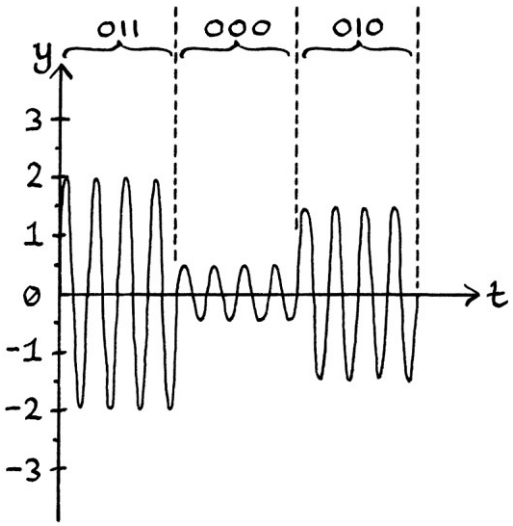
If we wanted to send the binary number for “a” (01100001”), we would have to send it as three separate changes of state, each containing 3-bits. This means that we would have to add an extra digit at the end of our binary number to make it a multiple of 3. We could use a one or a zero as long as the receiver of the message knew to ignore it. It is more common to pad out numbers with zeroes. In this

example, we will use a zero. The digits would become 011000010. We would send the following three-bit numbers:
011
000
010

These would require these amplitudes:
2 units
0.5 units
1.5 units
... all broadcast for one second each.

[In practice, we would not *have* to send them for one second each. We could choose any number of seconds or cycles as long as we were consistent – the time that each amplitude stays at the same level to indicate a number has to be the same. Another point is that the receiver must be aware of the timing and have the ability to decode the signal at that rate. If we were decoding the signal by hand or expecting interference, we would need to use longer times; if we were decoding the signal with electronics, and not expecting interference, we could use shorter times.]

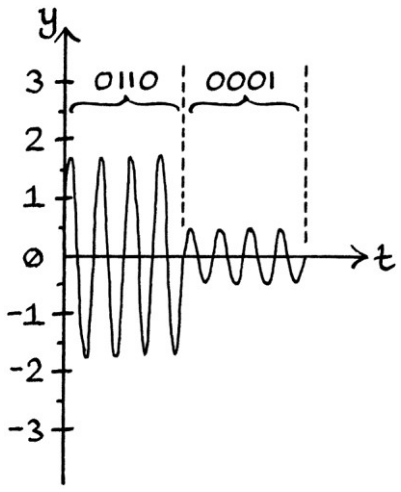
A signal encoding the above binary numbers using our particular system of 8-ASK looks like this:



16-ASK would allow sending four digits at a time: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111. A table showing one way of assigning amplitudes to these digits is as so:

Amplitude	**Bits represented by this amplitude existing for 1 second**
0.25 units	0000
0.50 units	0001
0.75 units	0010
1.00 units	0011
1.25 units	0100
1.50 units	0101
1.75 units	0110
2.00 units	0111
2.25 units	1000
2.50 units	1001
2.75 units	1010
3.00 units	1011
3.25 units	1100
3.50 units	1101
3.75 units	1110
4.00 units	1111

With 16-ASK, the letter “a” could be sent in just two parts: “0110” as an amplitude of 1.75 units for one second, and “0001” as an amplitude of 0.5 units for one second.

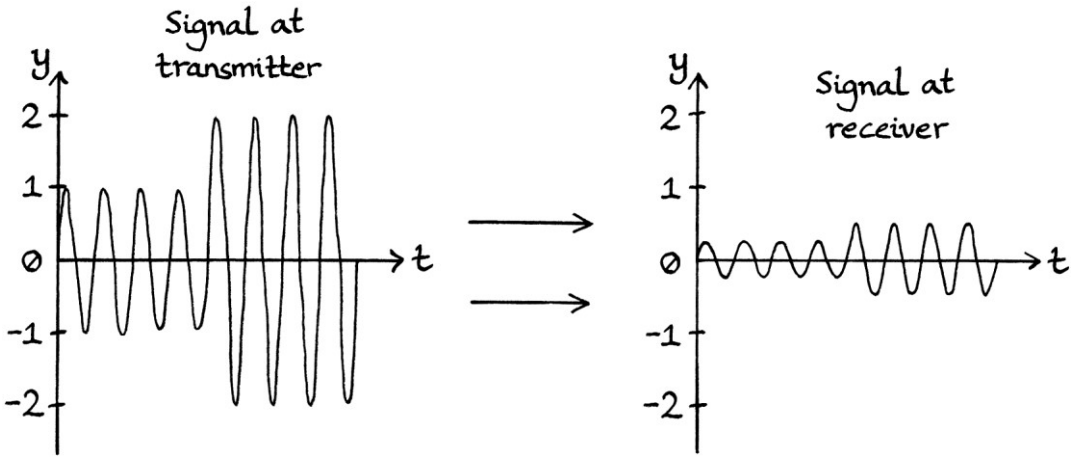


32-ASK would allow sending five digits at a time: 00000, 00001, 00010 and so on. If we used 256-ASK, we could send 8 digits at a time: 00000000, 00000001, 00000010 and so on. This would let us send the whole binary code for any letter of the alphabet in just one go.

The limits of ASK

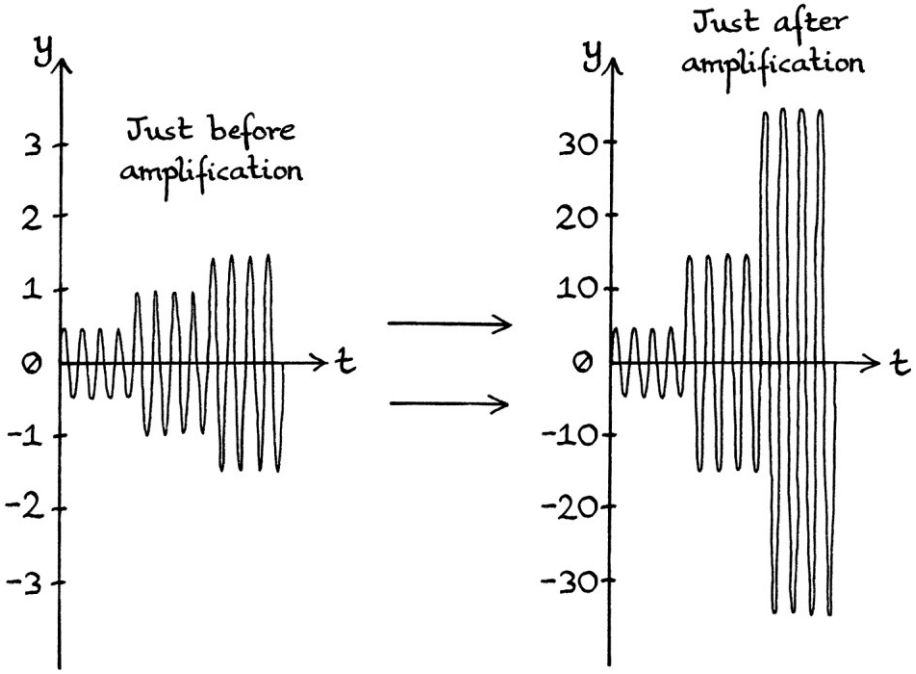
Although 256-ASK seems to save a lot of effort by sending binary digits as a large group all at the same time, in practice, it is not such a good idea. The trouble with having many different amplitudes is that it becomes much harder to distinguish between them. With 2-ASK, the difference between the higher amplitude and the lower one is obvious. With something such as 16-ASK or higher, it requires much more care to distinguish between the amplitudes. Not only is it harder to distinguish between them, but also any misreadings will result in errors that are more significant because more data is being sent in one go. With 2-ASK, a misreading will mean that one bit in the received message is incorrect; with 256-ASK, a misreading will mean that eight bits in the received message are incorrect.

The difficulty of distinguishing amplitudes becomes even more of a problem because of the way that the amplitude of a radio signal can be affected. The amplitude of a radio signal reduces as it moves further away from the source. We cannot say that a particular received instantaneous amplitude represents a particular value because when the receiver detects the signal, the amplitudes will have reduced in size. Therefore, a receiver actually needs to distinguish between the *relative* amplitudes. The received amplitude levels become smaller and smaller as they move further away from the transmitter, and the differences between them become harder and harder to distinguish. In practice, it would require a lot of accuracy to make 256-ASK work well in a radio signal.



Another problem is that radio waves can be subject to interference, and interference affects the amplitude of a signal more than the frequency or the phase.

Yet another problem is how the transmission of a radio signal requires an amplifier just before the antenna in order to increase the power so that the signal can travel a long way. Amplifiers do not necessarily increase the amplitude in a linear way – in other words, higher amplitudes might be amplified more than lower amplitudes, thus "stretching" the signal. This would make the receiver's job of distinguishing between amplitudes harder.



We could partially solve some of these problems by sending a preamble message indicating every possible amplitude value in turn. This would let the person receiving the message calibrate their equipment for detecting the correct levels of amplitude. In practice, it is easier to use the other forms of shift keying instead.

2-ASK is fine to send messages, but ASK becomes less reliable as more amplitude levels are brought into play.

Padding

With shift keying, we always need the number of bits that we send to be equal to, or a multiple of, the number of bits in a group. If we are sending groups of 8 bits, but we have a number that contains fewer than 8 bits, then we have to decide what to do about the absent bits. One solution is to put extra zero bits at the end. The extra zeroes are called "padding". For example, for an 8-bit group size, if we just wanted to send the number:
111
... then we could place zeroes at the end to make it the correct length:
1110000

Padding allows us to send a smaller number of bits than the group size, but with the risk that the padding be misinterpreted by the receiver as significant. To avoid the recipient thinking that the padding is part of the message, we could send the size of the entire message as binary bits before the actual data, or have a prior arrangement that the important part of the message will always be the same size.

Maths

We can calculate how many different states of amplitude are needed to send a particular number of bits by solving 2^x where "x" is the number of bits we want to send in one go. For example, if we want to send 4 bits in one go, we calculate:
$2^4 = 16$

This means that we would need 16 different amplitudes to send 4 bits in one go. We would need 16-ASK.

To calculate the number of bits that can be portrayed if we have a given number of different amplitudes, we need to calculate $\log_2 z$ [the base 2 logarithm of "z"], where "z" is the number of different amplitudes being used. In doing this, we are really solving $2^x = z$, where we know "z", but we do not know "x". For example, if there are 1024 different amplitudes, we can portray:
$\log_2 1024 = 10$ bits in one go.

Symbols

If we used 256-ASK, with one second for each particular amplitude state, we would be sending 8 bits of a binary number in one go. Although we are seeing only one change in amplitude, there are 8 bits of information being communicated in that one state of amplitude. We could say that each single amplitude level represents 8 bits of information.

If we used 2-ASK, then one second of a particular amplitude level would represent just 1 bit.

Each block of information (i.e. each group of bits) that is sent in one state of amplitude is called a "symbol". In other words, with 256-ASK, a symbol is 8 bits long because one amplitude state existing for a certain length of time represents 8 bits. With 2-ASK, a symbol is 1 bit long because a particular amplitude existing for a certain length of time represents just one bit. A symbol is one group of bits that is sent in one go.

We need to have the term "symbol" so we can distinguish the number of bits (ones or zeroes) being sent per second from the number of "groups of bits" being sent per second. If we are using 256-ASK, and change the amplitude once every second, then we are sending 8 bits per second, but just 1 symbol per second. If we are using 2-ASK, and change the amplitude once every second, then we are sending 1 bit per second, and also 1 symbol per second.

For the examples that we have looked at so far, "symbols per second" is really the same as "state changes per second" or, with ASK, "amplitude changes per second". However, it is possible to have a system that requires more than one state change to indicate a single piece of information, so symbols per second are not necessarily the same as state changes per second. [We will see this with Manchester encoding later on in this chapter.]

From all of the above, we can see that knowing how many symbols are sent per second does not tell us how many bits are sent per second. We have to know how many bits are within a symbol to know this.

The number of symbols sent per second is often called "baud", or sometimes, the "baud rate", where "baud" is another term for "symbols per second". The term "Baud" is named after the French Engineer Jean-Maurice-Émile Baudot. In a similar way to how "cycles per second" is more descriptive than "hertz", "symbols per second" is more descriptive than "baud", and, personally, I think is a better term to

use. “Symbols per second” is often abbreviated to “sps”. Be careful not to confuse this “sps” with the “sps” standing for “*samples* per second”. The context will usually indicate which is meant.

If someone says that a piece of equipment can send, say, 1000 symbols per second, it does not mean anything unless you know how many bits are in a symbol for that piece of equipment.

As a summary of this section:

- A bit is one unit of information that is either 0 or 1. It is a digit in the binary counting system.

- A symbol is a group of bits that is sent together as one unit and in one go. In the examples so far, a symbol has been sent in a single state change of a signal.

- “Bits per second” refers to how many bits are sent each second.

- “Symbols per second” refers to how many *groups* of bits are sent each second (with no mention of how many bits are in a group).

M-ary

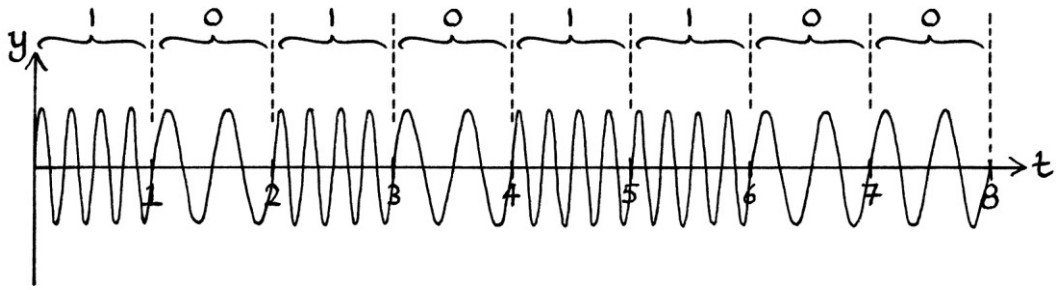
As we have seen, if we have two-state amplitude shift keying, it can be called “*binary* amplitude shift keying”. If we have four-state amplitude shift keying, it can be called “*quaternary* amplitude shift keying”. The words “binary” and “quaternary” both end in the suffix “-ary”. You will frequently see people use the word “m-ary” to refer to shift keying in a general way where there might be any number of states. [It is pronounced “em-aree”.] Using the term “m-ary” is a shorthand way of saying “shift keying where there are ‘m’ different states”, where “m” refers to a number that is either unknown or its exact value is unimportant to the explanation. Someone might say, “m-ary ASK.” A possibly better alternative would be to use the term “x-ASK”, where “x” refers to any number. Saying “x-ASK” would save time compared with saying “amplitude shift keying using any number of states”, or saying the list, “2-ASK, 4-ASK, 8-ASK, 16-ASK etc.” Generally, you will see people use the term “m-ary ASK” and not “x-ASK”.

Frequency shift keying

Frequency shift keying with our transmitter

To make the pictures clearer, in this section, we will say that our default wave has a frequency of 2 cycles per second and that each binary digit exists for one second. This means that a binary zero will be represented by 2 cycles over one second, and a binary one will be represented by 4 cycles over one second. [We have altered how the transmitter works to make the examples clearer – with amplitude shift keying, the default frequency was 4 cycles per second.]

In a similar way to how we can send messages with our transmitter using the "increase amplitude" button, so can we send messages with the "increase frequency" button. To do this, we double the frequency to indicate a one and leave it as normal to indicate a zero. To send the binary number:
10101100
... we would send this signal:



Note that the amplitude remains the same throughout.

Sending information in this way is called "frequency shift keying" or "FSK" for short. Frequency shift keying can be a better way of sending a message than amplitude shift keying because the frequency of a signal is much less likely to be altered along the signal's journey than the amplitude.

Frequency shift keying is analogous to someone pressing two consecutive keys on a piano with the same pressure. They might press the higher pitched one (the one with the higher frequency) to indicate a one, and the lower pitched one to indicate a zero.

In the above wave graph, it is possible to see the variation in frequency by seeing where the peaks become closer together and further apart. However, it is much easier to see what is happening in a frequency domain graph.

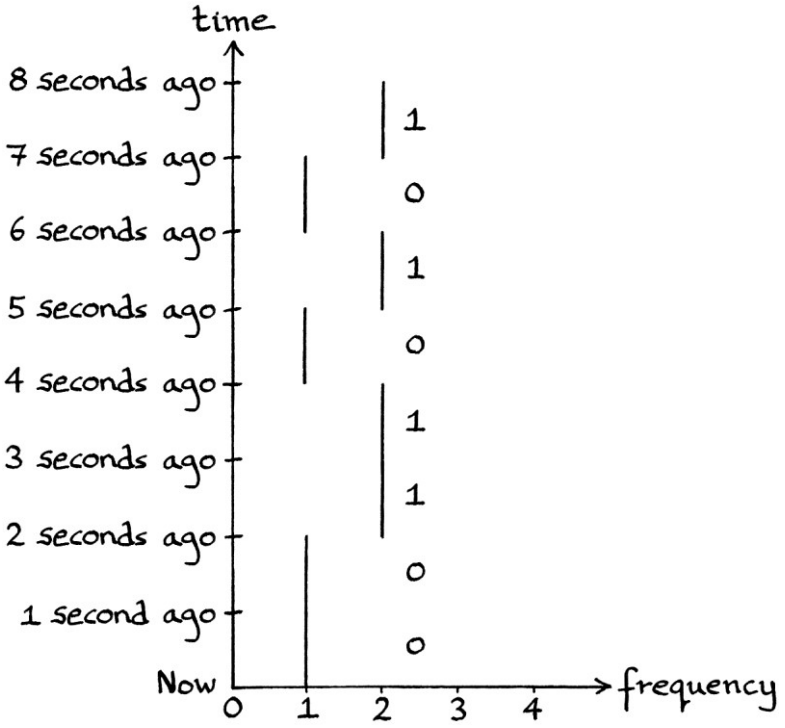
It pays to know that for frequency shift keying, there are really two types of frequency domain graphs:

- There is the graph that shows the frequencies that we *intend* to switch between, from the point of view of the formulas.

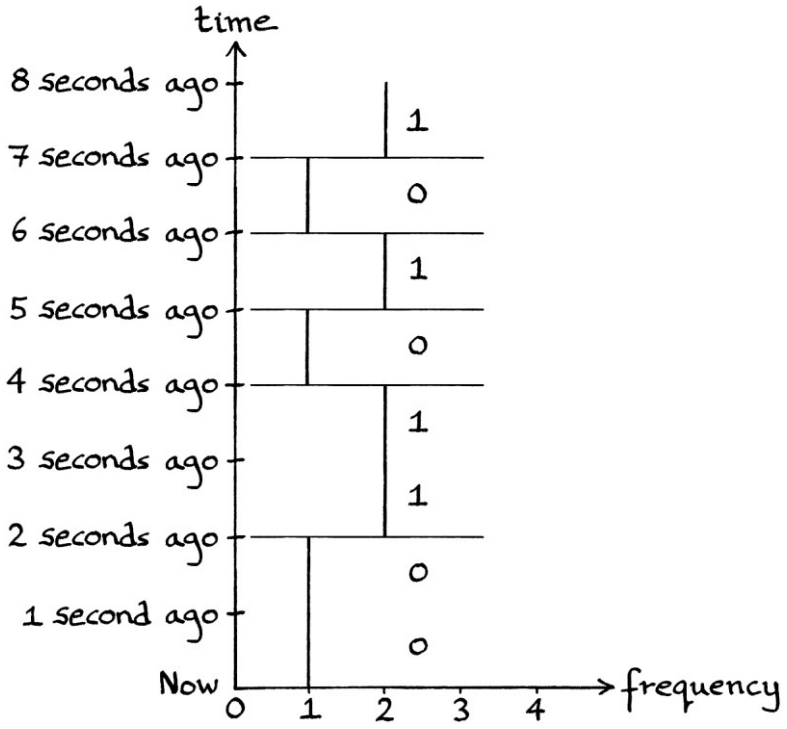
- There is the graph that shows the frequencies that actually existed in the signal. [Such a graph would be based on analysing the signal.]

We will call the first type of graph, the "intended frequency domain graph" and the second type of graph the "analysed frequency domain graph". Because no pure wave changes in frequency, a signal that switches between frequencies cannot be a pure wave. Therefore, such a signal must consist of two or more pure waves of different frequencies. In practice, for FSK, the signal will consist of extra side band waves around the moment of the switching. This means that a frequency domain graph based on analysing the signal will have more frequencies in it than a frequency domain graph showing just the frequencies we intended to switch between.

For our example above, our *intended* frequency domain graph appears as in the following picture. It shows the frequencies that we switch between over time. [The time axis is labelled in a similar way to how such a graph would appear on a computer screen if the signal were being shown in real time.] The ones and zeroes show the binary digit represented by the existence of each frequency.



As switching between frequencies creates sidebands, if we created a signal that involved switching between frequencies in the way explained in this example, the actual frequencies produced over time (and revealed by analysing the signal) *might* look something like this:

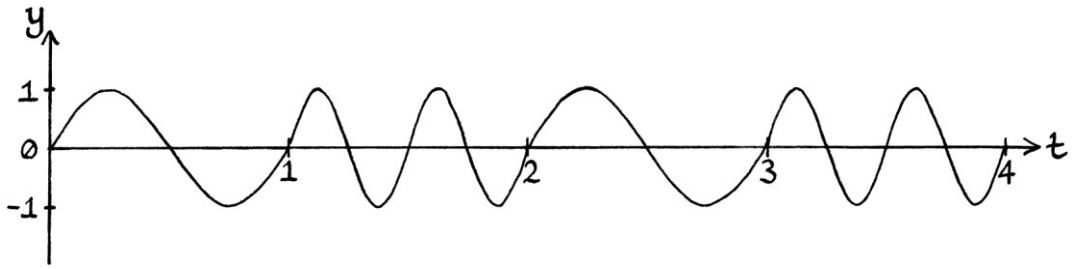


In the drawing, for the very brief moments when we change from 1 cycle per second to 2 cycles per second or back, there are multiple frequencies created. In practice, the side bands are unlikely to be as blatant as in the drawing. The number of frequencies existing in an *analysis* of a signal at any one moment depends on how the signal is being analysed (as well as the quality of the transmitter and receiver).

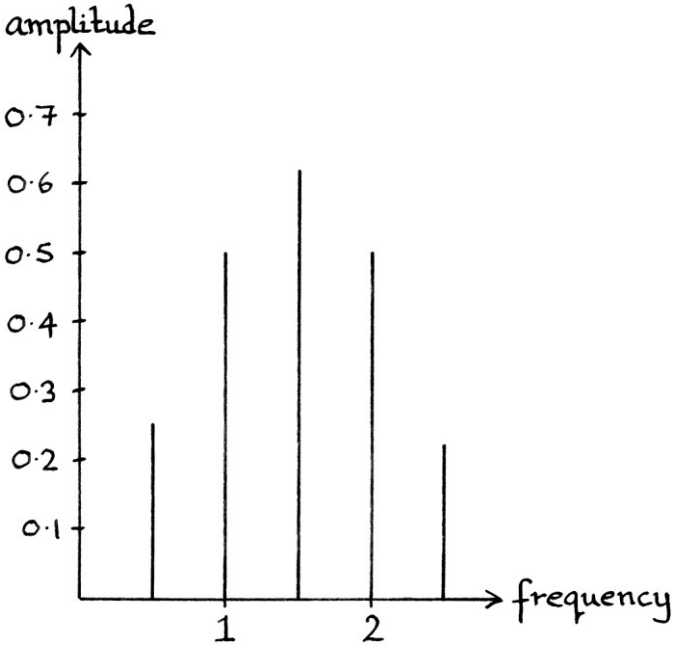
There is not a great deal of difference between the two graphs, but it pays to understand the difference between the frequencies that we want to change between and the actual effects of changing between those frequencies.

We will look at another example, but this time, a *periodic* frequency shift-keyed signal. We will send the digits 01010101010101... . The signal conveying this message will be periodic, which means that we can use the relatively simple Fourier series analysis on it to look for sidebands. We will use “y = sin 360t” for one second to indicate a zero, and “y = sin (360 * 2t)” for one second to indicate a one.

Our signal looks like this in the time domain:



It looks like this in an "analysed" frequency-domain graph with the y-axis as amplitude (as opposed to time). The waves with amplitudes less than 0.1 unit have been removed to make the graph clearer:



From the graph, we can see that our periodic switching between just two frequencies actually means that we are *constantly* transmitting 5 different waves. The five waves are:

"y = 0.25 sin ((360 * 0.5t) + 90)"
"y = 0.5 sin (360t)"
"y = 0.62 sin ((360 * 1.5t) + 270)"
"y = 0.5 sin (360 * 2t)"
"y = 0.22 sin ((360 * 2.5t) + 90)"

The waves we were switching between were:
"y = sin 360t"
"y = sin (360 * 2t)"

Frequency shift keying in practice

With our transmitter example, pressing the frequency button doubled the frequency. In practice, frequency shift keying would not require the frequency to double, and it would be more likely that the frequency would just be increased slightly instead. If the zero-indicating frequency were, say, 100 MHz, then the one-indicating frequency might be 100.01 MHz, or even 100.001 MHz. [In practice, frequency shift keying could not be performed legally around 100 MHz.] The difference in frequency used in FSK depends on where in the radio spectrum we are broadcasting, how much bandwidth we are allowed to use, how easily the signal can be received, and other factors.

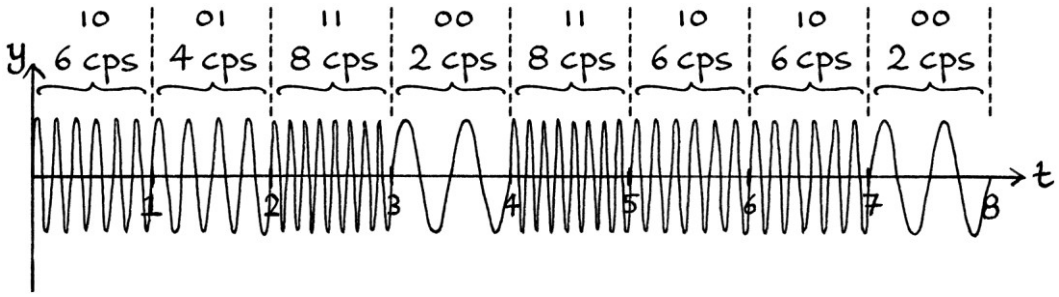
Many of the ideas relevant to amplitude shift keying are also relevant to frequency shift keying. It is possible to have more efficient frequency shift keying by using a higher number of frequencies to encode information. For example, instead of having two frequencies, we could have four frequencies. Each frequency would indicate 2 bits at the same time. We could have:

2 cycles per second for one second to indicate “00”
4 cycles per second for one second indicate “01”
6 cycles per second for one second to indicate “10”
8 cycles per second for one second to indicate “11”

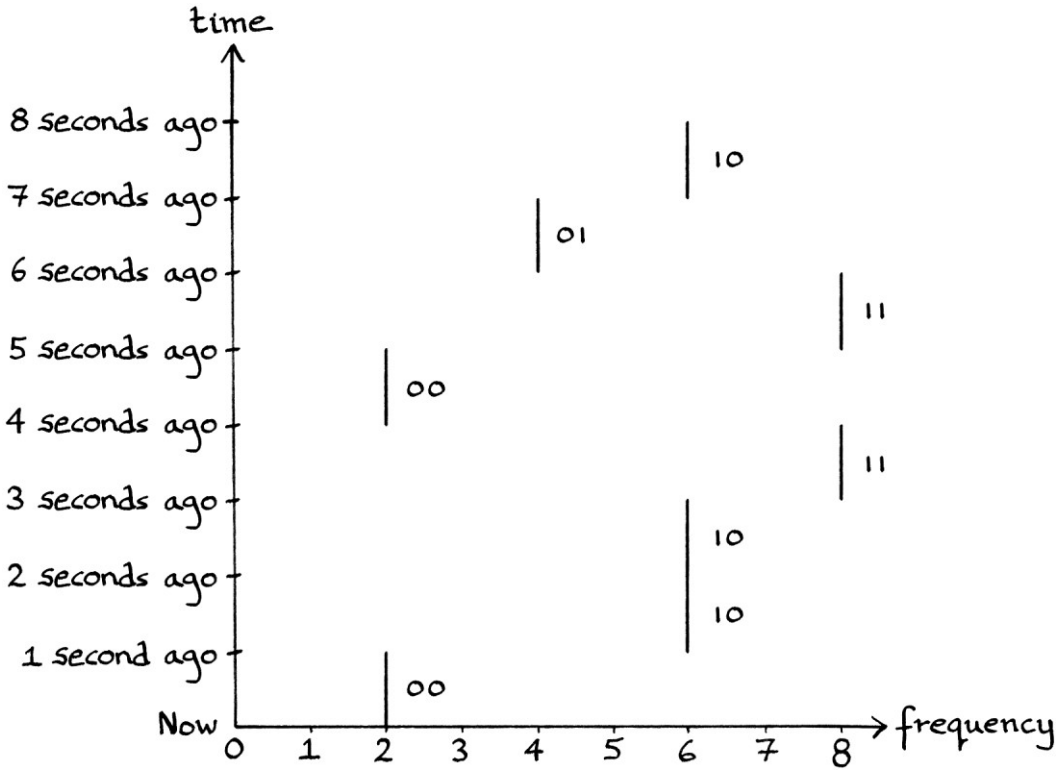
To transmit the binary number:
1001110011101000
... we would first split it into 2-bit sections:
10
01
11
00
11
10
10
00
... and then we would transmit the relevant frequencies to send this message.

We would transmit one second of each of the following waves in order:
6 cycles per second
4 cycles per second
8 cycles per second
2 cycle per second
8 cycles per second
6 cycles per second
6 cycles per second
2 cycle per second

The signal would look like this (where “cps” stands for “cycles per second):



When we use several different frequencies in a signal, it becomes harder to tell visually what they are. It is easiest to understand the characteristics of the signal we are sending in an “intended frequency” frequency domain graph:



This system is called 4-FSK because there are four states, or four frequencies, used. There are 2 bits in each state. As we are sending the message with one frequency change per second, the symbol rate is 1 symbol per second. The bit rate is 2 bits per second. [Note how the frequencies of the wave sections are not related to the symbol rate or the bit rate. It is the change from one frequency to another that is important in sending the message, and not the frequencies that are used. If our system switched between 40 cycles per second, 60 cycles per second, 1,000 cycles per second and a million cycles per second, but still changed the frequency once every second, the symbol rate would still be 1 symbol per second and the bit rate would still be 2 bits per second.]

If we used 256-FSK, we could send the entire binary number with just two state changes. If we were using amplitude shift keying instead of frequency shift keying, and using 256-ASK, we would have to be very careful in setting and observing the exact amplitudes because there is a limited amount of detectable differences in amplitude. When using 256-FSK, we are not as limited – even within one megahertz of the radio spectrum, it would be easy to distinguish between thousands of different frequencies. [One of the actual limits is how much of the spectrum we are legally allowed to use, and how much of *that* is not being used by other people. We definitely would not be allowed to use frequencies around 100 MHz legally for FSK because it would interfere with the FM radio broadcast band.]

If we were transmitting around 100 MHz, then an example of a possible arrangement for 256-FSK is as follows:

Frequency	**Meaning**
100.000 MHz	00000000
100.005 MHz	00000001
100.010 MHz	00000010
100.015 MHz	00000011
100.020 MHz	00000100
100.025 MHz	00000101
100.030 MHz	00000110
100.035 MHz	00000111
100.040 MHz	00001000
100.045 MHz	00001001
100.050 MHz	00001010
100.055 MHz	00001011
100.060 MHz	00001100
100.065 MHz	00001101
100.070 MHz	00001110

100.075 MHz 00001111
... and skipping a bit...
101.275 MHz 11111110
101.280 MHz 11111111

The presence of any one frequency at a particular time represents a particular group of 8 bits. This means that we would need to show only two frequencies to transmit our entire binary number of “1001110011101000”. In practice, for such a signal, it would make sense to start with a preamble to indicate that the message was about to start. In this case, the preamble could switch between the frequencies at either end – in other words the frequencies for 00000000 and 11111111, which are 100.000 MHz and 101.280 MHz.

If we used 256-FSK with one frequency change per second, the symbol rate would be 1 symbol per second. The bit rate would be 8 bits per second. A symbol would contain 8 bits.

We could use FSK that had even higher numbers of states. For example, it would not be impractical to use 4096-FSK. In practice, instead of using frequency shift keying with many frequencies all spread out over a large bandwidth, it is more common for there to be fewer levels and for multiple FSK streams of data to be sent at the same time on adjoining frequencies. This is called Frequency Division Multiplexing.

MFSK

A variation of frequency shift keying is called “multiple frequency shift keying” or “MFSK”. This uses more than one frequency at the same time. In this way, it is possible to use fewer frequency states to send the same number of bits.

A simple example is as follows. We will have 4 different frequencies, but we can transmit one or more of the frequencies at any one time. Transmitting two or more frequencies at the same time is the same as adding them, and as we know from part one of this book, different frequencies do not mix – it will always be possible to distinguish them even after they have been added together. The four frequencies we will use are:
100.000 MHz
100.005 MHz
100.010 MHz
100.015 MHz

The possible different combinations of frequencies that we can transmit at any one time are:

- 100.000 MHz on its own
- 100.005 MHz on its own
- 100.010 MHz on its own
- 100.015 MHz on its own

- 100.000 MHz and 100.005 MHz at the same time
- 100.000 MHz and 100.010 MHz at the same time
- 100.000 MHz and 100.015 MHz at the same time

- 100.005 MHz and 100.010 MHz at the same time
- 100.005 MHz and 100.015 MHz at the same time

- 100.010 MHz and 100.015 MHz at the same time

- 100.000 MHz, 100.005 MHz and 100.010 MHz at the same time
- 100.000 MHz, 100.005 MHz and 100.015 MHz at the same time
- 100.000 MHz, 100.010 MHz and 100.015 MHz at the same time
- 100.005 MHz, 100.010 MHz and 100.015 MHz at the same time

This means that there are 15 possible combinations of frequencies, which means that we can distinguish between 15 different states. We could assign a pattern of bits to each combination of frequencies in this way:

- Combination 1 will represent 0001
- Combination 2 will represent 0010
- Combination 3 will represent 0011
- Combination 4 will represent 0100
- Combination 5 will represent 0101
- Combination 6 will represent 0110
- Combination 7 will represent 0111
- Combination 8 will represent 1000
- Combination 9 will represent 1001
- Combination 10 will represent 1010
- Combination 11 will represent 1011
- Combination 12 will represent 1100
- Combination 13 will represent 1101
- Combination 14 will represent 1110
- Combination 15 will represent 1111

There is a problem here in that we have no combination to indicate 0000. We could solve this in two ways:

- We could portray only the numbers 000 to 111 instead of 0000 to 1111. This would work, but would be a lost opportunity for all the combinations that we would not be using.
- We could portray 0000 by switching the signal off completely.

For this example, we will portray 0000 by switching the signal off for one second.

The above system of associating frequencies with 4-bit numbers will work, but we can improve it slightly. The nature of binary, and how we have 4 possible frequencies, means that we can have a more systematic way of assigning each combination to a binary number. This method involves using a table with the possible frequencies as the heading:

100.000 MHz 100.005 MHz 100.010 MHz 100.015 MHz

The presence of a frequency at any particular time will indicate a "1"; the absence of that frequency will indicate a "0". This layout of frequencies and binary digits is different from the one above, but easier to remember. The full table looks like this:

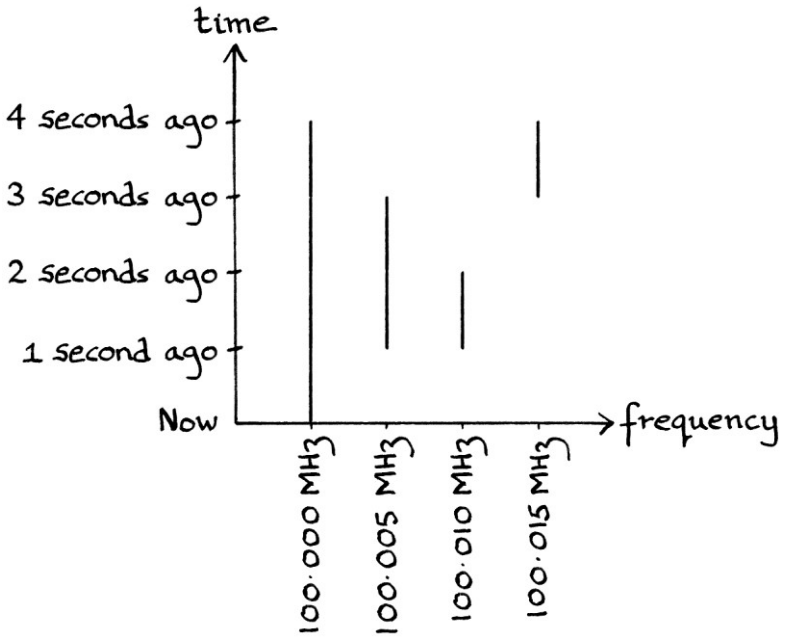
100.000 MHz	**100.005 MHz**	**100.010 MHz**	**100.015 MHz**
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

From this, we can encode 4 bits at a time using 4 frequencies, and, what is more, the frequencies as they appear in a frequency domain graph will be in order of the bits at that time. If we were using 4-FSK (as in using four frequencies, but only one at a time), we would be able to send only 2 bits at a time. With this version of what can be called 4-MFSK, we are sending 4 bits at a time. Although this is not a great difference, we can see how the idea lends itself to sending much more information if we used more frequencies.

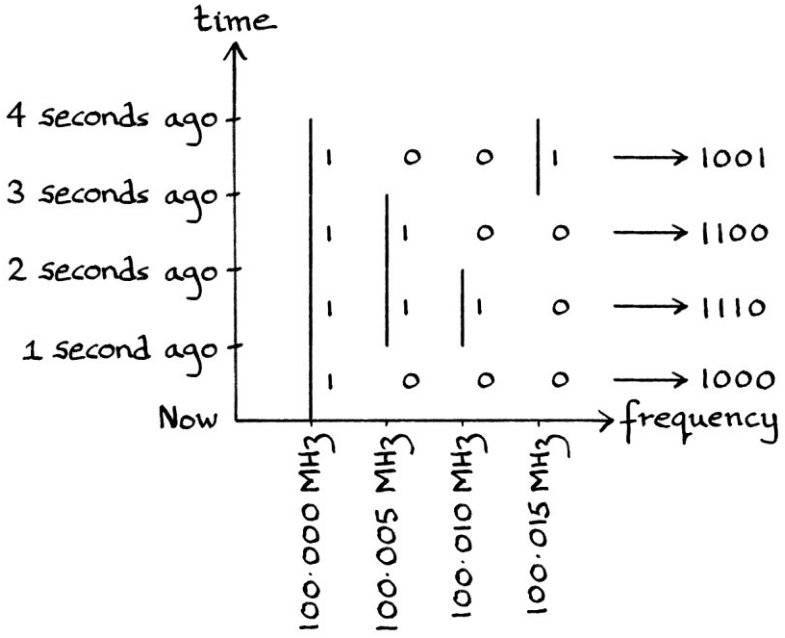
We will send the following binary number, which is the one we sent with 4-FSK earlier:
1001110011101000

First, we split it into 4-bit pieces
1001
1100
1110
1000

The sent 4-bit pieces appear like so in the “intended frequency” frequency domain graph:



We can mark the ones and zeroes that are represented by the existence or absence of the waves to make the meaning clearer.



The frequency domain graph itself becomes a look-up table for the binary digits being sent at any one moment. For any particular second, we can see which waves do or do not exist. If a wave exists, it represents a one; if it does not, its absence represents a zero. By observing all four frequencies at the same time, we can work out the 4 binary digits being represented at any one second.

As we are changing the frequency every second, in this system, we are sending one symbol per second or 4 bits per second. There are 4 bits in a symbol.

The signal as seen in the time domain would be made up of one-second sections of the added waves.

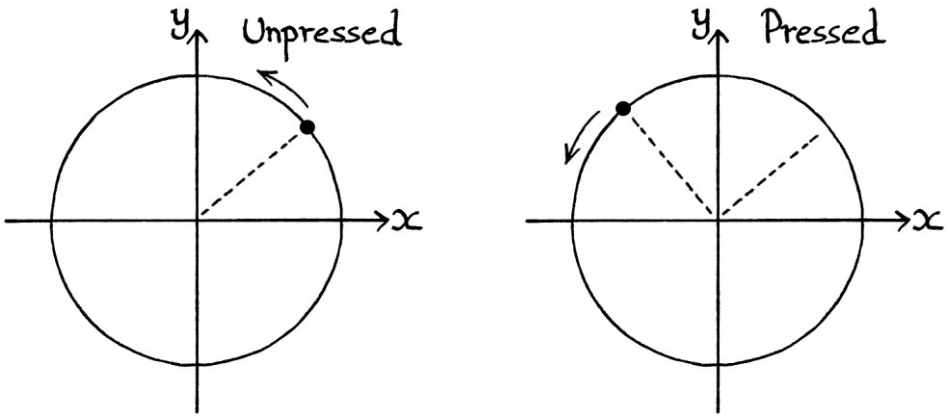
We can send more than one frequency at a time because different frequencies do not mix – it is possible to work out which frequencies were added together to make up a signal. It is worth noting that we could not send more than one *amplitude* of the same frequency at a time. What would be called "Multiple *Amplitude* Shift Keying" would not work. If we add two amplitudes together, it becomes impossible to work out which amplitudes were added. This is because different amplitudes *do* mix.

Phase shift keying

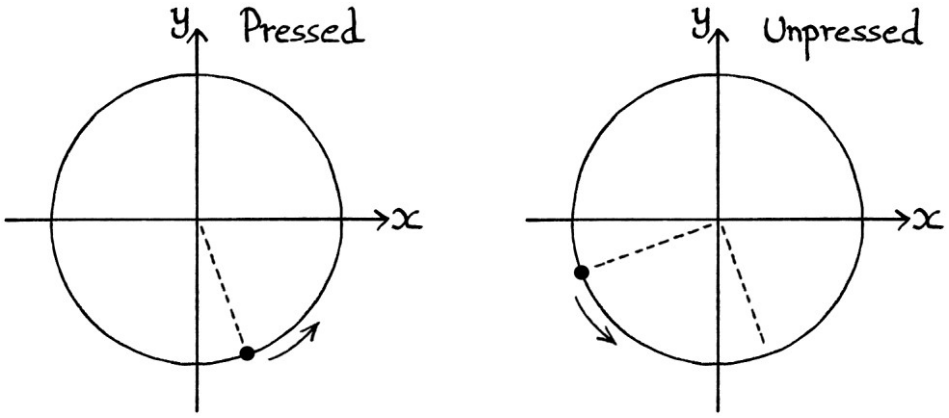
Phase shift keying on our transmitter

The final button on our transmitter increases the phase by 90 degrees while it is pressed down, and returns the phase to normal when it is released. This idea has similarities to amplitude shift keying or frequency shift keying, but is slightly harder to understand.

When it comes to the phase change from pressing the button, one way of thinking about it is to imagine that 90 degrees is added to the *instantaneous* phase for the time that the button is pressed. If we imagine the circle from which the wave could be said to be derived, we can think of the object that is rotating around the circle instantly jumping forwards 90 degrees when the button is pressed and continuing rotating at the same speed as it was before.



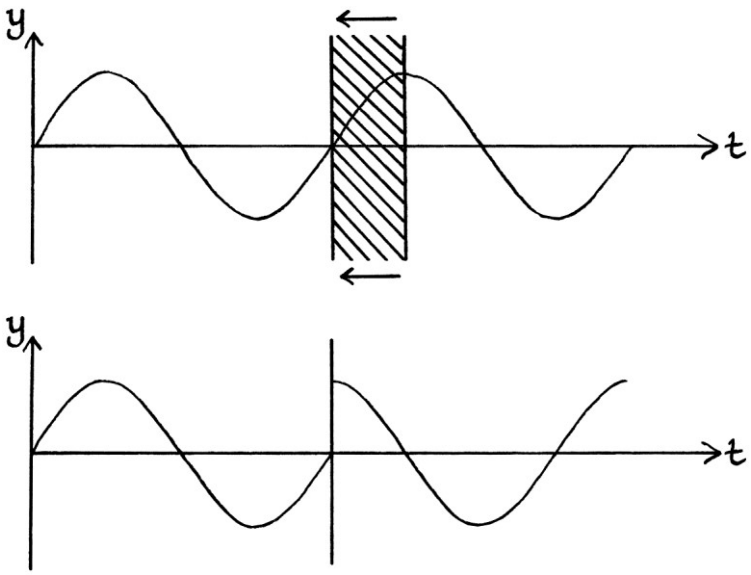
Then after the object has rotated some more, releasing the button will cause the object to jump immediately backwards by 90 degrees:



The object will then continue to rotate as normal.

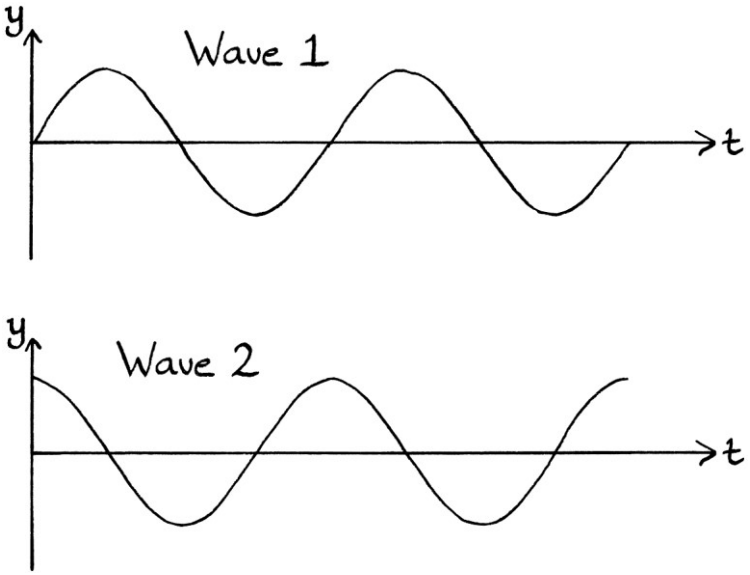
When it comes to the wave, we can think of the button press as having the effect of making the state of the wave suddenly jump forwards to how it would have been 90 degrees later. When the button is released, the state of the wave jumps back to how it would have been 90 degrees earlier. Pressing the button has the same effect as instantly sliding the wave from that point onwards to the left by 90 degrees. When the button is released, it is as if the wave is instantly slid to the right by 90 degrees.

In more detail, one can imagine that at the exact instant of the button press (the phase increase), time stops completely, the part of the wave 90 degrees in the future (quarter of the wave's cycle in the future) is shifted left along the time axis to the current time, skipping what would be there, and then time starts again. It is as if time has skipped quarter of a wave cycle.



When the phase returns to normal, it is as if time stops and an extra quarter of a cycle is inserted before the rest of the wave.

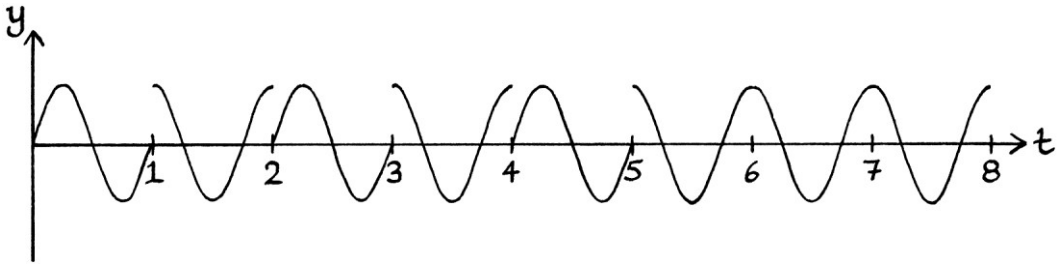
Yet another way of thinking about what is happening is to imagine two waves occurring at the same time – one with a phase of zero degrees, which we will call Wave 1, and the other with a phase of 90 degrees, which we will call Wave 2. We will imagine that we are watching a television showing Wave 1. When the "increase phase" button is pressed, it is as if the television instantly switches to showing Wave 2 instead. When the button is released, the television returns to showing Wave 1.



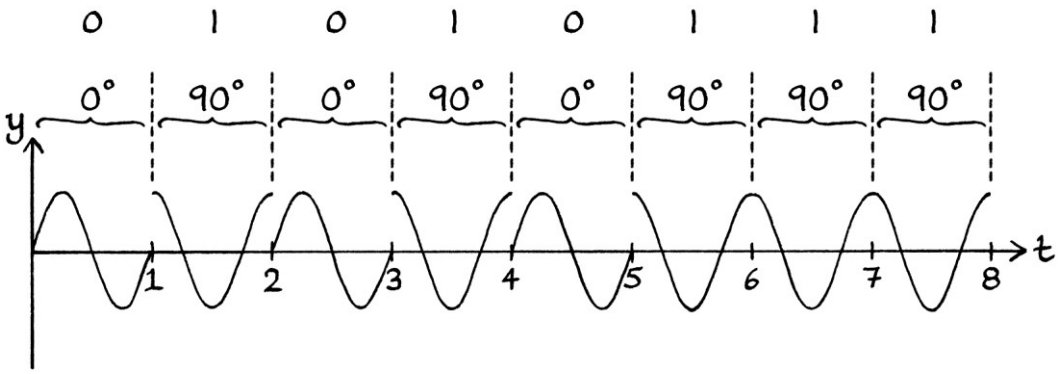
It is important to note that the state of the wave when the button is released is exactly as it would have been if the button had never been pressed at all. It is not as if the wave starts anew at that point.

Using our transmitter, we can indicate a zero by leaving the phase untouched, and we can indicate a one by increasing the phase by 90 degrees. We will say that our transmitter transmits a Sine wave with a frequency of 1 cycle per second (which is a different frequency from those we used for ASK and FSK).

To illustrate the idea, if we were sending the binary number:
01010111
... by switching the phase of a one-cycle-per-second wave with the formula:
“y = sin 360t”
... then the resulting signal would look like this:



The same picture, with the phases and their meanings written on the graph, is as so:



Note how it can be harder to see the changes in state when we are altering the phase, than when we are changing the amplitude or frequency. It is also harder to identify which state the wave is in at any point in time. Ideally, the person receiving the signal would have a reference wave with which to compare the signal, so that they could determine the phase at any particular moment in time.

Changing the phase in this way is called “phase shift keying” or “PSK” for short.

Although we were able to make analogies for ASK and FSK involving keys on a piano, it is difficult to do this for phase shift keying without it becoming very contrived. We could imagine two identical pianos and a piano player who hits the same key on each piano a fraction of a second apart. Assuming the note from each piano continues for a long time, and the player can mute the sound from each piano instantly, they could indicate a zero by muting the second piano, and indicate a one by muting the first piano. We also have to assume that whoever is listening can identify the phases, and that they have a reference piano with which they can compare the phases.

Sidebands

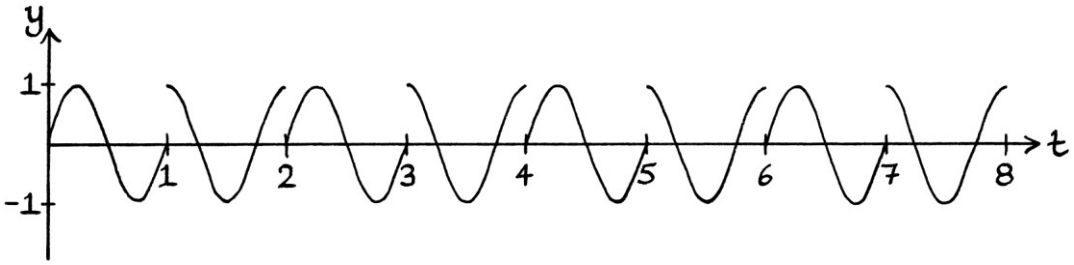
One question that people new to phase shift keying might ask is, “How is it possible for a wave to have a jump in it, such as can happen with phase shift keying?” The answer is that it is not possible for a *pure* wave to have a jump in it. However, as explained in part one of this book, when a wave stops being a pure wave, as the one in the above picture does, it becomes the sum of two or more pure waves of various frequencies, phases and amplitudes. Therefore, a wave that contains jumps is really the sum of multiple other pure waves. This in turn means that the frequency domain view of a signal subject to phase shift keying will have multiple

frequencies around the point of the phase change. This means that phase shift keying, *like all modulation,* creates waves with other frequencies. Careless use of phase shift keying (or any modulation) of radio waves could interfere with other nearby broadcasts.

We will use phase shift keying to portray the binary number:
0101010101...

We will use:
"y = 1 sin ((360 * 1t) + 0)"
... for one second to portray a zero, and:
"y = 1 sin ((360 * 1t) + 90)"
... for one second to portray a one.

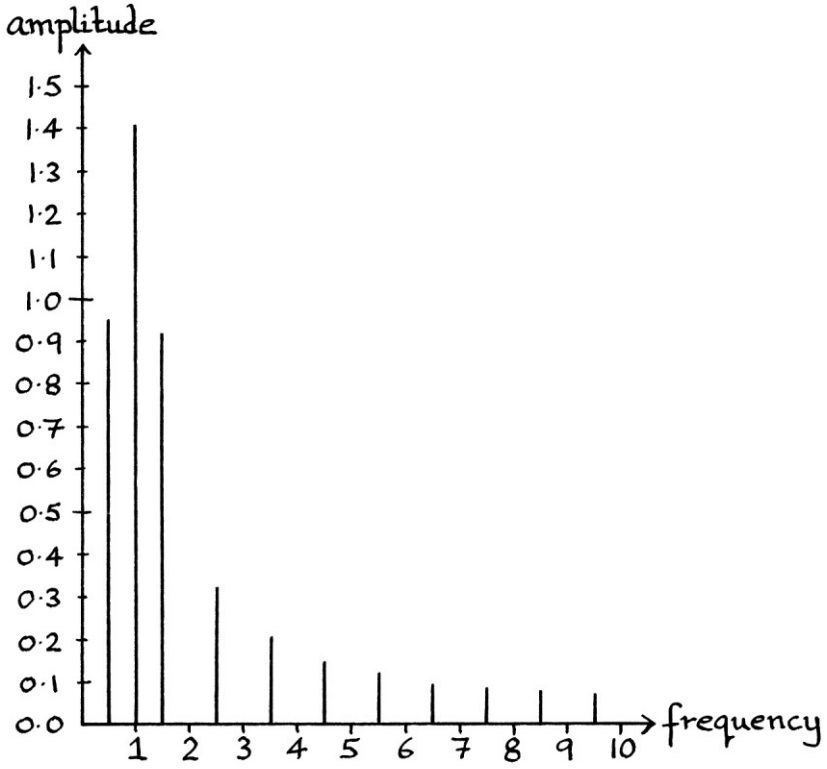
The signal looks like this:



Such a signal is periodic, so we can use Fourier series analysis to discover the characteristics of the pure waves that could be added to create it.

The frequency of this signal as a whole is 0.5 cycles per second. Therefore, the constituent waves will have frequencies that are integer multiples of 0.5.

The following graph shows the constituent waves (with the y-axis as amplitude):



From the graph, we can see that our PSK signal actually consists of eleven different waves (and more if we include amplitudes below 0.05 units). For the entire existence of our PSK signal, we are producing all these eleven waves. The reason for the large number of waves is that we have vertical jumps in our signal. As we saw in part one of this book, signals that contain vertical jumps have countless constituent waves. It is impossible to create a signal exactly by adding pure waves if that signal has vertical jumps in it, and any approximate reconstruction will involve an infinite number of waves (although many will have negligible amplitudes). Whether we end up with a periodic signal or not, the very nature of phase shift keying means that we will have significant sidebands, and we will be using more bandwidth. In the above example, we created a periodic signal, but normally, a PSK signal is unlikely to be periodic. Therefore, the extra waves would vary over time.

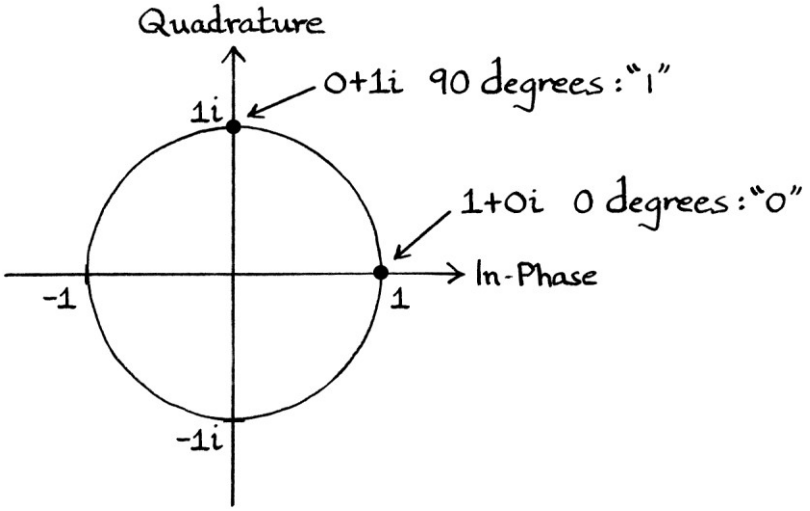
Constellation diagram

One way to look at the phase changes in our phase shift keying example is with a constellation diagram, such as we introduced in Chapter 7.

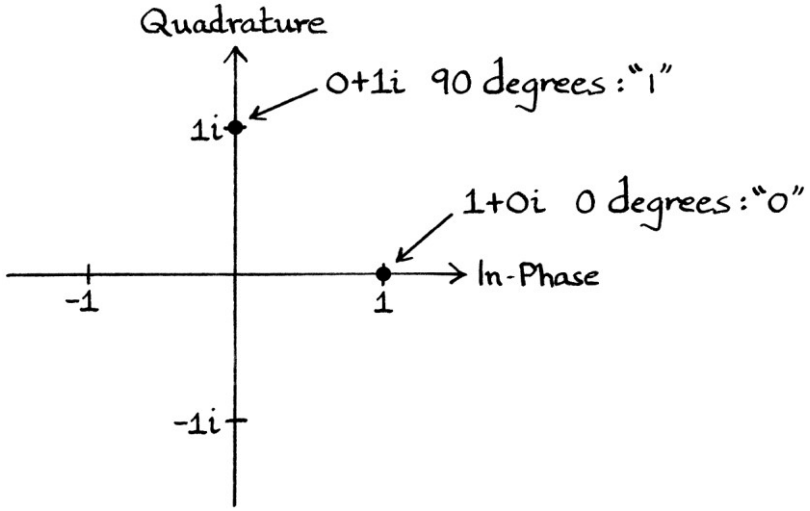
For our phase shift keying, the two waves we are using in our transmitter are:
"y = 1 sin ((360 * 1t) + 0)"
... and:
"y = 1 sin ((360 * 1t) + 90)"

We will use the zero-phase wave as the reference or "In-phase" wave. This appears at the coordinates of (1, 0) on the constellation diagram. As constellation diagrams usually use Complex numbers, we will say that it is at "1 + 0i" in the constellation diagram. [Note how I am still including the unnecessary ones and zeroes in Complex numbers to make everything clearer. Other people would probably not do this.] The wave with the phase of 90 degrees appears at the coordinates of (0, 1) or "0 + 1i" on the constellation diagram. The two points on the constellation diagram show the starting points of the two waves that we will switch between during our communication. Another way of thinking about this is that they show the two phases that are possible using our current phase shift keying method.

The constellation diagram can be drawn with the underlying circle included:



It is more usual for the constellation diagram to be drawn with just the relevant points without any circles:

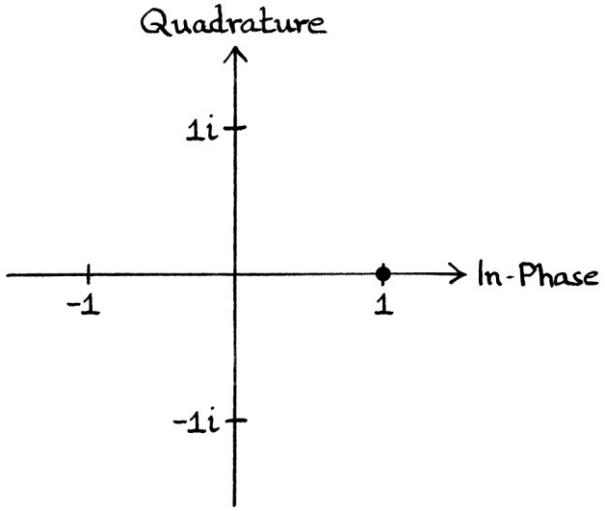


The constellation diagram shows the relative phases that indicate the values "0" and "1". For any signal, when the phase is zero, the signal is indicating a "0", and when the phase is 90 degrees, the signal is indicating a "1". Our particular arrangement of phase shift keying is summarised within the constellation diagram.

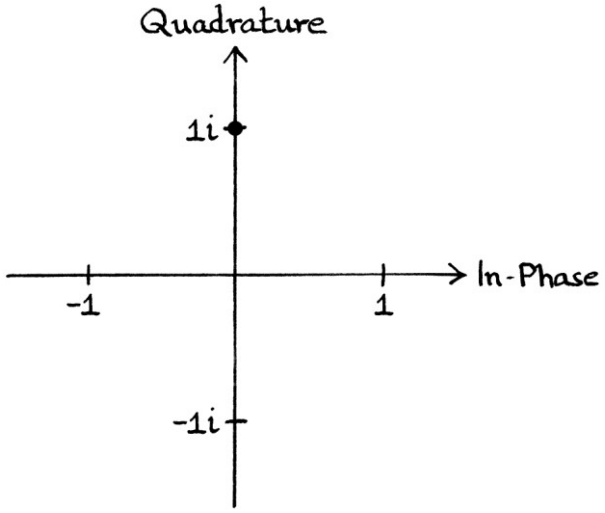
A picture of a constellation diagram is useful to show how we intend to use the range of phases that will convey a message. A radio receiver might also use a constellation diagram on a screen to show the phases that are being received. In such a case, it would be showing the *starting* phases of each wave that the signal jumps between – it would not be detecting the *instantaneous* phases of each wave (which would not be useful). Putting this in terms of the television analogy from earlier, the receiver would be detecting which wave we were viewing at any particular time, and it would be identifying those waves by their starting phase, which it would indicate by a dot in the appropriate place on the diagram.

If we observed a message sent using our "zero degrees – ninety degrees" phase shift keying, using a receiver that could detect the phases of the waves existing at any moment in time, we would see the detected phases jump from "1 + 0i" to "0 + 1i" and back again. Over time, we might see this:

The first second (meaning “zero”):

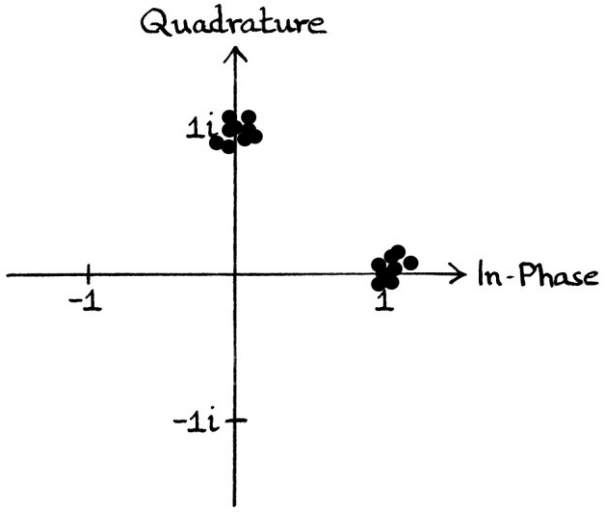


The second second (meaning “one”):

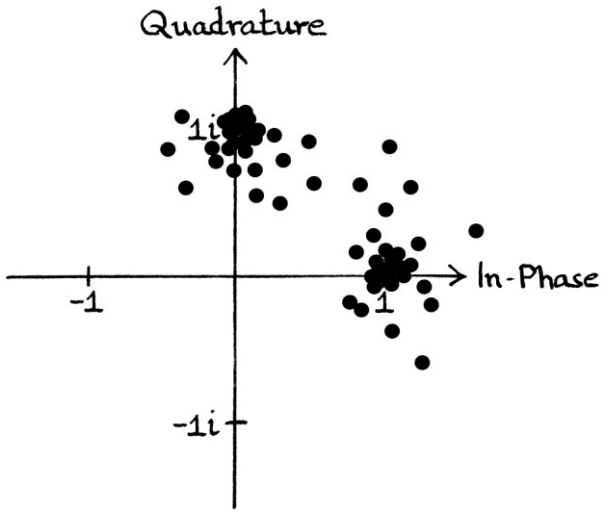


In practice, phase shift keying would change phase much more often than once a second, and the waves would have frequencies of thousands or millions of cycles per second. Therefore, it would not be useful to have a screen that showed just the current phase. The current phase might exist for only a fraction of a second before it changed, so a person would not be able to decode the signal visually. It is more useful to see all the phases that have been transmitted, and allow electronics or software to decode the signal instead. In this way, seeing the phases that have happened allows us to know how many phases are being used and what they are. It also tells us whether the transmitting equipment, the receiving equipment, and the decoding equipment (or software) are working correctly.

A real-world constellation diagram, as viewed on a screen, and made from detecting actual radio waves, would never be as neat as the constellation diagrams we have seen so far. For a "zero degrees – ninety degrees" phase shift keying system, instead of having two distinct single dots marking the starting phases of the waves in the signal, there would be multiple dots that were grouped seemingly randomly around where the phase points would be expected, as shown here:

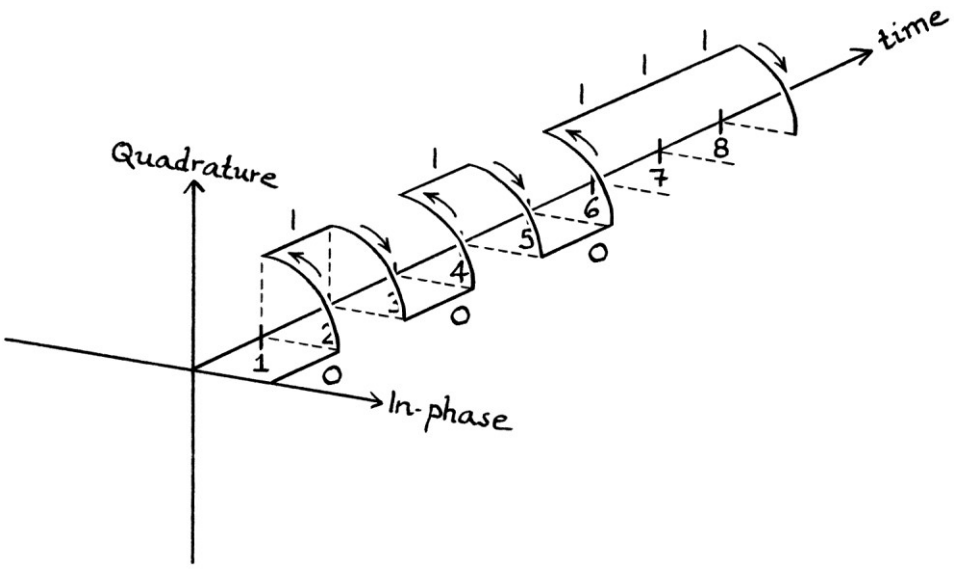


The reasons for this are due to minor flaws in how signals are transmitted and received, and flaws in how they are decoded. As long as the phase points are clustered around the correct point, the signal can still be decoded. Problems occur when the phase points stray too far from their proper location. In such cases, the signal might be decoded incorrectly. In the following picture, some of the received phase points are too misplaced to know what their meaning should be. Although we can tell that the phases are supposed to switch between zero and ninety degrees, it is impossible to know which phase every single point is supposed to represent:



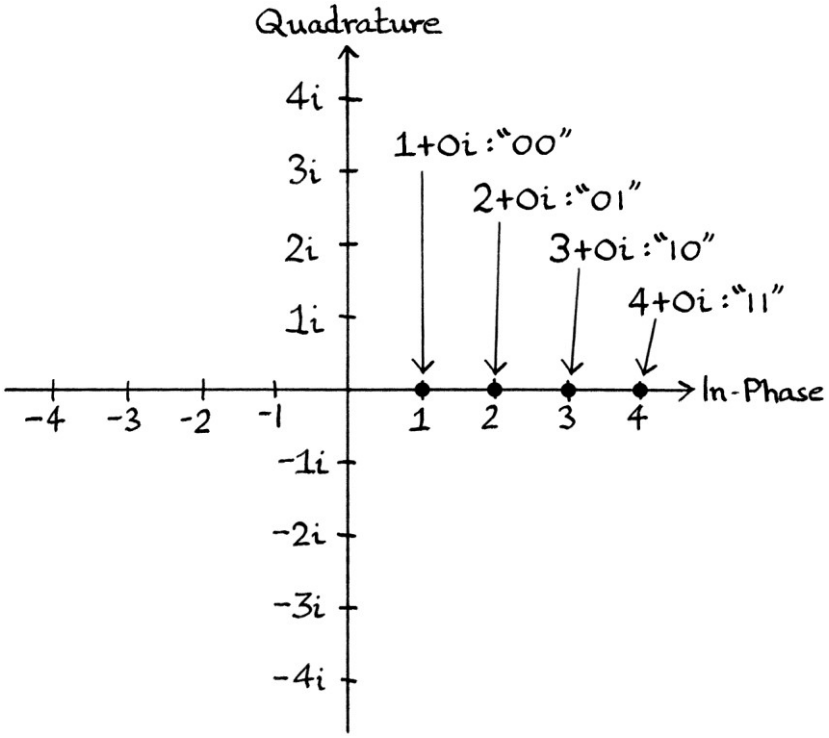
The constellation diagram is helpful to show the range of phases, but as with the circle chart or the Complex plane, we can get a slightly better understanding of a signal's behaviour by showing it as a three-dimensional graph with a time axis. Doing this with the circle chart or the Complex plane created the helix chart or the Complex helix chart. Doing this with the constellation diagram will create what I will call the "time-based constellation diagram".

The time-based constellation diagram shows the phase of the signal at any particular moment in time. Instead of just showing the phases as points, which would make the graph difficult to understand, the graph shows the movement from each point as well. For our original binary number of "01010111", sent using our system of phase shift keying, the phases over the length of the message would appear as so:



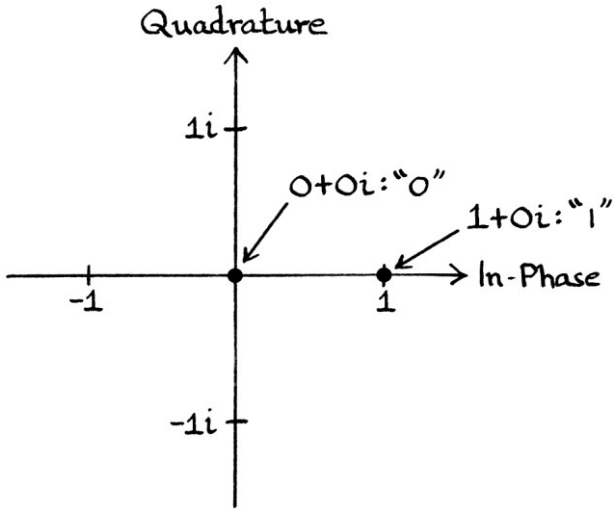
Note that this is not how the *signal* looks over time, and it is not how the helix from which a signal could be said to be derived looks over time – the diagram shows the waves that we switch between to send the message, and it identifies each wave by its starting phase. It would be easy to confuse the types of graph.

A two-dimensional constellation diagram can show the possible states in amplitude shift keying. For 4-ASK, such a diagram might look like this:



All the points are along the 0-degree line (the In-phase axis) because they have the same phase.

We can also use the constellation diagram to show the possible states in on-off keying:

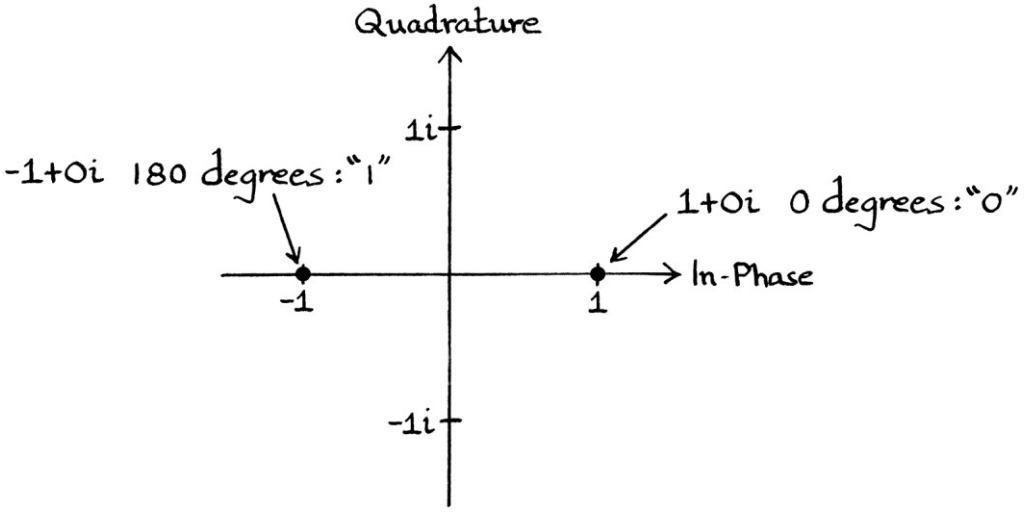


Constellation diagrams are less useful when the phases are all zero, but they still act as a guide as to which waves are being used.

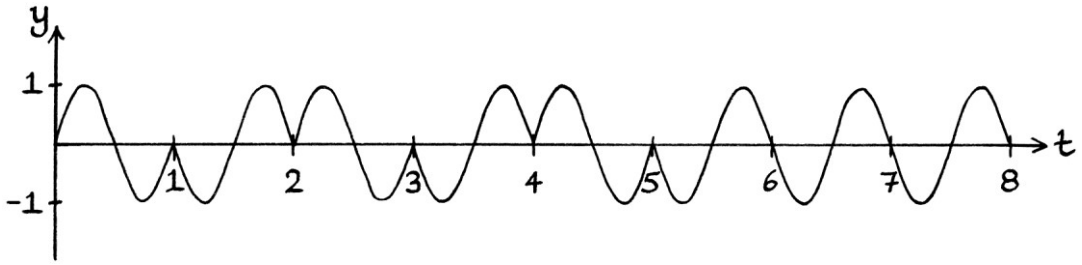
Phase shift keying in practice

As with amplitude shift keying and frequency shift keying, we can use several different changes in phase to send a message.

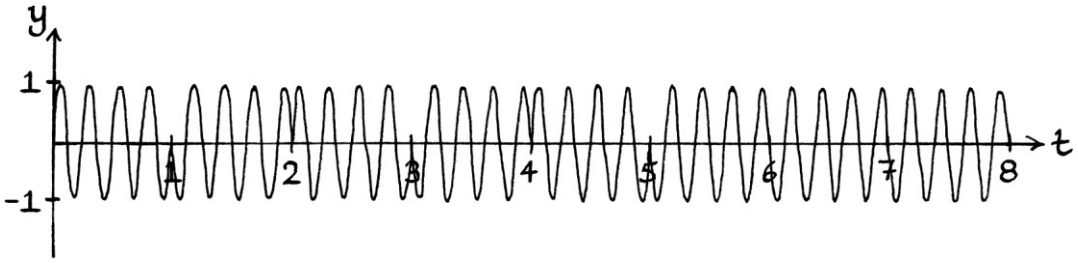
In the above examples, we used two different phases of 0 degrees and 90 degrees. This is called "2-PSK" or "BPSK", where the "B" stands for "binary" and refers to how there are two states, and not to how we are using it to encode binary. In practice, 2-PSK would be more likely to use phases of 0 degrees and 180 degrees. These are far enough apart that they will not be confused in bad conditions or on inferior equipment. The constellation diagram showing the two possible phases is as so:



A graph showing the binary number "01010111" encoded with 2-PSK, with phases of 0 degrees and 180 degrees, looks like this:



We could also have more cycles per phase state. For example, the message could be portrayed as follows, which is harder to decode visually:



Difficulties

One aspect of phase shift keying that makes it slightly more complicated than ASK or FSK is that to decode the message, it is necessary to know which phase represents "0" and which phase "represents "1". If we observe the signal from the start of the transmission and we know that it always starts with a one or a zero, then this is not a problem. However, if we miss the very beginning or do not know how it starts, we cannot know which phase represents which digit.

A receiver cannot know the literal starting phase when the transmitter started up, which is why receivers must deal in *relative* phases. That is why the constellation diagram has an axis called "In-phase" – a phase on this axis is used as a reference for the other phases. All the other phases are plotted relative to the reference phase. Given that, if we are using phase shift keying with two phases of 0 degrees and 180 degrees, without further information, it would be impossible to know which phase state we were in at any particular time. [If we were using phases of 0 degrees and 90 degrees, then it would be possible, but at the cost of not making the best use of the available space – if we are able to use quarter circles, then we could use more phase states to send more information as we will see shortly].

A receiver might not know the phase of the wave as it was transmitted, however, it would still be possible to see the phase *changes*. It would be possible to see how the phases changed from a particular starting phase. This means we would be able to decode something, but it might not be correct.

For two phases of 0 and 180 degrees, this binary number:
10101111
... might be misinterpreted as this binary number:
01010000

... because, although we can distinguish the phase changes, we do not know which phase state we are in, so we do not know whether a phase state represents a one or a zero.

Despite all of this, it might be possible to make deductions from the context to know which phase represented zero. If we were expecting ASCII text, then the nature of the received 8-bit binary numbers might indicate if we had the ones and zeroes the wrong way around. The text would make sense if the ones and zeroes were correct, but would be nonsense otherwise.

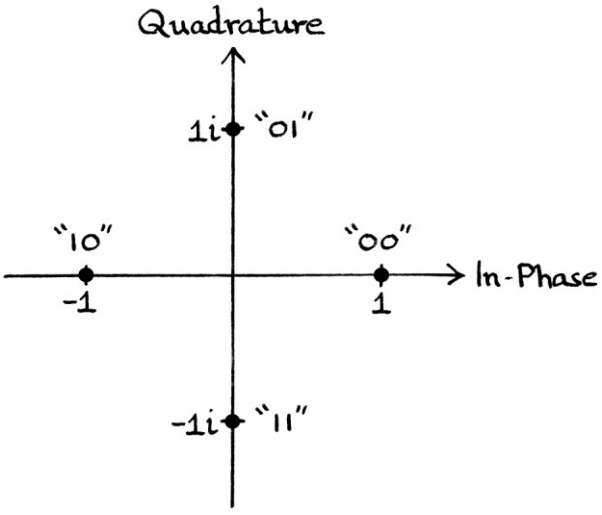
Another way to get around this problem is to have a preamble that starts with "0" (and to have the receiver know that the preamble always starts with "0").

Higher levels of PSK

If we were to use 4 different phases, we would be using what is called "4-PSK" [or "QPSK", where the "Q" stands for "quaternary" or "quad"], and a typical choice of the four phases would be:

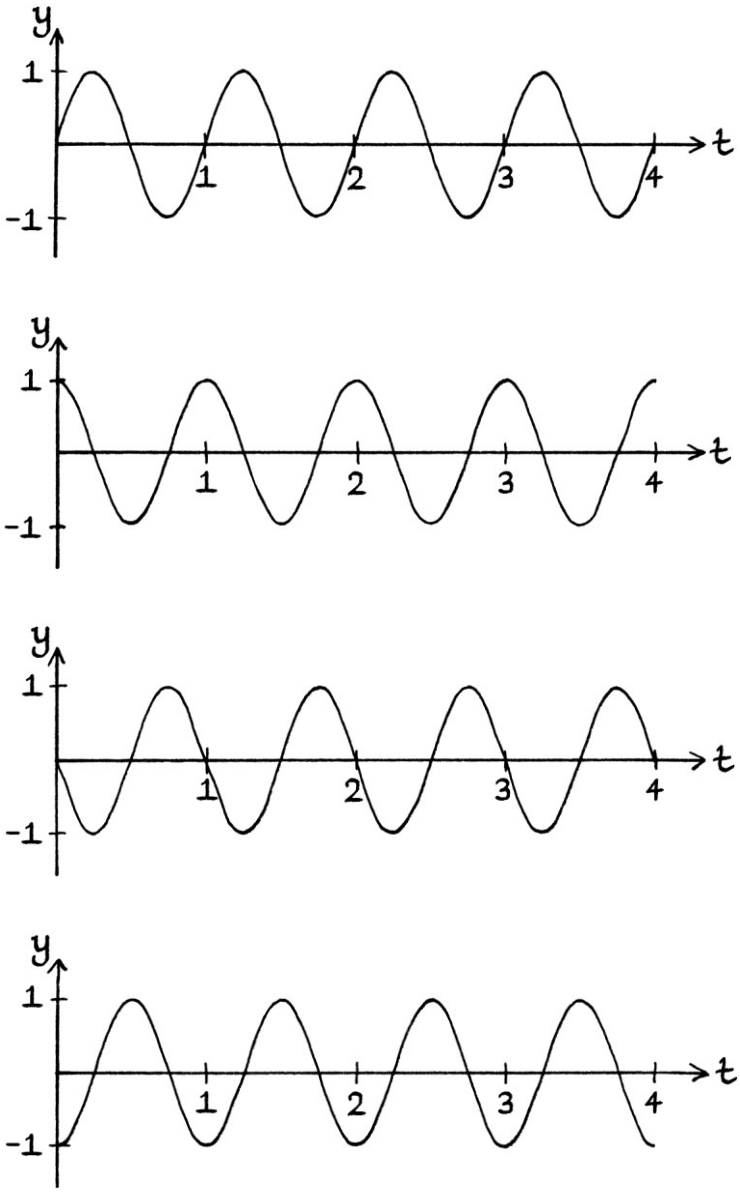
- 0 degrees represents "00"
- 90 degrees represents "01"
- 180 degrees represents "10"
- 270 degrees represents "11"

This 4-PSK system would appear as so on the constellation diagram:

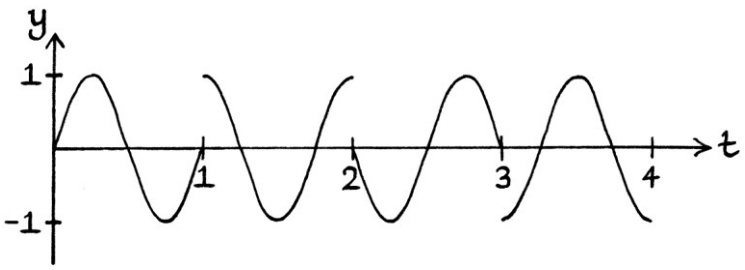


[I say a "typical" choice would be these phases, but strictly speaking, the phases could be anything such as 1, 5, 122 and 123 degrees. However, it is more straightforward to space the phases evenly around the circle, and by doing so, it will reduce the chances of misreading a particular phase.]

For our example, we will base the four states of phase on these four wave formulas. Each change in phase would essentially be switching from one wave to another:
"y = sin 360t"
"y = sin (360t + 90)"
"y = sin (360t + 180)"
"y = sin (360t + 270)"



An example of a signal portraying the binary digits 00, 01, 10, 11 is as so:

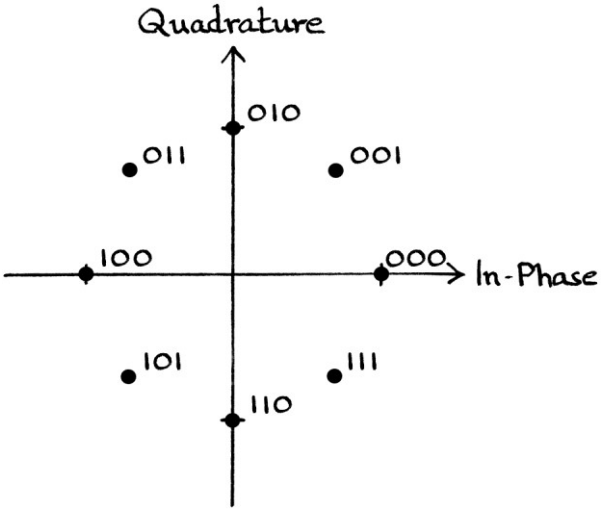


Higher levels of PSK are much harder to decode by looking at a graph than higher levels of ASK or FSK.

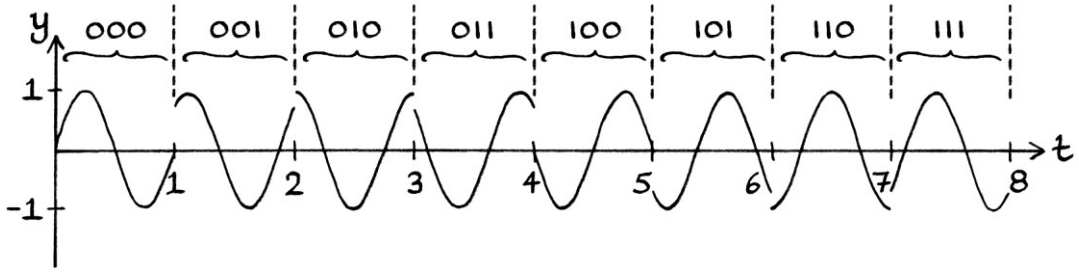
8 different phases ("8-PSK") could use this system:

- 0 degrees represents "000"
- 45 degrees represents "001"
- 90 degrees represents "010"
- 135 degrees represents "011"
- 180 degrees represents "100"
- 225 degrees represents "101"
- 270 degrees represents "110"
- 315 degrees represents "111"

The constellation diagram showing the range of phases and their meanings is as so (with the axes numbering removed to make the graph clearer):



A signal portraying the digits: 000, 001, 010, 011, 100, 101, 110, 111 would appear as so:



Variations of phase shift keying

There are countless variations of phase shift keying. In general, these improve the efficiency of the transmission by reducing the sidebands created by the phase changes, or they make decoding the signal easier.

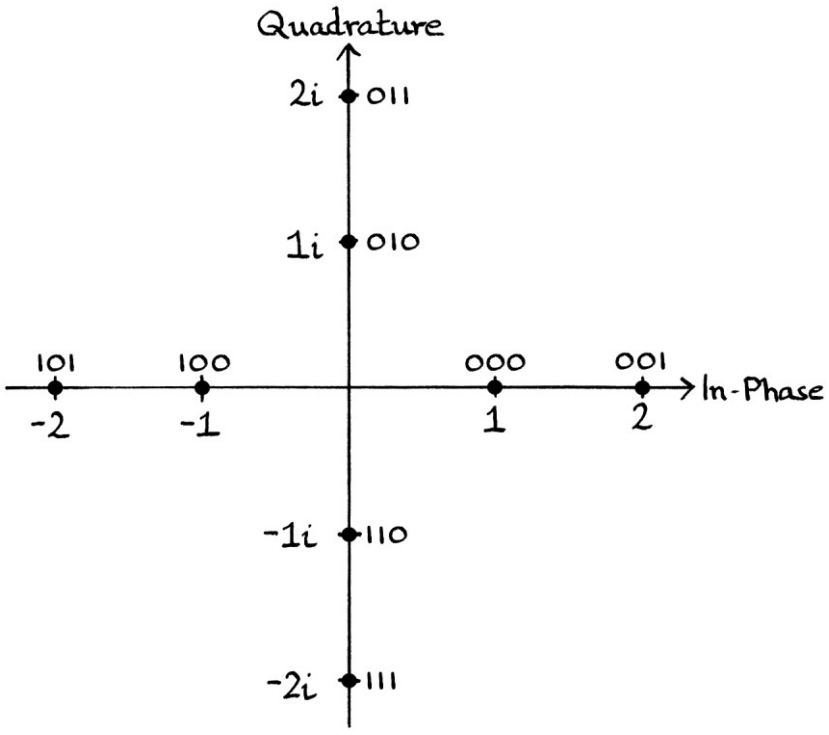
When decoding phase shift keying, it can help to have a reference wave with which the received signal can be compared. One type of phase shift keying does not require a reference wave because each phase change is relative to the previously seen phase. In other words, instead of having a number represented by a particular phase, a number is represented by the *change in phase* from the previous phase.

A simple example of this would involve representing a one as an increase in phase of 90 degrees, and a zero as a decrease in phase of 90 degrees. Therefore, if we had the binary digits 1101, we would first transmit a preamble to illustrate the timing. This could be 0101010101010101. The first "0" of the preamble would be sent with a phase of zero degrees. Then, the "1" would be sent with a phase of 90 degrees more than this (which is 90 degrees). The next "0" would be sent with a phase of 90 degrees less (which is 0 degrees), and so on until we get to the end of the preamble, which ends with "1" (which would be 90 degrees). Then the actual message (1101) starts with a "1", which would be portrayed with a phase 90 degrees more than the last phase, so it becomes 180 degrees. The next digit is "1" again, so the transmitted phase is 90 degrees more than the last phase, so it is 270 degrees. The next digit is "0", so the phase is 90 degrees less, which is 180 degrees. The next digit is "1", so the phase is 90 degrees more, which is 270 degrees. If we had a series of "1" digits, the phase would move around the circle anticlockwise in 90 degree jumps. If we had a series of "0" digits, the phase would move around the circle clockwise in 90 degree jumps.

The system as described is a simple example of what is called “differential phase shift keying” or “DPSK” for short. There are other, more complicated, methods of differential phase shift keying. Phase shift keying is unique among the types of shift keying because the phase relates to circles, so as the phase increases, we just move around the circle. With amplitude shift keying or frequency shift keying, a higher amplitude or frequency never rolls around to zero.

Phase and amplitude

It might be clear that we can combine the types of shift keying to increase the number of bits sent at one time. Of the combinations of shift keying, it is most common to combine PSK and ASK. We can portray such an idea with a constellation diagram:



In the above constellation diagram, there are four different phases, and each phase has two different amplitudes.

The full range of phases and amplitudes, and the binary digits they represent are as follows:

Phase	**Amplitude**	**Binary digits represented by this phase and amplitude**
0 degrees	1 unit	000
0 degrees	2 units	001
90 degrees	1 unit	010
90 degrees	2 units	011
180 degrees	1 unit	100
180 degrees	2 units	101
270 degrees	1 unit	110
270 degrees	2 units	111

[The table shows just one of the many possible ways that we could assign binary digits to the amplitudes and phases. We could just as easily choose other ways.]

We will say that every wave has a frequency of 1 cycle per second and that each state remains the same for one second. In this way, for each group of 3 bits, we are sending one of the following waves for one second:
“y = 1 sin (360 * 1t)”
“y = 2 sin (360 * 1t)”
“y = 1 sin ((360 * 1t) + 90)”
“y = 2 sin ((360 * 1t) + 90)”
“y = 1 sin ((360 * 1t) + 180)”
“y = 2 sin ((360 * 1t) + 180)”
“y = 1 sin ((360 * 1t) + 270)”
“y = 2 sin ((360 * 1t) + 270)”

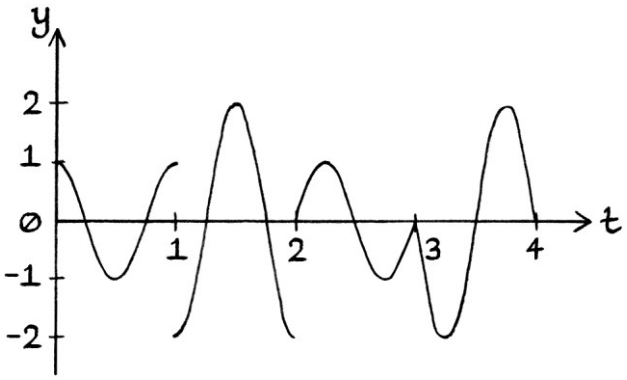
By having 4 different phases and 2 different amplitudes for each phase, we are able to portray 3 binary digits at a time. If we have one phase and amplitude change per second, we would be transmitting 3 bits per second, or one symbol per second where a symbol contains 3 bits.

We will say that we want to use the table to send this binary number:
010111000101.

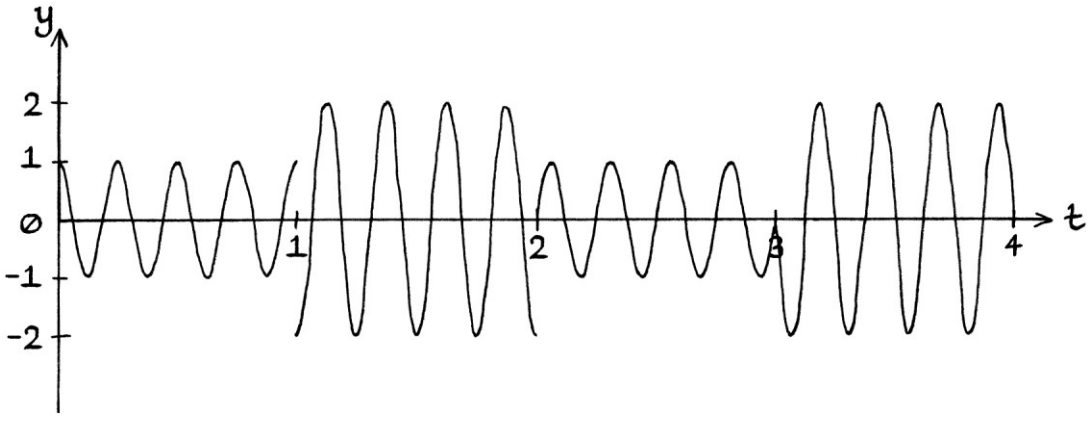
First, we split the digits into groups of 3:

010
111
000
101

Then, we create a signal using the relevant phases and amplitudes. It looks like this:



The same encoding using waves with frequencies of 4 cycles per second would look like this:



We could of course have more phases and more amplitudes, but with every extra phase or amplitude, the signal becomes harder to decode and more susceptible to being corrupted on its journey.

Quadrature amplitude modulation

Combining phase shift keying and amplitude shift keying is usually performed in a way that makes it seem much more complicated than it really is. The resulting signal will be the same, but the method of achieving it is different. Instead of creating the transmitted signal with the phase and amplitude shifts built in, the signal is created by altering just the amplitude of two different zero-phase waves and then adding them together. This has the same effect, and is easier to perform with electronics.

To understand the idea, it helps if you understand the following:

- Phase shift keying can be thought of as sending sections of different periodic pure waves with different phases at different times. In other words, we can think of PSK as switching from one pure wave with a particular phase to a different pure wave with a different phase.

- Amplitude shift keying can be thought of as sending sections of different periodic pure waves with different *amplitudes* at different times. In other words, we can think of ASK as switching from one pure wave with a particular amplitude to a different wave with a different amplitude.

- As we saw in Chapter 15, any pure wave with a non-zero phase can be portrayed as the sum of a Sine wave with zero phase and a Cosine wave with zero phase, both with the same frequency, and maybe with negative amplitudes. For example, this wave:
  "y = 2 sin ((360 * 2t) + 25)"
  ... can be thought of as the sum of these two waves:
  "y = 1.8126 sin (360 * 2t)"
  ...and:
  "y = 0.8452 cos (360 * 2t)"

  [The way to calculate which two zero-phase waves make up a non-zero phase wave is to imagine the phase point on the circle from which the wave could be said to be derived. The amplitude of the Sine wave will be the x-axis value of the phase point; the amplitude of the Cosine wave will be the y-axis value of the phase point. To put this another way, the zero-phase Sine wave will have an amplitude equal to the original wave as a Cosine wave at t = 0; the zero-phase Cosine wave will have an amplitude equal to the original wave as a Sine wave at t = 0. In the above example, the zero-phase Sine wave has an amplitude of "2 cos 25", and the zero-phase Cosine wave

has an amplitude of "2 sin 25". It is also worth remembering that adding any two pure waves with the same frequency results in a wave with the same frequency.]

- These ideas mean that we can perform carefully thought out amplitude shift keying on a Sine wave with zero phase and on a Cosine wave with zero phase, add the results together, and it will be as if we had performed both amplitude shift keying and phase shift keying.

In this method of combining PSK and ASK, there is a Sine wave and a Cosine wave that are both subjected *only* to amplitude shift keying, and then added together. The addition of the two amplitude-shift-keyed signals has the effect of creating a signal that incorporates amplitude shift keying *and* phase shift keying. The method is called "Quadrature Amplitude Modulation" and abbreviated to "QAM". The "quadrature" part of the name relates to how we start with two waves that are in "quadrature to each other" – in other words, two pure waves that have phases that are 90 degrees apart. These are a Sine wave and a Cosine wave. The "amplitude modulation" part of the name relates to how we are performing only amplitude shift keying on the two waves. The name is appropriate for the method to construct the transmitted signal, but it does not describe the actual transmitted signal particularly well. The name "quadrature amplitude modulation" really applies to how the idea is usually executed in electronics (and how the idea is usually taught). Given that the same result can be achieved without adding two waves in quadrature makes the name slightly redundant. A better name would be something such as "amplitude and phase shift keying" or "combined shift keying".

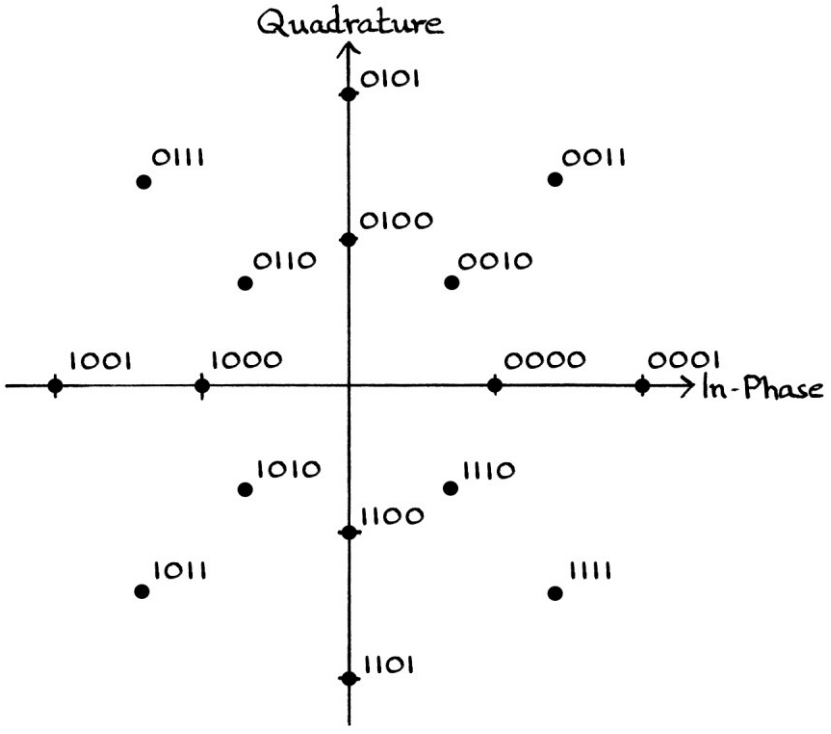
We will look at a simple example that uses 16 different states, and so can send 4 bits in one go. This is called 16-QAM. The simplest way of understanding this is to think of it as using 16 different waves, and switching from one to another to indicate a series of binary digits. We will start with the most straightforward explanation and then work backwards to how it is usually explained and executed.

For this example, we will use waves with a frequency of 2 cycles per second, and we will indicate each number by keeping the phase and amplitude the same for a full second. Both of these are arbitrary choices.

The 16 waves that we will use, and the 4 binary bits that each wave will represent, are as follows:

Wave	**Bits being represented**
“y = 1 sin ((360 * 2t) + 0)”	0000
“y = 2 sin ((360 * 2t) + 0)”	0001
“y = 1 sin ((360 * 2t) + 45)”	0010
“y = 2 sin ((360 * 2t) + 45)”	0011
“y = 1 sin ((360 * 2t) + 90)”	0100
“y = 2 sin ((360 * 2t) + 90)”	0101
“y = 1 sin ((360 * 2t) + 135)”	0110
“y = 2 sin ((360 * 2t) + 135)”	0111
“y = 1 sin ((360 * 2t) + 180)”	1000
“y = 2 sin ((360 * 2t) + 180)”	1001
“y = 1 sin ((360 * 2t) + 225)”	1010
“y = 2 sin ((360 * 2t) + 225)”	1011
“y = 1 sin ((360 * 2t) + 270)”	1100
“y = 2 sin ((360 * 2t) + 270)”	1101
“y = 1 sin ((360 * 2t) + 315)”	1110
“y = 2 sin ((360 * 2t) + 315)”	1111

The constellation diagram that demonstrates this table is as so (with the axis numbering removed to make it clearer):



We will say that one second's worth of one state indicates the relevant four digit binary number.

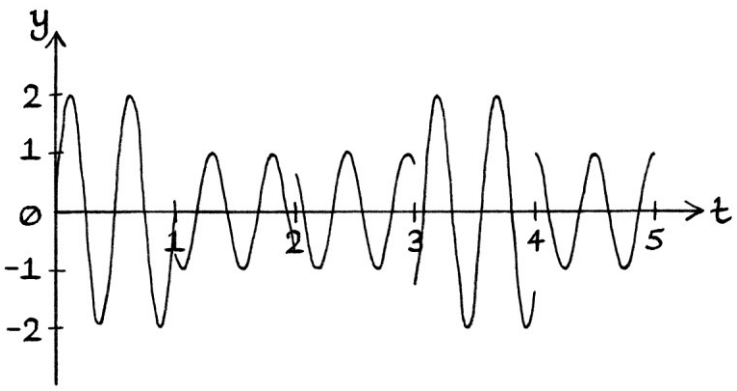
If we wanted to send the binary digits:
00011010011011110100
... we would split the number into 4-bit sections as so:
0001
1010
0110
1111
0100

Then for each group of bits, we would transmit the appropriate wave for just one section. Therefore, we would transmit the following:

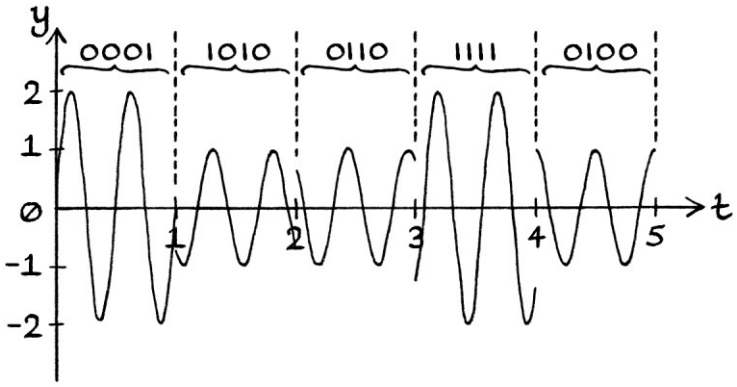
"y = 2 sin ((360 * 2t) + 0)" for one second
"y = 1 sin ((360 * 2t) + 225)" for one second (as if it had started at t = 0)
"y = 1 sin ((360 * 2t) + 135)" for one second (as if it had started at t = 0)
"y = 2 sin ((360 * 2t) + 315)" for one second (as if it had started at t = 0)
"y = 1 sin ((360 * 2t) + 90)" for one second (as if it had started at t = 0)

[Note how the one-second sections of waves are as if the wave had been in existence since the start. From a conceptual point of view, it is as if all the waves exist since the start, and we are switching focus from one wave to another. The waves do not start anew with their initial phase every time they appear. In this particular example, this does not make any difference as the cycles all align every second, but with other frequencies and state lengths, it might be important.]

The actual resulting signal looks like this:



With notes, it looks like this:



So far, this has been straightforward. The above signal is the actual signal that will be transmitted. However, normally with QAM, how that signal is created is different. With QAM, a Sine wave with zero phase is added to a Cosine wave with zero phase to produce that signal. With QAM, we have to split each of our waves with phases into the zero-phase Sine wave and zero-phase Cosine wave that when added would make that wave.

Note that splitting the waves into Sine waves with zero phases and Cosine waves with zero phases is only needed if doing so makes the process easier for the electronics in the transmitter. Otherwise, it is wasted effort. Most explanations do not point this out.

Here is a table showing the zero-phase Sine and Cosine wave sections that will be added together: [I have left in the waves with zero amplitudes.]

Binary digits being represented	**Wave with phase**	**Zero phase Sine and Cosine waves that when added create the wave with the wanted phase**
0000	“y = 1 sin ((360 * 2t) + 0)”	“y = 1 sin (360 * 2t)” “y = 0 cos (360 * 2t)”
0001	“y = 2 sin ((360 * 2t) + 0)”	“y = 2 sin (360 * 2t)” “y = 0 cos (360 * 2t)”
0010	“y = 1 sin ((360 * 2t) + 45)”	“y = 0.7071 sin (360 * 2t)” “y = 0.7071 cos (360 * 2t)”
0011	“y = 2 sin ((360 * 2t) + 45)”	“y = 1.4142 sin (360 * 2t)”

		“y = 1.4142 cos (360 * 2t)”
0100	“y = 1 sin ((360 * 2t) + 90)”	“y = 0 sin (360 * 2t)” “y = 1 cos (360 * 2t)”
0101	“y = 2 sin ((360 * 2t) + 90)”	“y = 0 sin (360 * 2t)” “y = 2 cos (360 * 2t)”
0110	“y = 1 sin ((360 * 2t) + 135)”	“y = −0.7071 sin (360 * 2t)” “y = 0.7071 cos (360 * 2t)”
0111	“y = 2 sin ((360 * 2t) + 135)”	“y = −1.4142 sin (360 * 2t)” “y = 1.4142 cos (360 * 2t)”
1000	“y = −1 sin (360 * 2t)”	“y = 0 cos (360 * 2t)” “y = 1 sin ((360 * 2t) + 180)”
1001	“y = 2 sin ((360 * 2t) + 180)”	“y = −2 sin (360 * 2t)” “y = 0 cos (360 * 2t)”
1010	“y = 1 sin ((360 * 2t) + 225)”	“y = −0.7071 sin (360 * 2t)” “y = −0.7071 cos (360 * 2t)”
1011	“y = 2 sin ((360 * 2t) + 225)”	“y = −1.4142 sin (360 * 2t)” “y = −1.4142 cos (360 * 2t)”
1100	“y = 1 sin ((360 * 2t) + 270)”	“y = 0 sin (360 * 2t)” “y = −1 cos (360 * 2t)”
1101	“y = 2 sin ((360 * 2t) + 270)”	“y = 0 sin (360 * 2t)” “y = −2 cos (360 * 2t)”
1110	“y = 1 sin ((360 * 2t) + 315)”	“y = 0.7071 sin (360 * 2t)” “y = −0.7071 cos (360 * 2t)”
1111	“y = 2 sin ((360 * 2t) + 315)”	“y = 1.4142 sin (360 * 2t)” “y = −1.4142 cos (360 * 2t)”

From looking at the table, if we want to send our message:
00011010011011110100
... we would split it into 4-bit sections as before:
0001
1010
0110
1111
0100
... and then we would choose the pairs of waves that represent these bits.

These are:
“y = 2 sin (360 * 2t)”
“y = 0 cos (360 * 2t)”
... together for the first four bits.

“y = −0.7071 sin (360 * 2t)”
“y = −0.7071 cos (360 * 2t)”
... together for the second four bits.

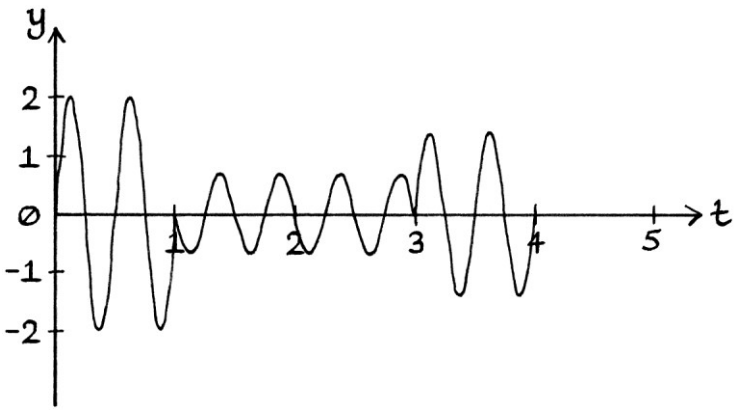
“y = −0.7071 sin (360 * 2t)”
“y = 0.7071 cos (360 * 2t)”
... together for the third four bits.

“y = 1.4142 sin (360 * 2t)”
“y = −1.4142 cos (360 * 2t)”
... together for the fourth four bits.

“y = 0 sin (360 * 2t)”
“y = 1 cos (360 * 2t)”
... together for the fifth four bits.

We will separate the two waves for each four bits, so that we have one signal based on Sine wave parts, and one signal based on Cosine wave parts. The Sine signal will consist of these parts:
“y = 2 sin (360 * 2t)” for the first second
“y = −0.7071 sin (360 * 2t)” for the second second
“y = −0.7071 sin (360 * 2t)” for the third second
“y = 1.4142 sin (360 * 2t)” for the fourth second
“y = 0 sin (360 * 2t)” for the fifth second.

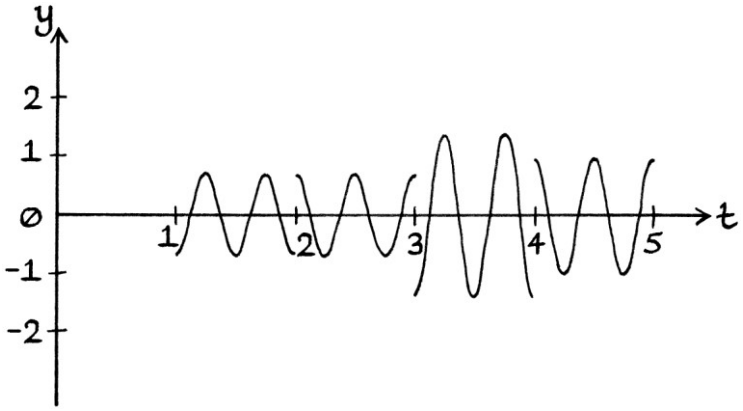
This signal looks like this:



As we can see, it is a Sine wave that has been subjected to amplitude shift keying (but where some of the amplitude shift keying has negative amplitudes).

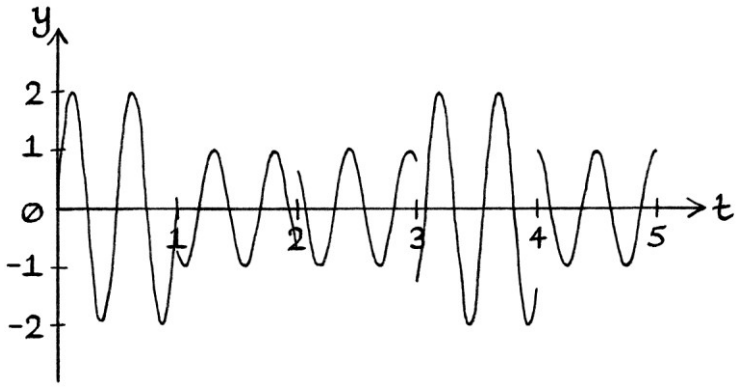
The Cosine signal will consist of these parts:
"y = 0 cos (360 * 2t)" for the first second
"y = −0.7071 cos (360 * 2t)" for the second second
"y = 0.7071 cos (360 * 2t)" for the third second
"y = −1.4142 cos (360 * 2t)" for the fourth second
"y = 1 cos (360 * 2t)" for the fifth second.

This signal looks like this:



As we can see, it is a Cosine wave that has been subjected to amplitude shift keying (but where some of the amplitude shift keying has negative amplitudes).

In quadrature amplitude modulation, part of the binary message is encoded into the Sine wave with amplitude shift keying, as we did here, and part of the binary message is encoded into the Cosine wave with amplitude shift keying, as we also did here. Then, the two signals are added together and transmitted. The two signals for this example, when added together, look like this:

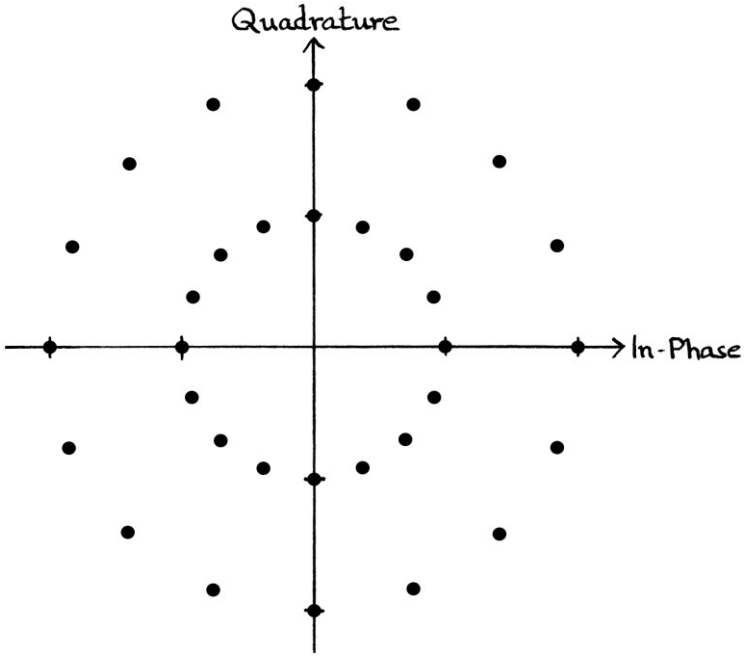


This is the same signal from earlier that we created using a combination of amplitude shift keying and phase shift keying.

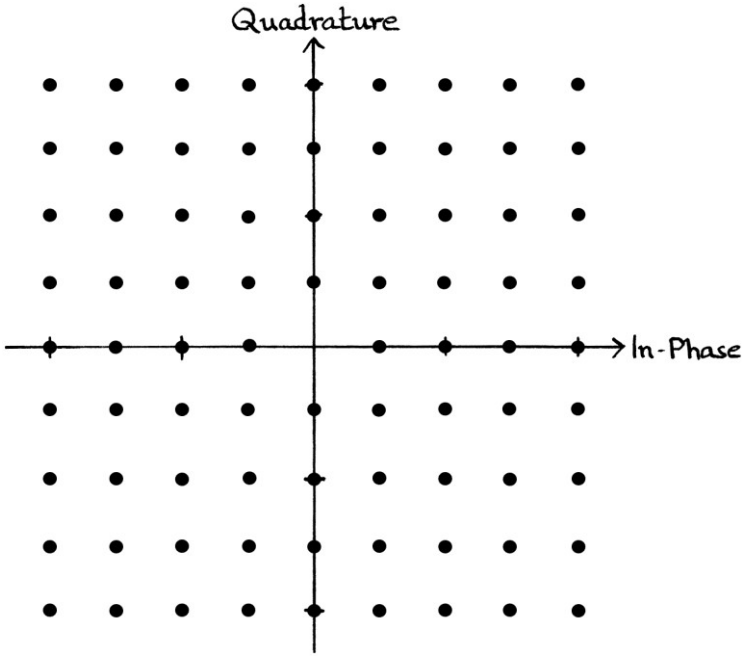
This has been a long and basic explanation of QAM to show that it is not particularly difficult to understand – it is just a more circuitous way of achieving ASK and PSK at the same time, by using ASK on Sine and Cosine waves with zero phases. On a piece of paper, it is much easier to calculate the final transmitted signal by using ASK and PSK. In electronics, it is easier to obtain the final signal by using ASK on Sine and Cosine waves with zero phases.

Square QAM

The type of QAM shown in the above example can be thought of as "circular" QAM – in the constellation diagram, the phases are at equal angles around the circle, and the amplitudes are the same for each "ring":



In practice, the form of QAM most commonly used, is "square" QAM. In this form, the different phases appear as in a square or rectangular grid like this:



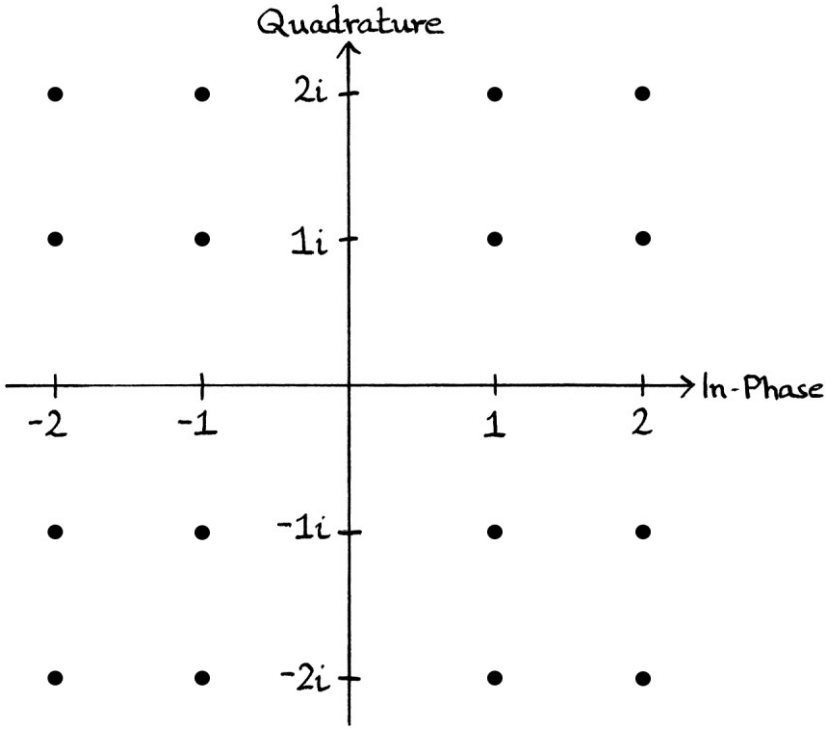
This form is so common that when most people think of QAM, they will be thinking of square QAM. In general, there is little need to use the adjective "square" to distinguish this type of QAM because this is what most people think of as QAM. One advantage of having the phase points in a grid is that the distance between each point is the same. With circular QAM, the distances vary according to how far away they are from the origin. When the phase points are the same distance away from each other, there is a more efficient use of space. This leads to more data being able to fit into a signal. It also means more data can be transmitted and decoded with less likelihood of confusion. As we will eventually see in this section, using a grid of points also means that it is easier to calculate the amplitudes of the zero-phase waves that will be used to indicate particular binary numbers.

[There are also countless variations of circular QAM, square QAM, and QAM in general.]

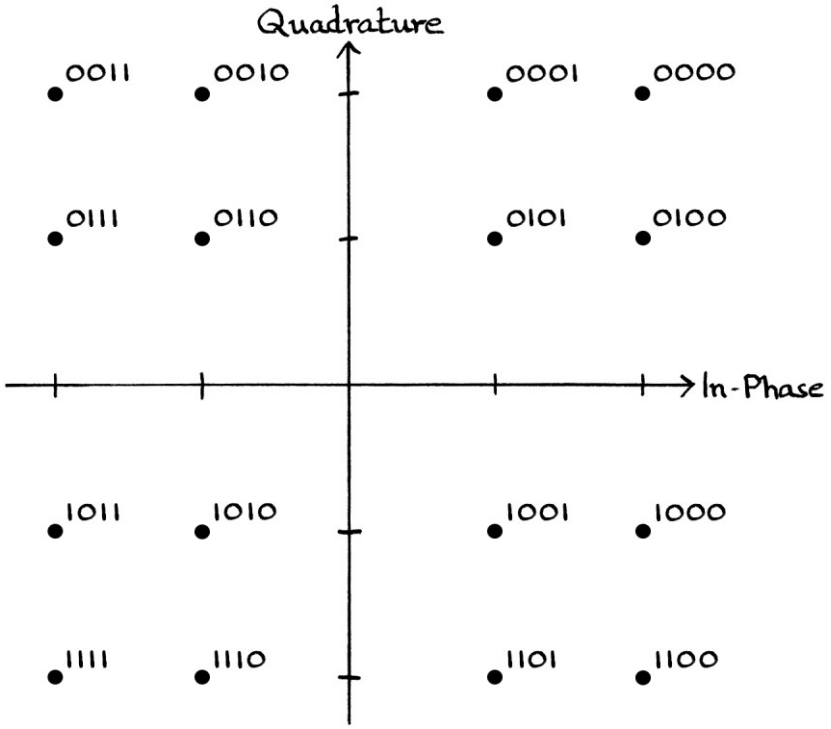
If you understood circular QAM, then square QAM is just *slightly* more complicated.

To achieve a square grid, the amplitudes cannot be the same for what would have been one ring of points. Similarly, to achieve a square grid, the phases cannot increase at an even rate.

As an example of Square QAM, we will use 16 different points as shown in this grid:



The bits represented by the points are as follows. These are the binary numbers from 0000 to 1111, placed right to left and from top to bottom.



[Note that this choice of which points represent which binary digits is arbitrary, and other systems of square QAM might use different layouts.]

What follows is a table showing the amplitude of each phase point (the distance of each point from the origin), and the angle of the phase point (in degrees). The amplitudes can be calculated by using Pythagoras's theorem. The angles can be calculated by using the arctan of the y-axis value divided by the x-axis value, or to be more accurate, the arctan of the Quadrature axis value divided by the In-phase axis value. As always, when using arctan, remember to check that the answer is the one that you want out of the two possible ones. You could also use a protractor to measure the angles. Since we are using a square grid, the angles and amplitudes are less pleasant than if we were using points around circles.

Binary number	Angle of point (degrees)	Distance from origin (amplitude)
0000	45	2.1213
0001	71.5651	1.5811
0010	108.4349	1.5811
0011	135	2.1213
0100	18.4349	1.5811
0101	45	0.7071
0110	135	0.7071
0111	161.5651	1.5811
1000	341.5651	1.5811
1001	315	0.7071
1010	225	0.7071
1011	198.4349	1.5811
1100	315	2.1213
1101	288.4349	1.5811
1110	251.5651	1.5811
1111	225	2.1213

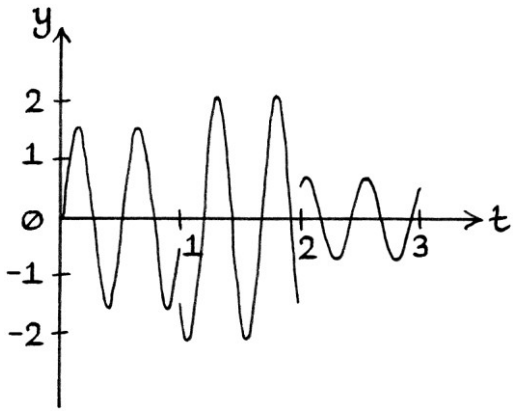
From the entries in the table, we can make up a list of the waves whose presence at any particular time would indicate each sequence of binary bits. We will give each wave the arbitrary frequency of 2 cycles per second:

Sequence of bits	Wave that will represent this sequence of bits
0000	“y = 2.1213 sin ((360 * 2t) + 45)”
0001	“y = 1.5811 sin ((360 * 2t) + 71.5651)”
0010	“y = 1.5811 sin ((360 * 2t) + 108.4349)”
0011	“y = 2.1213 sin ((360 * 2t) + 135)”
0100	“y = 1.5811 sin ((360 * 2t) + 18.4349)”
0101	“y = 0.7071 sin ((360 * 2t) + 45)”
0110	“y = 0.7071 sin ((360 * 2t) + 135)”
0111	“y = 1.5811 sin ((360 * 2t) + 161.5651)”
1000	“y = 1.5811 sin ((360 * 2t) + 341.5651)”
1001	“y = 0.7071 sin ((360 * 2t) + 315)”
1010	“y = 0.7071 sin ((360 * 2t) + 225)”
1011	“y = 1.5811 sin ((360 * 2t) + 198.4349)”
1100	“y = 2.1213 sin ((360 * 2t) + 315)”
1101	“y = 1.5811 sin ((360 * 2t) + 288.4349)”

1110 “y = 1.5811 sin ((360 * 2t) + 251.5651)”
1111 “y = 2.1213 sin ((360 * 2t) + 225)”

Supposing we were indicating each 4-bit number by transmitting the relevant waves for one second, then we could just use the above table as a guide to which waves to transmit. For example, if we wanted to send these binary digits:
100011110101
... we would split them into groups of four as so:
1000
1111
0101
... then transmit the relevant waves for each four bits. This means that we would transmit:
“y = 1.5811 sin ((360 * 2t) + 341.5651)” for one second, followed by:
“y = 2.1213 sin ((360 * 2t) + 225)” for one second, followed by:
“y = 0.7071 sin ((360 * 2t) + 45)”.

The signal would look like this:



To fit in with how QAM is usually thought of as adding an amplitude shifted Sine wave with zero phase to an amplitude shifted Cosine wave with zero phase, and transmitting the result, we will show how that idea works for the above example. [Remember that splitting the waves into Sine waves with zero phases and Cosine waves with zero phases is only needed if doing so makes the process easier for the electronics in the transmitter. Otherwise, there is no need.]

First, we need to adjust the table so that we know which pairs of Sine waves with zero phases and Cosine waves with zero phases represent each 4-bit binary number.

Binary number	Wave with phase	Zero-phase Sine and Cosine waves
0000	“y = 2.1213 sin ((360 * 2t) + 45)”	“y = 1.5 sin (360 * 2t)” “y = 1.5 cos (360 * 2t)”
0001	“y = 1.5811 sin ((360 * 2t) + 71.5651)”	“y = 0.5 sin (360 * 2t)” “y = 1.5 cos (360 * 2t)”
0010	“y = 1.5811 sin ((360 * 2t) + 108.4349)”	“y = −0.5 sin (360 * 2t)” “y = 1.5 cos (360 * 2t)”
0011	“y = 2.1213 sin ((360 * 2t) + 135)”	“y = −1.5 sin (360 * 2t)” “y = 1.5 cos (360 * 2t)”
0100	“y = 1.5811 sin ((360 * 2t) + 18.4349)”	“y = 1.5 sin (360 * 2t)” “y = 0.5 cos (360 * 2t)”
0101	“y = 0.7071 sin ((360 * 2t) + 45)”	“y = 0.5 sin (360 * 2t)” “y = 0.5 cos (360 * 2t)”
0110	“y = 0.7071 sin ((360 * 2t) + 135)”	“y = −0.5 sin (360 * 2t)” “y = 0.5 cos (360 * 2t)”
0111	“y = 1.5811 sin ((360 * 2t) + 161.5651)”	“y = −1.5 sin (360 * 2t)” “y = 0.5 cos (360 * 2t)”
1000	“y = 1.5811 sin ((360 * 2t) + 341.5651)”	“y = 1.5 sin (360 * 2t)” “y = −0.5 cos (360 * 2t)”
1001	“y = 0.7071 sin ((360 * 2t) + 315)”	“y = 0.5 sin (360 * 2t)” “y = −0.5 cos (360 * 2t)”
1010	“y = 0.7071 sin ((360 * 2t) + 225)”	“y = −0.5 sin (360 * 2t)” “y = −0.5 cos (360 * 2t)”
1011	“y = 1.5811 sin ((360 * 2t) + 198.4349)”	“y = −1.5 sin (360 * 2t)” “y = −0.5 cos (360 * 2t)”
1100	“y = 2.1213 sin ((360 * 2t) + 315)”	“y = 1.5 sin (360 * 2t)” “y = −1.5 cos (360 * 2t)”

1101	“y = 1.5811 sin ((360 * 2t) + 288.4349)”	“y = 0.5 sin (360 * 2t)” “y = −1.5 cos (360 * 2t)”
1110	“y = 1.5811 sin ((360 * 2t) + 251.5651)”	“y = −0.5 sin (360 * 2t)” “y = −1.5 cos (360 * 2t)”
1111	“y = 2.1213 sin ((360 * 2t) + 225)”	“y = −1.5 sin (360 * 2t)” “y = −1.5 cos (360 * 2t)”

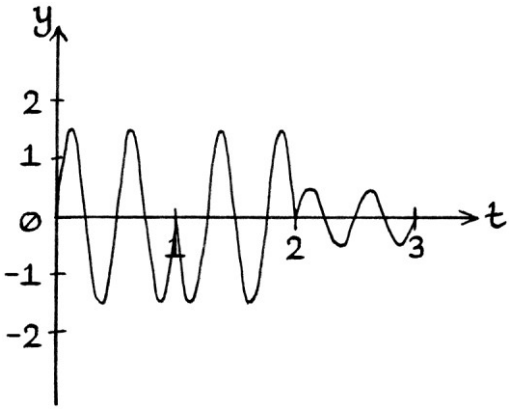
You might notice that the amplitudes of the Sine waves and Cosine waves are the x-axis and y-axis coordinates respectively of each phase point. It would be quicker to use this fact to get this table quickly, than to go through the stage of calculating the phases and amplitudes first. This is an advantage of using a square or rectangular grid as is done with square QAM.

Now that we have the pairs of Sine waves with zero phases and Cosine waves with zero phases, we can send our binary number again. The number was:
100011110101
... which we put into groups of four as so:
1000
1111
0101

We then find the pairs of waves for these 4-bit numbers. These are:
“y = 1.5 sin (360 * 2t)” and “y = −0.5 cos (360 * 2t)”
“y = −1.5 sin (360 * 2t)” and “y = −1.5 cos (360 * 2t)”
“y = 0.5 sin (360 * 2t)” and “y = 0.5 cos (360 * 2t)”

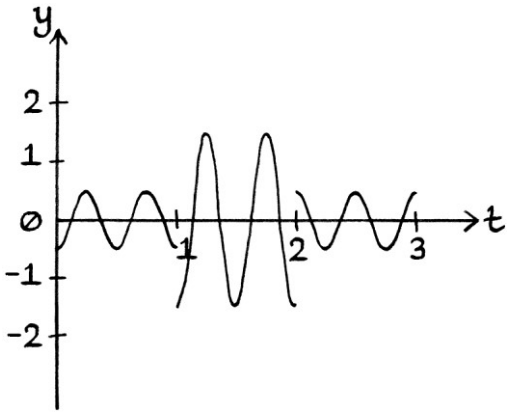
We put the Sine waves together to make a signal that will consist of:
“y = 1.5 sin (360 * 2t)” for one second
“y = −1.5 sin (360 * 2t)” for one second
“y = 0.5 sin (360 * 2t)” for one second.

This signal looks like this:

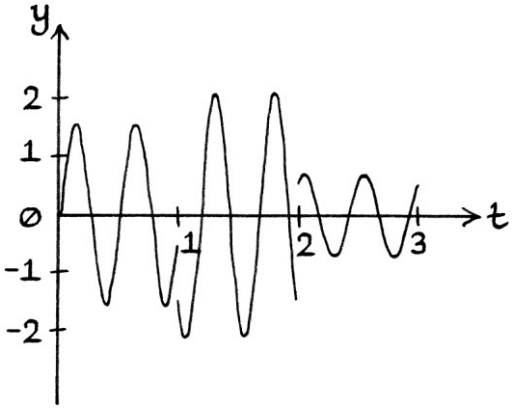


We then put the Cosine waves together to make a signal that will consist of:
"y = −0.5 cos (360 * 2t)" for one second
"y = −1.5 cos (360 * 2t)" for one second
"y = 0.5 cos (360 * 2t)" for one second.

This signal looks like this:



We then add the two signals together to produce the final signal that will be transmitted. This is:



The signal is identical to the one from earlier, as it should be.

Thoughts

If we are using square QAM, then it is fairly straightforward to make the look-up table of Sine waves with zero phases and Cosine waves with zero phases. For each phase point, the amplitude of the Sine wave will be the x-axis position of the phase point, and the amplitude of the Cosine wave will be the y-axis position of the phase point. From that, we can create the "Sine" signal and the "Cosine" signal that will be added together to make the final signal that is transmitted. Conversely, with Square QAM, it is more effort to create the table of waves *with* phases. It is harder because we need to use Pythagoras's theorem and arctan.

If we are using circular QAM, then it is very straightforward to make the look-up table of waves with phases – the phases are equally spaced around the circle and the amplitudes are the same for each ring. Conversely, it is *slightly* more effort to create the look-up table of Sine waves with zero phases and Cosine waves with zero phases. To do this we have to use Sine and Cosine.

Whether it is better to use the method of adding two separate signals, or just to jump to the final signal in one go, really depends on how the signal is being constructed before it is transmitted. If we were using a "simple" electronic circuit, it would probably be easier to create (or imagine) a square QAM table of zero-phase Sine and Cosine waves, and use it to create two waves that are then added and transmitted. If we were using a computer and a software defined radio, it might be easier to use a table of waves with phases, and jump to the final signal in one go.

If we did not mind whether we used circular QAM or square QAM, and we were using a computer, the effort in creating the look-up table would mostly be irrelevant to any decisions. This is because the table needs to be created only once, and after it is created, it can be used any number of times. After its creation, the only effort involved is in reading it.

More thoughts

At first glance, QAM, when done with two zero-phase waves, seems complicated, but if you understand the steps to get there, it is reasonably straightforward.

QAM is used in such fields as digital terrestrial television in Europe among other places. At the time of writing, digital television in the UK uses 64-QAM for normal television channels, and 256-QAM for high definition channels. The system for television is slightly more complicated than the one described in this section because several QAM signals are transmitted in parallel at the same time in different frequency bands.

Other combinations

Although it is possible to combine FSK and ASK, and FSK and PSK, the most common combination of keyings is ASK and PSK. As we will see near the end of this chapter, frequency and phase are related to each other, so altering the phase too quickly can affect the frequency. Therefore, combining FSK and PSK needs more care than QAM.

The best keying

When it comes to radio transmissions, the shift keying that is most appropriate for the data being transmitted really depends on several criteria:

- The complexity of the electronics needed to perform the method. Phase shift keying, for example, would have been more expensive and difficult to perform decades ago than it is now.

- The energy required to transmit the signal using the method.

- The bandwidth (as in the number of different frequencies used) taken up by the method. Some methods use more bandwidth than other methods, and so would be less suitable if there is limited available bandwidth for the signal.

- The centre frequency of the signal. If a transmission method uses a one-megahertz band of frequencies, then the frequencies will reach either side of the centre frequency by 500 KHz. Therefore, it would not be possible to use that method at, say, 100 KHz - there is not enough room under 100 KHz to accommodate the 500 KHz of frequencies.

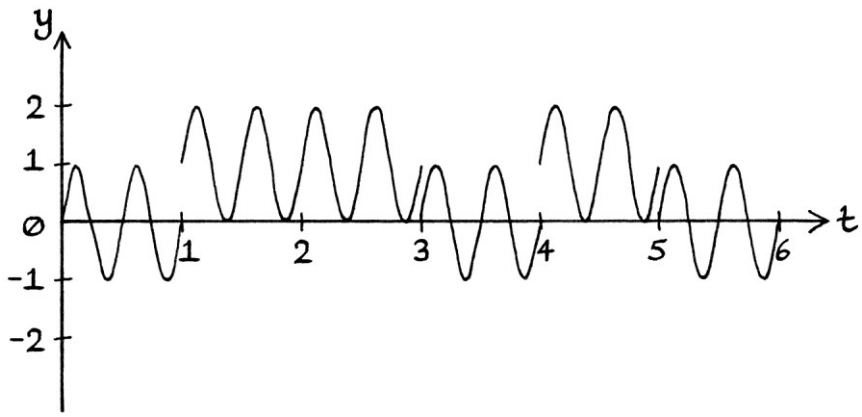
- How much interference there is at the frequency being used. If there is a lot of interference, then it might not be possible to correctly receive higher levels of ASK, for example.

Mean level shift keying

It is possible to adjust the mean level of a wave to encode a message in the same way that we did with amplitude shift keying, frequency shift keying and phase shift keying. We will call this “mean level shift keying”, which we will abbreviate to “MLSK”. We would not be able to use this method with radio waves, which, when travelling in the real world, have zero mean level. However, there are other phenomena and entities that have behaviour that can be described with waves, and for some of them, mean level shift keying would be possible. Waves in electric circuits are one such type.

For MLSK, we could keep the mean level at zero to indicate a zero, and raise it to 1 unit to indicate a one. As an example, we could say that:
“y = 0 + sin (360 * 2t)” indicates a zero
“y = 1 + sin (360 * 2t)” indicates a one
... where the frequency of 2 cycles per second is an arbitrary choice to make the graphs in this explanation clearer.

If we had the binary number:
011010
... we could encode it using MLSK as so:



We could have different levels of mean level shift keying. For example, we could have 4-MLSK with these waves:
"y = 0 + sin (360 * 2t)" indicates 00
"y = 1 + sin (360 * 2t)" indicates 01
"y = 2 + sin (360 * 2t)" indicates 10
"y = 3 + sin (360 * 2t)" indicates 11

The limit to the number of mean levels would be dependent on the entity that was creating the waves.

The mechanical device with the rotating arm from Chapter 32 would be able to encode data using MLSK, and furthermore, it would be able to encode data using two different mean levels at the same time.

Speed shift keying

When it comes to distance-based waves, we have another attribute that can be altered – the spatial frequency. We will look at this idea, and why it is not particularly useful for shift keying. Spatial frequency shift keying would be a system that encoded information into a wave by varying the spatial frequency of a distance-based wave. As a basic example, if a wave had a spatial frequency of 10 cycles per metre, we could use that to indicate a "0" and increase its spatial frequency to 20 cycles per metre to indicate a "1".

The concept of spatial frequency shift keying is the least useful of all the shift keyings. The other shift keyings (ASK, FSK, PSK, MLSK) are independent of each other, and could all be combined to be used at the same time. [Such a combination would need to be done carefully so that the changes in the state of one attribute could not be misinterpreted as the change of a different attribute.] Spatial frequency, on the other hand, is related to the temporal (time-based) frequency and the speed of the wave. For a particular temporal frequency, if we changed the speed of the wave, we would change its spatial frequency. For a particular wave speed, if we alter the temporal frequency, we would change the spatial frequency. We can know these two facts to be true by the formula that connects wavelength, speed and frequency:
*wavelength = speed * period*
... which becomes:
1 / spatial frequency = speed * (1 / temporal frequency)
... which is:
1 / spatial frequency = speed / temporal frequency
... which is:
temporal frequency / spatial frequency = speed
... which is:
*temporal frequency = speed * spatial frequency*

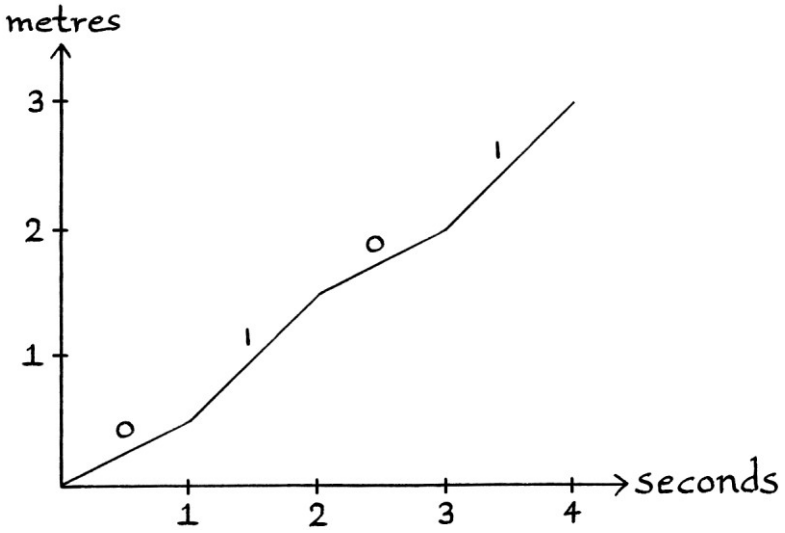
By changing the spatial frequency, we are also changing the temporal frequency or the speed, or both. Spatial frequency shift keying loses its usefulness because it is not an independent characteristic of a wave.

A slightly better shift keying idea for distance-based waves is “speed shift keying”. In this idea, the *speed* is varied to indicate binary digits. As an example, we could have two states:

- A speed of 100 kilometres per hour to indicate a “0”
- A speed of 110 kilometres per hour to indicate a “1”

A sequence of binary digits could be encoded into the movement of a distance-based wave by increasing and decreasing the speed. Such an idea can be displayed on a graph showing the distance travelled.

Here is an example of encoding the digits 0101 into the speed of an entity as seen on a "distance travelled" graph:



When the speed is 0.5 metres per second for one second, the moving entity is representing a zero; when the speed is 1 metre per second, it is representing a one.

There are four main problems with speed shift keying:

- The first problem might be apparent if you think of the Doppler effect (as mentioned in Chapter 33). If the wave is of a type where the characteristics are measured as the fluctuations pass a fixed place, then the *perceived* temporal frequency will rise and fall as the speed of the wave increases and decreases. Therefore, the speed will affect the *measured* temporal frequency. We could still use speed shift keying with such entities, but it could not be used at the same time as frequency shift keying. Given that, it would make more sense just to use frequency shift keying instead.

- The second problem is that for some entities the speed is directly connected to the frequency. The most obvious example that we have seen so far is the toy tortoise. Its speed is dependent on how fast the wheels rotate, yet how fast the wheels rotate also dictates its frequency. The faster the wheels rotate, the faster it moves and the higher the frequency of the movement of its head. A similar problem occurs with the wood pigeon – the faster the pigeon flaps its wings, the faster it moves. Again, we could still use speed shift keying with such entities, but not at the same time as frequency shift keying.

- The third problem is that altering the speed of most entities takes time to do. Entities generally cannot change their speed instantly. However, depending on what is creating the wave, this problem can also apply to the amplitude, frequency, phase and mean level of a wave.

- The fourth problem is that it can be difficult or impossible to alter the speed of some entities. Technically, we could alter the speed of sound by changing the temperature of the air through which the vibrations are travelling. We could alter the speed of light by changing the medium through which the light is travelling. However, the effort in doing either of these would probably not be worthwhile.

If the wave is of a type where:

- the entity creating the wave is moving
- we can control the speed of the entity
- the temporal frequency is independent of the speed
- we can observe the fluctuations and the speed without needing to measure them from a fixed place

... then the speed and the temporal frequency will be independent of each other. In such cases, varying the speed will not affect the real or perceived temporal frequency, and speed can then be considered independent of the other characteristics of the wave. We could then use speed shift keying as well as frequency shift keying, amplitude shift keying, phase shift keying and mean level shift keying all at the same time.

Looking at the types of waves from Chapters 31 to 34, we could not use speed shift keying as an independent type of shift keying with:

- Sound. This is because we need to measure sound from a fixed place, and so it is subject to the Doppler effect. Another reason is that we cannot easily control the speed of sound.
- Light. This is because we cannot easily control the speed of light.
- The toy tortoise – The wheels of the toy tortoise are connected to its neck, so as it increases in speed, its temporal frequency increases.
- The toy steam train. The steam train is the same as the toy tortoise.
- The wood pigeon. The wood pigeon's speed is directly related to how fast it flaps its wings. It is analogous to the toy tortoise. Therefore, its temporal frequency is related to its speed. However, if we could alter the movement of the air through which it travelled, we could treat the speed as an independent attribute.

- The corrugated metal sheet. The measurements are made at a fixed place that the metal sheet passes. Therefore, increasing the speed will increase the perceived temporal frequency.

We *would* be able to use speed shift keying as an independent form of shift keying with:

- The beach ball. The beetle moving around the beach ball moves around the ball with a frequency that is independent of the speed that the beach ball moves through space.
- The mechanical device with the rotating arm. The rotation of the arm is independent of the speed of the mechanical device.
- The toy tortoise if it had a motor that controlled the movement of its neck independently of the movement of the wheels.
- The toy steam train if it had a motor that controlled the movement of the chimney independently of the movement of the wheels.

More on shift keying

Modulation of a wave to convey information does not have to be through the radio band of the electromagnetic spectrum, or even with electromagnetic radiation at all. We can modulate any phenomenon that has a wave characteristic that can be controlled. Of course, the main advantage of modulating radio waves is that radio waves travel a long distance, allowing messages to be sent to people far away. Modulating the wave characteristics of, say, a stationary spinning wheel would not be as useful for sending messages [although it might be useful if someone could see the wheel from a distance]. Other waves might not be as useful as radio waves, but they could still have some uses.

With some types of wave, it is much harder, or impossible, to control individual aspects such as phase, but it is always possible to use on-off keying.

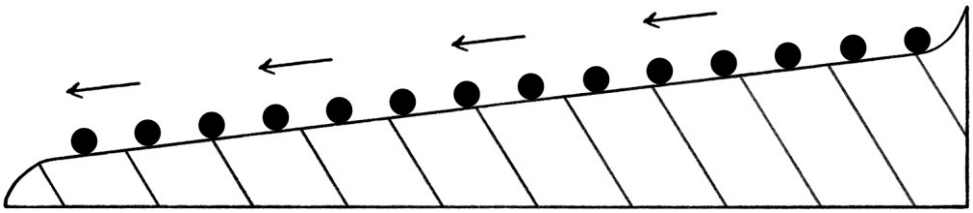
The most obvious phenomena that can be used to send messages using modulated waves are electromagnetic radiation, voltage and current in cables, and sound. With all three we could perform ASK, FSK, and PSK. With voltage and current in cables, we could also perform MLSK.

The device with the spinning arm in Chapter 32 could send messages using ASK, FSK, PSK, two simultaneous versions of MLSK (vertical and horizontal), and speed shift keying.

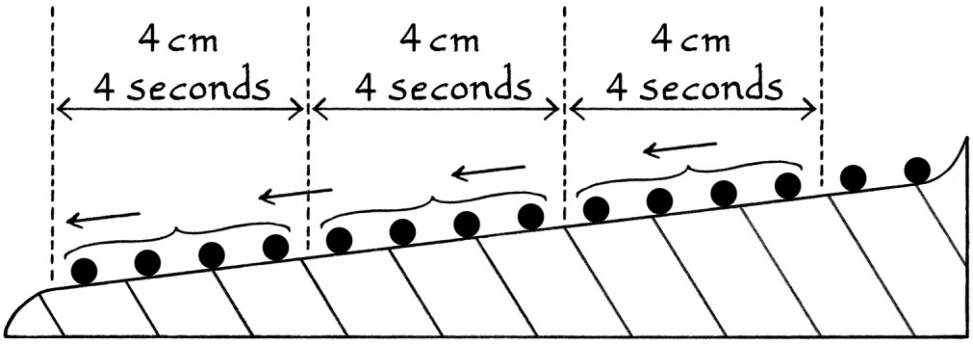
Shift keying using marbles

Any regularly occurring event can be modulated to encode data if it is possible to control it. [Not all shift-keying types apply to every situation though.] As an example, we will look at using marbles to perform various types of shift keying. In this way, marbles are analogous to the peaks of waves.

We will imagine a series of evenly spaced marbles rolling down a slope:



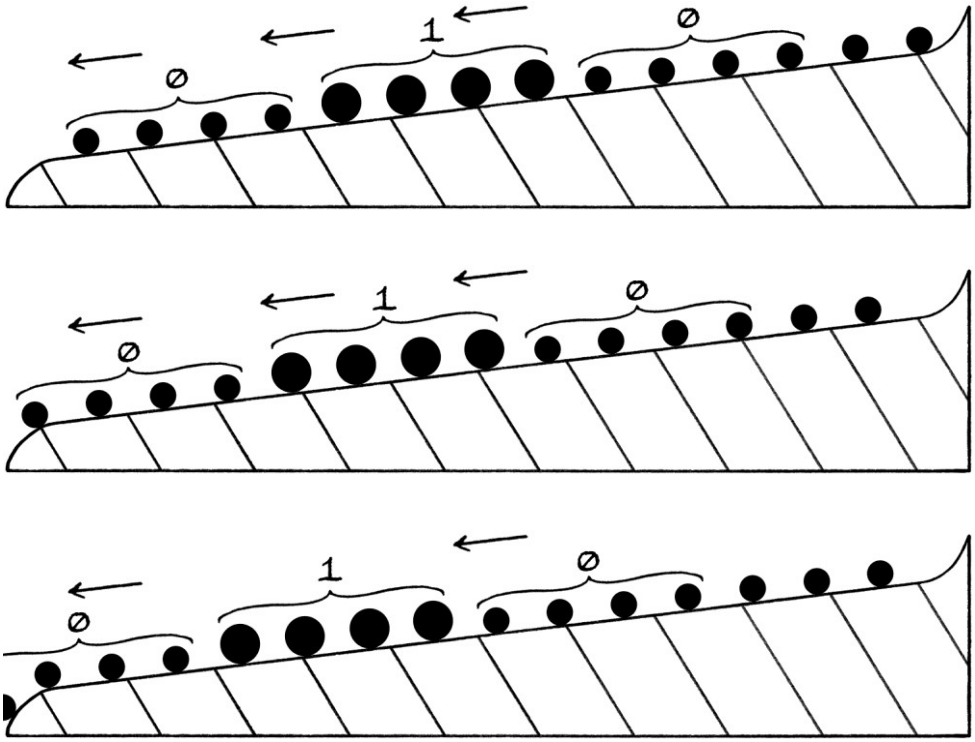
We will say that one marble is released every second, and that the slope is gentle enough and long enough that at any one time we can see several marbles on the slope. In the default state, there are four marbles in one 4-centimetre section of the slope. In other words, 4 seconds worth of marbles take up 4 centimetres.



For our shift keying, we will look at groups of marbles in four-second periods or over four-centimetre lengths. The characteristics of a group of marbles over four seconds will indicate the state of indicating a one or a zero. By default, there are four marbles in a four-second group.

ASK

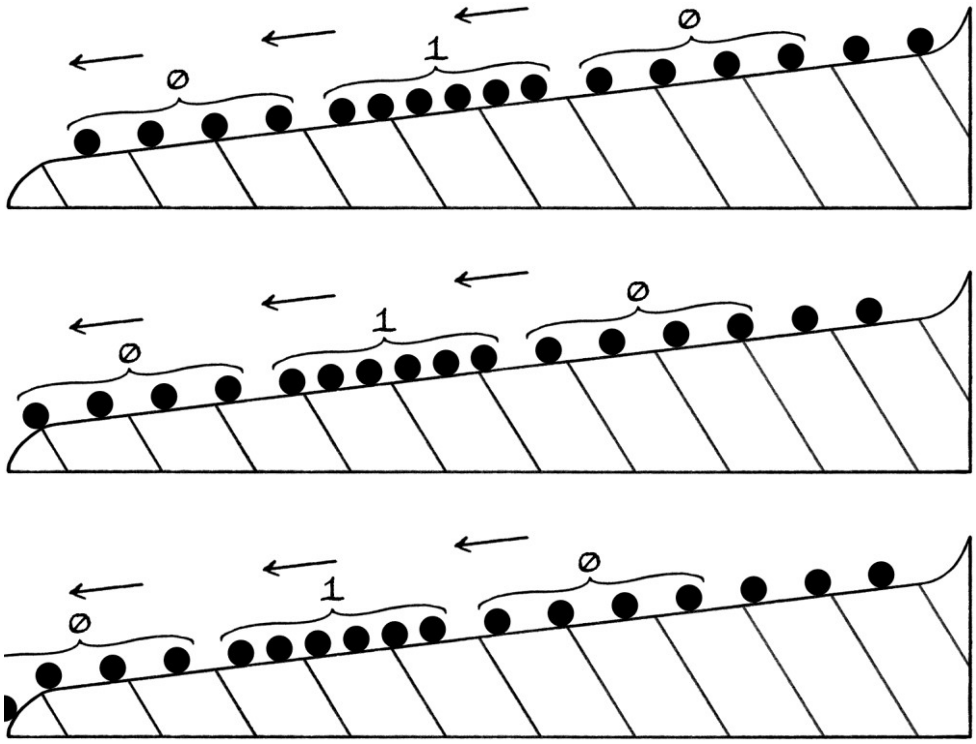
First, we will use the marbles to perform amplitude shift keying. The marbles will be released at the original steady rate of one every second, but to indicate a "1", we will use larger marbles for the four second period of each state. Therefore, four seconds' worth of large marbles indicates a "1", and four seconds' worth of normal marbles indicates a "0". The binary digits "010" would be portrayed as so:



FSK

For frequency shift keying, we can have the rate of marble release as one every second (a frequency of 1 marble per second) to indicate a "0", and 1.5 every second (a frequency of 1.5 marbles per second) to indicate a "1". The marbles will be closer together when indicating a "1" than they are when they are indicating a "0". This means that we will have four seconds' worth of close marbles to indicate a "1", and four seconds' worth of normally spaced marbles to indicate a "0".

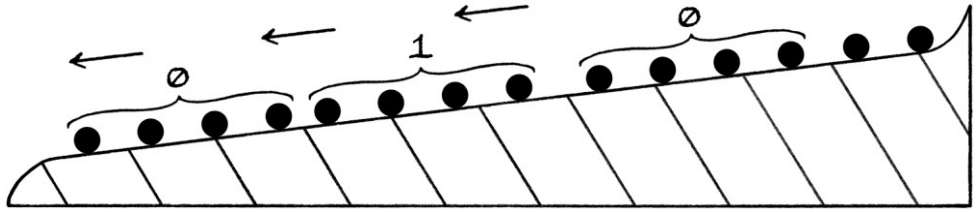
The binary digits “010” would be portrayed as so:

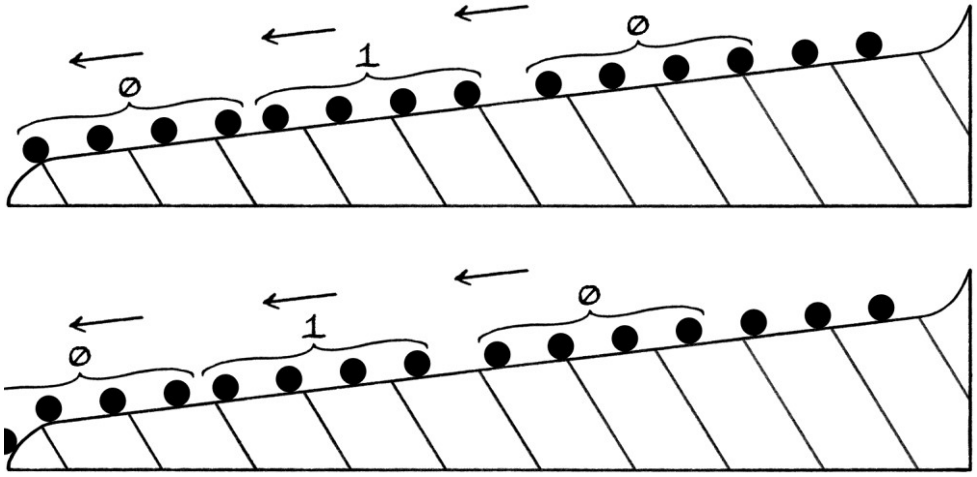


We can observe the frequency either by timing when each marble is released or by looking at the whole slope of marbles.

PSK

We can also use phase shift keying with the marbles. If we are switching from a “0” to a “1”, we will reduce the gap between one four-second batch of marbles and the next. If we are switching from a “1” to a “0”, then we will increase the gap. On the slope, the marbles will still be spaced at one every second, except at the points when the state changes from “1” to “0” or “0” to “1”. The binary digits “010” would be portrayed as so:





One thing to notice is that we would have no way of knowing if the first batch of marbles represented a phase meaning a "0" or a phase meaning a "1". There would be nothing with which to compare it. We could solve this problem by always starting with a series of default marbles before the message starts, or by using a "reference" slope. We could have a second slope with marbles being released at the same rate, and rolling at the same speed, and for which there are never any phase changes. We could then compare the marbles on our reference slope with those on our actual slope – if they are in the same position, then we know the phase on the actual slope represents a "0"; otherwise, it represents a "1".

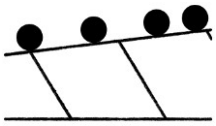
This example of phase shift keying is a good way of noticing a difficulty occurring in all phase shift keying. If we had enough consecutive seconds of ones or zeroes, we would not be able to tell whether we were in a lower phase state or a higher phase state by just looking at the slope – all the marbles would be equally spaced. Which state the marbles are in is only obvious when there is a change in state. There are two ways in which we can know the state of the marbles:

- We can observe the changes from one state to another. This method fails if we have a very long duration of one state and have forgotten (or did not know) which state we were in to start with.

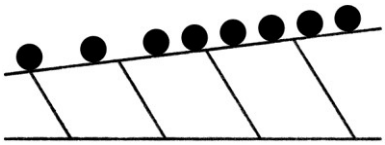
- We can have a reference slope that constantly shows one phase state.

The marble idea also helps show that phase shift keying and frequency shift keying have similarities. If an observer has not been keeping track of the marbles that have passed, they might mistake the second marble in a frequency shift with the first marble in a phase shift and vice versa.

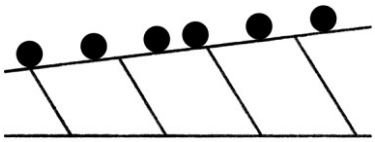
With no context, the following picture could be a change in frequency starting at the second-last marble, or it could be a change in phase starting at the last marble:



If we had the rest of the marbles, we could tell which was happening. A change in frequency would look like this:

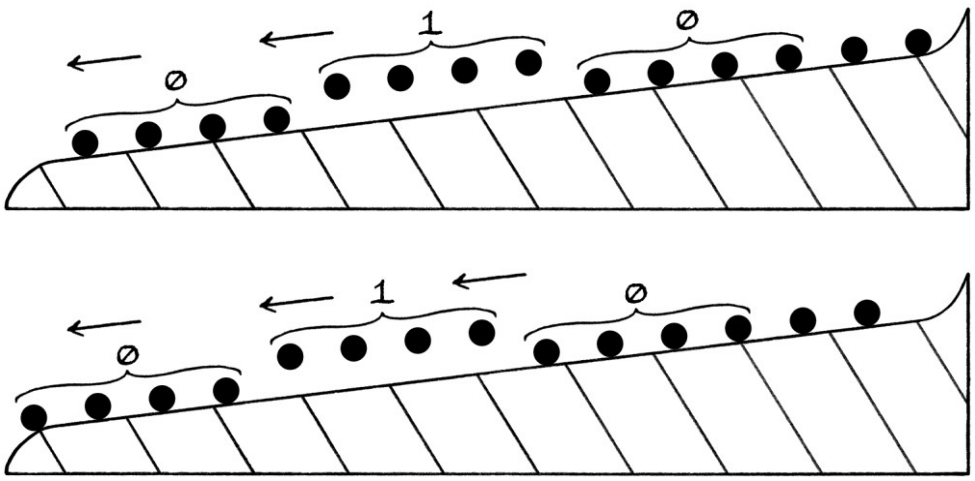


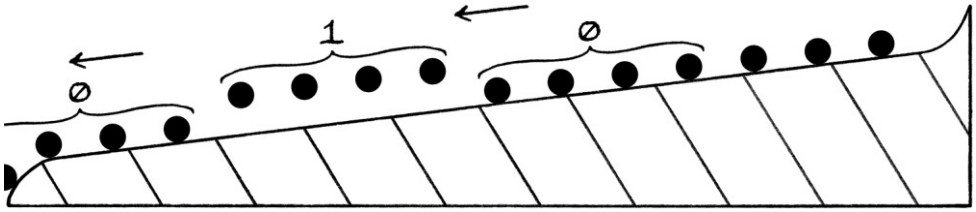
A change in phase would look like this:



Mean level shift keying

We can perform mean level shift keying by having marbles that can float in the air. [We could say that they are magic marbles or that jets of air keep them in the air]. A four-second batch of floating marbles indicates a "1"; a four-second batch of marbles that are not floating indicate a "0". The binary digits "010" would be portrayed as so:

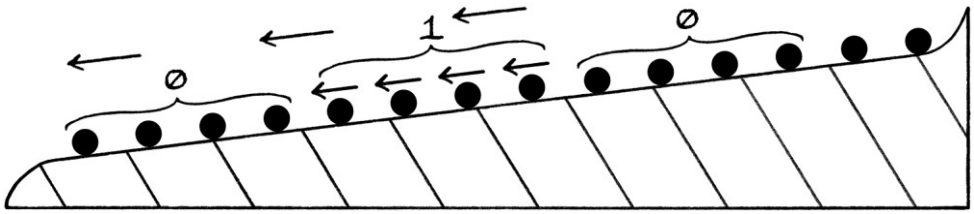


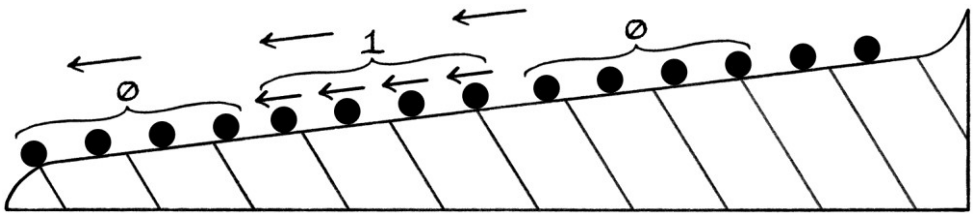


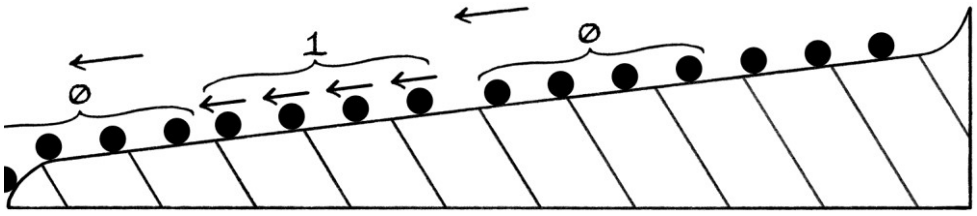
Speed shift keying

We could also portray speed shift keying, but this requires more thought. To do this without it being the same as frequency shift keying, we have to keep the *release* of the marbles at the same rate, but have the marbles roll down the slope at two different speeds that we can control. Therefore, we will say that we can control the speed of the marbles – maybe we have jets of air that propel them or maybe some marbles contain spinning weights that make them roll faster. We will say that the marbles that indicate zeroes move at 1 centimetre per second, and that the marbles that indicate ones move at 1.01 centimetres per second. This means that in practice, it would be harder to distinguish between the speeds, but, on the other hand, there would be less chance of the faster marbles bumping into the slower ones.

At any one moment in time, the "faster-speed marbles" on the ramp would have the same appearance as those representing phase shift keying – there would be four-second long groups of marbles with longer or shorter gaps separating the groups. The difference between the two types of shift keying would be that the *speed* of a group of marbles rolling down the ramp would be different. In the following pictures, the faster marbles are indicated with little arrows above them.







When we introduce speed shift keying, we have to change the way we observe the frequency shift keying and phase shift keying. To distinguish between speed shift keying and frequency shift keying or phase shift keying, we need to observe the marbles on the slope over time – we cannot distinguish them from a static picture.

All shift keyings

We can combine the shift keyings so that all five systems are used at the same time. We can use:

- Amplitude shift keying by varying the size of the marbles for a four-second batch of marbles.
- Frequency shift keying by varying the number of marbles released per second for a four-second batch of marbles.
- Phase shift keying by increasing or decreasing the gaps between the batches of marbles.
- Mean level shift keying by varying whether the marbles in a four-second batch float above the slope or not.
- Speed shift keying by varying the speed of the marbles in a four-second batch when they are on the slope.

We will make a table to say which binary numbers are represented by a particular state. In the table, when a particular state represents a “0”, it means that it is the default state. For example, if the amplitude is 0, it refers to the standard sized marble; if the amplitude is 1, it refers to the larger size of marble. The binary number we are representing is based on whether the different possible attributes are changed from normal or not.

Speed	Mean level	Phase	Frequency	Amplitude	Full binary number
0	0	0	0	0	00000
0	0	0	0	1	00001
0	0	0	1	0	00010
0	0	0	1	1	00011
0	0	1	0	0	00100
0	0	1	0	1	00101
0	0	1	1	0	00110
0	0	1	1	1	00111
0	1	0	0	0	01000
0	1	0	0	1	01001
0	1	0	1	0	01010
0	1	0	1	1	01011
0	1	1	0	0	01100
0	1	1	0	1	01101
0	1	1	1	0	01110
0	1	1	1	1	01111
... and so on until:					
1	1	1	0	1	11101
1	1	1	1	0	11110
1	1	1	1	1	11111

Using just one of the shift keying types on its own, we would be able to encode only one bit at a time – either a one or a zero. By using all five, we can encode 5 bits in one go.

As an example of using this in practice, we will encode the following binary number:
010101010111111

First, we split this into 5-bit groups:
01010
10101
11111

To portray the first 5-bit binary number, 01010, we need:

- The speed of a four-second batch of marbles to indicate a “0”. In other words, it is the default speed.
- The mean level of that four-second batch of marbles to indicate a “1”. In other words, the marbles will be floating in the air.
- The phase of that batch to indicate a “0”. In other words, it will be the default phase.
- The frequency of that batch to indicate a “1”. In other words, there will be more marbles in that batch than usual.
- The amplitude of that batch to indicate a “0”. In other words, the marbles will be the standard size.

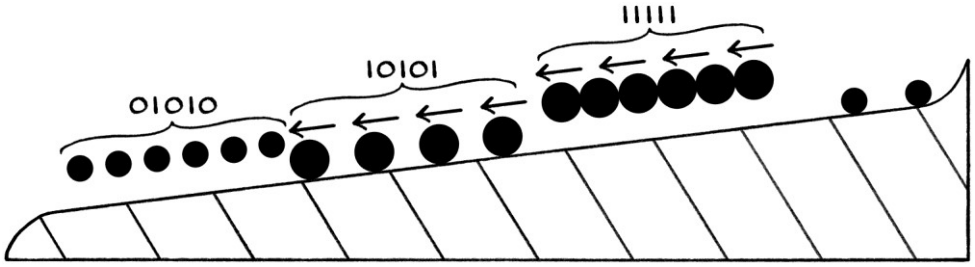
To portray the second 5-bit binary number, 10101, we need:

- The speed of a four-second batch of marbles to indicate a “1”. In other words, this batch will move at a faster speed.
- The mean level of that four-second batch of marbles to indicate a “0”. In other words, the marbles will be touching the slope.
- The phase of that batch to indicate a “1”. In other words, the marbles in this batch will all be slightly closer to the previous batch.
- The frequency of that batch to indicate a “0”. In other words, the marbles will be released at the normal rate.
- The amplitude of that batch to indicate a “1”. In other words, the marbles will be of a larger size.

To portray the third 5-bit binary number, 11111, we need:

- The speed of a four-second batch of marbles to indicate a “1”. In other words, this batch will move at a faster speed.
- The mean level of that four-second batch of marbles to indicate a “1”. In other words, the marbles will be floating above the slope.
- The phase of that batch to indicate a “1”. In other words, the marbles in this batch will all be slightly closer to the previous batch.
- The frequency of that batch to indicate a “1”. In other words, there will be more marbles in that group.
- The amplitude of that batch to indicate a “1”. In other words, the marbles will be of a larger size.

The fully encoded digits would be portrayed as so:



[Note that in this picture, there is the risk that the combination of the phase shift and the speed shift will make the second batch of marbles catch up with the first batch. Ideally, there would be a greater gap between all of the marbles, but this is harder to fit into a drawing.]

Other uses

If we ignore speed shift keying and MLSK, we could use the ideas in this section to encode information into something as simple as a string of beads.

Shift keying thoughts

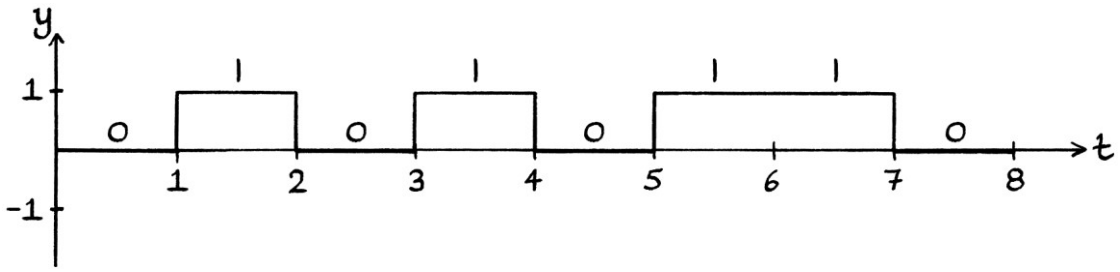
When it comes to shift keying, the signal needs to stay in a “1” state or a “0” state for a minimum number of cycles for the changes in state to be detectable by the receiver. How many cycles we need depends on many factors, including how the signal is being decoded, atmospheric conditions affecting the signal (if it is a radio or sound signal), the frequency, and some mathematical rules. An obvious example of how there is a minimum number of cycles relates to how if a person were decoding the signal by hand, they would not be able to detect a change in, say, amplitude if that change lasted for only a thousandth of a second. On the other hand, they would be able to detect a change if it lasted for a second. A computer might be able to detect changes in amplitude if they lasted for just one cycle (which at 100 MHz would be 0.00000001 seconds), and also for partial cycles, although there would be a minimum length of partial cycle that could be detected. [Changing the amplitude halfway through a cycle would create more unwanted frequencies as the signal would ultimately be the sum of more pure waves of different frequencies.]

Automation

It would be rare for the three main types of shift keying (ASK, FSK, PSK) to be performed by hand. It is more likely that a computer or electronics would control the amplitude, frequency or phase of a carrier wave. In which case, the process would be performed by creating a square wave that represented the binary digits, and that in turn would be multiplied by, or added to, the carrier wave to create the final signal. The actual process might involve such a square wave, or the encoding might be done in one go without a square wave, in which case, it still essentially uses the *idea* of a square wave. In this section, we will look at square waves encoding binary digits. The reason we need to know all of this will become clearer when we use the square waves to encode data into waves.

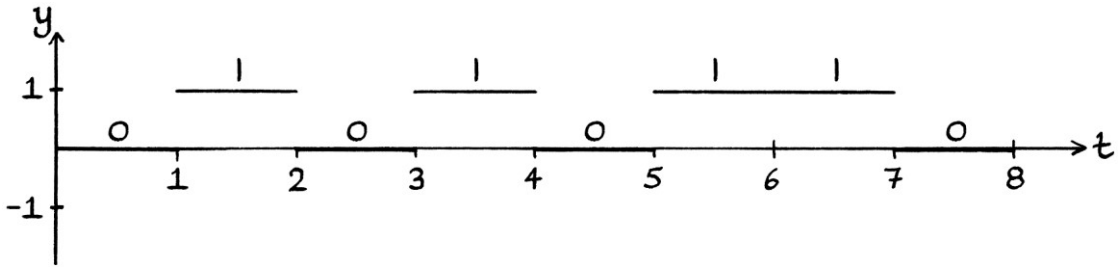
Square waves

We will say that we have this binary number:
01010110
... and we want to encode it with shift keying. To encode it well, we would first need to create a square wave based on these binary digits. The square wave would portray the bits and also the timing of when the bits change. One possible way of portraying the digits is as so:



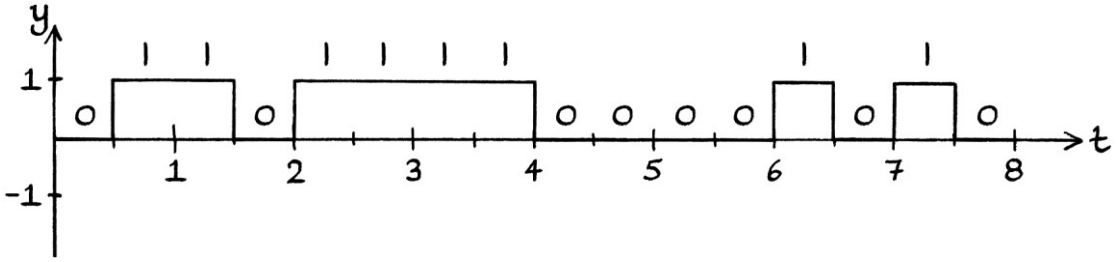
When the square wave is at its highest (1 unit), it is representing a one; when it is at its lowest (0 units), it is representing a zero. This square wave has a mean level of 0.5 units, instead of 0 units, so that it can be useful for encoding shift keying. The square wave is a representation of the binary number, and most importantly, it is a representation that incorporates timing. In this particular square wave, each state lasts for one second. In other words, a “1” lasts for one second, and a “0” lasts for one second.

Although I have drawn the square wave with vertical lines connecting the horizontal lines, it could just as easily be drawn without the vertical lines:



This square wave can be used to alter the frequency, amplitude or phase of a pure wave. We do this with the multiplication or addition of an attribute of a pure wave with our square wave. We will see these processes later in this chapter. There are countless other ways we could set up this square wave depending on what we want to do and how we want to do it. For example, the states could change more quickly or more slowly, the amplitude could be higher or lower, and the mean level could be higher or lower, and so on.

As another example, the following square wave contains the binary number:
0110111100001010
... where each digit stays the same for just half a second:

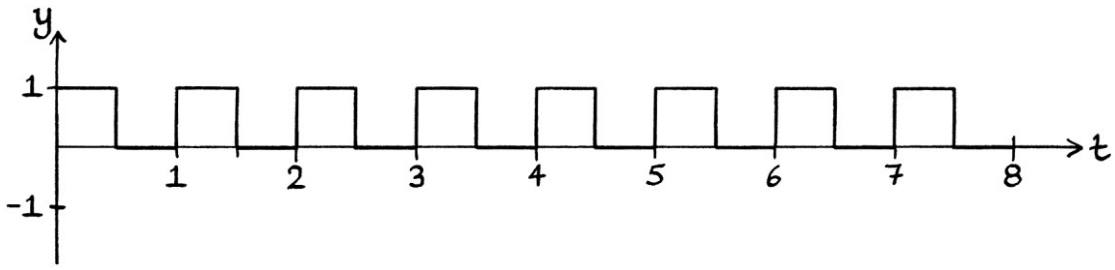


Having the binary number in the form of a square wave greatly simplifies some aspects of shift keying, and it is also a helpful step towards understanding other types of modulation.

Clocks

Converting a sequence of bits into a square wave with timing is called “line coding”. We could do it by hand, but it is much quicker with a computer or electronics.

The standard method of turning a sequence of bits into a square wave involves what is called a “clock”. A clock, in this sense, is a digital counter that changes its state from 1 to 0 and from 0 to 1 at fixed intervals of time. An example of the output of a very slow clock is as follows:



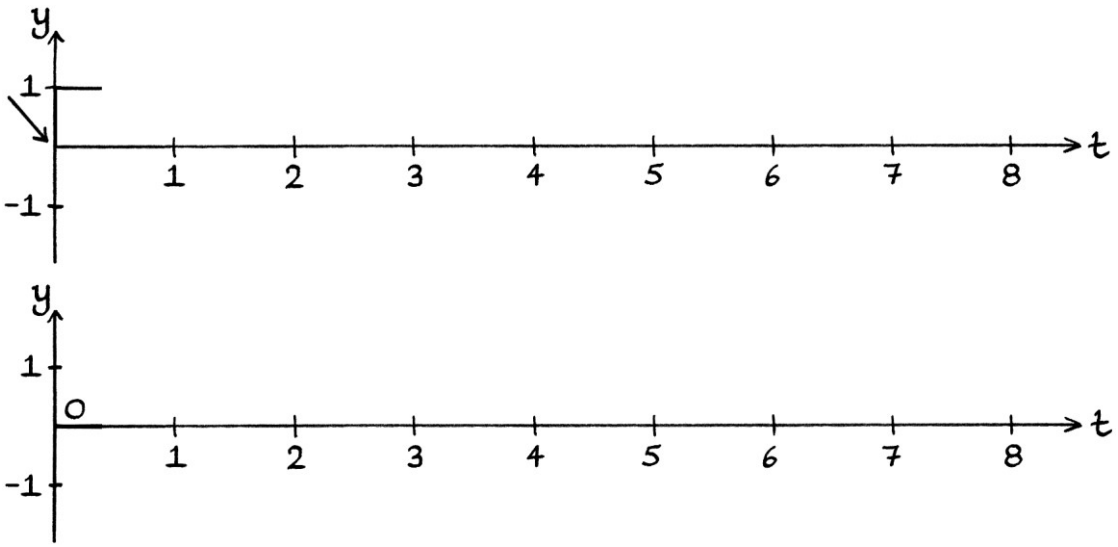
This particular clock moves from zero to one and back once every second. Although it changes its state (from 0 to 1 or 1 to 0) twice every second, the significant aspect is that it completes one cycle every second. The clock shown in the above picture first moves from zero to one at 0 seconds, which is essentially the same as saying that it starts at 1.

Clocks appear in computers and digital electronics where they are the ultimate controller of time based events. In a computer processor, for example, every instruction is executed according to the timing of the state changes of the clock. Every event takes place according to the beat of the clock. In a modern computer processor, a clock might change its state billions of times a second. For the purposes of shift keying, the clock can be one that changes much more slowly – say once every tenth of a second or so.

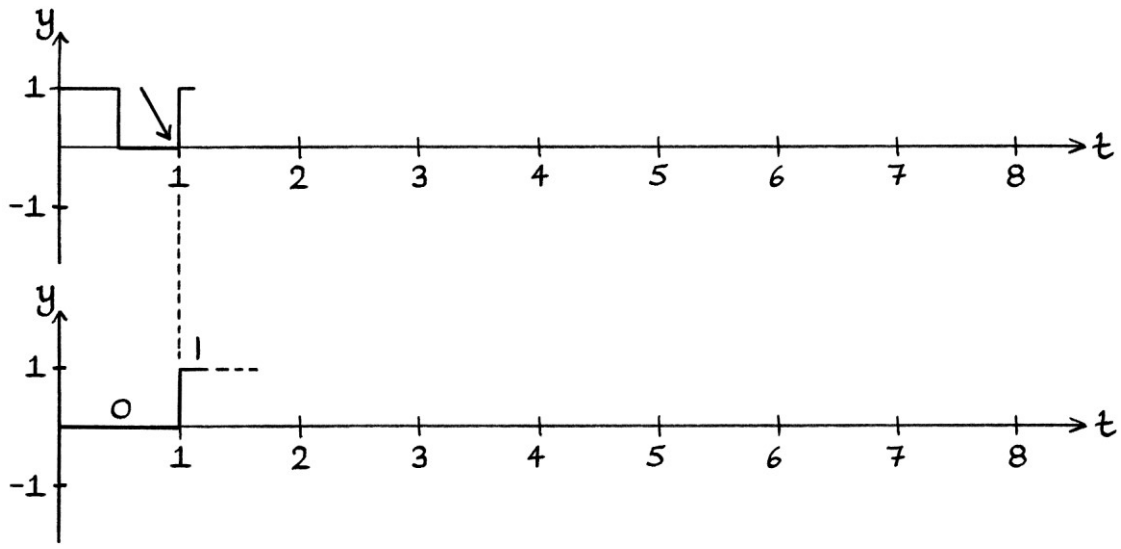
To create our square wave, we will say we have a clock that changes state twice every second. [A state change is a change from 0 to 1 or 1 to 0, which means that the clock wave completes one cycle in 1 second, so it has a frequency of 1 cycle per second.] A computer or electronics will read one bit from our number every time the clock changes from 0 to 1 (which happens once every second). The y-axis value of the resulting square wave will be set at either 1 or 0 depending on the digit of our binary number that has just been read. The y-axis values will stay fixed at that level until the next time that the clock changes from 0 to 1 again, which happens after one second.

We will start with the binary number “01010110” and the clock about to start. The clock starts by moving from 0 to 1, which it does at t = 0. Therefore, the computer reads the first bit (“0”) from our number, and sets our output signal to “0”. The output signal will stay at “0” until the clock next moves from 0 to 1, at which time, the next digit will be read.

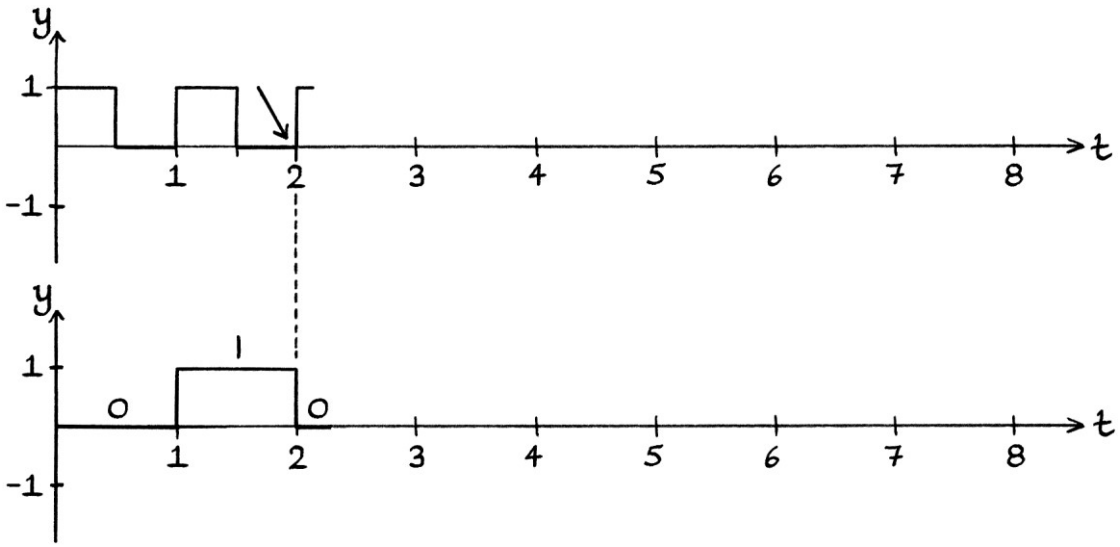
[The first rise of the clock from 0 to 1 is the hardest to visualise because it is not clear that any rising is taking place. Later rises are more straightforward.]



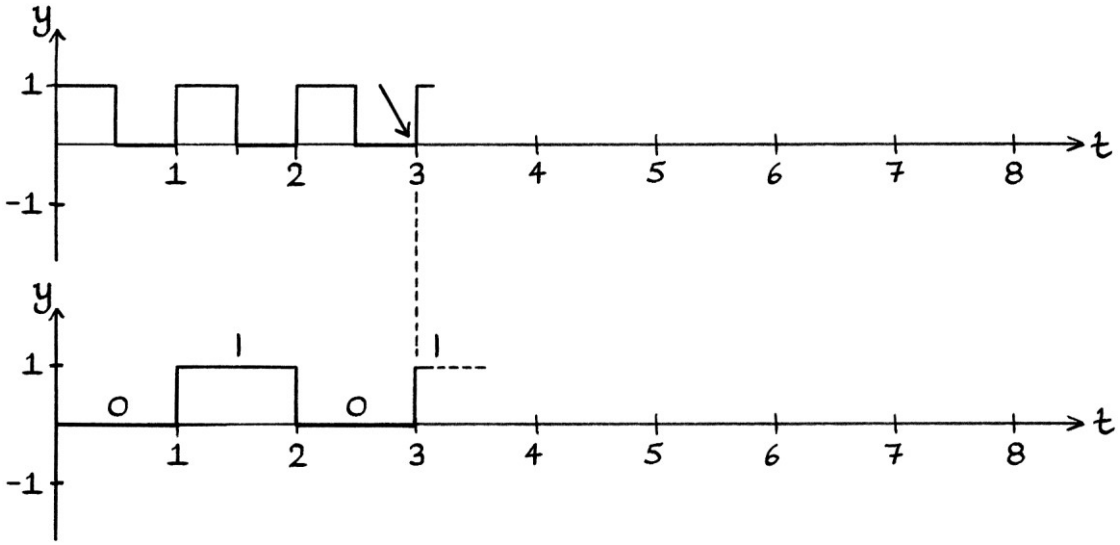
The clock next changes from 0 to 1 at t = 1 seconds. The next digit in our list of digits is “1”. Therefore, the computer sets the value in our output signal to 1 unit. The value remains the same in the output signal until the clock next rises from 0 to 1 again, when the next digit will be read.



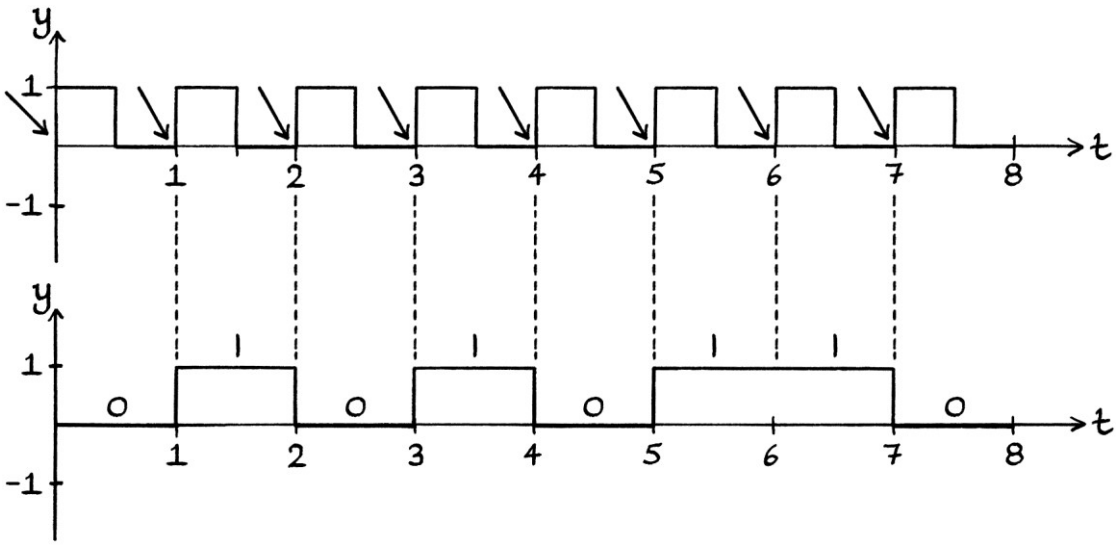
The next time the clock changes from 0 to 1 is at t = 2 seconds. The next digit in our list of digits is “0”. Therefore, the computer sets the value in our output signal to 0 units, and keeps it there until the next time that the clock rises from 0 to 1 again.



The next time the clock changes from 0 to 1 is at t = 3 seconds. The next digit in our list of digits is “1”. Therefore, the computer sets the value in our output signal to 1 unit, and keeps it there until the next time that the clock rises from 0 to 1 again.



The process continues until we have completed our square signal:



We end up with a square wave that portrays our binary number with the bit changes occurring every second.

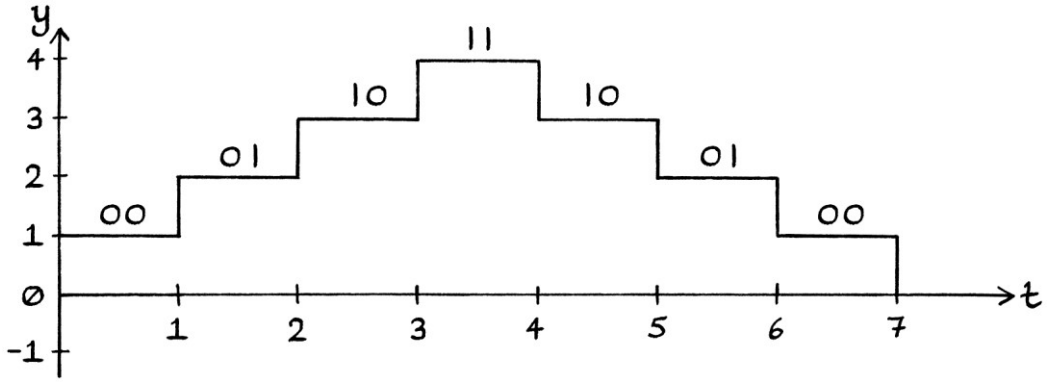
Without using some kind of time-based event, we would not be able to convert a sequence of binary bits into something involving time.

It does not really matter if you do not understand the idea of clocks – you can understand shift keying perfectly well without them. Clocks make more sense when you experience them in a variety of situations. One way to understand them is to try to think up your own method of creating a square wave that indicates the bits of a binary number with each digit evenly spaced over time. You will probably end up with the system as described.

Multi-level square waves

As well as having two-state square waves, we can also have square waves with more than one level. These can be used with multi-level shift keying such as 4-ASK, 4-FSK and 4-PSK.

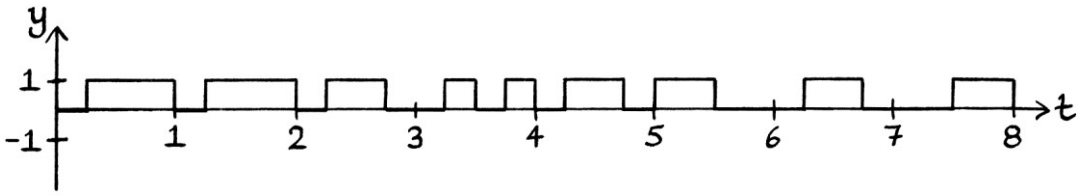
As an example, we could have a square wave such as the following one. This can alter the amplitude, frequency or phase of a carrier wave by addition or multiplication to encode two binary digits into any one state:



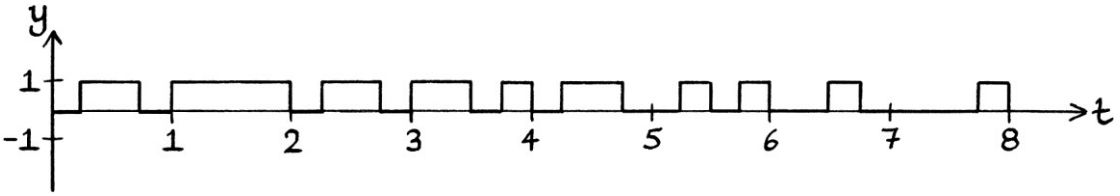
More on square waves

If we were receiving a shift-keyed signal, when we decoded it, we might have one step where we had the binary digits as a square wave. Such a signal might continue for many seconds, and it can be good to get used to such an idea.

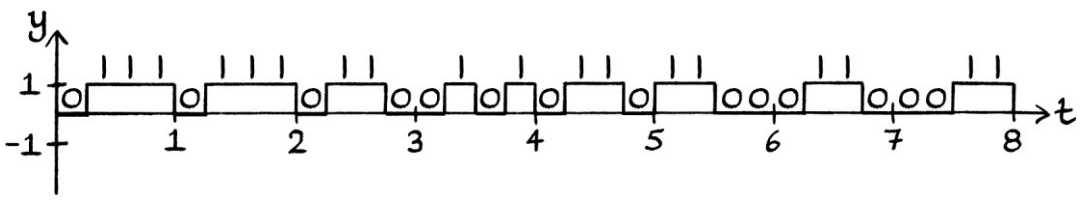
As an example, we might decode a received wave to end up with this square wave:

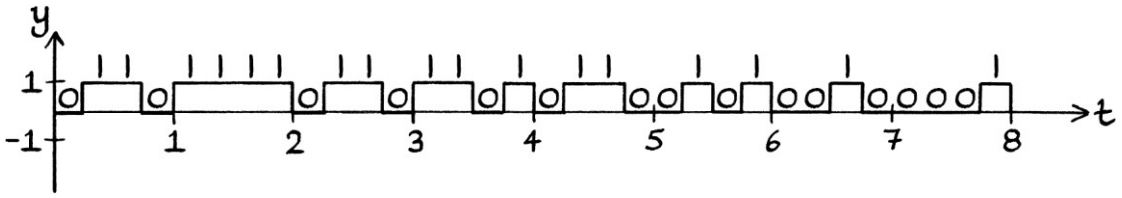


... followed by this square wave:



We will guess the timing of each one and zero, and mark the ones and zeroes on the graphs:





We can then write out the binary digits in full for each graph:
01110111011001010110110001100011
01101111011011010110010100100001

We will split these into groups of 8 bits. [As we will see in Chapter 40 on binary and hexadecimal, by convention, bits are grouped into eights or multiples of eight.] When we put the binary bits into groups of eight, we can more easily spot patterns that will suggest what the numbers mean.

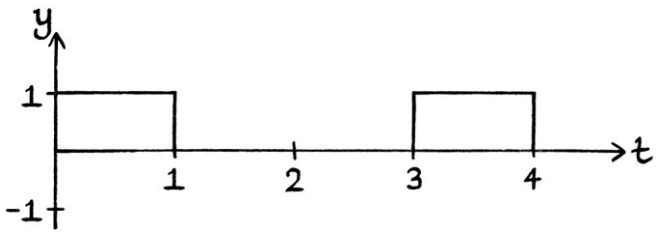
01110111
01100101
01101100
01100011
01101111
01101101
01100101
00100001

Experience will tell us that these groups are valid ASCII characters. Therefore, the binary is readable text. Each of the 8-bit numbers refers to a letter of the alphabet, and we can decode them by looking them up in an ASCII table. I will show an ASCII table in Chapter 40. For now, it is enough to know that the characters for each binary number are as follows:

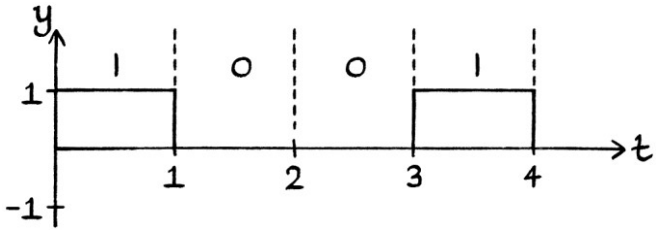
01110111	“w”
01100101	“e”
01101100	“l”
01100011	“c”
01101111	“o”
01101101	“m”
01100101	“e”
00100001	“!”

Therefore, the text in the signal says, “welcome!”

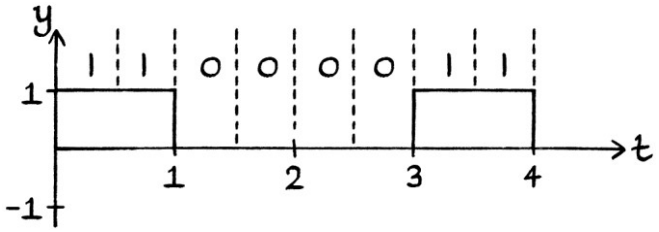
One problem that is easiest to see when using square waves is that if the receiver is not aware of the timing, it can be difficult to know how many ones or zeroes are being described by any section. If we are given this square wave:



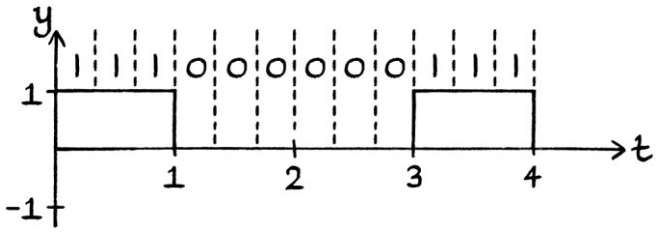
... then we could interpret it like this, with one digit per second:



... or we could interpret it like this, with two digits per second:



... or we could interpret it like this, with three digits per second:

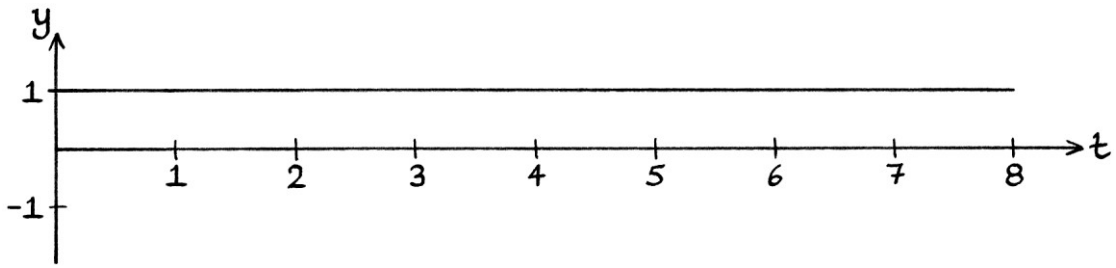


... and so on. The only ways to tell how many bits are being sent per second are:

- To know in advance (perhaps by seeing a preamble).
- To check if the received data makes sense for its context.
- To make an assumption, perhaps incorrectly, that the shortest single state is the length of one digit.

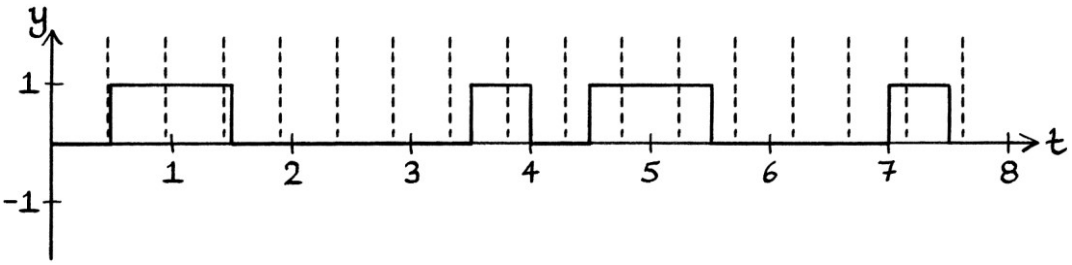
We had the same problem earlier in this chapter when using on-off keying. The problem applies to all shift keying.

A more extreme example is this signal:



The above signal consists entirely of ones, but we cannot tell how many ones without more information.

A similar problem can occur when we have a long message and there is a slight difference in the timekeeping of the transmitter and the receiver. As an example, we will imagine that a transmitter is sending one digit every second, but the receiver has a timer that is running too quickly. The receiver will end up trying to decode impossible states, or end up with nonsense. The following graph shows the signal as it is sent with the time being that of the sender. The dotted lines indicate the timing of the receiver in half seconds. As time goes on, the receiver becomes more and more out of synchronisation with the intended timing. The receiver will not be able to decode the message correctly.



[Clearly in this example, **we** can see the changes in state, but the problems arise when we automate the process.]

Trying to discover the timing of a shift-keyed signal is called “clock recovery”. When decoding a signal, we need to know or work out the behaviour of the original clock that was used to encode it in the first place.

There are several ways to solve timing problems. The most simple is to have a preamble – this indicates how long each digit will be. Alternatively, if we were using phase shift keying, we could use differential phase shift keying where the phase will change with each digit regardless of whether it is the same as the one before or not. This means that we can observe the signal to know the timing. It will not even matter if the timing speeds up and slows down.

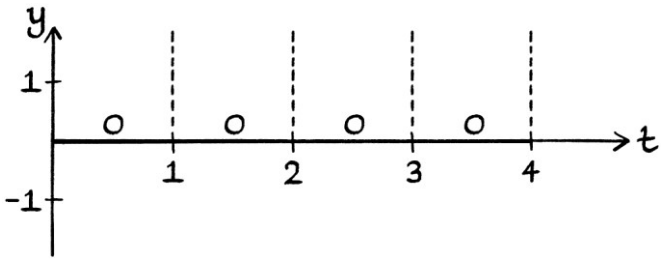
Other ways of solving timing problems involve altering the square wave so that it changes with every digit. Any types of shift keying will then use the altered square wave.

Manchester encoding

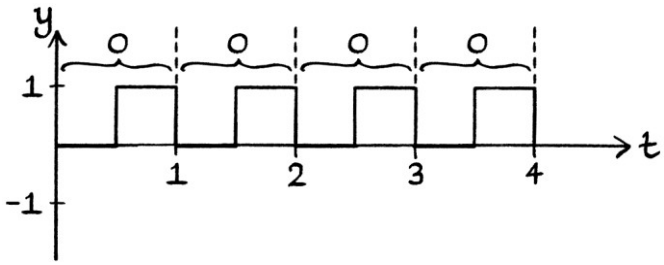
A common system for indicating the timing for the duration of a message is called "Manchester encoding". Manchester encoding changes the state of the square wave halfway through the time for one binary digit depending on the binary digit being sent.

In the following examples, if a zero is being sent in the message, then the actual signal will start at zero and then move to one halfway through. If a one is being sent, the signal will start at one and then move to zero halfway through.

For example, if we were sending the binary digits "0000", the original square wave would look like this:

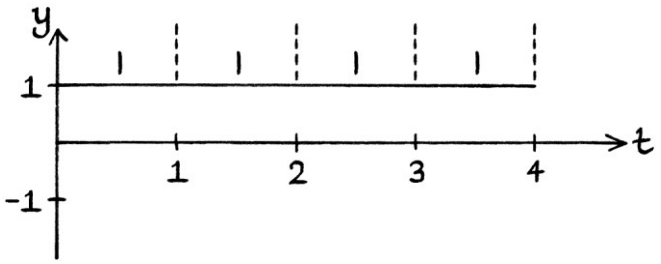


... but the Manchester-encoded square wave would look like this:

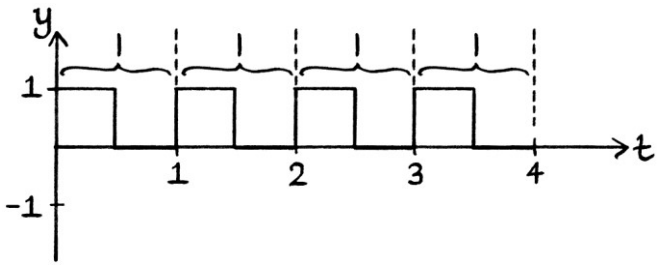


Each zero followed by a one in the Manchester-encoded square wave signifies a zero. In this way, we know that we have a run of sequential zeroes, we know where they start and end, and we know the timing for each digit. Of course, we have to know that we are using Manchester encoding or else, the signal will be misinterpreted as a sequence of pairs of zeroes and ones.

If we were sending the binary digits “1111”, the original square wave would look like this:

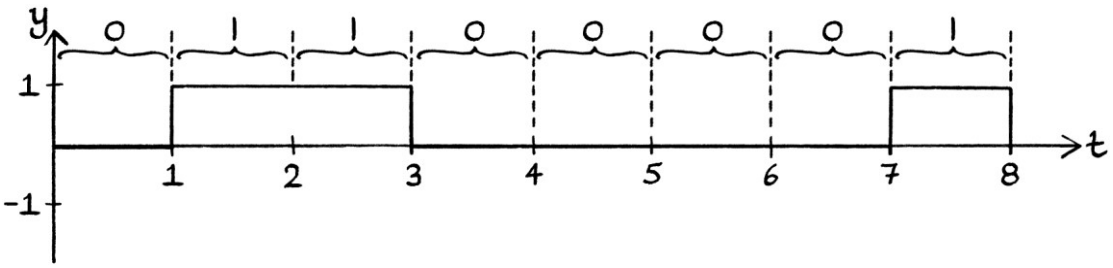


... but the Manchester-encoded square wave would look like this:

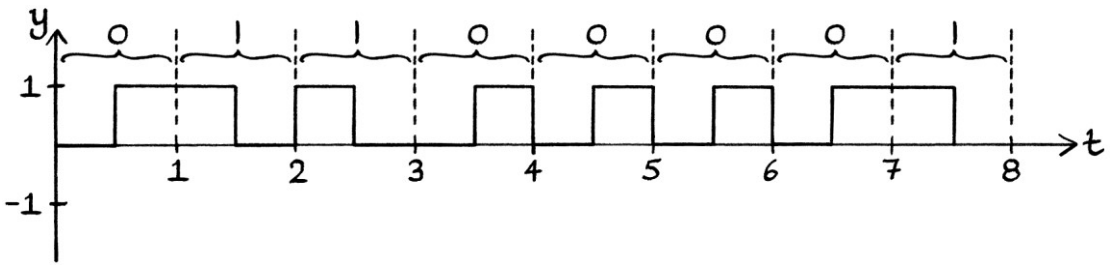


In the above Manchester-encoded signal, each “one” is followed by a “zero”, and each pair indicates that we are intending to send a one.

If we wanted to send the binary digits “01100001”, the original square wave would look like this:

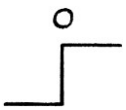


... but the Manchester-encoded square wave would look like this:

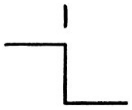


Once the signal is encoded with Manchester encoding, we would then use our desired system of shift keying as normal with the new square wave.

Manchester encoding does not have to switch between zero and one. It might switch between −1 and +1, or between any two states that are useful for the task being performed. We can call the states “low” and “high”. Therefore, in the examples so far, we can say that a zero in the message has been represented by the square wave changing from low to high halfway through the time for that digit:



... and a one in the message has been represented by the square wave changing from high to low halfway through the time for that digit:



To complicate everything, there are two implementations of Manchester encoding in general use. First, there is the type that we have just seen where a zero is represented by a switch from low to high, and a one is represented by a switch from high to low. Second, there is the opposite system, where a zero is represented by a switch from high to low, and a one is represented by a switch from low to high. This means that we need to know, guess, or work out which version of Manchester encoding we are receiving in order to decode such a message.

The disadvantage of Manchester encoding is that it doubles the amount of data that we have to send. The advantage of Manchester encoding is that we remove any possible timing problems. Whether it is useful to use Manchester encoding depends on the situation.

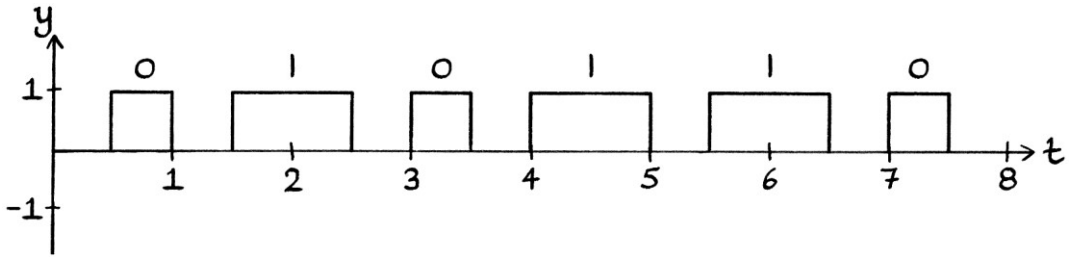
There are numerous alternatives to Manchester encoding, and instead of using any of them, we could indicate the timing of a signal by sending a preamble first. However, the preamble cannot help with very long messages where the sender and receiver's timekeeping might diverge from each other. It is also of no use when the receiver does not see the beginning of the message, as happens with a television receiving a digital television broadcast.

Pulse width modulation

Pulse width modulation ("PWM") is another method of encoding the data that can help with timing. In its most basic form, there is no indication of timing, and the system works by sending a short burst of one state to indicate a zero, and a longer burst of that same state to indicate a one. Between the bursts to indicate zeroes and ones, the signal continues at its default state. Pulse width modulation is essentially the same as a form of Morse code where there are only two letters: a dot and a dash. The dot means "zero" and the "dash" means one.

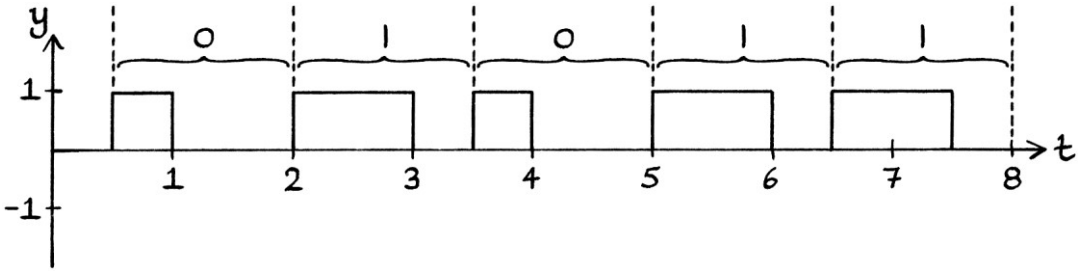
With pulse width modulation, for on-off keying, there might be, say, half a second of signal to indicate a zero, followed by no signal for a moment, then a full second of signal to indicate a one, followed by no signal, and so on. For frequency shift keying, there might be half a second of a particular higher frequency to indicate a zero, followed by the default frequency for a while, followed by the higher frequency for one second to indicate a one, followed by the default frequency, and so on.

The square wave for sending the digits 010110 might look like this:



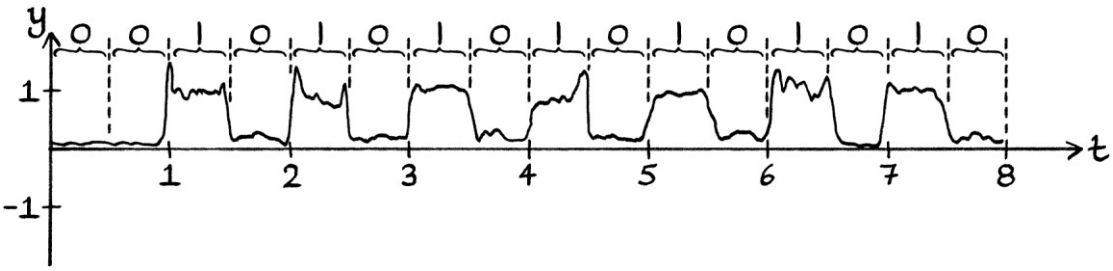
A slightly more complicated version of pulse width modulation keeps the same amount of time for each digit being sent. Therefore, a "zero" will have more space after it to keep it to the same length as a "one". In this way, the method is better for keeping track of the timing.

The square wave for sending the digits 01011 would look like this:



Reality

In practice, if you see a square wave made from *receiving* radio waves or monitoring an electronic circuit, the lines will seldom be as clean as in the above graphs. Such a signal might be as distorted as the one shown here:



Decoded square waves are seldom perfect in real life for many reasons, among which are:

- There can be corruption due to interference before the signal is received.
- There can be imperfections in receiving or transmitting equipment.
- There can be inaccurate measuring equipment.
- There can be imperfect mathematical processes used to analyse the signal.
- The practicalities of real world switching mean that a clean and instant change from one state to another is difficult to achieve.

Automation: OOK

We will now give a purpose to the above discussion of square waves. When we have a square wave containing our binary message, we can use it for automated shift keying.

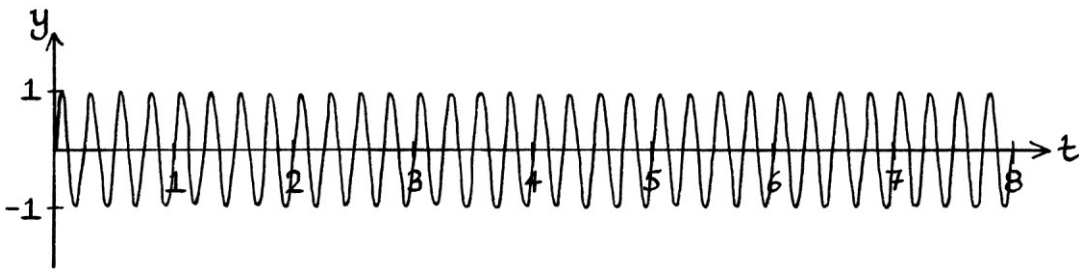
For the following methods of automating shift keying, these are just example methods that *could* be used. In practice, they might not be the best way to control radio waves.

On-off keying

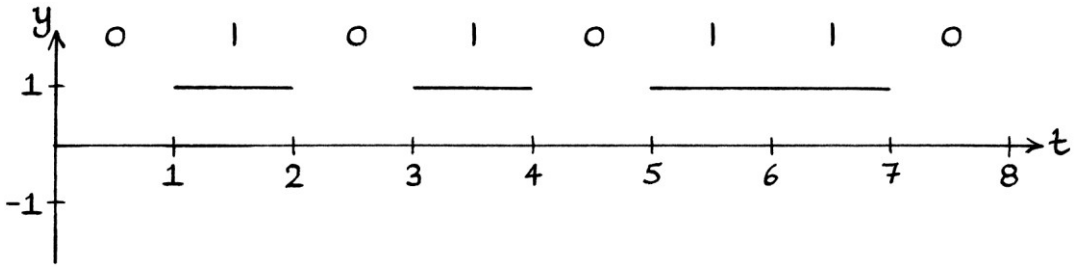
If we were performing on-off keying (OOK), we could create a square wave containing the digits we want to send, where the values switch between y = 0 (to indicate a zero), and y = 1 (to indicate a one). We then multiply the square wave by our carrier wave. In this way, each y-axis value from the square wave would be multiplied by the corresponding-by-time y-axis value from the carrier wave. When the square wave's instantaneous amplitude is zero, the resulting signal will have an instantaneous amplitude of zero. When the square wave's instantaneous amplitude is one, the resulting signal will have an instantaneous amplitude of one.

For this example, we will use a carrier wave with the formula:
"y = sin (360 * 4t)"

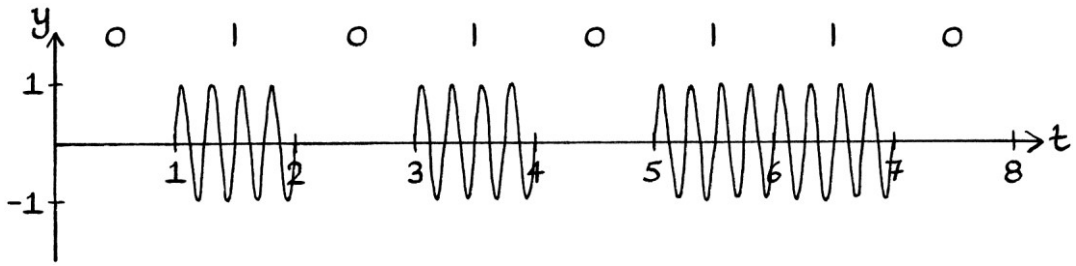
Our carrier wave looks like this:



Our square wave containing the details of our binary digits (01010110) looks like this:



After multiplying the carrier wave by our square wave, we end up with this signal:



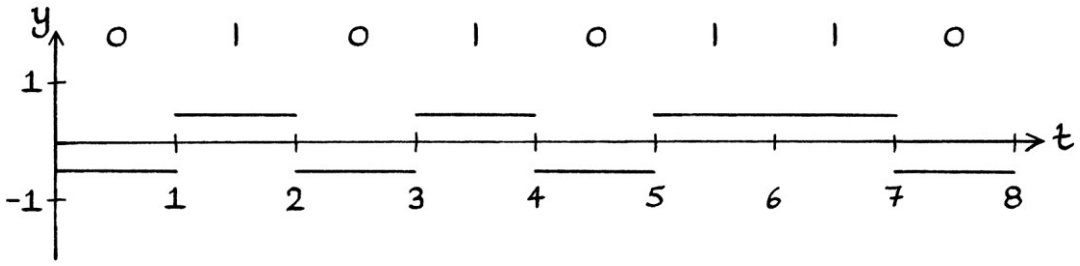
An instantaneous amplitude of 0 units for one second represents a binary 0. An instantaneous amplitude of 1 unit for one second represents a binary 1.

If the y-axis value of the square wave at any particular time is "X" units, then the formula for the resulting signal will be:
"y = X * sin (360 * 4t)"

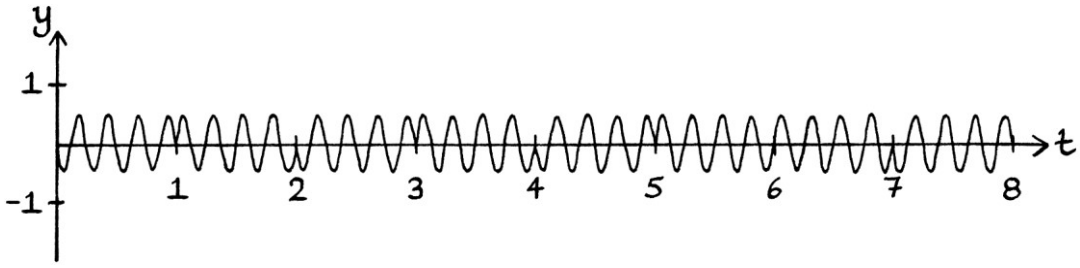
A general formula for the resulting signals in general would be:
"y = X * A sin ((360 * ft) + ϕ)"

In this example, we multiplied the square wave with the whole formula of the carrier wave. For this example, this is the same as multiplying the square wave by the *amplitude* of the carrier wave. This is because the carrier wave had a zero mean level. If our formula had a non-zero mean level, we would need to make sure that that became zeroed too. Therefore, ideally, we should give the formula for the resulting OOK signal as:
"$y = X * (h_s + A \sin ((360 * ft) + \phi))$"

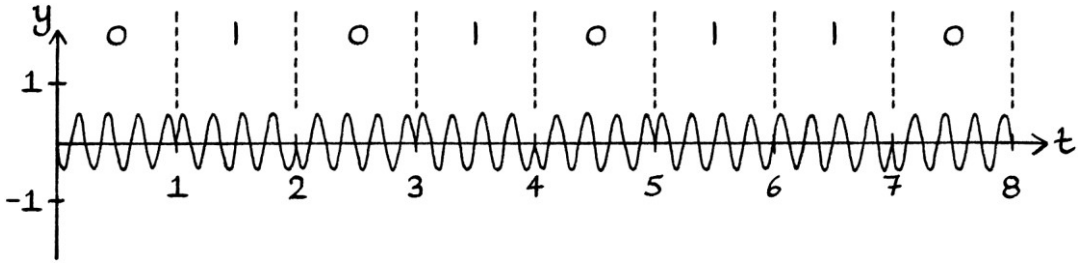
Our square wave switched between zero and one. This means that it has a non-zero mean level. If the square wave had a zero mean level and switched between +0.5 units and −0.5 units, the resulting signal would have a maximum instantaneous amplitude of 0.5 units and a minimum instantaneous amplitude of −0.5 units. This means that there would no obvious change in the resulting signal apart from sections of the signal being flipped upside down when the signal was representing a 1. In this case, the square wave would look like this:



... and the resulting signal would look like this:



We have ended up performing two-state phase shift keying instead of on-off keying. A phase of 180 degrees represents a one, and a phase of 0 degrees represents a zero. With the binary digits marked on the graph, the signal looks like this:

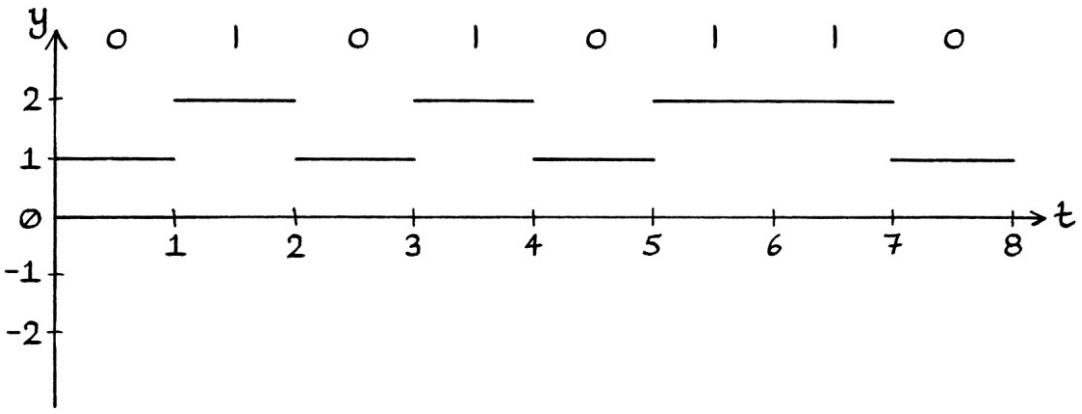


Automation: ASK

ASK with multiplication

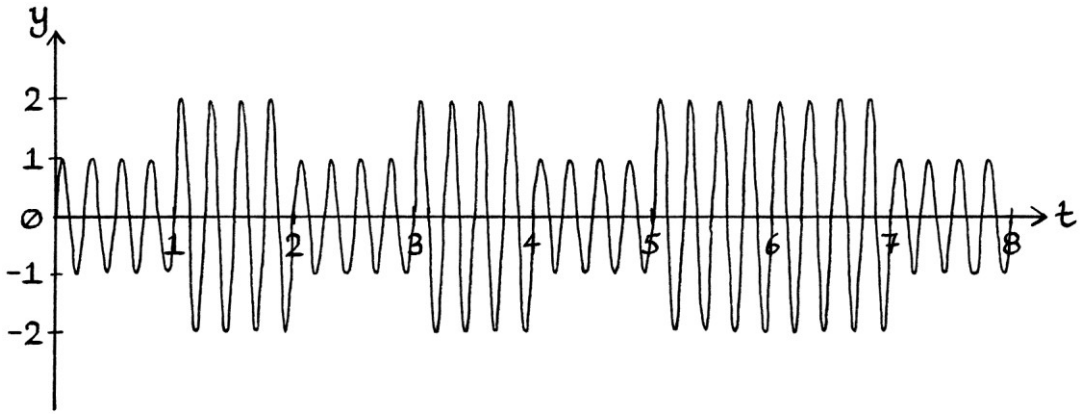
To perform amplitude shift keying, we could use multiplication or addition with a square wave. It is more straightforward to use multiplication, so we will look at that first.

The difference between amplitude shift keying and on-off keying is that for ASK, we need to raise the mean level of the square wave so that the wave is entirely above zero. We do not want the instantaneous amplitude ever to become zero because that would produce on-off keying instead of amplitude shift keying. For this example, the maximum y-axis value of the square wave will be 2; the minimum y-axis value will be 1. Our square wave containing the binary number 01010110 looks like this:

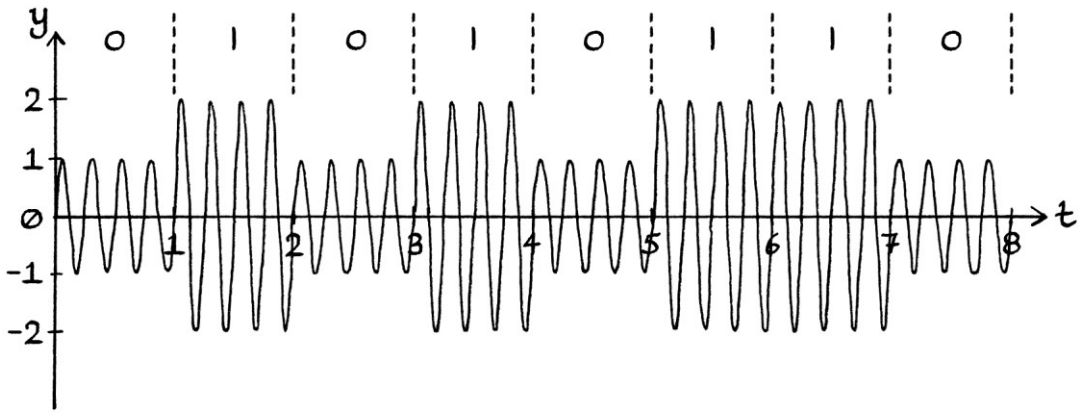


We will multiply the new square wave by the *amplitude* of the carrier wave: "y = sin (360 * 4t)".

We end up with this signal:



An instantaneous amplitude of 1 unit for one second represents a binary 0. An instantaneous amplitude of 2 units for one second represents a binary 1. The graph with the binary digits marked on it looks like this:



As with on-off keying, if the y-axis value of the square wave at any particular time is "X" units, then the formula for the resulting signal in this example will be:
"y = X * A sin (360 * 4t)"

A general formula that takes into account a carrier wave with a mean level is as so:
"$y = h_s + (X * A) \sin ((360 * ft) + \phi)$"

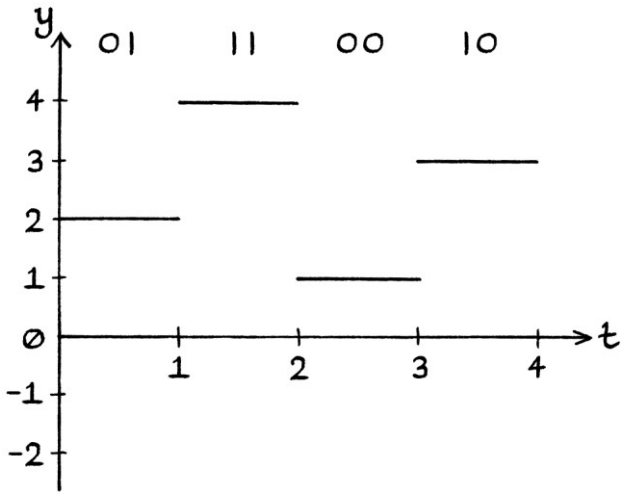
Higher levels of ASK

We can also automate higher levels of amplitude shift keying. We will perform 4-ASK. The most straightforward way to do this is to create a square wave that has 4 levels in it. Each level of the square wave indicates 00, 01, 10 or 11.

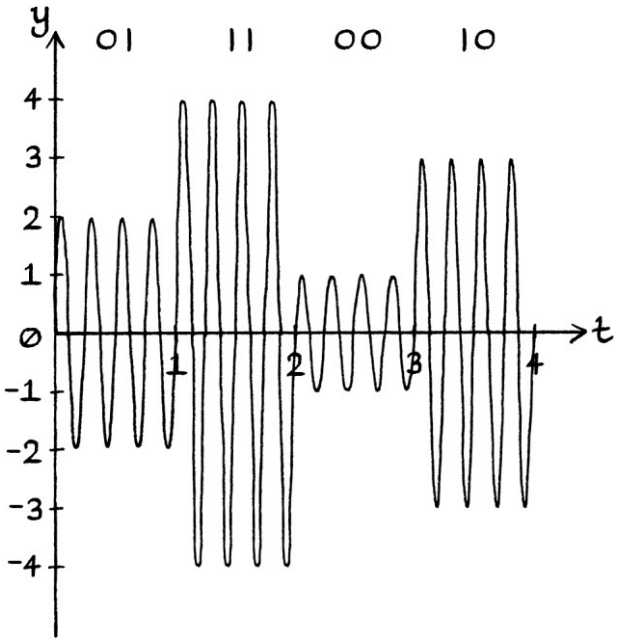
For the following example:

- The binary digits "00" are represented by the instantaneous amplitude of the square wave being 1 unit.
- The digits "01" are represented by an instantaneous amplitude of 2 units.
- The digits "10" are represented by 3 units.
- The digits "11" are represented by 4 units.

We will have a square wave containing the digits “01110010”. It looks like this:



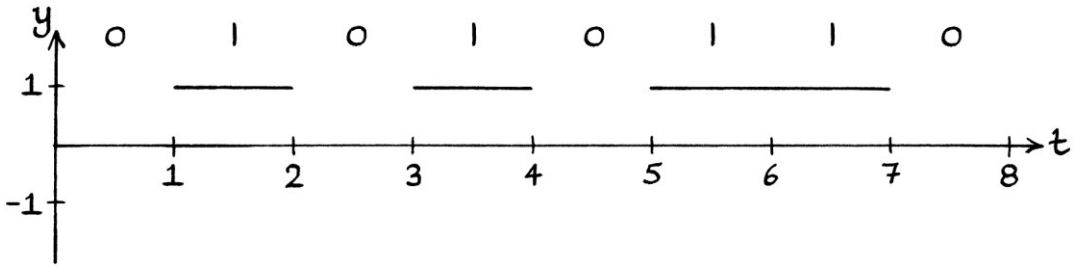
We then multiply our new square wave by the *amplitude* of our carrier wave. The resulting signal looks like this:



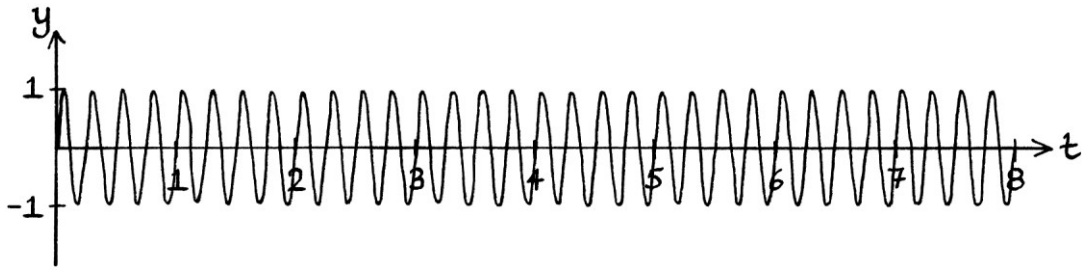
ASK with addition

A slightly more convoluted method of amplitude shift keying involves *adding* the y-axis values of the square wave to the amplitude of the carrier wave. In other words, if the y-axis value of the square wave at any time is “X”, and the carrier wave is:
“y = 2 sin (360 * 4t)”
... then the resulting amplitude-shift-keyed signal will be:
“y = (2 + X) sin (360 * 4t)”

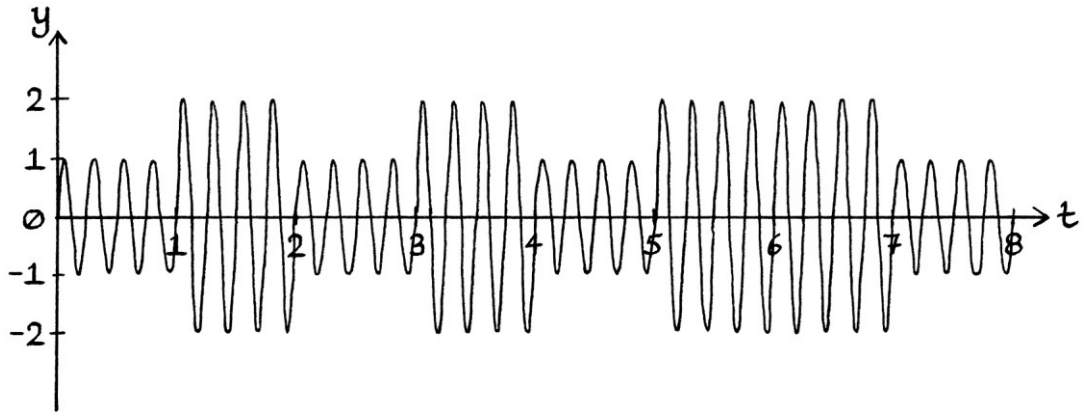
As an example, if we have this square wave that switches between zero and one:



... and we have this carrier wave: “y = sin (360 * 4t)”:



... then the resulting signal will look like this:



This result is identical to when we multiplied a square wave switching between 1 and 2 units with our carrier wave.

At the times when the square wave is at zero, the instantaneous amplitude of the carrier wave has zero added to it, so stays at 1 unit. At the times when the square wave is at one, the instantaneous amplitude of the carrier wave has one added to it, and so it rises to 2 units. Therefore, the carrier wave’s instantaneous amplitude becomes either 1 or 2 units.

We could also scale the y-axis values of the square wave to make the changes in amplitude more noticeable. For example, if we wanted the resulting signal to switch between average amplitudes of 1 and 3 units, we would multiply the square wave's y-axis values by 3 first, and then add them to the carrier wave's amplitude:
"y = (2 + 3X) sin (360 * 4t)"

Scaling the y-axis values before the addition makes the addition method as versatile as the multiplication method.

Whether it is better to use addition or multiplication to achieve amplitude shift keying depends on the type of waves we are modulating and the way the modulation is being executed (for example, with electronics, a computer, a mechanical device and so on).

Automation: FSK

FSK with multiplication

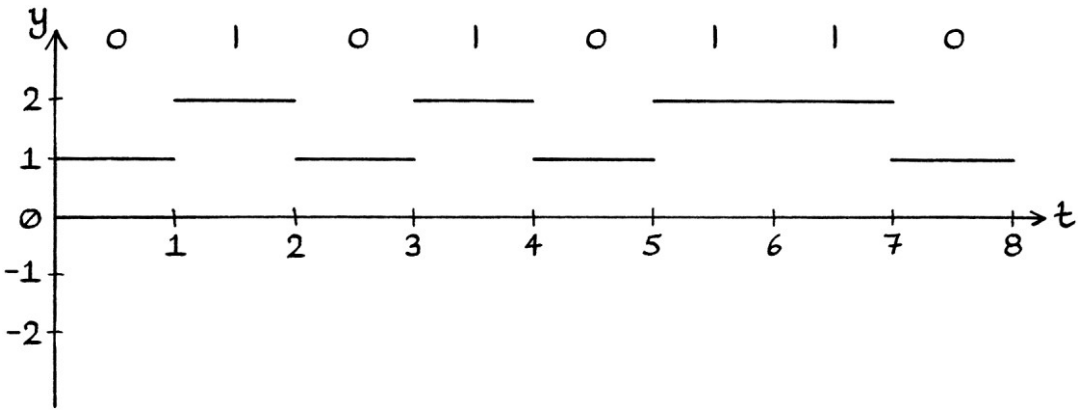
Frequency shift keying can be performed with multiplication or with addition, as well as other methods. Strictly speaking, it does not matter in which way the frequency is changed as long as the changes are related to the instantaneous amplitudes of the square wave, and as long as they can be decoded by the receiver.

In this section, we will look at frequency shift keying using multiplication. We will use a carrier wave with the formula "y = sin (360 * 4t)".

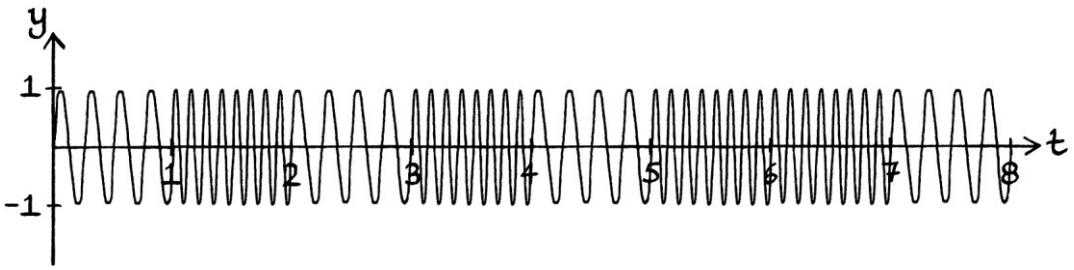
To perform frequency shift keying, we will use multiplication on the frequency within the formula. This means that if the y-axis value of the square wave at any particular time is "X" units, then the formula for the resulting signal will be:
"y = A sin (360 * X * ft)"

For two state frequency shift keying (2-FSK), we want the frequency of the resulting signal to switch between two states: 4 cycles per second and 8 cycles per second. Therefore, the square wave needs to switch between 1 unit and 2 units. Its mean level will be 1.5 units. For the binary number "01010110", the square wave will be the same as for the 2-ASK example.

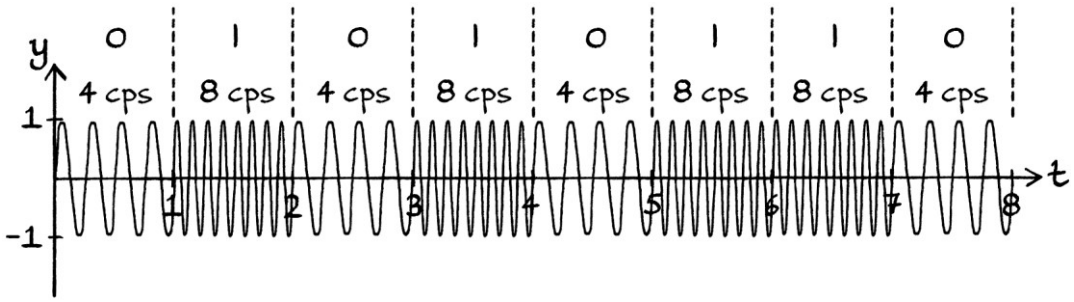
It will look like this:



The resulting signal looks like this:



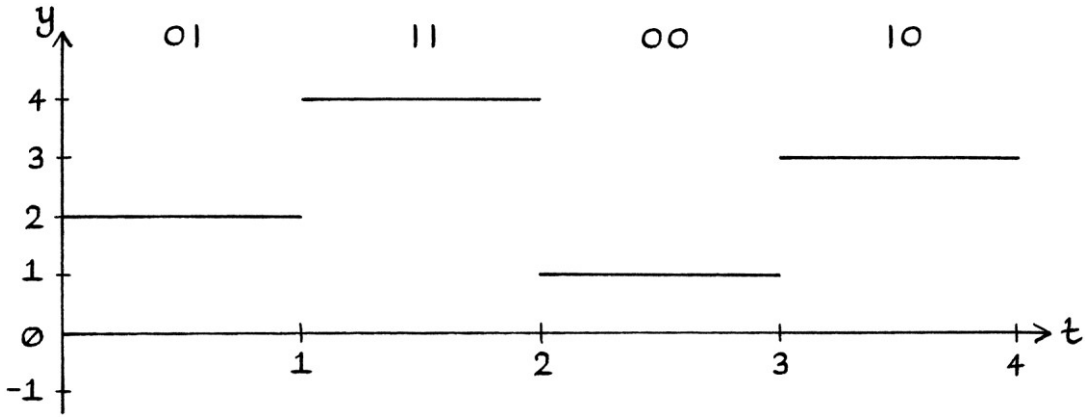
The same picture with more information is as so:



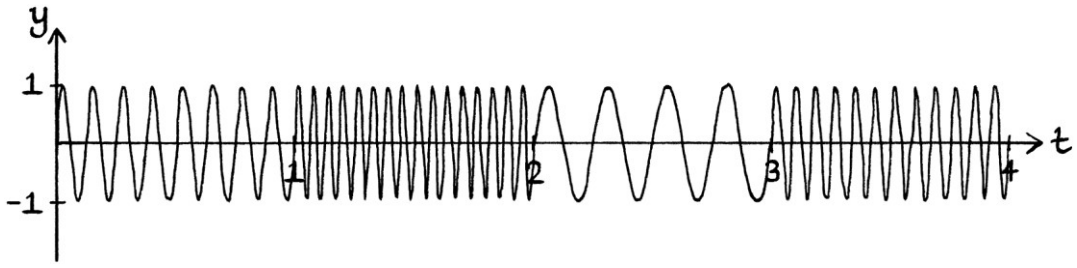
A general formula for frequency shift keying using multiplication is:
"$y = h_s + A \sin ((360 * X * ft) + \phi)$"
... where "X" is the instantaneous amplitude of the square wave at any one time.

Higher levels of FSK

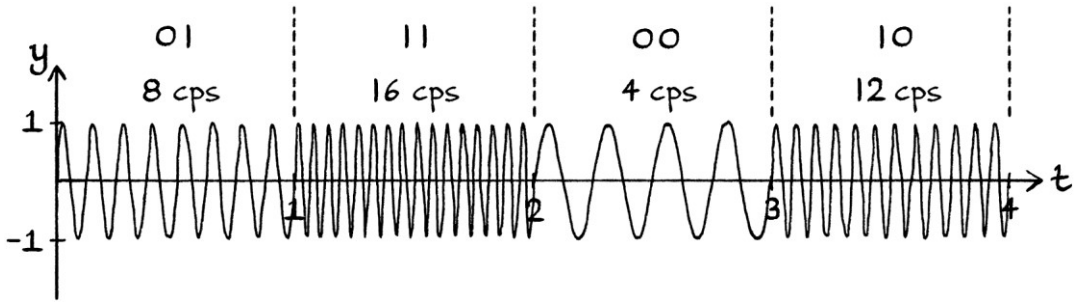
We can perform higher levels of frequency shift keying with multiplication. We will look at 4-FSK. We will use the same square wave as from the 4-ASK example, which encoded the binary digits "01110010". That square wave used one unit for one second to represent "00", two units for one second to represent "01", three units to represent "10", and four units to represent "11". The square wave looked like this:



The carrier wave will be "y = sin (360 * 4t)". When the frequency of the carrier wave is multiplied by the instantaneous amplitude of the square wave, we end up with this signal:



With more details, the signal looks like this:



FSK with addition

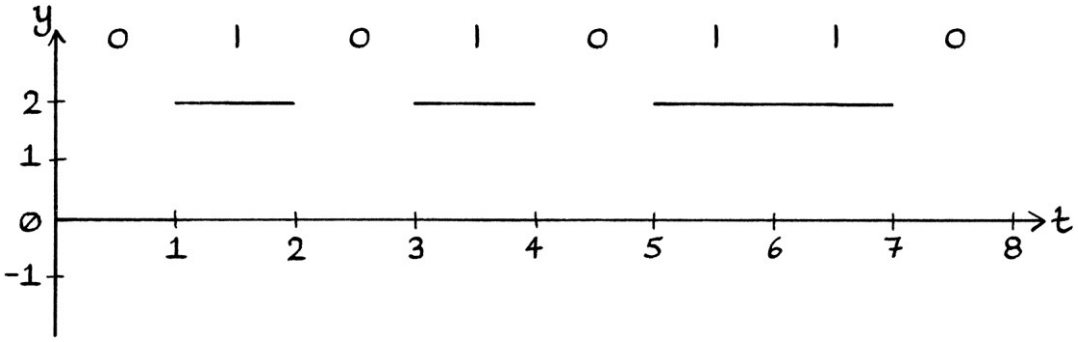
A second way to do frequency shift keying is by *adding* the instantaneous amplitude of the square wave to the frequency of the carrier wave. For this, it is clearest in pictures if the square wave switches between zero and a higher number. [It can also switch between positive and negative numbers.]

If the y-axis value of the square wave at any particular time is "X" units, then the formula for the resulting signal will be:
"y = A sin (360 * (f + X) * t)"

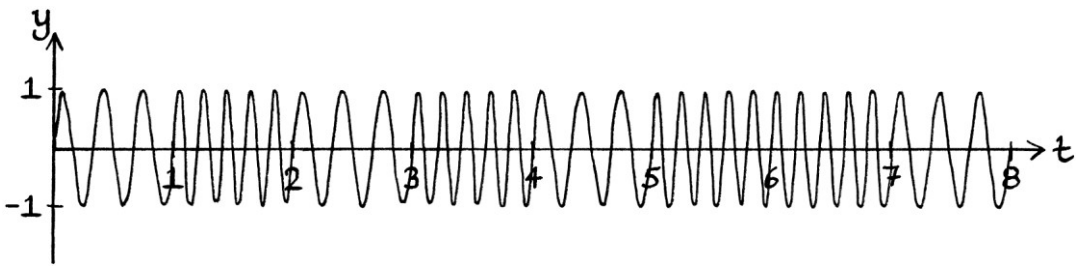
If our carrier wave is "y = sin (360 * 3t)", and our square wave switches between zero and two, the resulting signal would consist of sections of:
"y = sin (360 * 3t)"
... and:
"y = sin (360 * 5t)"

[We will make the square wave switch between zero and two, as opposed to zero and one, or zero and another number, solely so that the effect is clearer in a picture.]

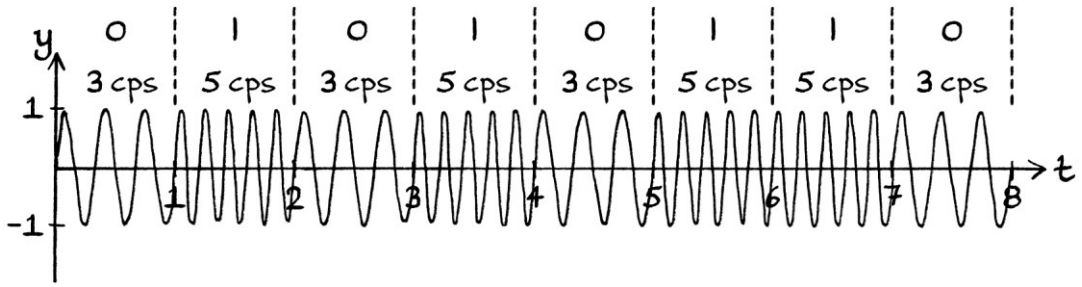
If we have this square wave representing the binary digits "01010110":



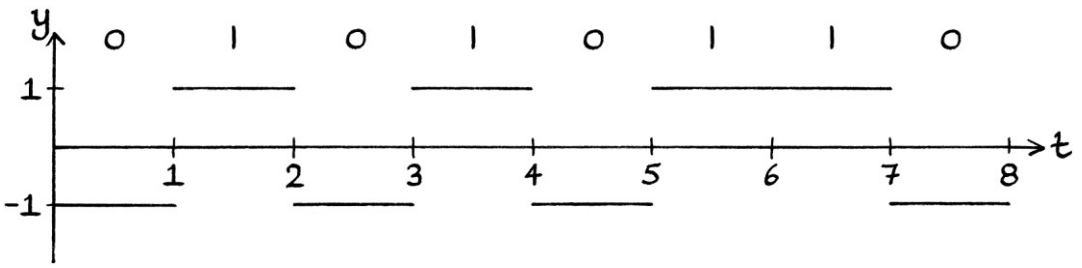
... then the resulting signal would look like this:



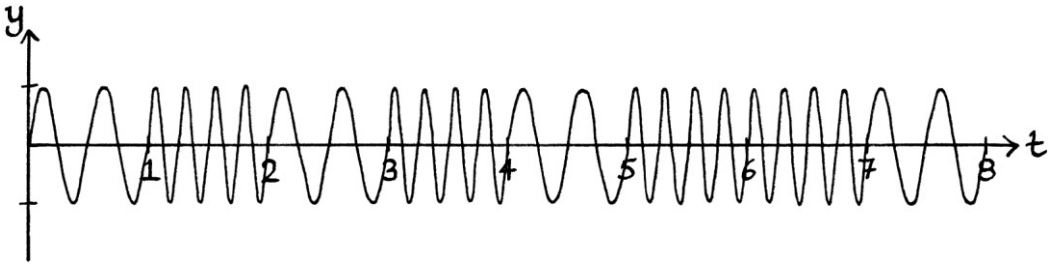
With the frequencies and binary digits marked, it looks like this:



Instead of just having positive y-axis values in the modulating signal, and so just increasing the instantaneous frequency of the carrier wave, we could have both positive and negative y-axis values. This would result in the instantaneous frequency of the carrier wave increasing and decreasing. As an example, we will use a carrier wave with the formula “y = sin (360 * 3t)”, and the following square wave that switches between −1 and +1 units:



The frequency-shift-keyed signal switches between instantaneous frequencies of 2 cycles per second and 4 cycles per second:



Higher frequencies

Supposing we wanted a larger or smaller jump in frequency, we could scale the y-axis values of the square wave before we added them to the frequency. This might be useful for higher frequencies where a change of one cycle per second would be hard to detect. If we had a carrier wave with a frequency of 10 MHz, and the square wave switched between zero and one, we could multiply the y-axis values of the square wave by, say, 10,000 before we added them to the frequency.

Limits

For a carrier wave with a particular frequency, there is a limit to the number of state changes that can occur per second before the message becomes corrupted. Ideally, the state changes should not occur more often than the number of cycles per second, and the changes in state should align with the starts and ends of the cycles.

Phase modulation

There is a third way of performing frequency shift keying, and that is by altering the phase of the carrier wave. We will look at this in Chapter 37 on phase modulation.

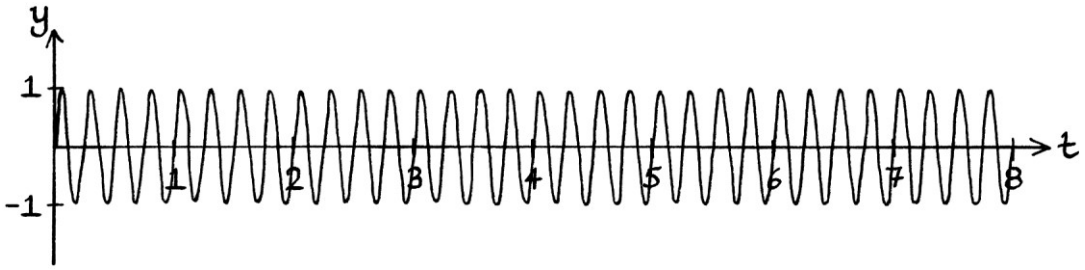
Automation: PSK

There are several ways to perform phase shift keying based on the characteristics of the square wave.

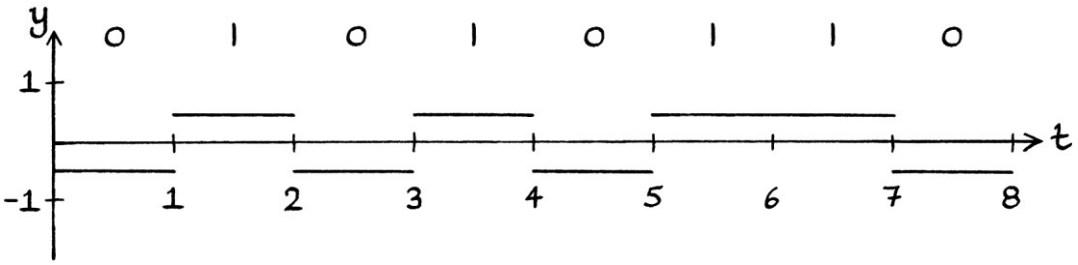
PSK by multiplying the amplitude

When we looked at automating on-off keying, we saw one way of performing two state phase shift keying, which involved multiplying the *amplitude* of the carrier wave by a square wave with a zero mean level. If the square wave switches between +1 and −1, then it will have the effect of making the resulting signal have a phase of 0 degrees or 180 degrees.

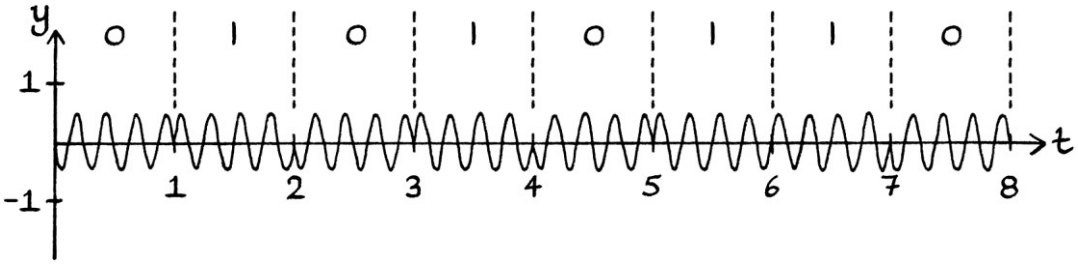
If we have this carrier wave:



... and our binary number (01010110) is encoded as this square wave:



... then the resulting signal will look like this:

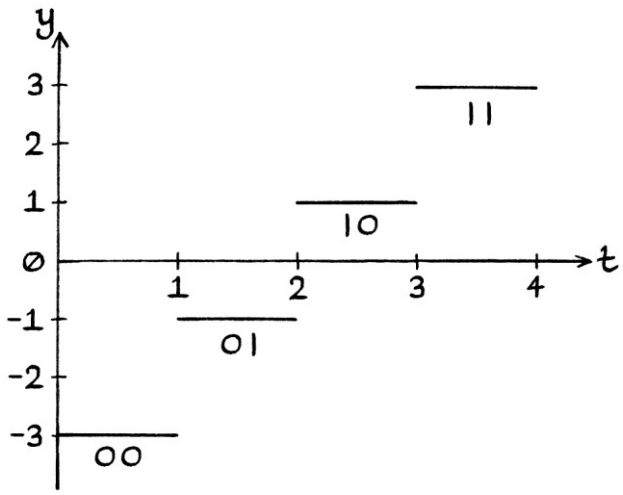


In this case, the phases are 180 degrees different from how we might want them, but this makes no real difference to decoding if the receiver knows this.

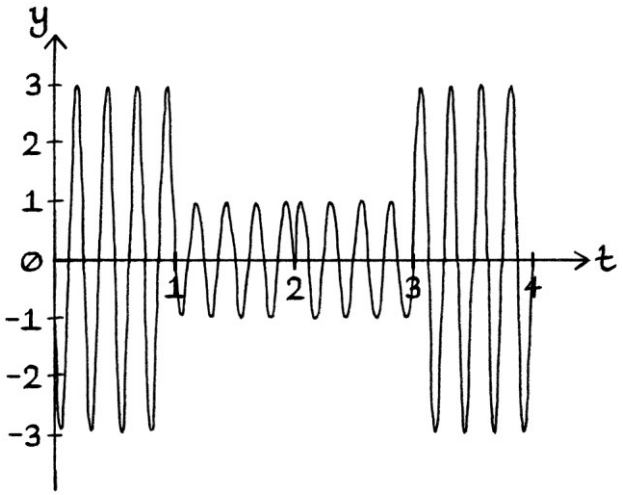
If "X" is the instantaneous amplitude of the square wave at any particular time, and the square wave is either +1 or −1, then 2-PSK can be performed using this formula:
"y = (X * A) sin ((360 * ft) + ϕ)"

[If the square wave switches between any value and *zero*, then this formula will instead produce on-off keying; if it switches between any two positive non-zero values (or any two negative non-zero values), then the formula will instead produce amplitude shift keying.]

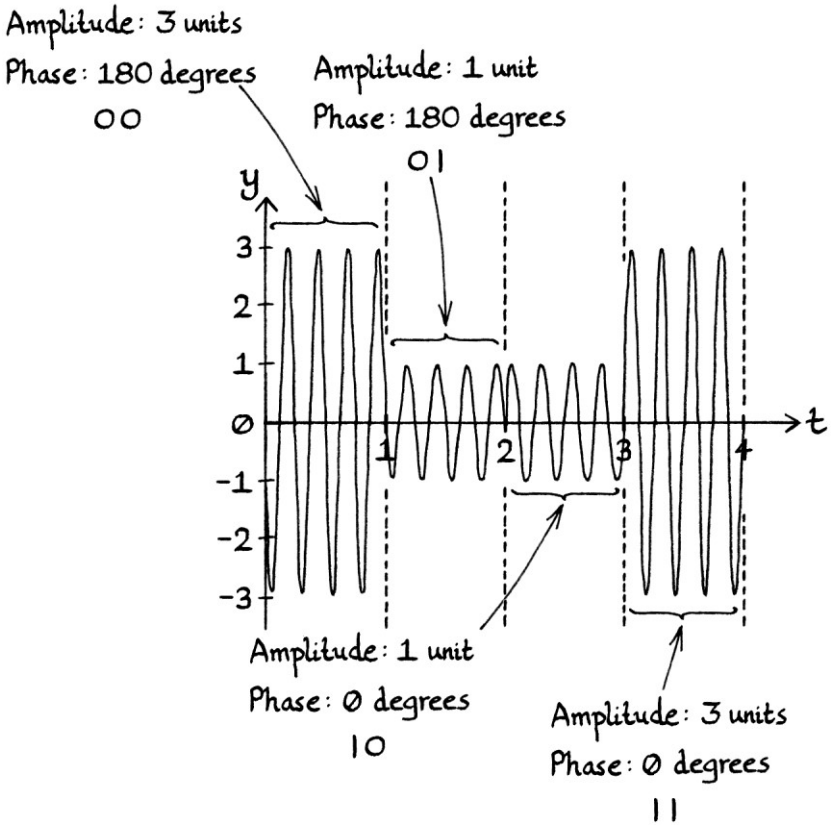
We can also perform a combination of 2-PSK and different levels of ASK at the same time using this idea. As an example, we will use the following 4 state square wave that encodes the digits “00011011”:



When we multiply the square wave against our carrier wave, we end up with this signal:



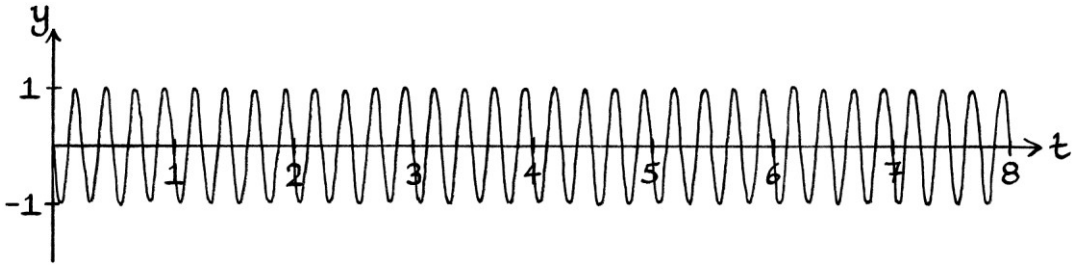
The resulting signal has two different amplitudes and two different phases. Each combination of an amplitude and a phase represents 2 bits:



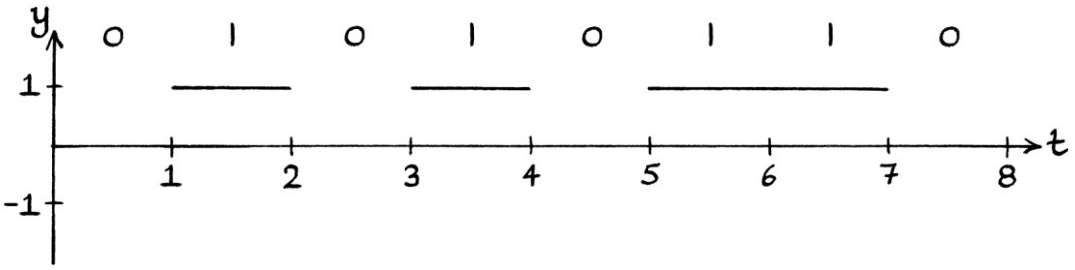
PSK by multiplying the phase

We can also produce phase shift keying by multiplying the phase of the carrier wave against the instantaneous amplitude of the square wave. This requires more thought than the previous examples of shift keying – if the original phase of the carrier wave is zero, then multiplying it by any value will keep it at zero, and so make no difference at all. We can get around this problem by starting with a non-zero phase. For example, if the phase of the carrier wave is 180 degrees, we could use multiplication for 2-PSK and change the phase successfully. We could have the instantaneous amplitude of the square wave as zero to indicate a zero, and have it as one to indicate a one. Multiplying the phase of the carrier wave by zero would set the phase to zero; multiplying it by one would leave the phase as 180 degrees.

We will use this carrier wave:
“y = sin ((360 * 4t) + 180)” in degrees

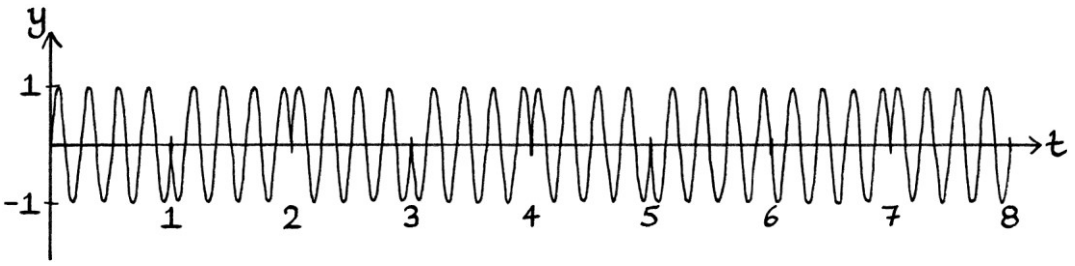


We will use the following square wave that switches between 0 units and 1 unit to encode the binary number “01010110”:

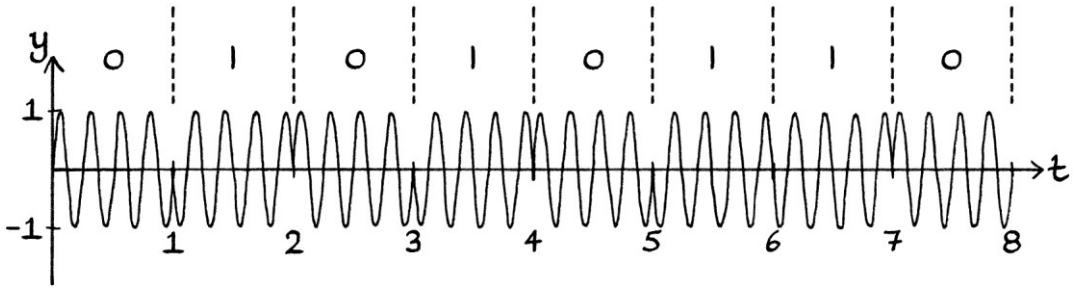


If “X” is the instantaneous amplitude of the square wave at any particular moment in time, then the resulting signal will have the formula:
“y = sin ((360 * 4t) + (180 * X))” in degrees

The resulting signal looks like this (with the t-axis numbering written underneath to make the graph clearer):



Here is the same graph showing the digits that are being represented at any one moment in time:



[We could achieve a similar effect by having the square wave switch between 1 unit and 2 units. The resulting signal would switch between frequencies of 180 degrees and 360 degrees, which is also 0 degrees. The downside to this is that the resulting phases would be the opposite of what we wanted. However, we could adjust the square wave so that a 2 represented a binary 0 and a 1 represented a binary 1.]

We can use higher levels of phase shift keying too, but each level requires using a different starting phase for the carrier wave. If we wanted to use 4-PSK, the carrier wave would need to have a phase of 90 degrees, and the square wave would need to switch between zero, one, two, and three units. In such a case:
0 * 90 = 0 degrees, and represents “00”
1 * 90 = 90 degrees, and represents “01”
2 * 90 = 180 degrees, and represents “10”
3 * 90 = 270 degrees, and represents “11”.

8-PSK would need the carrier wave to have a phase of 45 degrees.

A general formula for phase shift keying, when using multiplication of the phase is:
“$y = h_s + A \sin ((360 * ft) + (X * \phi))$”
... where:

- “X” is the instantaneous amplitude of the square wave at any one moment in time.
- “ϕ” is the phase of the carrier wave. It is non-zero and of an appropriate angle for the range of values by which it will be multiplied.

PSK by adding to the phase

[In this section, we will use both radians and degrees.]

Instead of multiplying the phase of the carrier wave by the state of the square wave, we can add to the phase of the carrier wave instead. We can do this in two different ways:

- The first way is to have the square wave switch between a minimum of 0 units and a maximum of under 2π units (if we are using radians), and then add the instantaneous amplitudes to the phase of the carrier wave. For degrees, the square wave would have to switch from a minimum of 0 units to a maximum under 360 units. The downsides to this method are that, for radians, the square wave would need to switch between unpleasant values (such as irrational numbers), and, for degrees, the square wave would need to have very high y-axis values. Using this method, the formula for our modulated signal would be:
  "y = sin ((2π * ft) + (ϕ + X))" in radians, or:
  "y = sin ((360 * ft) + (ϕ + X))" in degrees
  ... where "X" is the instantaneous amplitude of the square wave at any one time.

  Given that it would make things simpler to give the carrier wave a phase of zero radians or degrees, these formulas can become:
  "y = sin ((2π * ft) + X)" in radians, or:
  "y = sin ((360 * ft) + X)" in degrees

  In this way, we are not so much *adding* to the phase as *setting* the phase. However, we will still refer to this method as "addition".

- The second way is for the square wave to switch between zero and values that are less than one. If we are working in radians, we multiply the values from the square wave by 2π to end up with a result between 0 and just under 2π. If we are working in degrees, we multiply the values by 360. We then add the result to the phase of the carrier wave. The formulas for this are as so:
  "y = sin ((2π * ft) + (ϕ + (2π * X))" in radians, or:
  "y = sin ((360 * ft) + (ϕ + (360 * X))" in degrees
  ... where "X" is the instantaneous amplitude of the square wave at any one time.

Again, it makes things simpler to give the carrier wave a phase of zero radians or degrees, in which case, we can give the formulas as so:
"y = sin ((2π * ft) + (2π * X))" in radians, or:
"y = sin ((360 * ft) + (360 * X)" in degrees

Again, in this case, we are not so much *adding* to the phase as *setting* the phase.

We could also use variations of the two methods. For the purposes of phase shift keying, the second method is clearest. It allows us to have simpler values in the square wave. Therefore, we will use that method in the following examples.

If we were using 2-PSK, the square wave would have states of 0 units and 0.5 units. If we were working in radians, we would multiply this by 2π, and set the phase of the carrier wave to the result. The resulting phase would be either 0 radians or π radians. If we were working in degrees, we would multiply by 360, and end up with either 0 degrees or 180 degrees.

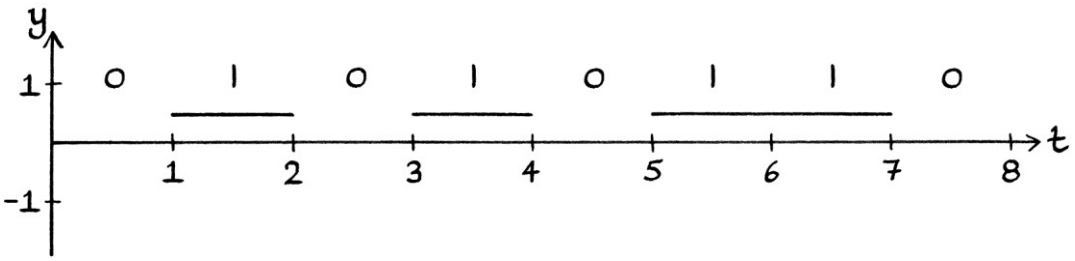
If we were using 4-PSK, the square wave would have possible values of 0, 0.25, 0.5 and 0.75 units. If we were working in radians, after the multiplication, we would end up with phases of 0, 0.5π, π and 1.5π radians. If we were working in degrees, we would have phases of 0, 90, 180 and 270 degrees.

If we were using 8-PSK, the square wave would have possible values of 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75 and 0.875 units. For radians, multiplying these by 2π would give phases of 0, 0.25π, 0.5π, 0.75π, π, 1.25π, 1.5π and 1.75π radians. For degrees, multiplying the values by 360 would give us phases of 0, 45, 90, 135, 180, 225, 270, and 315 degrees.]

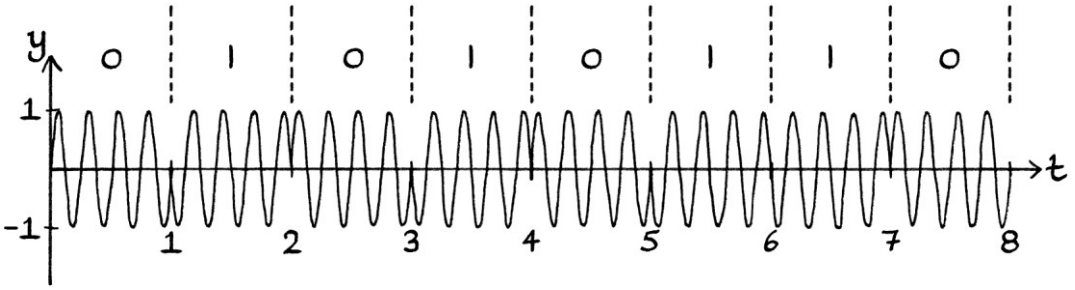
We will look at an example of 2-PSK while using radians. We multiply the state of the square wave by 2π, and then add that to the phase of the carrier wave. As the phase of the carrier wave is zero, this is the same as making the carrier have that phase.

For our carrier wave, we will use this wave:
"y = sin (2π * 4t)" in radians

We will use a square wave that contains the binary digits “01010110”. A y-axis value of zero represents a binary zero; a y-axis value of 0.5 represents a binary one:



For every moment in time, we take a y-axis value from the square wave, multiply it by 2π, and then set the carrier wave’s phase to the result. The resulting signal looks like this:



This is the same resulting signal as when we used PSK with multiplication and a square wave that encoded the same digits, but that switched between 0 and 1 unit.

The formula for the resulting signal is:
“y = sin ((2π * 4t) + (2π * X))”
... where “X” is the instantaneous amplitude of the square wave at any one time.

The “addition” methods of phase shift keying require more work to calculate the instantaneous phase of the carrier wave than when using multiplication on the phase. On the other hand, with the “addition” methods, we can use the same carrier wave no matter what the level of PSK we are doing. The “multiplication-of-the-phase” method requires the carrier wave to have a different phase depending on whether we were using 2-PSK, 4-PSK and so on.

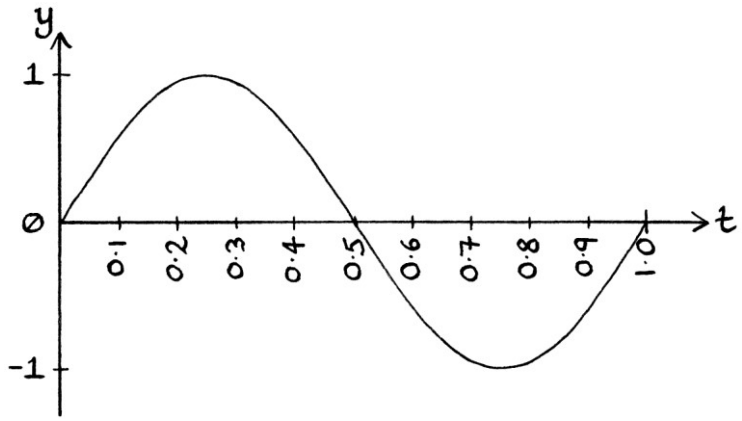
These examples show how phase shift keying could be performed in theory. In practice, the way it is done depends on many factors.

Frequency and phase

Constantly increasing phases

There is an interesting idea that is more obvious once you have become used to shift keying – it is possible to change the frequency of a wave by repeatedly increasing its phase as time progresses.

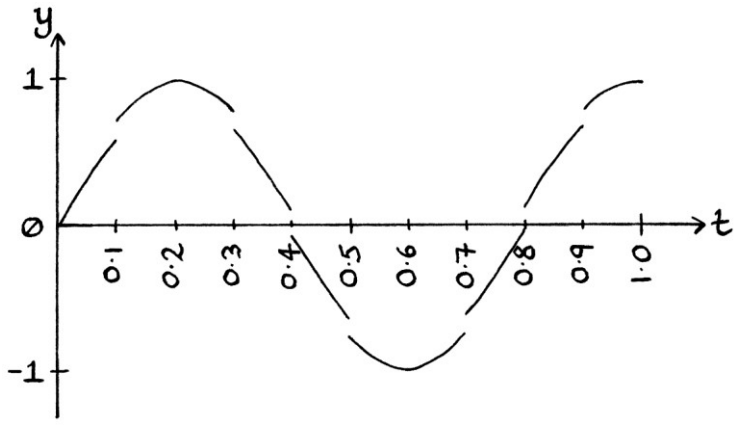
As a simple example of what this means, we will start with this wave in degrees: "y = sin 360t":



If we increase the phase of the wave by 10 degrees every 0.1 seconds, the phase and the formulas for each 0.1 second interval will be as follows:

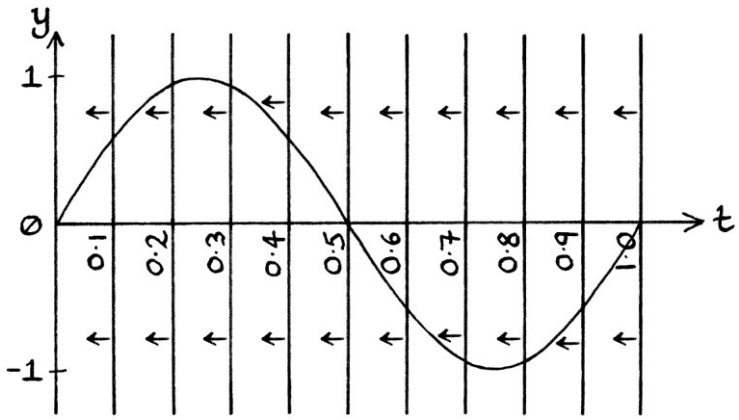
Time (seconds)	**Phase during this time**	**Formula during this time (as if the wave had existed since t = 0)**
0.0 to 0.1	0	y = sin 360t
0.1 to 0.2	10	y = sin (360t + 10)
0.2 to 0.3	20	y = sin (360t + 20)
0.3 to 0.4	30	y = sin (360t + 30)
0.4 to 0.5	40	y = sin (360t + 40)
0.5 to 0.6	50	y = sin (360t + 50)
0.6 to 0.7	60	y = sin (360t + 60)
0.7 to 0.8	70	y = sin (360t + 70)
0.8 to 0.9	80	y = sin (360t + 80)
0.9 to 1.0	90	y = sin (360t + 90)

The resulting “wave” looks like this:



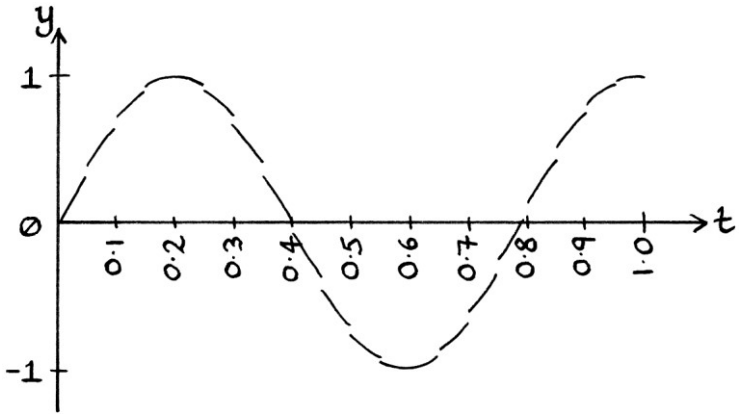
Although this “wave” is blocky, we can see that constantly increasing the phase of the wave in this way is approximately creating a new wave with a higher frequency. This new wave has a period of approximately 0.8 seconds, and therefore, a frequency of approximately 1.125 cycles per second.

Constantly increasing the phase increases the frequency because we are constantly bringing bits of the wave from future times to earlier times. Every part of the wave is brought earlier than it would have been otherwise.

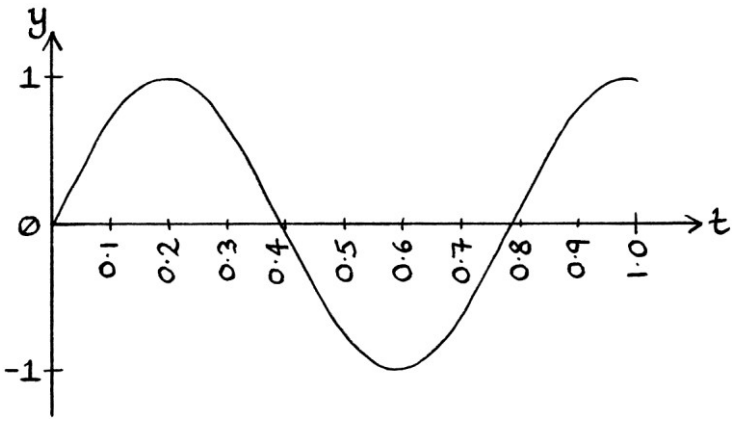


If we had only one jump in phase, the wave would continue at the same frequency after the jump. As we are constantly increasing the phase, the overall frequency is changing.

We can make a smoother wave by changing the phase at shorter intervals. For example, the following “wave” was created by changing the phase at intervals of 0.05 seconds, and increasing the phase by 5 degrees each time:



The following wave was created by changing the phase at intervals of 0.01 seconds, and increasing the phase by 1 degree each time. In a drawing, it is indistinguishable from the pure wave it resembles.



The above wave has a period of about 0.78 seconds, and so has a frequency of 1.28 cycles per second. The previous two waves had the same period and frequency, but it was impossible to read the waves as accurately.

In each of the three graphs, we are raising the phase at an average rate of 100 degrees per second. [We can calculate this as "increase in phase over a particular time" divided by "the length of that time". For example, 10 degrees ÷ 0.1 seconds = 100 degrees per second.] Therefore, the formula for the resulting wave can be written in terms of the phase increase as:
"y = sin (360t + 100t)"
... where:

- Sine is working in degrees and all angles are given in degrees
- The frequency is 1 cycle per second.
- The phase is increasing at 100 degrees per second.

What might be clear from this formula is that we can add the frequency part "360t" and the phase part "100t" together and have this formula:
"y = sin (460t)"
... where the constant phase increases have become absorbed into the frequency part of the formula.

We can then split the frequency correction part of this formula (360) to have a formula that matches how we would normally write it: [We divide 460 by 360]:
"y = sin (360 * 1.2778t)"

This means that 1.2778 cycles per second is the actual frequency of the above three graphs, although we were achieving this frequency by constantly adding to the phase.

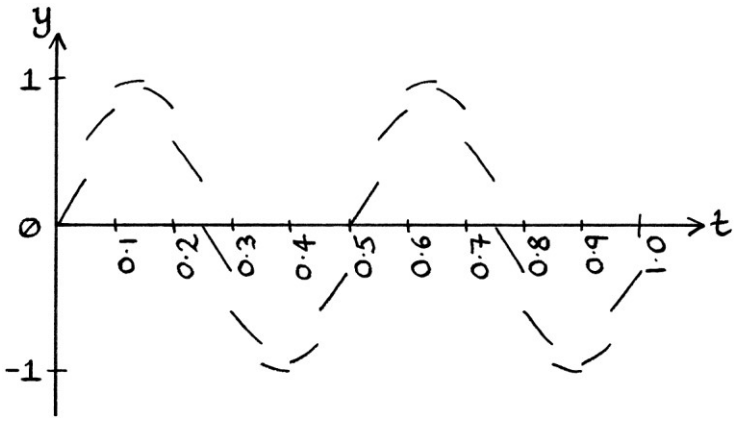
Another example

We can adjust the frequency of a wave using continuously increasing phases to turn this wave:
"y = sin 360t"
... into the equivalent of this wave:
"y = sin (360 * 2t)"

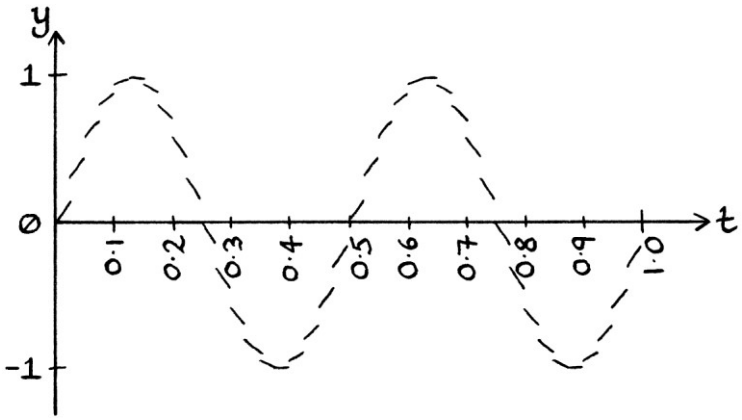
The two-cycle-per-second wave can be rephrased to be this:
"y = sin (720t)"
... which is also:
"y = sin (360t + 360t)
... where:

- The first “360t” is the angular frequency of the wave (or we could say it is the frequency of the wave (1 cycle per second) multiplied by the 360 degree “time correction”.
- The second “360t” is the ever-increasing phase of the wave. It increases at a rate of 360 degrees every second.

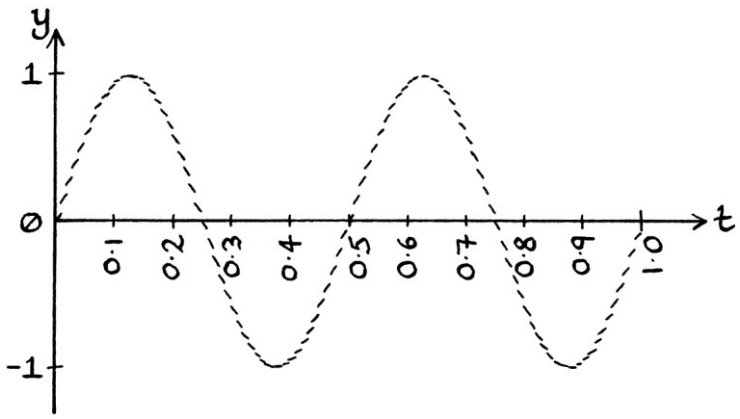
This means that if we start with the wave “y = sin 360t”, we can double its frequency by increasing its phase at a rate of 360 degrees per second. To demonstrate this most simply, we will increase its phase by 18 degrees every 0.05 seconds (which is the same average rate). The result looks like this:



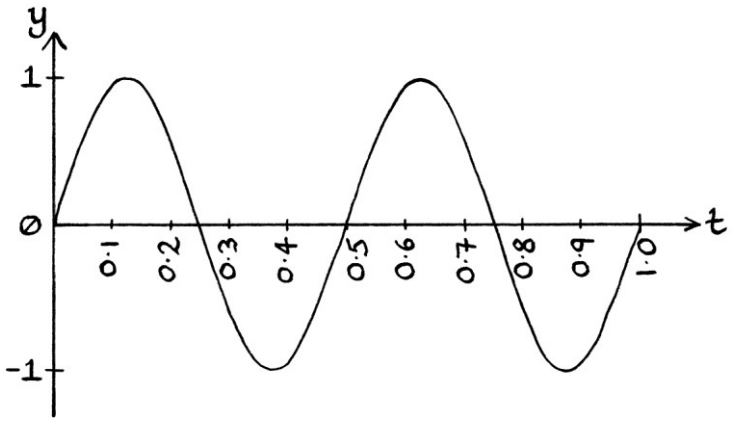
Keeping the same average rate of phase increase, if we increase the phase of “y = sin 360t” by 9 degrees every 0.025 seconds, we will end up with this:



If we increase its phase by 3.6 degrees every 0.01 seconds, we will have this:



If the phase increased at a continuous rate of 360 degrees per second, we would have a perfectly drawn two-cycle-per-second wave:



Thoughts

From all of the above, we can see that we can simulate a particular level of frequency by using continuous increases in phase. Being pedantic, it is actually the case that frequency **is** a continuous increase in phase. Frequency is the rate of phase change. To put this in the language of calculus, frequency is the derivative of phase. Conversely, phase is the integral of frequency.

The idea that frequency is a constant increase in phase is a reasonably important one. This section could have been in Chapter 7 (on phase) or in Chapter 11 (on frequency) of part one of this book, but it is a subject that is much easier to understand after seeing phase shift keying. Explaining it before now would have made it seem a much more complicated idea.

Given that frequency can be altered by continuously changing the phase, it is possible to perform frequency shift keying, and in fact all frequency modulation, by carefully altering the phase of the carrier wave. [It is also the case that careless phase shift keying might accidentally change the frequency of a wave.]

More details about shift keying

In practice, you might never see mean level shift keying or speed shift keying.

In all the examples of on-off keying and shift keying, if we were to use square waves that incorporated Manchester encoding, pulse width modulation or other methods of indicating timing, the resulting signals would end up incorporating those methods.

ASK, FSK, and PSK generally use methods and rules to minimise the used bandwidth, or to make decoding easier. These include such things as only changing the amplitude or frequency at the end of a cycle and switching between phases in a less straightforward manner.

Seeing ASCII being sent in a non-amateur radio signal would be relatively rare because it makes sense to compress the data first to reduce its size. Once compressed, the data would no longer look like ASCII. Another fact about ASCII is that it dates from a time when computer memory was expensive and it paid to use as few bytes as possible. Therefore, it concentrates on a subset of the Latin alphabet, and it is not as suited to other alphabets. A better system is Unicode, which can portray characters from any language without any chance of confusion. Unicode uses 8 binary digits, 16 binary digits, or more, depending on the type of Unicode being used and the type of character being portrayed. The letters of most alphabets of the world can be portrayed with just 16 binary digits each.

It is common for data to be sent not as just data, but instead in the form of “packets”. A packet is a series of bits that contain a portion of the data to be sent, but also other information such as the size of the piece of data, where in a message it belongs, and checksums that tell the receiver if the data has been corrupted on its journey. Sending data as packets requires more information to be sent, but solves many transmission and reception problems.

Chapter 36: Amplitude modulation

Modulating waves with waves

Amplitude shift keying, frequency shift keying and phase shift keying are fairly simple ways to understand modulation. However, they are more complicated to implement with electronics, which is why books usually explain them after introducing non-digital modulation. In my opinion, it is easier to understand non-digital modulation after learning about shift keying.

As we saw in the previous chapter, messages sent with shift keying can be automated by using a square wave containing the details of the binary digits. That square wave can be used to alter the amplitude, frequency, or phase of the carrier wave. Non-digital modulation works in the same way, but instead of using a square wave, it alters the amplitude, frequency, or phase of a carrier wave *with another wave or signal*. If we want to broadcast a piece of music, it is easier to use the sound signals of the music to modulate the carrier wave, than to convert the music to a digital format and use a square wave to modulate the carrier wave. While shift keying can use square waves that mirror the binary data to alter a wave, "normal" modulation uses other signals to alter waves. This can be harder to visualise, but conversely, it is easier to perform. It should be much easier to understand now that we have looked at shift keying.

Technically, the terms "amplitude modulation", "frequency modulation" and "phase modulation" include amplitude shift keying, frequency shift keying and phase shift keying. However, in this and the next two chapters, we will be looking at the non-digital aspects of amplitude modulation, frequency modulation and phase modulation.

In these chapters, I will use degrees or radians depending on which is the simplest or clearest for the example.

Amplitude modulation

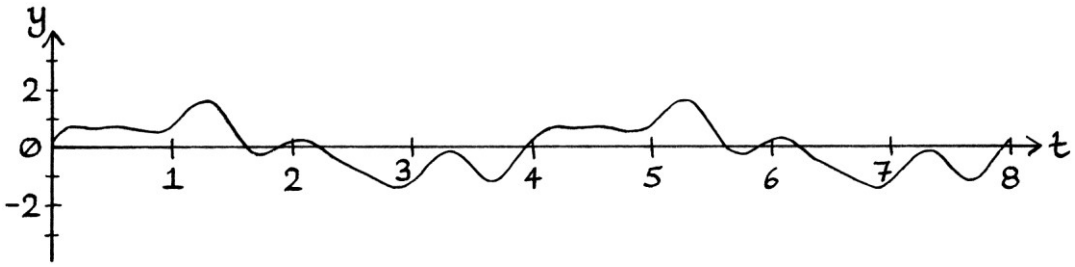
We looked at amplitude modulation in Chapter 16 in the section “More aspects of multiplication”. It is worth looking at that section again to refresh your memory on the terminology.

Amplitude modulation encodes a signal into a wave by altering the wave’s instantaneous amplitude in a way relating to the instantaneous amplitude of that signal. In other words, the instantaneous amplitude of the carrier wave is altered according to the instantaneous amplitude of the signal being sent.

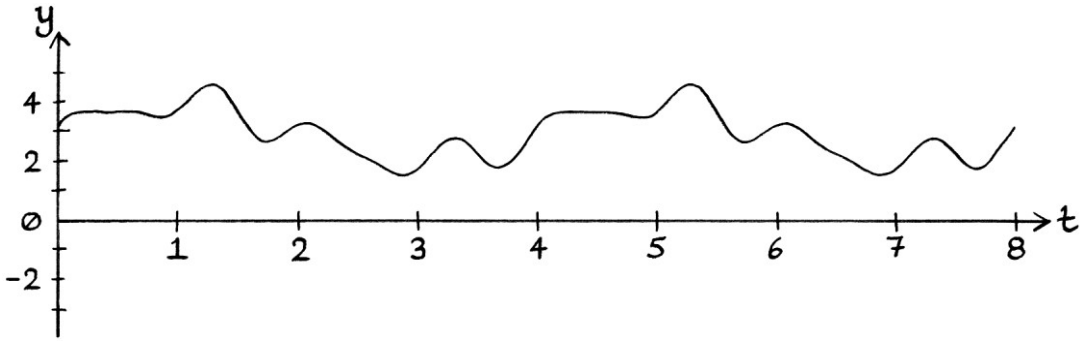
Amplitude modulation is performed by giving a slower frequency signal (the modulating signal, which is the message) a positive mean level, and then multiplying it by a faster frequency wave (the carrier wave). This produces a new signal consisting of waves with frequencies centred around the frequency of the faster wave. The new signal contains the slower frequency signal in its envelope, where the “envelope” is the shape formed by joining up the peaks of the signal. Amplitude modulation is generally used with sound signals in radio broadcasts – the frequency of an audible sound is too low to be sent as it is using radio waves. We cannot easily convert an audible sound to a radio signal of the same frequencies and have it travel as radio waves. Therefore, we have to let the sound piggyback on a faster frequency radio wave instead. This also has the benefit of allowing us to transmit sounds at various chosen frequencies, so that they do not interfere with other broadcasts.

Amplitude modulation with wave-like signals essentially works in the same way as amplitude shift keying. With amplitude shift keying, we multiplied the amplitude of our carrier wave by the square wave containing the encoded binary data. With amplitude modulation, we multiply the amplitude of our carrier wave by the wave-like signal. With amplitude shift keying, the square wave had to switch between states that were positive and non-zero (for example, between 1 and 2). It could not become zero or we would have ended up with on-off keying. It could not become negative or we would have ended up with phase shift keying. With amplitude modulation, similarly, the signal we want to send must be entirely non-zero and positive. If the signal is not entirely above zero, a mean level must be added to it to make it so.

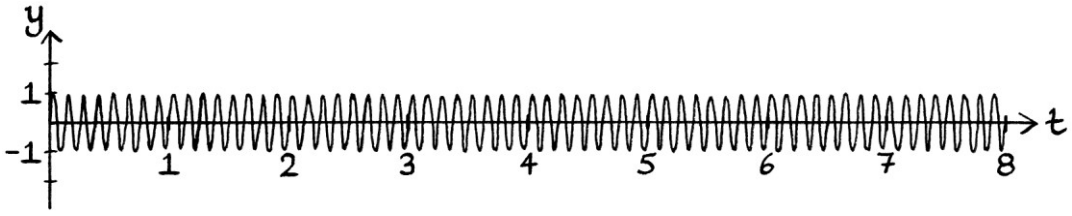
As an example of amplitude modulation, we will say that we want to send this audio signal, which has a mean level of zero units:



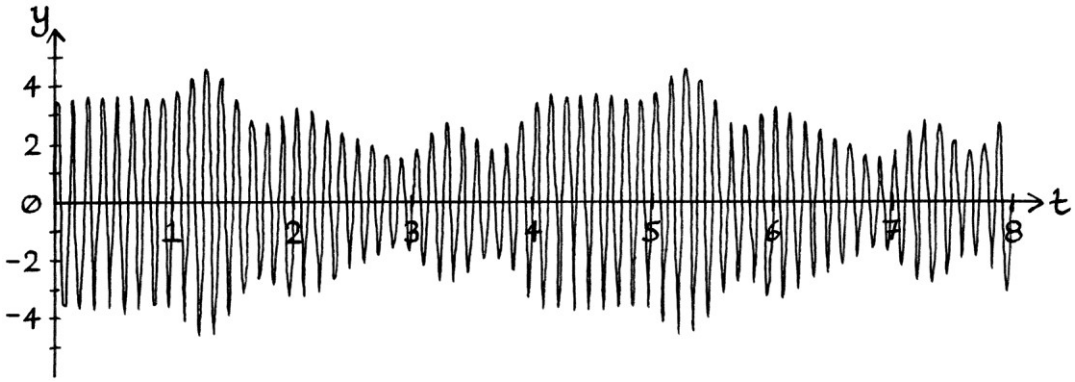
As this signal has parts that are zero or negative, we must make it entirely non-zero and positive. Therefore, we give it a non-zero mean level. For this example, we will add 3 units to the entire signal, thus giving the signal a mean level of 3 units.



We will then multiply the new signal by the following wave (the carrier wave), which has a frequency of 8 cycles per second:

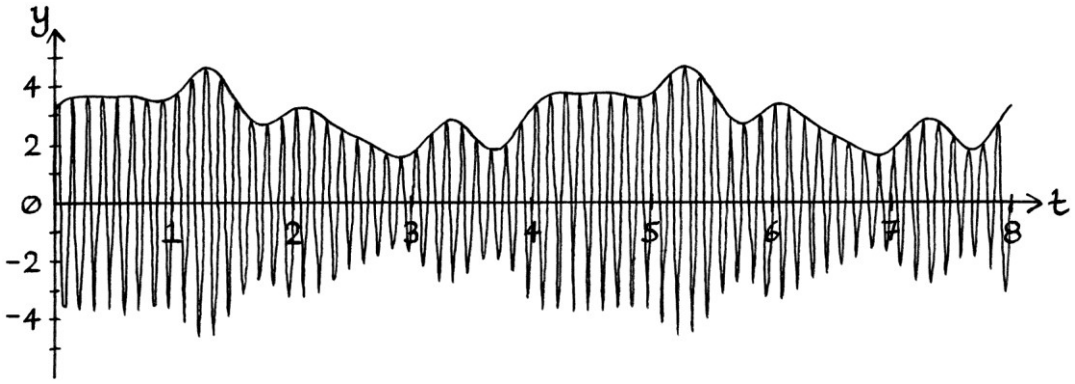


The result is this signal:



[If the frequency of the carrier wave were fast enough, we could transmit our resulting signal as a radio signal.]

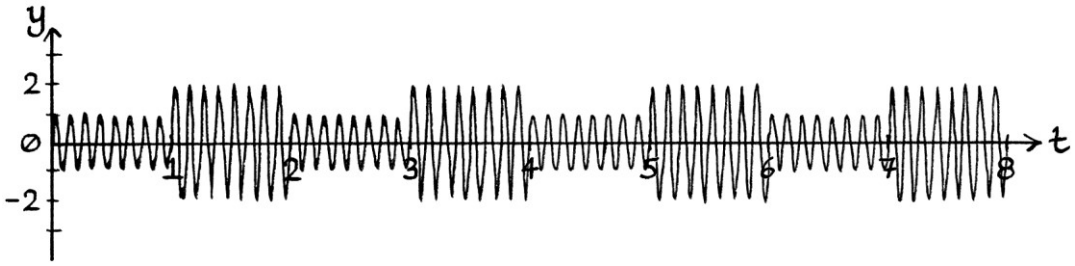
The original audio signal is visible in the "envelope" of the resulting signal. In other words, if we draw a line connecting all the peaks of the resulting signal, we will see the shape of the original signal:



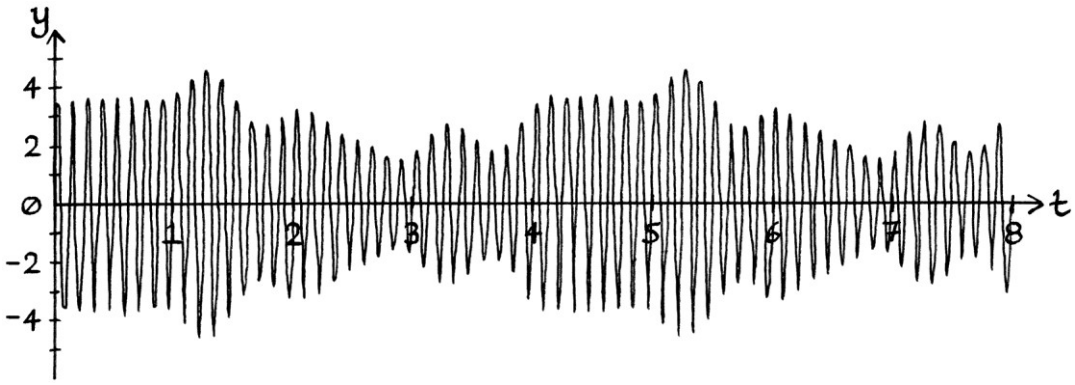
Remember that the resulting signal is a sum of waves as all impure signals are. The carrier wave that was used to create the resulting signal will still exist as one of the constituent waves.

Similarities with ASK

When we were using amplitude shift keying, we multiplied our carrier wave by the square wave. (The square wave contained the binary digits to be sent, and was entirely non-zero and positive). For 2-ASK, the result was a signal that had two levels of amplitude.

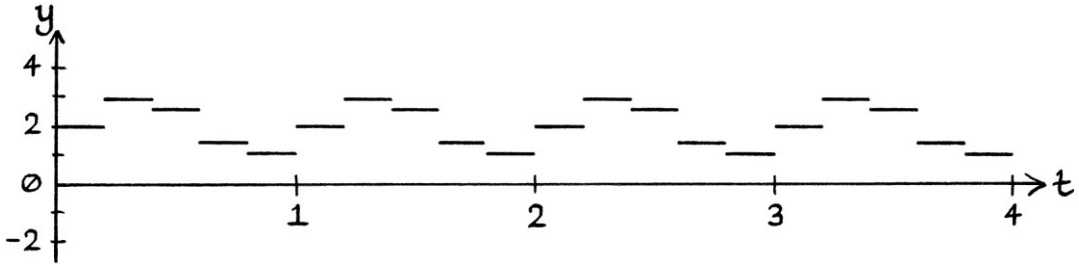


When we perform non-digital amplitude modulation, we multiply our carrier wave by the signal that we want to send. (The signal that we want to send is the modulating signal and is entirely non-zero and positive). Instead of the result having two levels of amplitude, it has as many levels of amplitude as there are different instantaneous amplitudes in the modulating signal.

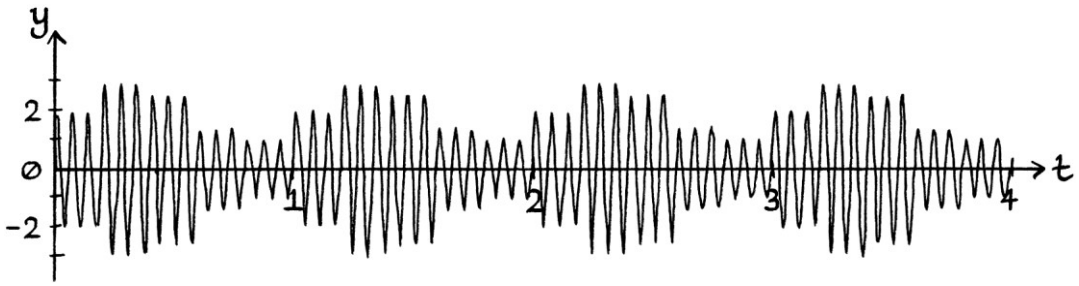


In ASK, the changes in amplitude are abrupt and obvious; in non-digital amplitude modulation, the changes are continuous. We can think of non-digital amplitude modulation as essentially being ASK with an infinite number of states.

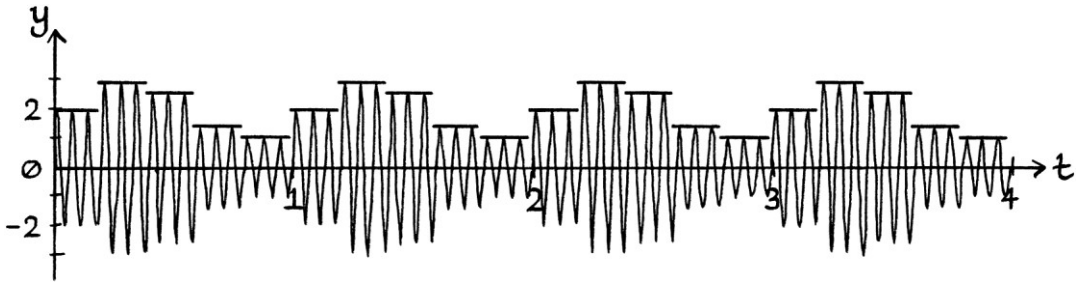
This is easier to visualise if we first use amplitude modulation to encode a Sine wave that has been broken up into a few levels such as this:



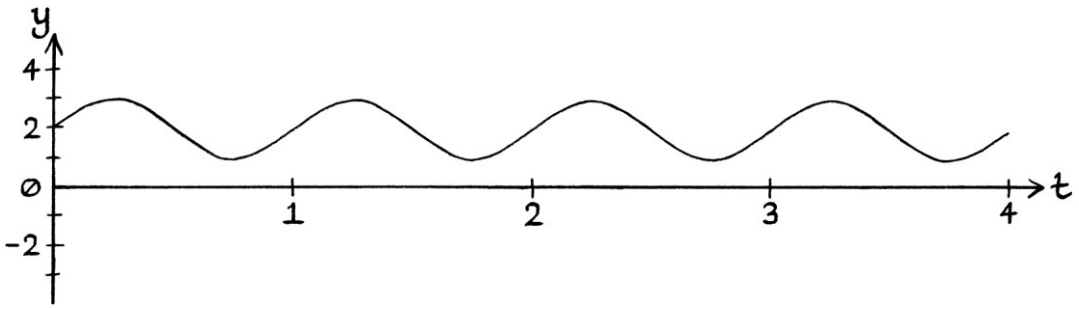
The resulting signal looks like this:



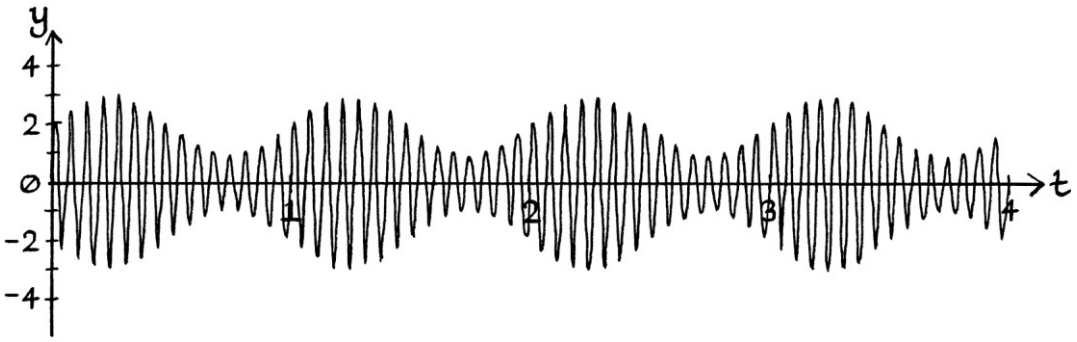
The “envelope” shows the original signal:



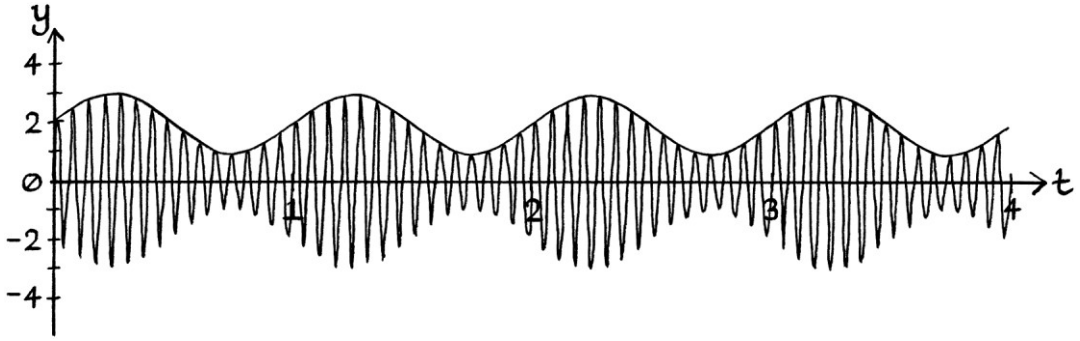
If we keep the original Sine wave unbroken like this:



... we end up with this signal:



The original wave is visible in the envelope:



One big difference between non-digital amplitude modulation and ASK is that, with ASK, every state is important and must be distinguished. In non-digital amplitude modulation, it is the general curve of the instantaneous amplitudes that is important.

Decoding amplitude modulation

There are several ways to decode the signal to retrieve the original modulating signal. One way is to multiply the received signal by the carrier wave (the faster frequency wave) that we used to encode it in the first place. Doing this creates a new signal consisting of the sum of:

- The original signal with its constituent waves at their original frequencies, but with half their original amplitudes.
- The original signal but with its constituent waves' frequencies centred around *twice* the frequency of the carrier wave, and with quarter of their original amplitudes.
- A mean level.

[The maths for this was explained in Chapter 16.]

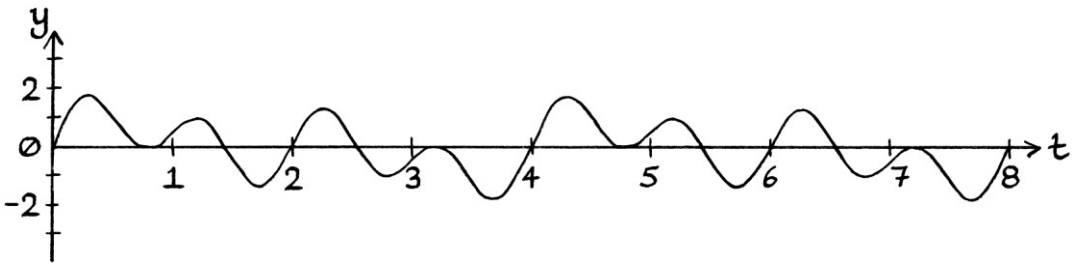
We remove the mean level. We then filter out the higher frequency waves to end up with just the decoded signal. In other words, we use a low pass filter. If we want to end up with exactly the same waves as were originally encoded, we can double the amplitudes of the retrieved waves. For radio broadcasts, the exact amplitudes of the retrieved waves are not as important as the relative amplitudes. The original amplitudes will be unknowable because the amplitudes become reduced as they travel from the source. Any decoded amplitude will be altered by the volume control on the radio by the listener anyway.

The frequency domain

As we saw in part one of this book, when we alter any pure wave in a way that stops it being a pure wave, it instantly becomes the sum of two or more pure waves of various amplitudes, frequencies and phases. This means that any form of modulation results in a signal consisting of a sum of waves of various amplitudes, frequencies and phases. When it comes to amplitude modulation, the pattern of new waves is straightforward as they will always have frequencies that are mirrored each side of the carrier frequency.

We will go through the steps to encode and decode an audio signal with a carrier wave as viewed in the frequency domain. To keep things simple, we will use a periodic audio signal. This means that we can use Fourier series analysis on the result, and create a frequency domain graph with the y-axis as amplitude.

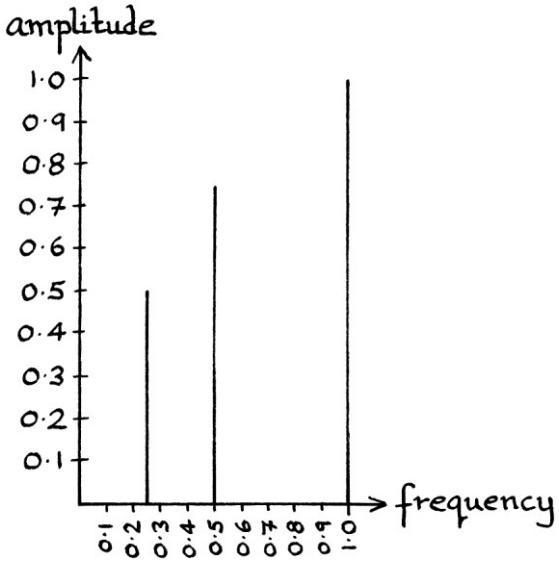
We will say that we want to send this audio signal:



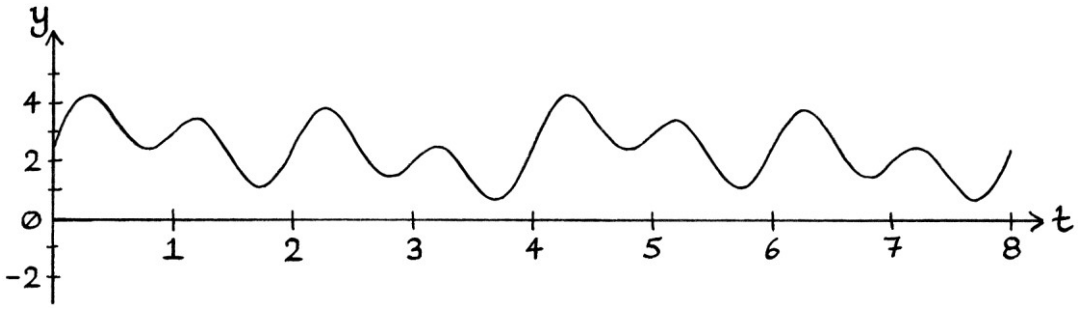
This signal has a frequency of 0.25 cycles per second. It repeats once every four seconds. [At first glance, it appears to have a frequency of 0.5 cycles per second, and to repeat once very two seconds.]

The signal is the sum of the following waves:
"y = 0.5 sin (2π * 0.25t)"
"y = 0.75 sin (2π * 0.5t)"
"y = sin 2πt"

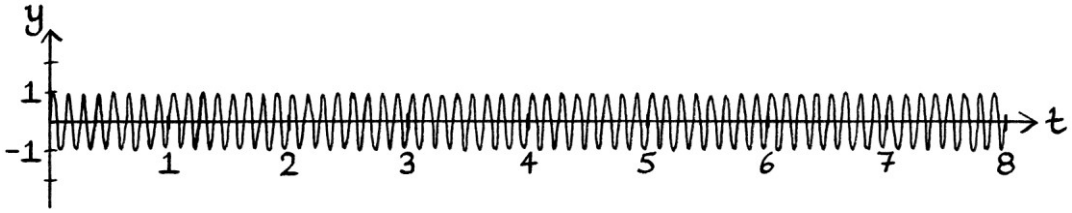
These appear as follows in a frequency domain graph with the y-axis as amplitude:



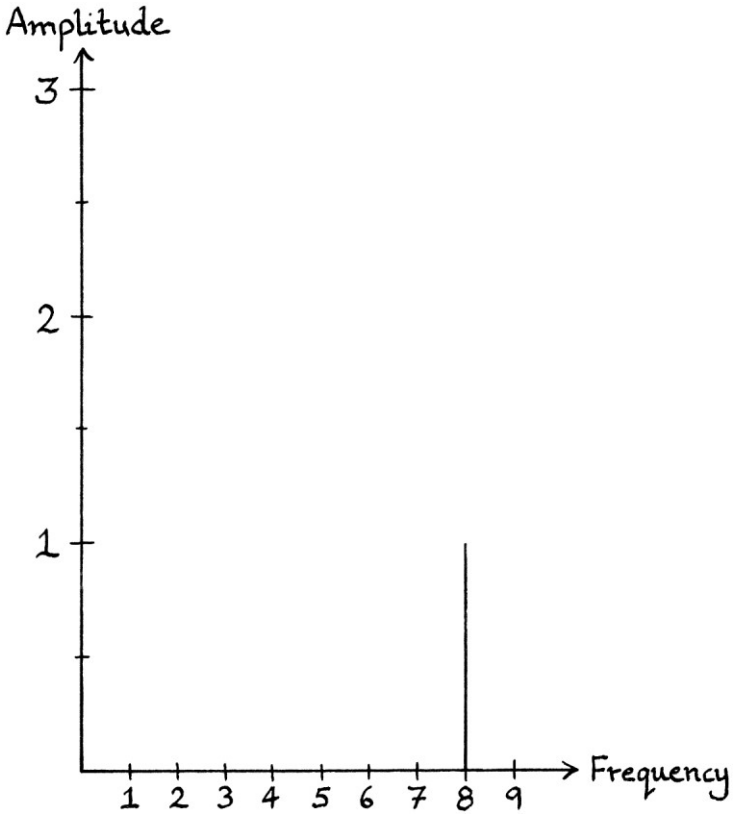
We give the signal a mean level to make it entirely non-zero and positive: [The mean level would not show up in a frequency domain graph.]



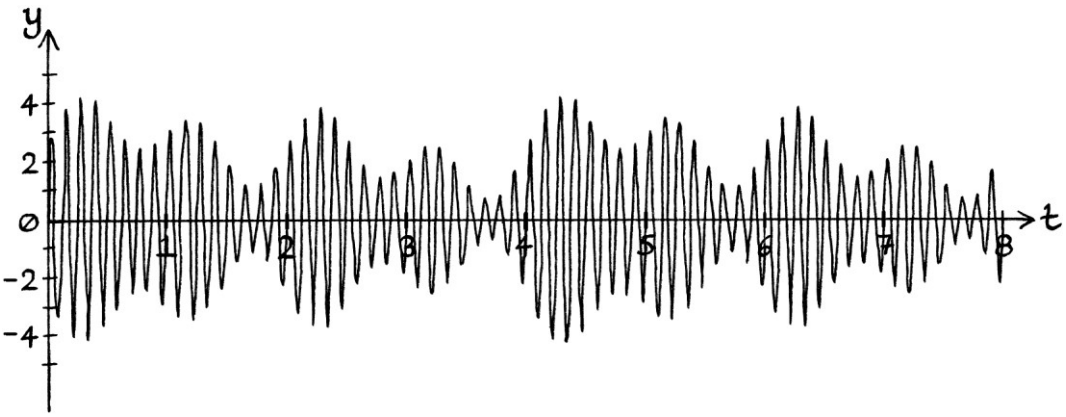
We then multiply it by the carrier wave, "y = sin (2π * 8t)":



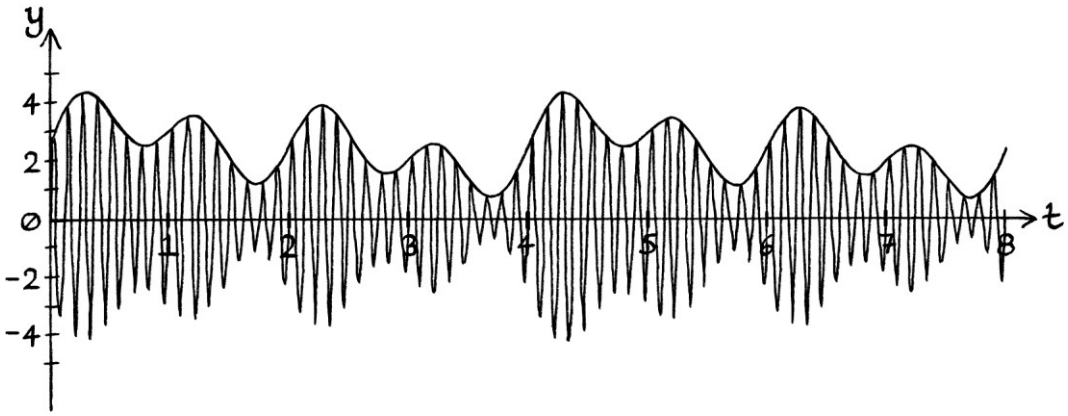
... which looks like this in the frequency domain (drawn to a different scale from the previous frequency domain graph):



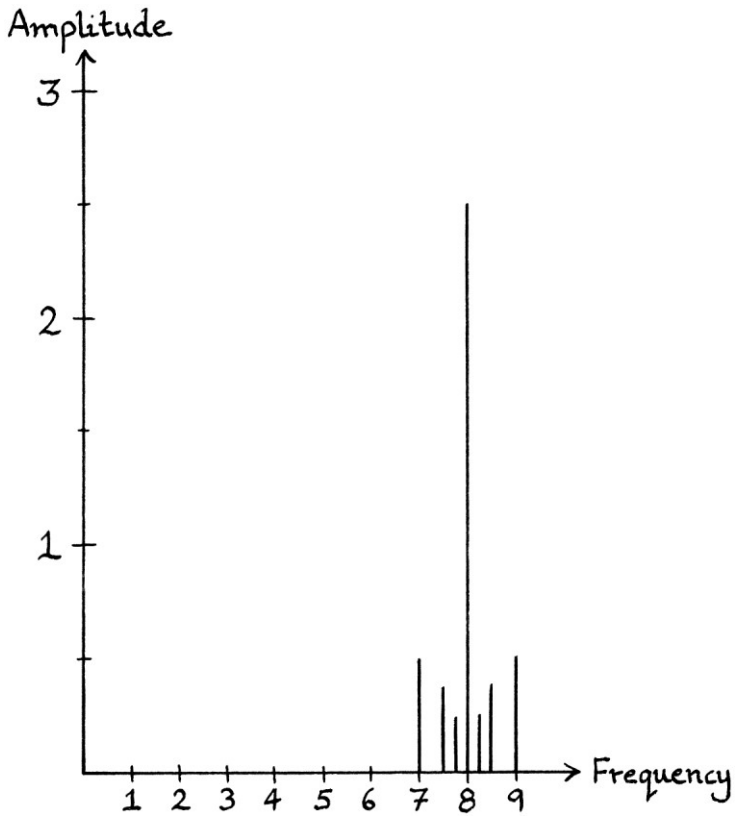
The result is this signal:



The original signal is visible in the envelope:



The resulting signal looks like this in a frequency domain graph:



The signal now consists of waves with frequencies mirrored around the carrier wave's frequency of 8 cycles per second. The waves that make up the signal are:

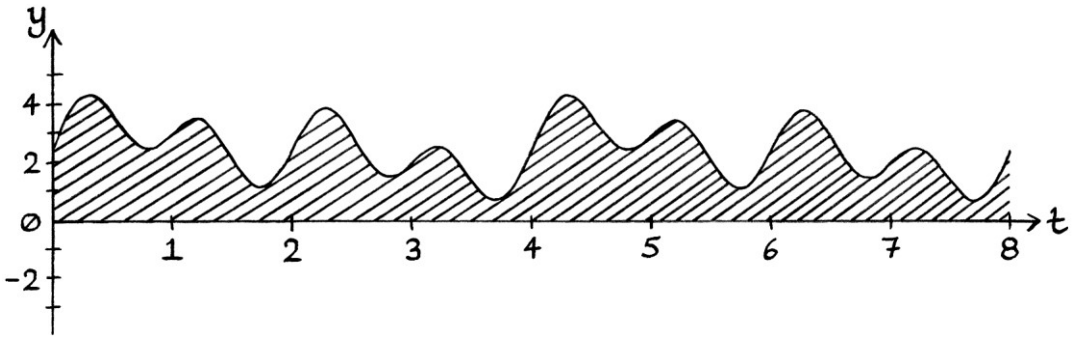
"y = 0.5 sin ((2π * 7t) + 0.5π)"
"y = 0.375 sin ((2π * 7.5t) + 0.5π)"
"y = 0.25 sin ((2π * 7.75t) + 0.5π)
"y = 2.5 sin (2π * 8t)"
"y = 0.25 sin ((2π * 8.25t) + 1.5π)
"y = 0.375 sin ((2π * 8.5t) + 1.5π)
"y = 0.5 sin ((2π * 9t) + 1.5π)

For reference, the three waves in the original signal were:
"y = 0.5 sin (2π * 0.25t)"
"y = 0.75 sin (2π * 0.5t)"
"y = sin 2πt"

The essence of these three waves appears in the resulting signal. In the resulting signal, there are two waves that have frequencies 0.25 cycles per second either side of 8 cycles per second. There are two waves with frequencies 0.5 cycles per second either side of 8 cycles per second. There are two waves with frequencies 1 cycle per second either side of 8 cycles per second. These six waves have amplitudes that are half the values of the original three waves.

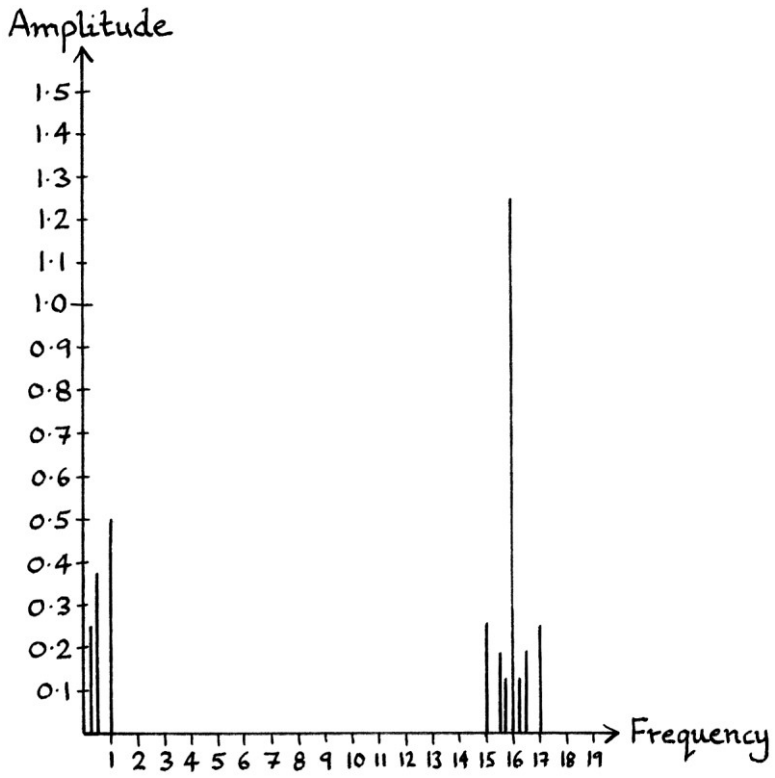
If we were doing amplitude modulation for the purposes of broadcasting them as radio waves, we would have multiplied the original signal by a much faster carrier wave. We would then have been able to transmit the result. For the purposes of a simple example and clearer graphs, it is easier to use slower waves.

To decode the resulting signal, we multiply it by the carrier wave again. We end up with the following signal. The frequency is too fast to make a clear drawing, so I have replaced the curves of the signal with cross-hatching.



[Note how the original signal is still visible in the envelope.]

The signal looks like this in a frequency domain graph:



The waves that make up this signal are:
“y = 0.25 sin (2π * 0.25t)”
“y = 0.375 sin (2π * 0.5t)”
“y = 0.5 sin (2πt)”
“y = 0.25 sin (2π * 15t)”
“y = 0.1875 sin (2π * 15.5t)”
“y = 0.125 sin (2π * 15.75t)”
“y = 1.25 sin ((2π * 16t) + 1.5π)”
“y = 0.125 sin ((2π * 16.25t) + π)”
“y = 0.1875 sin ((2π * 16.5t) + π)”
“y = 0.25 sin ((2π * 17t) + π)”
... and a mean level of 1.25 units.

The first three waves in this list are the waves that made up the original signal, but with half their amplitudes. If we remove the mean level and filter out any frequencies faster than that of the carrier wave (8 cycles per second), we will end up with our original signal with half the amplitudes. To obtain the original signal, we can double the resulting amplitudes.

Thoughts

Multiplication

It is worth pointing out that using *multiplication* to modulate the carrier wave is just a simple method of creating the modulated signal. Theoretically, we could create the identical signal by just transmitting carefully chosen pieces of periodic waves of frequencies centred around the carrier frequency, and ignore the idea of multiplication completely. Similarly, we could decode the signal by observing the waves around the carrier frequency, and deducing how the original audio signal must have looked from that. As it is, for many situations, it is much simpler to use multiplication.

Normally, an audio signal would be an aperiodic signal, as opposed to a periodic one. Therefore, the modulated signal would also be an aperiodic signal, and would consist of countless sections of periodic waves. Therefore, the frequency domain graph would need to be a time-based one to show the full situation.

Formulas

As we have seen, amplitude modulation causes the instantaneous amplitude of a faster frequency wave to be altered according to the instantaneous amplitude of a slower frequency signal. This can be expressed in terms of formulas.

If the carrier wave has the formula:
“$y = A \sin ((2\pi ft) + \phi)$”
... then the modulated signal will have the formula:
“$y = (X * A) \sin ((2\pi ft) + \phi)$”
... where “X” is the instantaneous amplitude of the signal we want to encode at any moment in time (by which I mean the y-axis value of the modulating signal at any particular moment in time). The signal we want to encode must have a non-zero mean level that is high enough that the envelope of the result does not cross the t-axis.

This is the same formula as if we were performing amplitude shift keying.

If we had the original faster frequency wave (the carrier wave) as:
"y = sin (2π * 252,000t)"
... and the modulating signal (the message we want to send) as the pure wave:
"y = 2 + sin (2π * 440t)"
... then the modulated signal would be:
"y = (2 + sin (2π * 440t)) * sin (2π * 252,000t)"

At every moment in time, the instantaneous amplitude of the carrier wave will be scaled by the y-axis value of the modulating wave at that particular time.

Carrier waves

Amplitude modulation relies on our being able to see the original modulating signal in the envelope of the modulated signal. The envelope is the curve that we obtain by joining up the peaks of the modulated signal. How accurately the envelope represents the original signal depends on how close the peaks are to each other. If the peaks are far apart, then the envelope might miss out important parts of the original signal. The modulated signal is actually carrying a *discrete* version of the original signal – in the modulated signal, the original signal exists as a series of individual points (the peaks) at evenly spaced moments in time. [We will look at discrete signals in Chapter 39.] The more points that make up the original signal (in other words, the more peaks there are), the more accurate the representation of the original signal will be. Therefore, it pays to use a carrier wave that has a sufficient frequency for the message that is being sent.

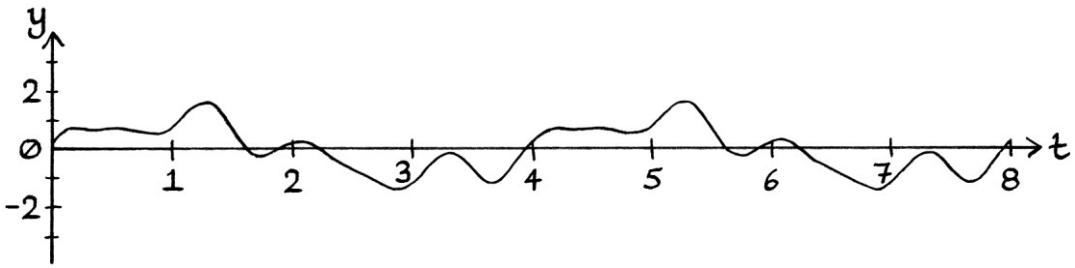
Mean levels

From the "y = (X * A) sin ((2πft) + ϕ)" formula, we can see why multiplication performs amplitude modulation. We can also see why only the modulating signal should have a non-zero mean level. The reasons in more detail are as follows:

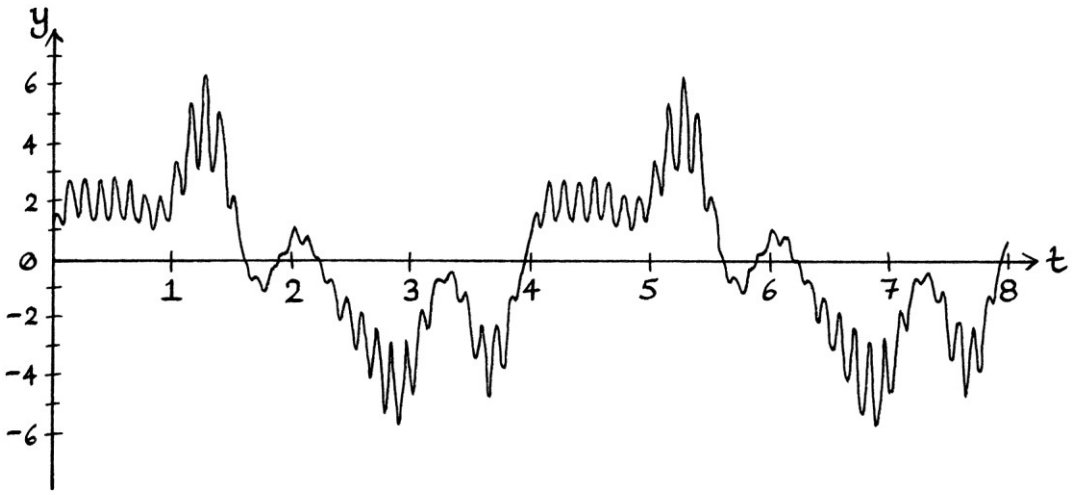
If both the carrier wave and the modulating signal had non-zero mean levels, multiplying the two waves together would result in a non-zero mean level in the result, as well as corrupting the result. [I explained multiplication of waves in Chapter 16.]

The reason we need the mean level added to the modulating signal is more obvious when we think about *amplitude shift keying* using multiplication. In such cases, the square wave must switch between values that are all non-zero and positive. Otherwise, there will be times when the multiplication will zero out the carrier wave and possibly alter the phase. The only way the square wave can switch between entirely positive values is if it has a mean level that keeps all the square wave's values above y = 0.

In the first example of this chapter, we encoded this signal:

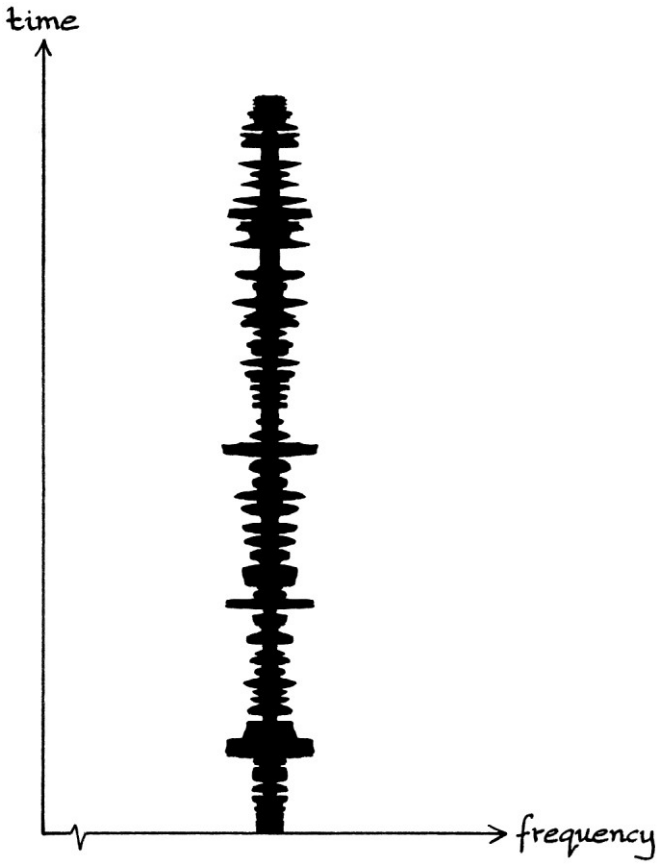


If we had added the mean level to the carrier wave instead of to the signal, we would have ended up with the following signal, which is much less useful (if not completely useless):



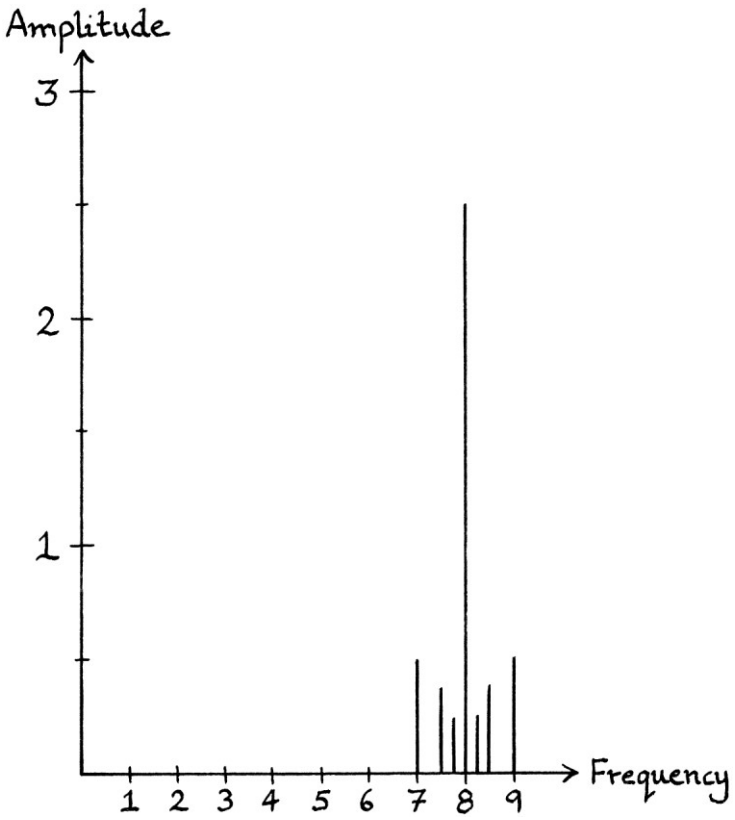
Recognising amplitude modulation

Radio broadcast amplitude modulation is easy to recognise on a real-life frequency domain plot on a computer screen. There will be various frequencies over time, all mirrored around one central frequency. At any one time, the frequencies will generally be closer to the centre frequency. Here is a drawing based on a broadcast from a Long Wave radio station:



Single sideband

When viewed in the frequency domain, an amplitude-modulated signal is mirrored around the frequency of the carrier frequency, as is seen in this example from earlier:



This means that the signal is really the sum of two sets of waves – there is one set with particular amplitudes and phases, and frequencies below the carrier wave's frequency, and there is a duplicate set with the same amplitudes, the negative phases, and frequencies above the carrier wave's frequency. There is also the carrier wave itself.

If, for some reason, we could not see the carrier wave, we would still know its frequency (but not its amplitude or phase) because we know it must be in the centre. Similarly, if we could only see the waves on one side of the centre, we could recreate the missing half because we know it would be a mirror image of the half that we did have. Given all of that, we really only need one half of the waves (as viewed in the frequency domain) without the carrier wave to convey the signal. When we want to decode such a signal, we can recreate the missing parts, and decode the signal as normal.

This idea is used in radio broadcasts in situations where the transmitter power is limited or there is not much available bandwidth. A modulated signal is created as normal, and then it is passed through a high pass filter or a low pass filter to remove one half of the constituent waves and the carrier wave. The remaining signal has fewer constituent frequencies, and so takes up less bandwidth. It also requires less power to transmit, which means that the transmitter can either save energy or transmit the remaining parts of the signal further. [As explained in Chapter 15, a "low pass filter" lets frequencies lower (as in slower) than a chosen value pass on to the next stage of the process. It blocks higher frequencies. A "high pass filter" lets frequencies higher (as in faster) than a chosen value pass on to the next stage of the process. It blocks lower frequencies.]

This type of amplitude modulation is called "single sideband" modulation because there is only one "sideband". In other words, only one side of the bandwidth of the signal (as viewed in the frequency domain) is used to convey the message. Single sideband is often abbreviated to "SSB".

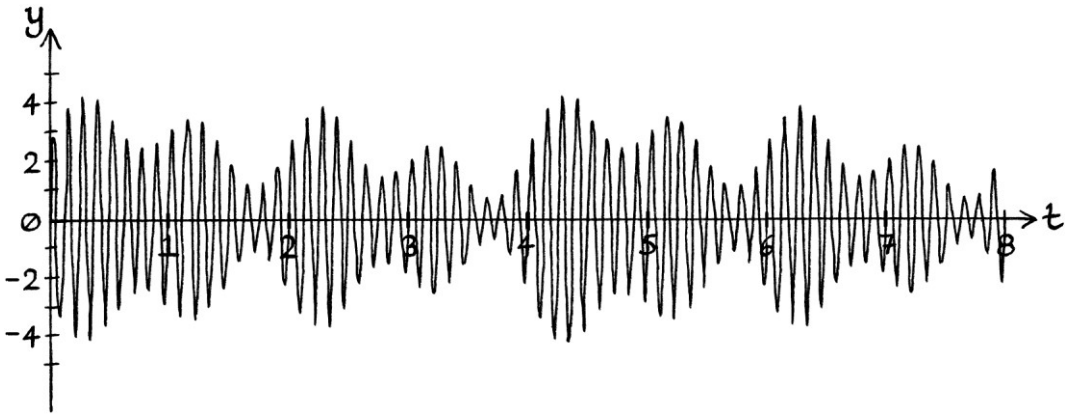
Single sideband has two main types:

- If we remove the higher frequency half of the signal (with a low pass filter), while leaving the lower half, the modulation is called "lower sideband amplitude modulation" or just "lower sideband". It is abbreviated to "LSB".

- If we remove the lower frequency half of the signal (with a high pass filter), while leaving the higher half, the modulation is called "upper sideband amplitude modulation" or just "upper sideband". It is abbreviated to "USB".

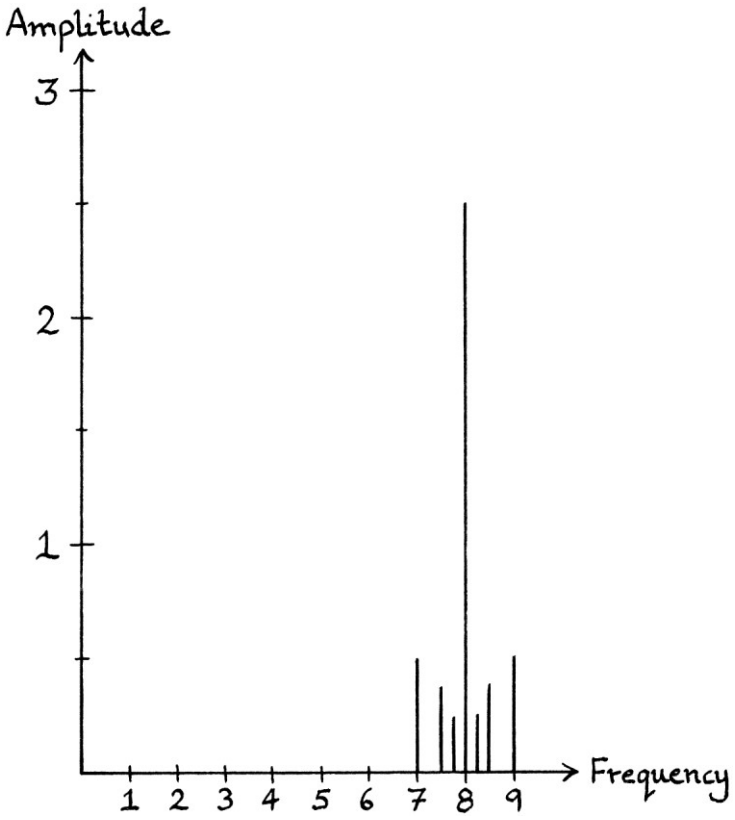
Single sideband is used a great deal by amateur radio broadcasters. Other uses include broadcasting meteorological information.

As an example of single sideband with low frequency waves, we will look at the previous example where the *modulated* signal looked like this:



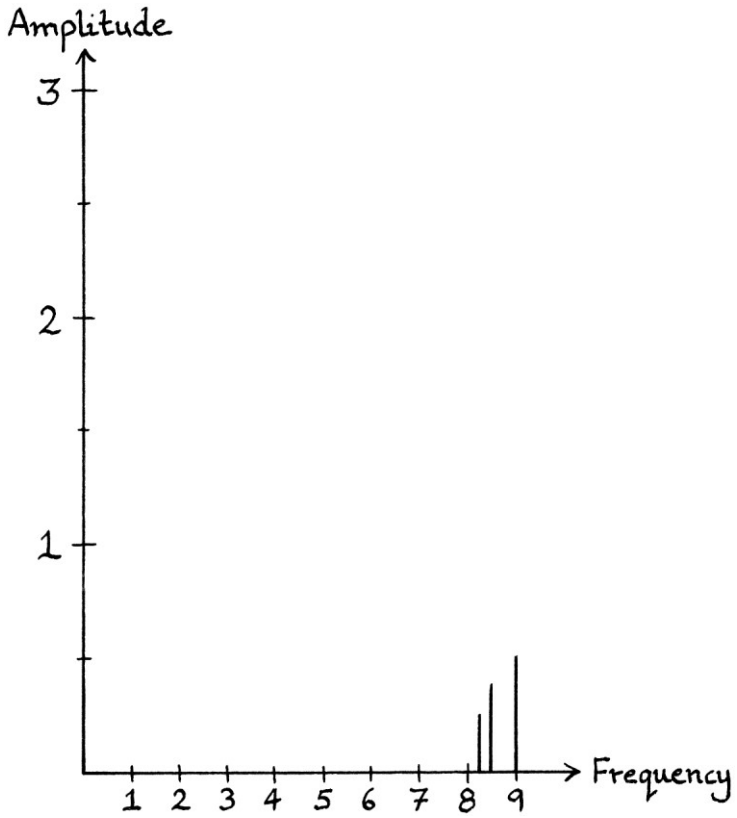
The constituent waves of this signal are:
“y = 0.5 sin ((2π * 7t) + 0.5π)”
“y = 0.375 sin ((2π * 7.5t) + 0.5π)”
“y = 0.25 sin ((2π * 7.75t) + 0.5π)”
“y = 2.5 sin (2π * 8t)”
“y = 0.25 sin ((2π * 8.25t) + 1.5π)”
“y = 0.375 sin ((2π * 8.5t) + 1.5π)”
“y = 0.5 sin ((2π * 9t) + 1.5π)”

The frequency domain graph looks like this:

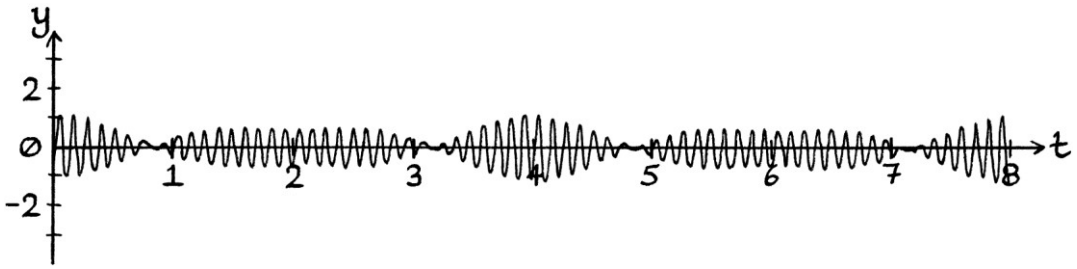


Upper sideband

For upper sideband, we only want to transmit the higher frequency half of the constituent waves. Therefore, we filter out the lower frequency waves and the carrier wave by using a high pass filter. We are left with a signal consisting of just these constituent waves:
"y = 0.25 sin ((2π * 8.25t) + 1.5π)"
"y = 0.375 sin ((2π * 8.5t) + 1.5π)"
"y = 0.5 sin ((2π * 9t) + 1.5π)"
... which look like this in the frequency domain:



The actual signal that is transmitted is this:

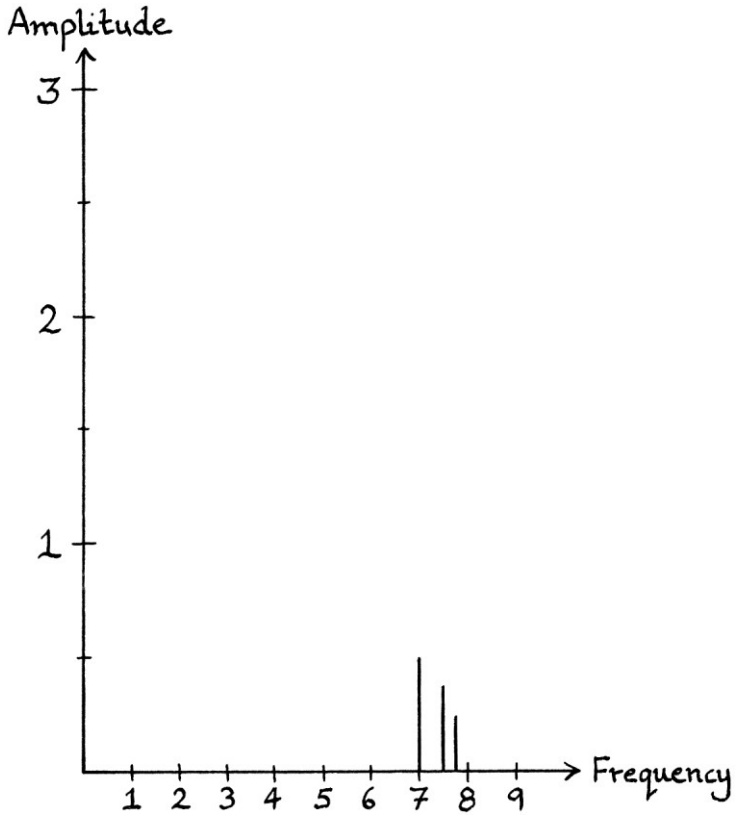


Although the signal has fewer constituent waves in it, it still contains enough information to recreate the original signal. As there are fewer constituent waves, using the same power to transmit it will cause it to travel with a higher perceived amplitude, and so it will heard further away. It will also use up less bandwidth, so be less likely to interfere with other broadcasts. One downside is that it is more effort to decode it in the standard way (through multiplication). To decode the signal, first, we would have to add the carrier wave, and then we would have to mirror the existing waves to the other side of the carrier wave so that the signal is symmetrical when viewed in the frequency domain. The extra effort in decoding SSB signals makes receivers more complicated and expensive than normal AM receivers.

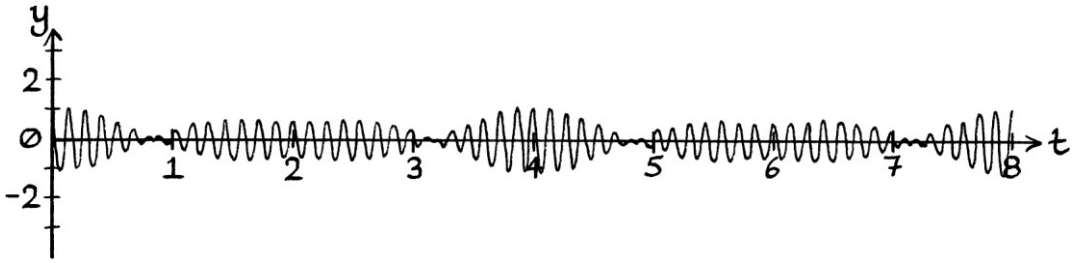
Lower sideband

If we want to send a lower sideband signal, we filter out the carrier wave and the higher frequency constituent waves by using a low pass filter. We would be left with these waves:
"y = 0.5 sin ((2π * 7t) + 0.5π)"
"y = 0.375 sin ((2π * 7.5t) + 0.5π)"
"y = 0.25 sin ((2π * 7.75t) + 0.5π)"

In the frequency domain, the signal would look like this:



The actual signal being sent looks very similar to the upper sideband signal. It looks like this:



To decode the signal, we must first recreate the carrier wave and the missing half of frequencies. Then, we can decode it in the normal way.

Abbreviations

Some common abbreviations that are used when discussing single sideband are as follows. There are several ways to say the same thing.

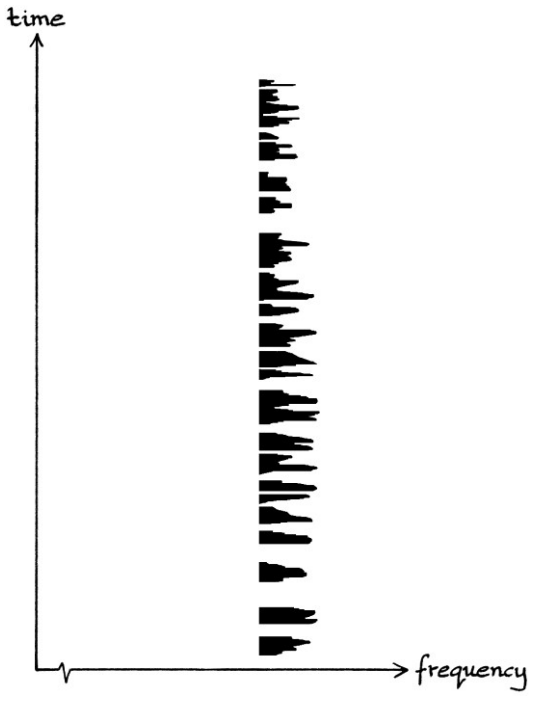
- SSB: Single sideband. The general term for when a signal is sent with half of its frequencies removed.
- USB: Upper sideband. A signal that has had the frequencies below that of the carrier wave removed, as well as the carrier wave itself.
- LSB: Lower sideband. A signal that has had the frequencies above that of the carrier wave removed, as well as the carrier wave itself.
- SSB-SC: Single sideband suppressed carrier: Another name for single sideband. Only one sideband is transmitted and the carrier wave is removed.
- USB-SC: Upper sideband suppressed carrier. Another name for USB. The “SC” part distinguishes it from signals where the carrier is still transmitted.
- LSB-SC: Lower sideband suppressed carrier. Another name for LSB.
- DSB-SC: Double sideband, suppressed carrier. A signal where both sidebands are transmitted as normal, and only the carrier wave is removed.

Thoughts

There is one significant side effect of single sideband that makes it different from normal amplitude modulation. If the audio signal modulating a carrier wave is completely silent, an AM broadcast will consist of a signal containing just the carrier wave, while an SSB broadcast will not produce any signal at all. We can see why this so by imagining an eternally long silent audio signal. It will be a straight line at y = 0 for all time. The audio signal is given a mean level so it can modulate the carrier wave. As it starts as a straight line at y = 0, it ends up as a straight line at a positive non-zero y-axis value. When it is multiplied by the carrier wave, the result is just the carrier wave with a slightly different amplitude. Therefore, the carrier wave remains a pure wave, and therefore, there are no other frequencies in the signal. When we turn an amplitude-modulated signal into a single sideband signal, we remove the carrier wave and either all the frequencies above it or all the frequencies below it. If the only wave in the full AM signal is the carrier wave, then for a single sideband signal where we have removed the carrier wave, there will be nothing left at all.

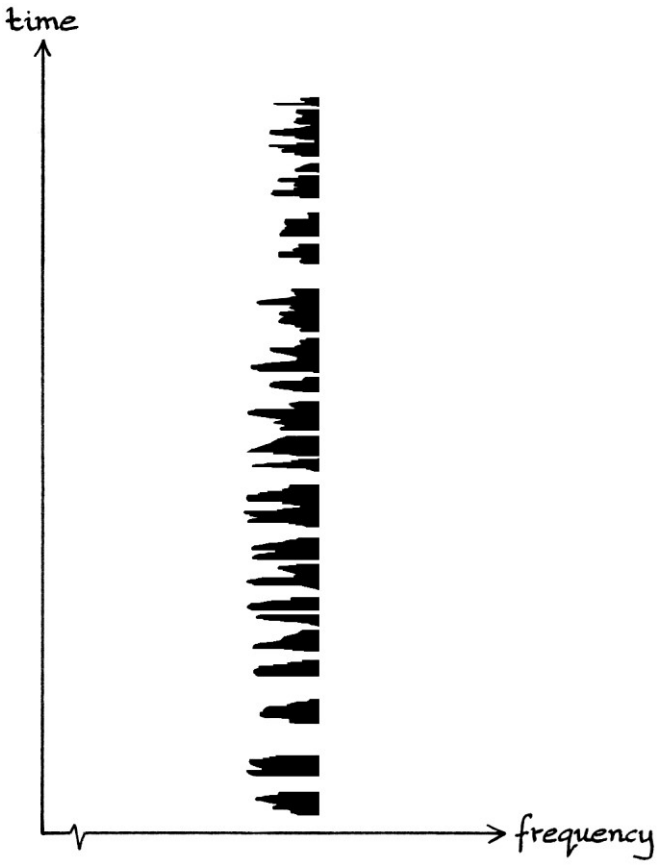
Recognising SSB

If we are observing real-time radio broadcasts in a frequency domain plot on a computer screen, it is easy to spot upper sideband and lower sideband broadcasts by the shape of the signal. Upper sideband looks like this:



Upper sideband has the entire signal to the right of a particular frequency (higher than a particular frequency). There is a distinct boundary to the left of the signal, and there might be gaps in the signal when there is silence. To listen to the broadcast, we would need to tune to the frequency at the left-hand side of the signal.

Lower sideband has the entire signal to the left of a particular frequency (lower than a particular frequency). There is a distinct boundary to the *right* of the signal. As with upper sideband, there might be gaps in the signal when there is silence. To listen to the broadcast, we would need to tune to the frequency at the right-hand side of the signal. Lower sideband looks like this:



Another way to recognise USB and LSB is by the sound they make when decoded incorrectly. If we use a normal AM receiver to listen to a voice that has been broadcast as a USB or LSB signal, we might hear what sounds like a muffled underwater voice. If we have a receiver that can decode USB and LSB, and we pick the wrong one of the two, the voice might sound like that of a high-pitched muffled robot. If we choose the correct decoding type, but tuned slightly away from the correct frequency, the voice will be distorted and slightly higher or lower in pitch than it should be.

Chapter 37: Phase modulation

[In this chapter, we will use radians.]

One might expect non-digital *frequency* modulation to be as simple as frequency shift keying, but, in fact, it is more complicated. Therefore, we will look at phase modulation before we look at frequency modulation.

Phase modulation alters a carrier wave's phase according to the instantaneous amplitude of a given signal. This is exactly what we were doing when we automated phase shift keying, but with PSK, we used a square wave to alter the phase of a carrier wave. Phase shift keying is just a type of phase modulation where the modulating signal is a square wave. Non-digital phase modulation can use any signal to alter the phase.

Unlike amplitude modulation or frequency modulation, phase modulation is seldom used non-digitally. You will seldom see phase modulation unless it is in the form of phase shift keying.

Non-digital phase modulation is a lot harder to visualise than phase shift keying, and without being introduced to PSK first, it can be a difficult subject to understand. Whereas PSK involved sudden jumps in phase, phase modulation performed with audio signals has continuous alterations in phase. It is difficult to discern visually that the phase has changed, or even to tell that phase modulation is being used.

As with PSK, non-digital phase modulation can be done with either multiplication or addition. Addition is slightly easier to understand, so we will concentrate on addition in this explanation. With the addition method of phase shift keying , we could use either of two methods (using waves in radians):

- We could use a square wave that switched between values from 0 units to just under 2π units, and set the instantaneous phase of the carrier wave to those values.
- We could use a square wave that switched between values from 0 units to just under 1 unit, and then multiply the values by 2π and set the phase of the carrier wave to the result.

When performing non-digital phase modulation, we can still use the basis of both these ideas:

- We can have all the y-axis values of our modulating signal fluctuate between 0 units and just under 2π units, and set the instantaneous phase of the carrier wave to those values. The formula for this method would be:
  "y = A sin ((2π * ft) + X)"

- We can have all the y-axis values fluctuate between 0 units and just under 1 unit, then multiply each value by 2π, and then set the instantaneous phase of the carrier wave to those values. The formula for this would be:
  "y = sin ((2π * ft) + (2π * X))"

[We could use variations of these methods. For example, for the first method, the instantaneous amplitudes could vary between just over −π units and just under +π units, and for the second method, they could vary between −0.5 units and +0.5 units.]

When we use *phase shift keying*, the square wave that modulates the carrier wave can be created by us to our own specifications. When we use non-digital phase modulation, we will usually be working with a signal that we did not create, and therefore, it will almost certainly require scaling to fit in with what we want to do. Therefore, the choice is whether we scale it to be below 2π units, or scale it to be below 1 unit and then multiply it by 2π. Ultimately, it makes no difference which we do because the results will be the same. [Scaling between 0 and 1 is essentially using "whole circle angle units" as first described in Chapter 22.]

As with phase shift keying with addition, we are not so much *adding* to the phase as *setting* the phase. However, we will still call this "addition" to be consistent with the previous chapter.

The essence of the formulas is that the instantaneous phase of the carrier wave increases and decreases in accordance with the instantaneous amplitude of the modulating signal. This is the same as for phase shift keying using addition. The instantaneous amplitude of the modulating signal needs to be kept between a particular maximum and minimum so that the addition to the phase never results in a phase that rolls around – it should never be possible that two different instantaneous amplitudes can result in the same phase in the resulting signal. If we were using the first method and the instantaneous amplitudes reached from 0 units to 2.1π units, then any value over 2π would end up producing the same phase as a value from 0 to 0.1π units would do. If we were using the second method, and

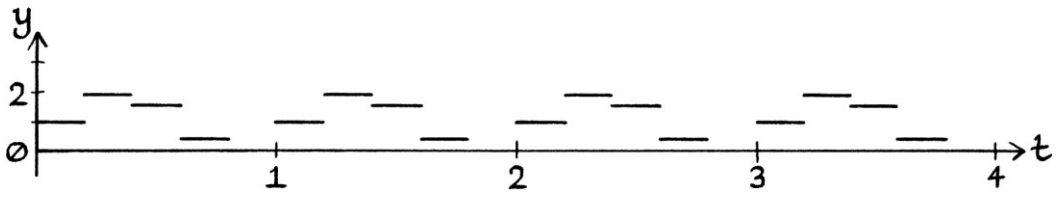
there were any instantaneous amplitudes of 1 unit or higher, they would result in the phase wrapping around, and be equivalent to instantaneous amplitudes of 0 units and higher.

We do not have to use the entire range of instantaneous amplitudes from 0 to just under 2π units (for the first method), or from 0 to just under 1 unit (for the second method), but it makes the best use of the available phases if we do. Conversely, to avoid having amplitudes of exactly 2π units or more (for the first method) and amplitudes of 1 unit or more (for the second method), it can be useful to keep the range of instantaneous amplitudes slightly lower than the maximums. [Depending on what we are actually doing, it might not matter too much if a few resulting phases roll around slightly.]

In the following examples, we will use the first method of addition – our signal will have instantaneous amplitudes from 0 to just under 2π units. We might or might not make the full use of the available values, depending on what is clearest for the graphs and the explanation. [Note that when we performed phase shift keying, we used the second method with values from 0 to just under 1 unit, and then we multiplied the values by 2π.]

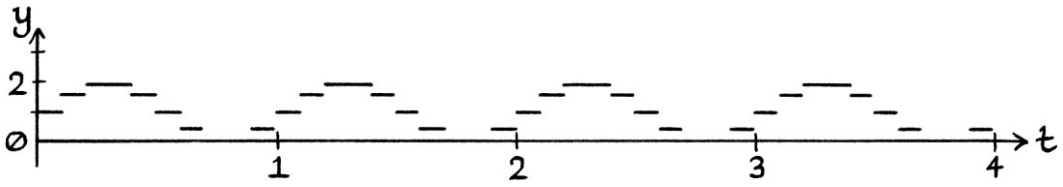
Visualising phase modulation

Phase modulation can be difficult to visualise. An easy way is to think of it is as being the same as phase shift keying, but with an infinite number of levels. We will imagine that we are using phase shift keying on a “Sine wave” that has been split into levels as so:

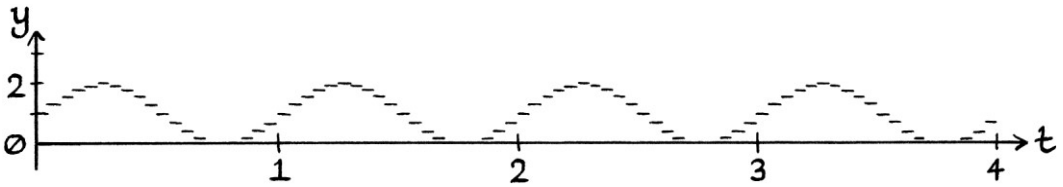


We can easily imagine such a signal being phase shift keyed.

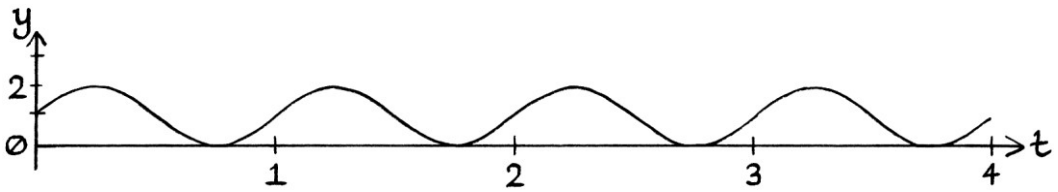
We can then imagine it with more levels, so that it becomes more wave-like:



And more levels still:



Eventually, we can imagine it having infinite levels like this:

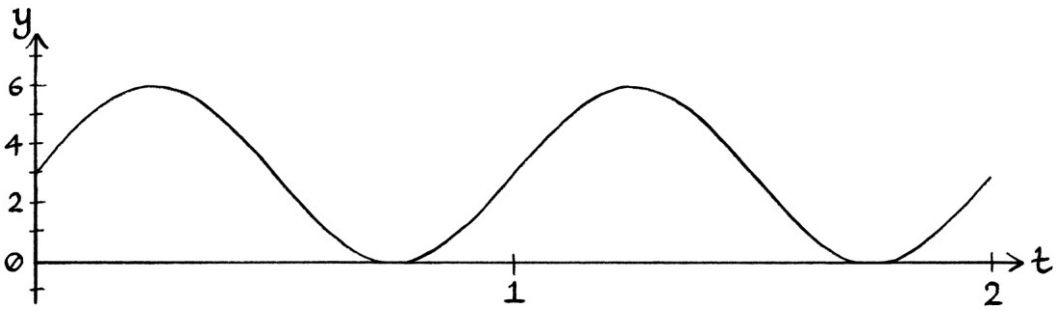


Examples

We will look at some examples of phase modulation with addition.

Example 1

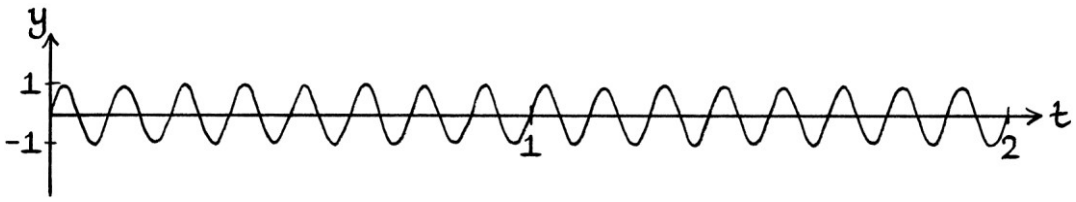
We will start with the wave "y = 3 + 3 sin (2π * 1t)" as the modulating wave:



The y-axis values (the instantaneous amplitudes) range from 0 units up to 6 units. This means that the instantaneous phases of the resulting signal will vary from 0 radians up to 6 radians. The angle of 2π radians is the same as 6.2832 radians, so we are making good use of the available phases, but without risking going too high and wrapping around.

If our modulating wave had a different maximum or minimum, we could shift the minimum to y = 0 and scale the maximum to just under 2π units. [If we were working in degrees, we would best have the maximum as just under 360 units. Everything would still work, but using degrees makes the wave y-axis harder to draw. In this example, it is more convenient to work in radians, and it is more convenient to go up to 6 units.]

Our carrier wave will be "y = sin (2π * 8t)":

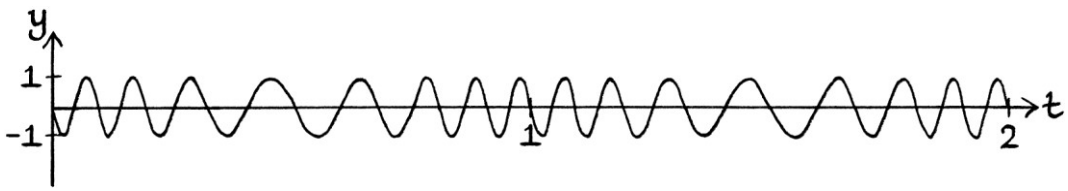


If "X" is the instantaneous amplitude of the modulating wave at any time, then our resulting formula will be:
"y = sin ((2π * 8t) + X)"

The exact (but harder to read) formula for the resulting signal at any moment in time will be:
"y = sin ((2π * 8t) + (3 + 3 sin (2π * 1t)))"

This means that the phase of the modulated signal will be constantly changing. The instantaneous phase (in radians) at any moment in time will be the same as the instantaneous amplitude (in units) of the modulating wave.

The resulting signal looks like this:

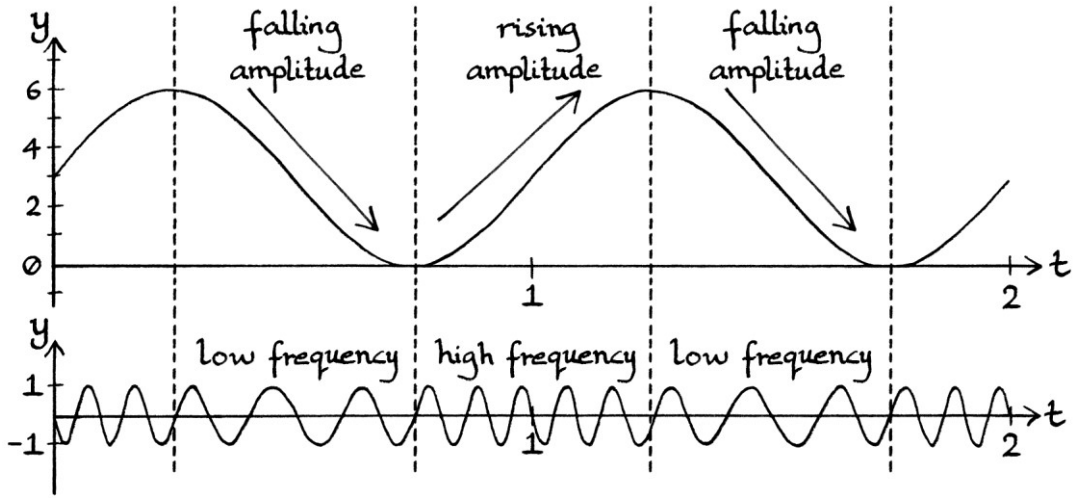


Note how the curve starts “upside down”. This is because the initial y-axis values start at 3 units and rise from there. The initial instantaneous phases, therefore, start at 3 radians and rise from there. The angle of 3 radians is just under π radians. It is about 172 degrees.

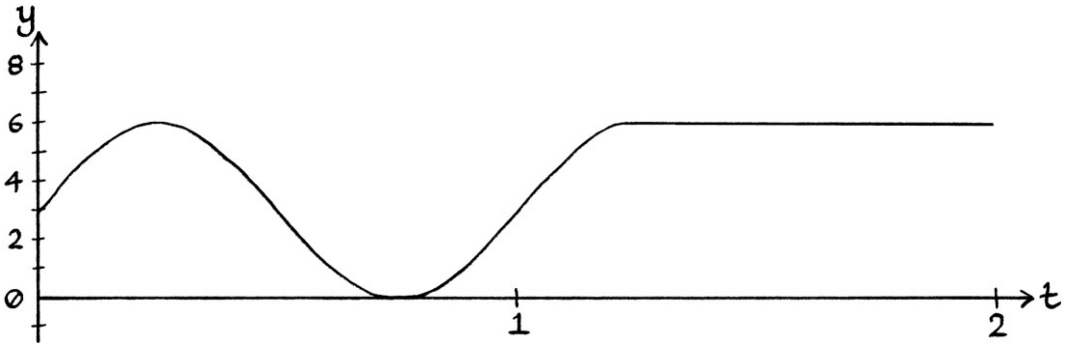
When the instantaneous amplitude of the modulating wave is low, the instantaneous phase is low. When the instantaneous amplitude of the modulating signal is high, the instantaneous phase is high.

An important observation is that the resulting signal speeds up or slows down depending on the nature of the modulating wave at any particular time – the resulting signal’s instantaneous *frequency* is changing. Its frequency is changing due to the reasons explained in the section “frequency and phase” in Chapter 35. That section explained that constantly increasing the phase of a wave increases the frequency. This is because we are constantly bringing parts of the wave from future times to earlier times. The cycles speed up. Similarly, constantly decreasing the phase decreases the frequency because we are constantly delaying parts of the wave. Therefore, the cycles take longer to appear.

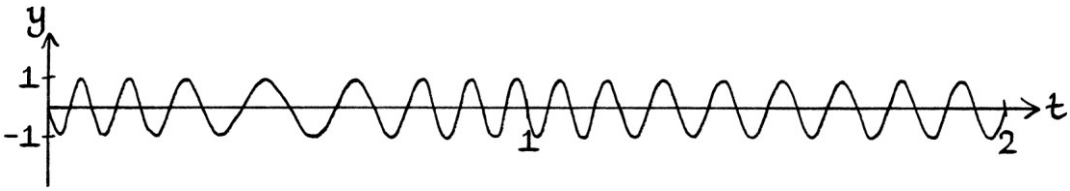
In this example, the phase increases and decreases as the instantaneous amplitude of the modulating wave rises and falls. Whenever the instantaneous amplitude of the modulating wave is continually rising, the phase of the carrier wave will be continually increasing. Therefore, at those times, the instantaneous frequency of the resulting signal will be higher. Whenever the instantaneous amplitude of the modulating wave is continually decreasing, the phase of the carrier wave will be continually decreasing too. Therefore, at those times, the instantaneous frequency of the resulting signal will be lower.



When the instantaneous amplitude of the modulating wave is high, the phase added to the carrier wave will be high. When the instantaneous amplitude of the modulating wave is low, then the phase added to the carrier wave will be low. However, the instantaneous *frequency* of the result at a particular time depends on a *change* in phase. Therefore, if the phase were high and remained high, the instantaneous frequency would stay at a fixed value. Similarly, if the phase were low and remained low, the instantaneous frequency would stay at a fixed value. As an example, we will change our modulating wave so that it stays at 6 units after 1.25 seconds:



We will keep the carrier wave as before with a frequency of 8 cycles per second. Now the resulting signal looks like this:



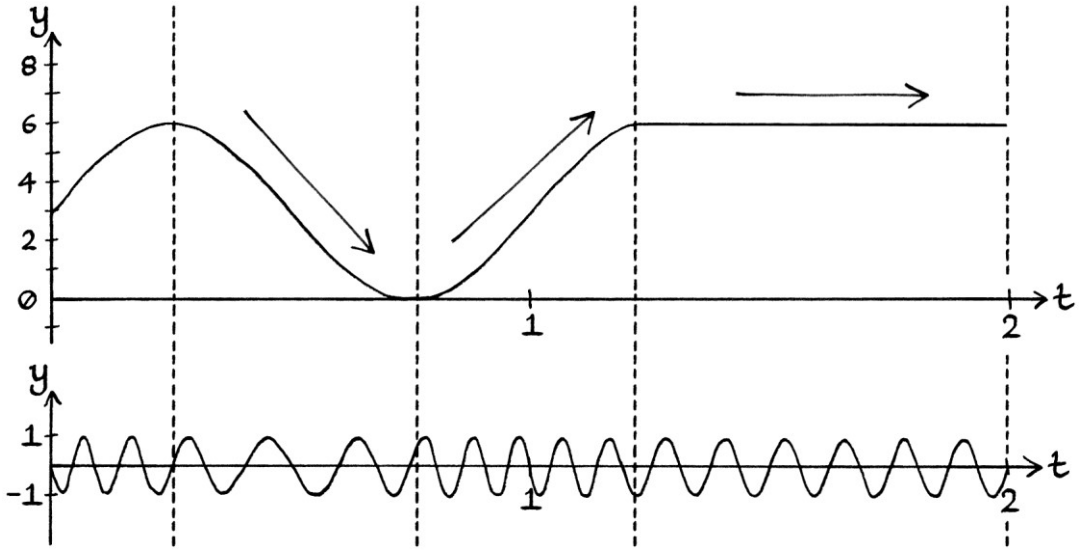
In the above signal, the instantaneous frequency starts at *roughly* 10 cycles per second, and gradually decreases until about 0.25 seconds. During this time the instantaneous amplitude of the modulating signal is rising. The rising instantaneous amplitude causes a rising phase in the carrier wave, and the rising phase causes the instantaneous frequency of the result to be faster than it would be otherwise. As the rate of increase of the instantaneous amplitude of the modulating wave decreases (as it approaches its maximum of 6 units), the frequency of the result begins to slow down. The underlying rule for this is that an increasing phase causes a faster frequency. Strictly speaking, an increasing phase **is** a faster frequency. A phase increasing at a constant rate is the same as a fixed faster instantaneous frequency for the time that the phase is changing. In this example, the phase increase is not completely constant – it slows down as the modulating signal reaches 6 units – but the increase still demonstrates the idea.

When the instantaneous amplitude of the modulating signal falls, so does the phase of the result, and because the fall is continuous, this results in a slower instantaneous frequency in the result for that time.

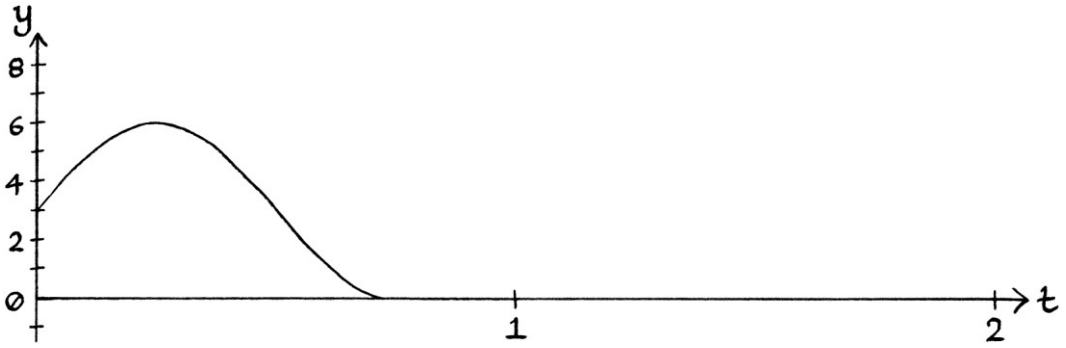
When the instantaneous amplitude of the modulating signal rises again, it makes the phase of the carrier rise continuously, which causes the frequency to be higher.

At 1.25 seconds, the modulating signal becomes fixed at 6 units for the rest of its existence. The phase of the carrier wave becomes fixed at 6 radians. As the phase is neither rising nor falling, the instantaneous frequency of the result is unaffected and stays at 8 cycles per second for the rest of the signal, which is the same frequency as that of the original carrier wave. It is only a continuous increase in the phase or a continuous decrease in the phase that alters the frequency. If the phase stays the same, then the frequency returns to its default, which, in this case, is the carrier wave's frequency of 8 cycles per seconds.

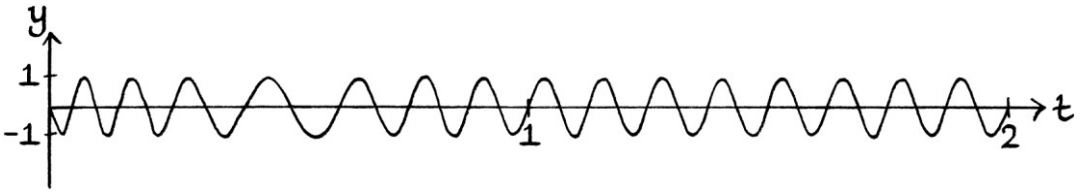
The modulating signal and the modulated signal together are as so:



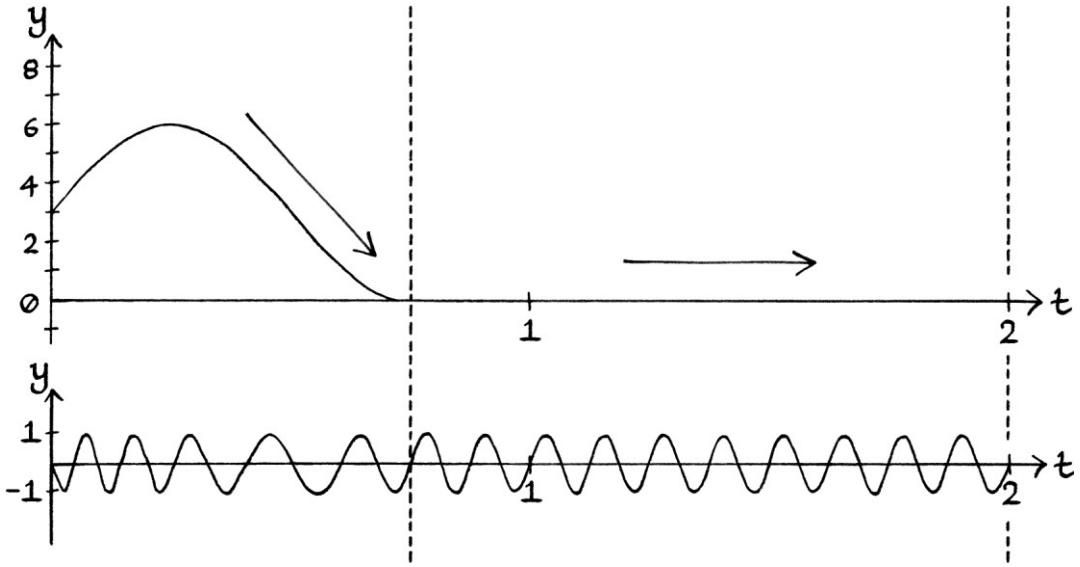
Now we will change the modulating wave so that it stays low (at 0 units) at 0.75 seconds:



We will keep the carrier wave the same as before at 8 cycles per second. The phase-modulated result is as so:



After 0.75 seconds, when the instantaneous amplitude of the modulating signal is 0 units, the phase of the carrier wave will be 0 radians. As the phase is neither rising nor falling, the instantaneous frequency of the result stays at 8 cycles per second.

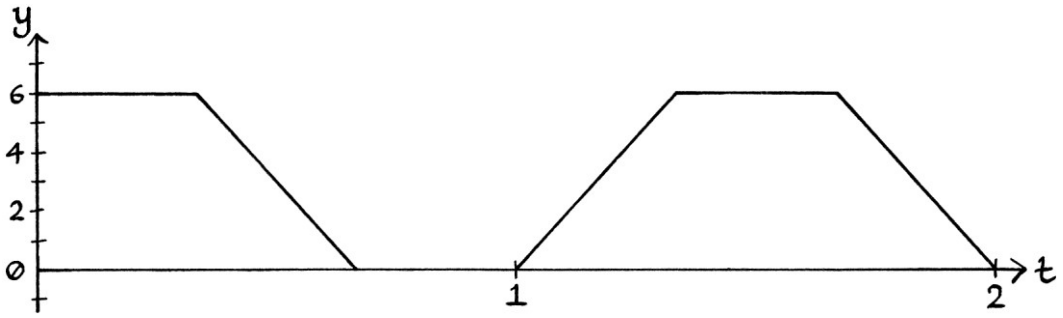


Summary so far

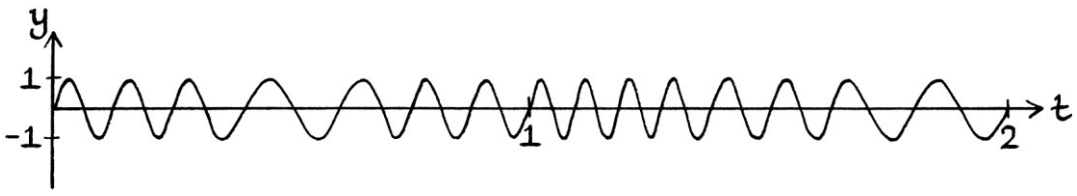
From all of the above, we can see that phase modulation results in an increased instantaneous frequency while the instantaneous amplitude of the modulating signal is rising, and a decreased frequency while the amplitude is falling. The frequency stays as that of the carrier wave whenever the amplitude stays at a fixed level. These are important observations because they are the basis of non-digital *frequency* modulation, which we will look at later in this chapter, and in Chapter 38.

Example 2

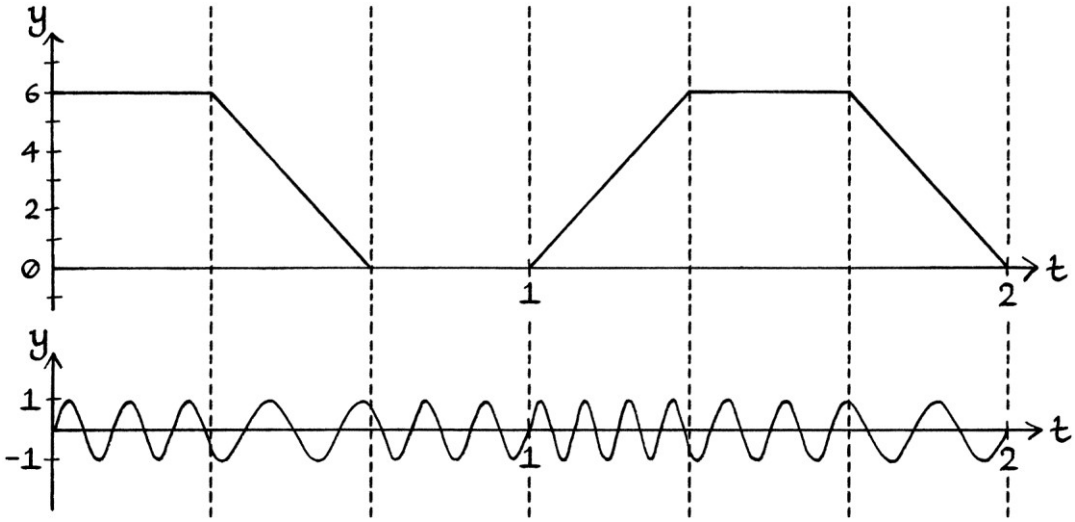
As a simpler example of the effect that phase modulation has, we will look at a more basic example than a Sine wave. We will use the following signal to phase modulate an 8-cycle-per-second carrier wave:



The resulting signal looks like this:



The two signals together are as so:



While the modulating signal has a stationary instantaneous amplitude, the result has a stationary instantaneous frequency of 8 cycles per second, which is the frequency of the carrier wave. It does not matter whether the amplitude is 6 units or 0 units at these times.

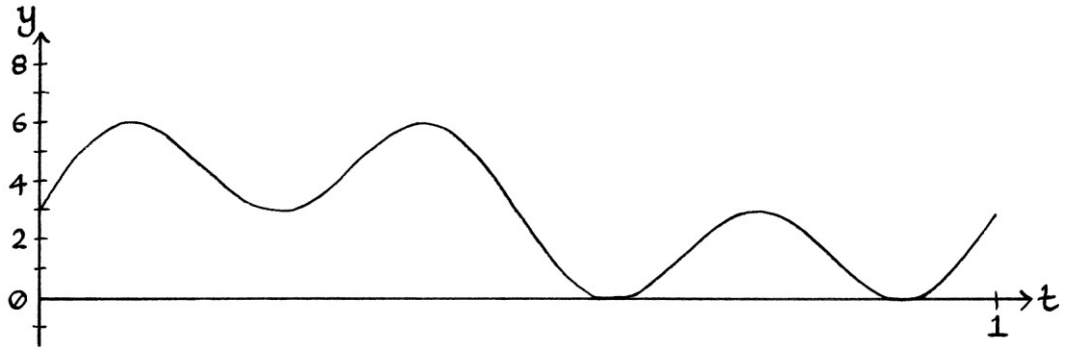
While the instantaneous amplitude of the modulating signal is falling, it does so at a constant rate of about 18 units per second or 1.8 units per 0.1 seconds. At these times, the resulting signal has a slower, but fixed frequency of 5 cycles per second.

When the instantaneous amplitude of the modulating signal is rising, it does so at a constant rate of about 18 units per second. At these times, the resulting signal has a faster, but fixed frequency of 11 cycles per second.

The constantly decreasing instantaneous amplitude of the modulating signal lowers the frequency of the carrier wave by 3 cycles per second. The constantly increasing instantaneous amplitude of the modulating signal raises the frequency of the carrier wave by 3 cycles per second.

Example 3

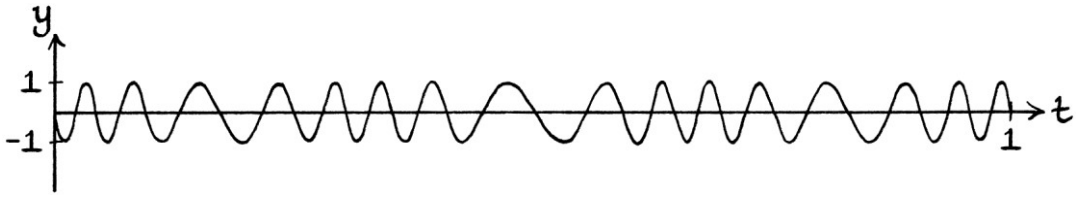
For the next example, we will use the following signal to phase modulate a carrier wave with a frequency of 16 cycles per second.



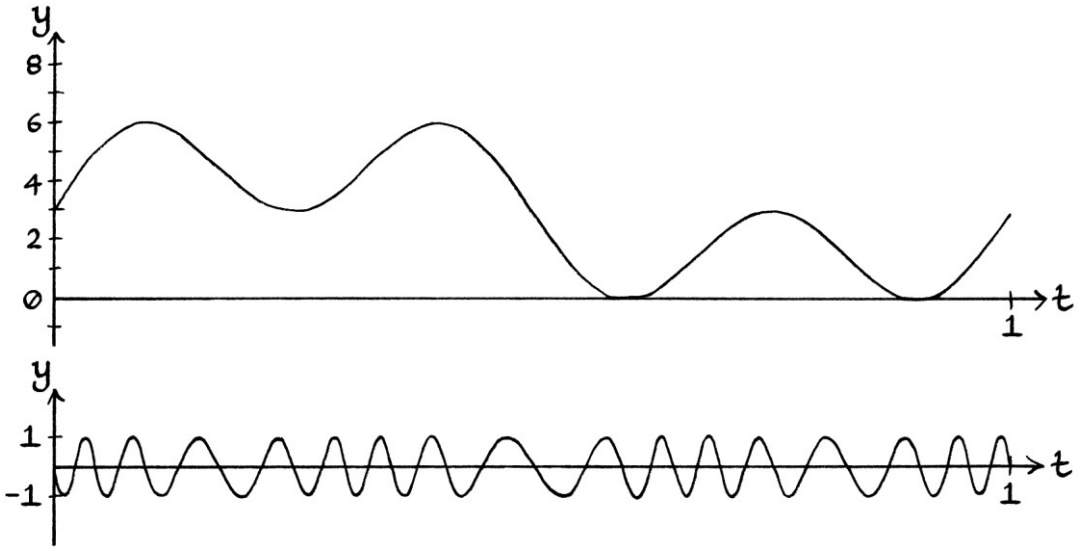
The above signal is the sum of:
"y = 3 + 1.9486 sin (2π * 1t)"
... and:
"y = 1.9486 sin (2π * 3t)"
... so we could say that its formula is:
"y = 3 + 1.9486 sin (2π * 1t) + 1.9486 sin (2π * 3t)"

[The signal has the characteristic look of the sum of two waves of the same amplitude and with a frequency ratio of 3 : 1.]

The signal fluctuates between y = 0 and y = 6. The resulting signal looks like this:

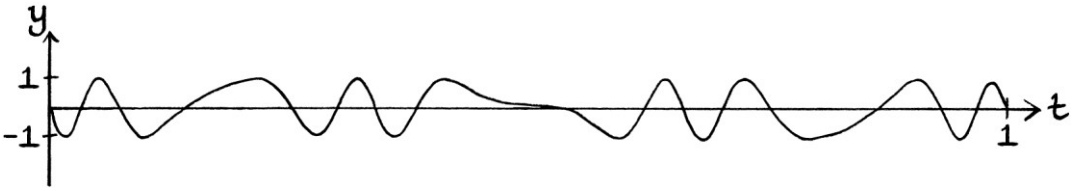


The two signals together for comparison are as so:



This gives a good illustration of what to expect when using phase modulation to encode a signal into a carrier wave.

For interest's sake, if we had used a carrier wave with a slower frequency, the resulting signal would have become corrupted, in the sense that the "cycles" would not all have reached to the maximum and minimum. The following signal is the result of phase modulating the example from before:
"y = 3 + 1.9486 sin (2π * 1t) + 1.9486 sin (2π * 3t)"
... with a carrier wave with a frequency of just 8 cycles per second:

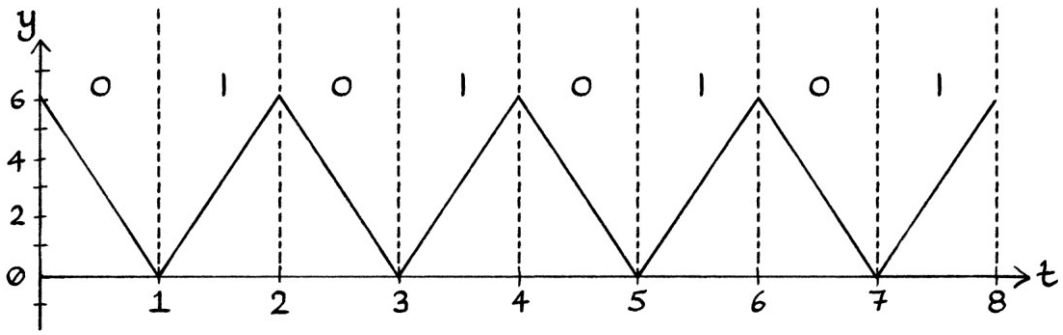


It turns out like this because the carrier wave's frequency is not fast enough to deal with the more complicated modulating signal. Whether the resulting signal is usable or not depends on how it is going to be used and how it is going to be decoded. If it happens to be the case that only one modulating signal combined with an 8-cycle-per-second carrier wave could have created this signal, then the signal could be decoded correctly. However, from a radio transmission point of view, it might be better if the peaks all reached the maximum, and the dips all reached the minimum.

FSK with phase modulation

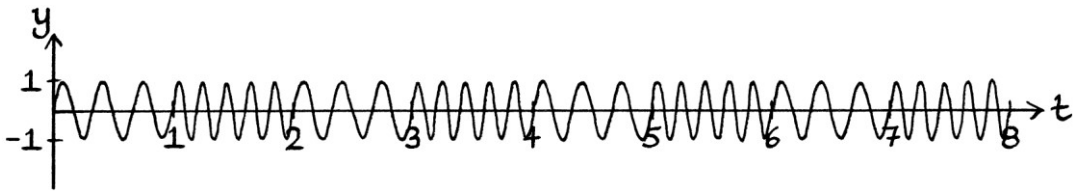
Given how a continually increasing phase produces a higher frequency, and how a continually decreasing phase produces a lower frequency, it is possible to control the frequency of a wave by constantly raising or lowering its phase. This means that we can use phase modulation to control the frequency of a wave, which in turn means that we can use phase modulation to perform frequency shift keying. [This also means that we can use phase modulation to perform non-digital frequency modulation, but we will look at that in the next chapter.]

Normally, if we were doing *frequency* shift keying, we would use a square wave that represented those digits, and then we would add the square wave to (or multiply the square wave by) the frequency in the formula of the carrier wave. As we are going to be using phase modulation for frequency shift keying, we cannot use a square wave. Instead, we must use a signal where a constantly rising line represents a one and a constantly falling line represents a zero. For the binary digits, "01010101", we could use the following signal, which varies between 0 units and 6.2 units (2π is 6.2832). [I am using 6.2 units instead of 6 units to make better use of the available units and phases. This becomes more important in these examples.]



When the instantaneous amplitude is falling, the phase of the carrier will be falling at a constant rate. Therefore, the resulting signal will have a fixed frequency that will be slower than that of the original carrier wave. When the instantaneous amplitude is rising, the phase of the carrier will be rising at a constant rate. Therefore, the resulting signal will have a fixed frequency that will be faster than that of the original carrier wave.

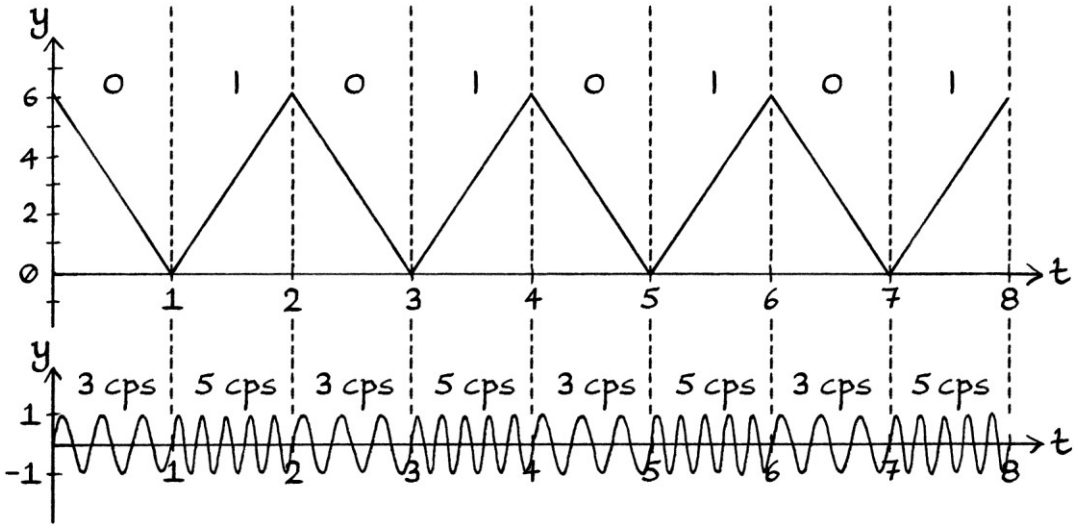
For a carrier wave of 4 cycles per second, the resulting signal will look like this:



[I am using a carrier wave of 4 cycles per second for the sole reason that it makes the resulting picture clearer.]

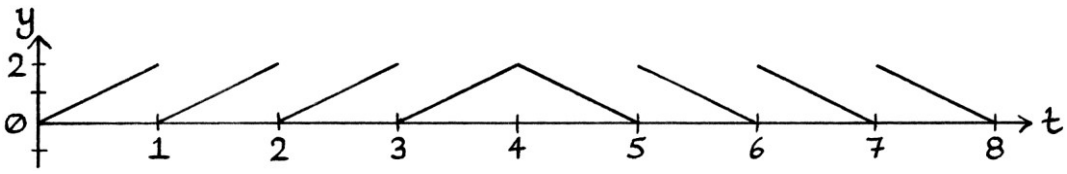
The different frequencies are clear to see. Whenever the modulating signal is falling, the resulting instantaneous frequency is slower (3 cycles per second); whenever the modulating signal is rising, the resulting instantaneous frequency is faster (5 cycles per second). We have achieved frequency shift keying by using phase shift keying.

The two signals together with notes are as so:

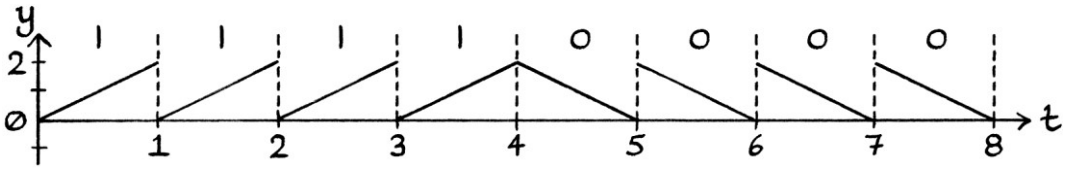


In our modulating signal, the y-axis values rise from 0 to 6.2 units, so the phases rise from 0 to 6.2 radians. One potential problem with the values not rising to very near to 6.2832 units (2π units) is that if we have binary numbers with two or more consecutive ones or zeroes, there will be an instant jumps in phases after each bit that might affect the continuity of the resulting signal.

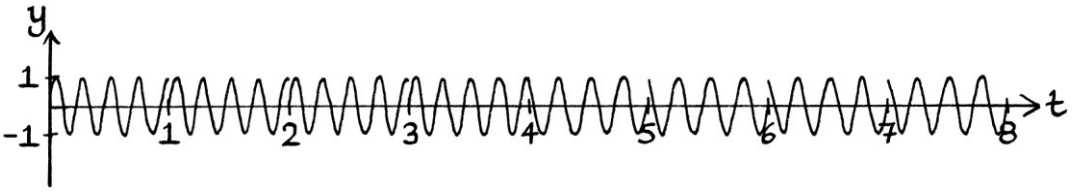
An extreme version of this can be seen if we only have y-axis values rising from 0 to 2 units, as shown here, where the modulating signal is encoding the digits "11110000":



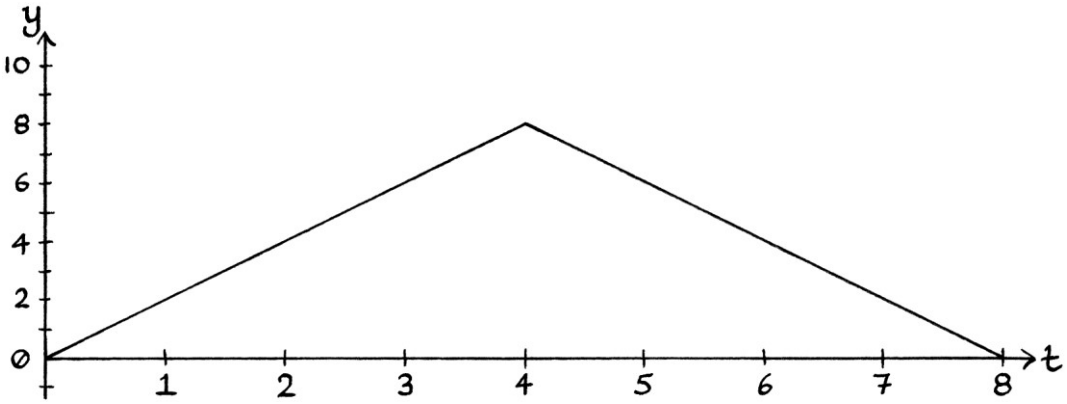
The same graph with the digits that are being represented is as so:



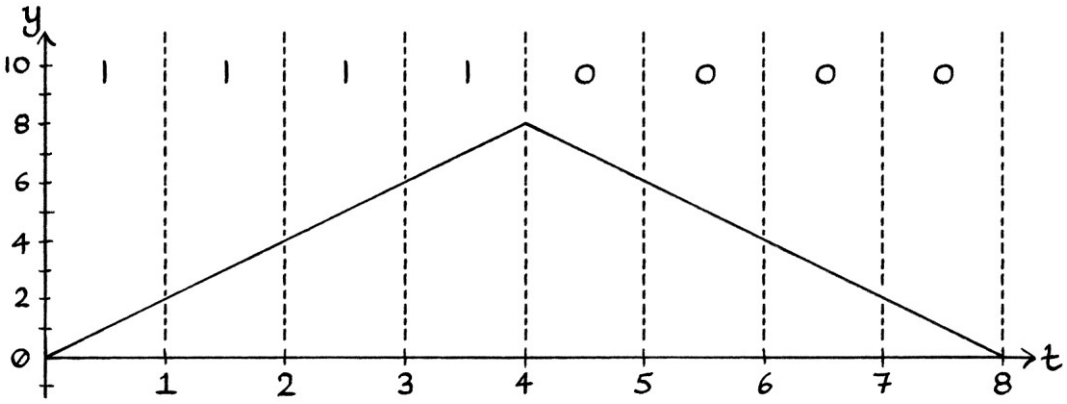
The resulting modulated signal contains vertical jumps (discontinuities).



One way to stop producing the jumps would be for the diagonal lines representing consecutive ones or zeroes to carry on, as shown here:



The same graph with the digits marked is as so:

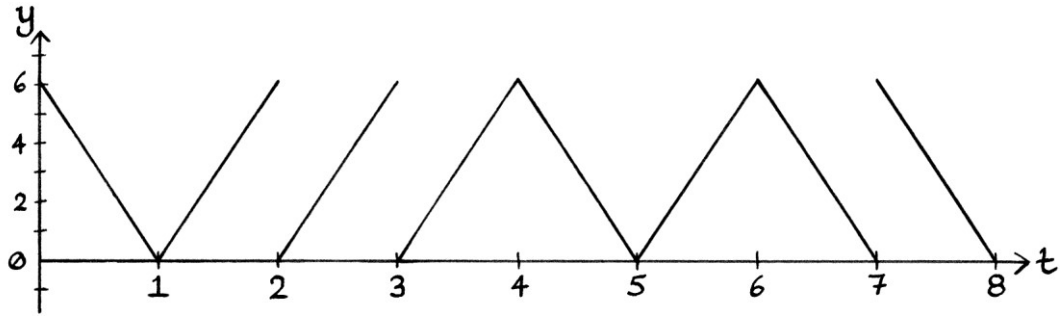


However, this can become unwieldy if there is a very long sequence of ones or zeroes. For example, a run of 100 ones would reach a very high y-axis value.

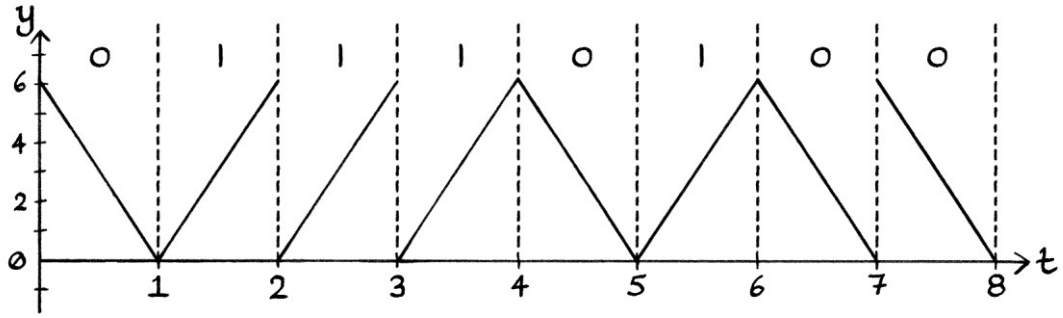
The simplest solution is to have the maximum y-axis values (instantaneous amplitudes) as close to 2π units (6.2832 units) as possible without equalling it or going over it. The closer the y-axis values are to 2π, the smaller any phase jumps will be. If the diagonals move from 0 units to 6.2 units, it will be hard to detect any jumps in phase. In this way, if we have consecutive ones or zeroes, there will be a visible jump in the *instantaneous amplitude* in the modulating signal, but that jump will essentially be a continuous movement when it comes to the *phase* that is added to the carrier wave. This is because 0 radians and 6.2 radians are nearly the same as each other. [If we were working in degrees, the instantaneous amplitudes would have to rise and fall between 0 and just under 360 degrees instead.]

FSK and gradients

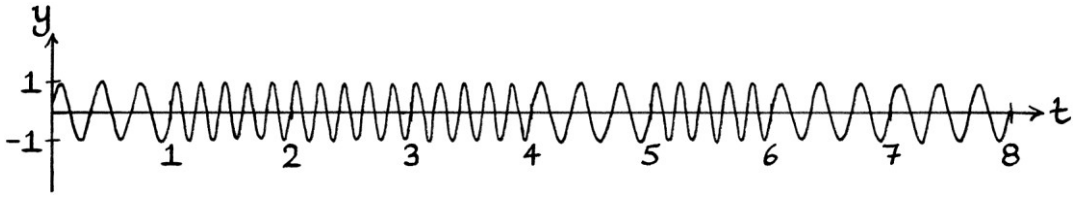
We will now investigate the underlying basis for using phase modulation to perform frequency shift keying. We will use phase modulation to perform frequency shift keying to encode the binary digits “01110100”. We will use the following modulating signal, where the y-axis values rise from 0 units up to 6.2 units to indicate a one, and fall from 6.2 units down to 0 units to indicate a zero. Each rise or fall lasts for one second.



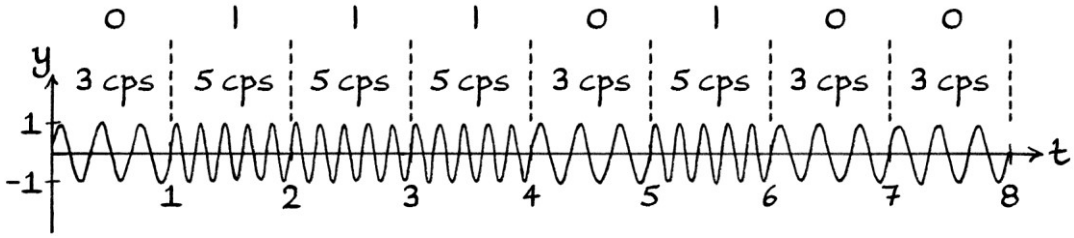
With the digits marked, the graph looks like this:



We will use a carrier wave with a frequency of 4 cycles per second. The resulting modulated signal looks like this:

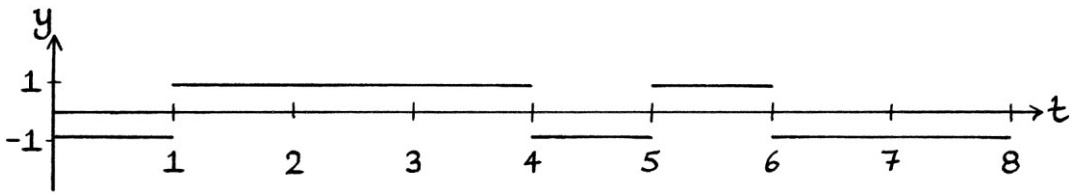


With the binary digits and the frequencies marked, it looks like this:

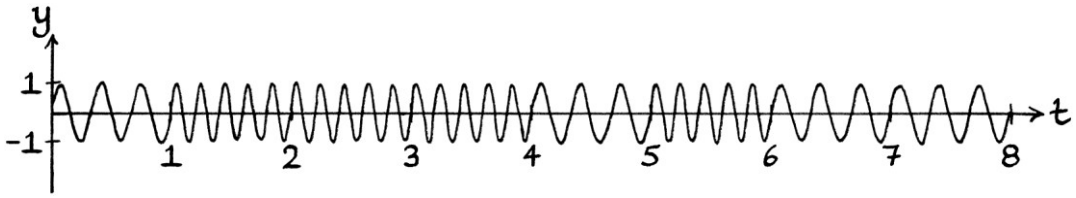


Whenever the modulating signal is falling, the resulting instantaneous frequency is slower (3 cycles per second). Whenever the modulating signal is rising, the resulting instantaneous frequency is faster (5 cycles per second).

If we had wanted to achieve the same result by encoding the "01110100" signal using *frequency shift keying* and addition, we would have needed to use this square wave (where the values switch between just above −1 and just below +1).



With a carrier wave of 4 cycles per second, the resulting signal would have ended up as so:



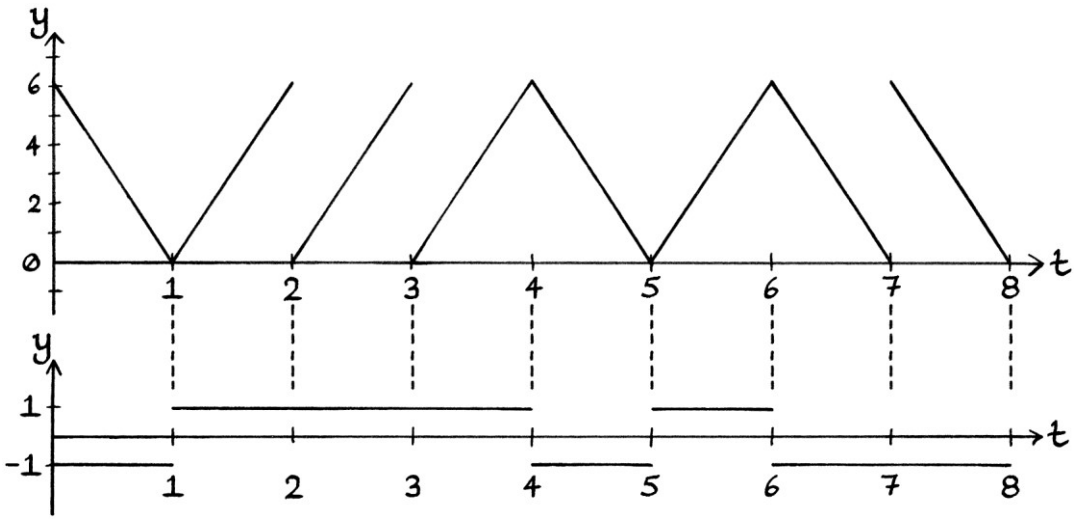
This is *exactly* the same result as when we used phase modulation with the diagonal wave. Having two ways to achieve frequency shift keying reinforces how frequency is a continuous increase or decrease in phase.

In the phase modulation FSK example, the y-axis values of the modulating signal were either increasing at a rate of 6.2 units every second, or falling at a rate of 6.2 units every second. Therefore, the instantaneous phases of the carrier wave were altered either to increase at a rate of 6.2 radians every second, or to fall at a rate of 6.2 radians every second. The original carrier wave had a frequency of 4 cycles per second. The instantaneous frequencies of the result were either 1 cycle per second faster than the carrier wave or 1 cycle per second slower. In the non-phase-modulation FSK example, the y-axis values of the modulating signal were either under +1 units or above −1 units. The carrier wave had a frequency of 4 cycles per second. As before, the instantaneous frequencies of the result were either 1 cycle per second faster than the carrier wave or 1 cycle per second slower.

An interesting observation about the above situation is that when the instantaneous amplitudes of the frequency modulation FSK square wave signal are multiplied by 6.2, they are actually the rate of change of the instantaneous amplitudes of the phase modulation FSK "diagonal" signal. [Remember that the values of the diagonal wave moved from 0 units up to 6.2 units.] We can think of this the other way around: the "+1, −1" FSK square wave shows the rates of change of the PSK diagonal signal, after the y-axis values of the PSK signal have been

divided by 6.2. It is also the case that the square wave at any particular time shows the gradient of the diagonal signal at that same time, after *that gradient* has been divided by 6.2. The simplest way to express this mathematically is to say that the square wave is the *derivative signal* of the diagonal signal if we divide all the y-axis values of the diagonal signal by 6.2. The diagonal signal, after every y-axis value has been divided by 6.2, is *one* of the anti-derivatives of the square wave. To put this slightly more mathematically:

$$fsk\ square\ signal" = \frac{1}{6.2}\int psk\ diagonal\ signal\ \ dt$$



The number 6.2

It is important to note that the value 6.2 appears in the integral formula because we are using instantaneous amplitudes from 0 to 6.2 units in our diagonal modulating signal to alter the phase of the carrier wave. We want the carrier wave's phases to vary between 0 and 6.2 radians. This makes good use of the available phases in a full circle, which rise from 0 to 2π radians (2π is 6.28318531). If we used instantaneous amplitudes from 0 to 2π then the formula would be:

$$fsk\ square\ signal = \frac{1}{2\pi}\int psk\ diagonal\ signal\ dt$$

If we used instantaneous amplitudes from 0 to 3 units, the formula would be:

$$fsk\ square\ signal = \frac{1}{3}\int psk\ diagonal\ signal\ dt$$

If we were working in *degrees*, we would want to make the best use of the available range of degrees in a circle. Therefore, we would use amplitudes from 0 to just under 360 degrees – perhaps 359 degrees – and our formula would be:

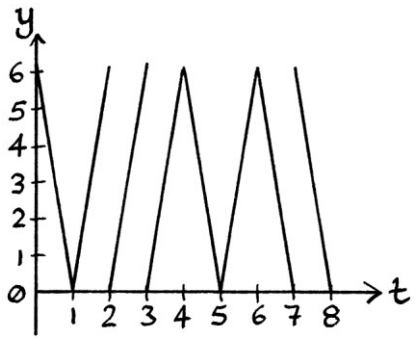
$$fsk\ square\ signal = \frac{1}{359}\int psk\ diagonal\ signal\ dt$$

[Of course, whenever calculus is used around waves, it is better to use radians and not degrees, but in this situation, we are dealing with straight lines, so it does not matter.]

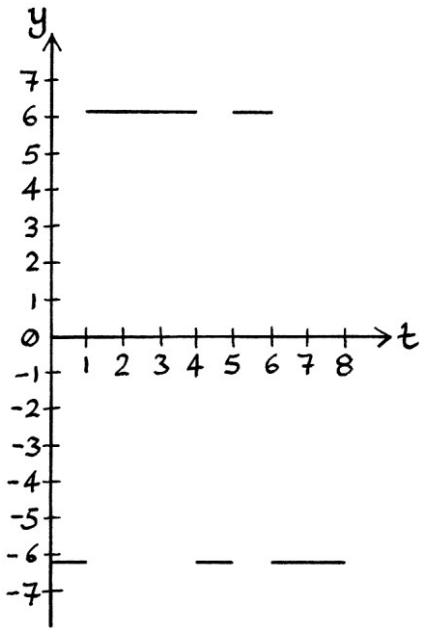
I am explaining this to demonstrate that there is nothing magical or particularly significant about the appearance of 6.2 in the formulas. It is only there because we want to be able to access the full range of phases, and we are working in radians.

Gradients in the pictures

Remember that the gradient of a line is the change in the y-axis divided by the change in the x-axis (or in this case, the t-axis). It is the rise divided by the tread. To keep the graphs in this explanation clearer, I have drawn the graphs with the axes to different scales. Therefore, the literal gradient *as measured on one of the pictures* is not the actual gradient as calculated by dividing the change in the y-axis by the change in the t-axis. If we draw the graphs with the axes to the same scale, we would have a drawing of the correct gradients, but the graphs become harder to read. Here is the diagonal wave with the axes drawn to the same scale as each other:



Here is the derivative square wave signal drawn with the axes to the same scale: [It has not been divided by 6.2]



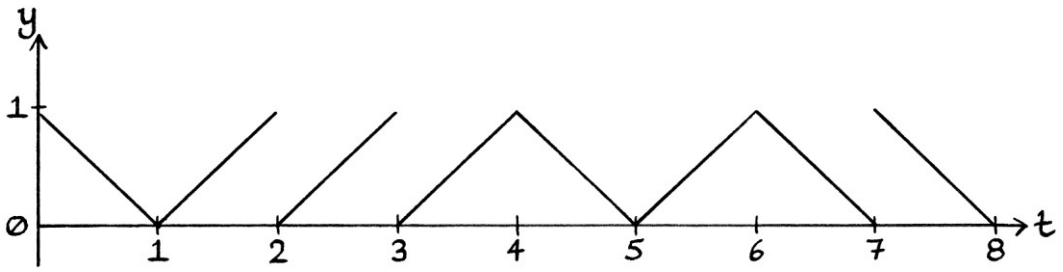
Changing the y-axis units

If we had chosen a different maximum value for the PSK diagonal signal to move up to, then the corresponding FSK square wave would have been the derivative of that signal after that signal had been divided by the maximum value.

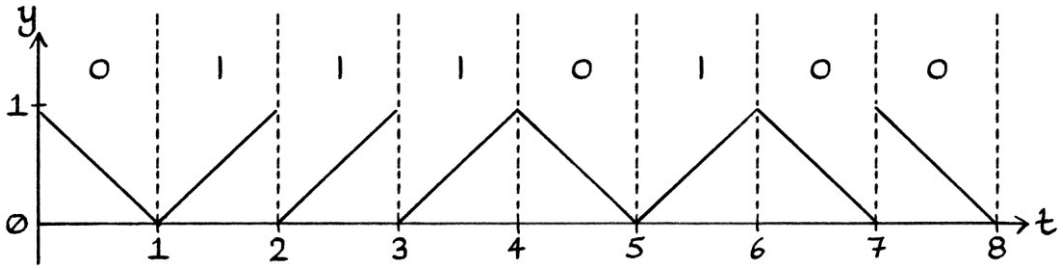
When we first started using phase shift keying with addition, or phase modulation with addition, we had two options:

- We could have the y-axis values of the modulating signal vary between 0 and just under 2π, and use them directly to set the instantaneous phases of the carrier wave.
- We could have the y-axis values of the modulating signal vary between 0 and just under 1, and then multiply them by 2π before using them to set the instantaneous phases of the carrier wave.

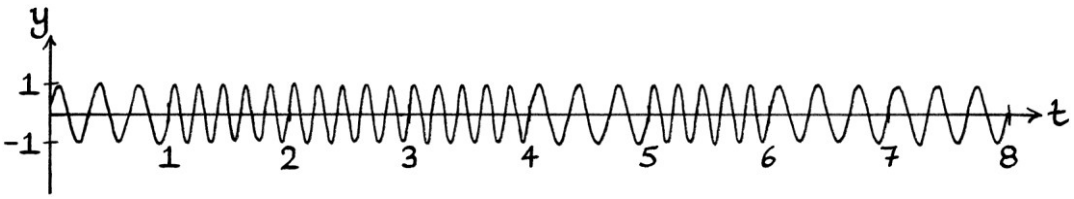
So far, we have been using the first of these ideas. However, when it comes to derivatives and integrals, it makes sense to use the second of these ideas. As a demonstration, we will use a modulating signal that varies between 0 and just under 1 to encode the binary digits “01110100”:



With the digits marked, the signal looks like this:



We take each y-axis from the signal, multiply it by 2π, and set the instantaneous phase of the carrier wave to the result. We will use the same carrier wave as before with a frequency of 4 cycles per second. The result, as expected, is the same as when we used the previous method:

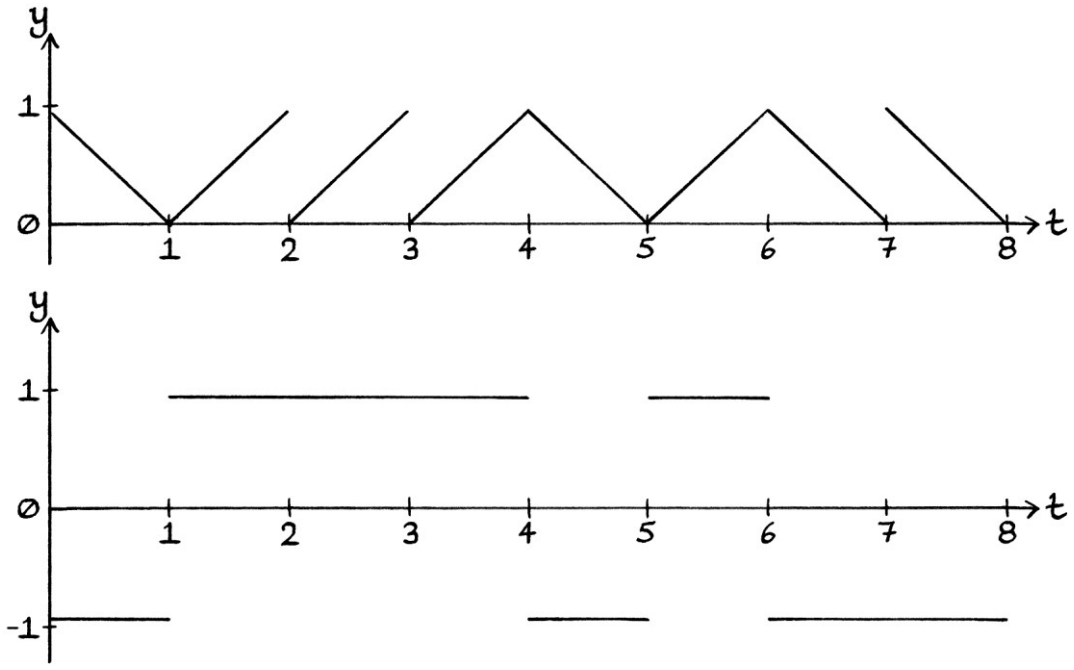


[There might be the tiniest difference caused by how far our diagonal signal is below 1, and by how far our previous diagonal signal was below 2π. If these were both, say, 99% of the maximum, then the graphs would be identical. If one was 99% of the maximum and the other was 98% of the maximum, then there would be a slight difference, but not enough to be visible in a drawing.]

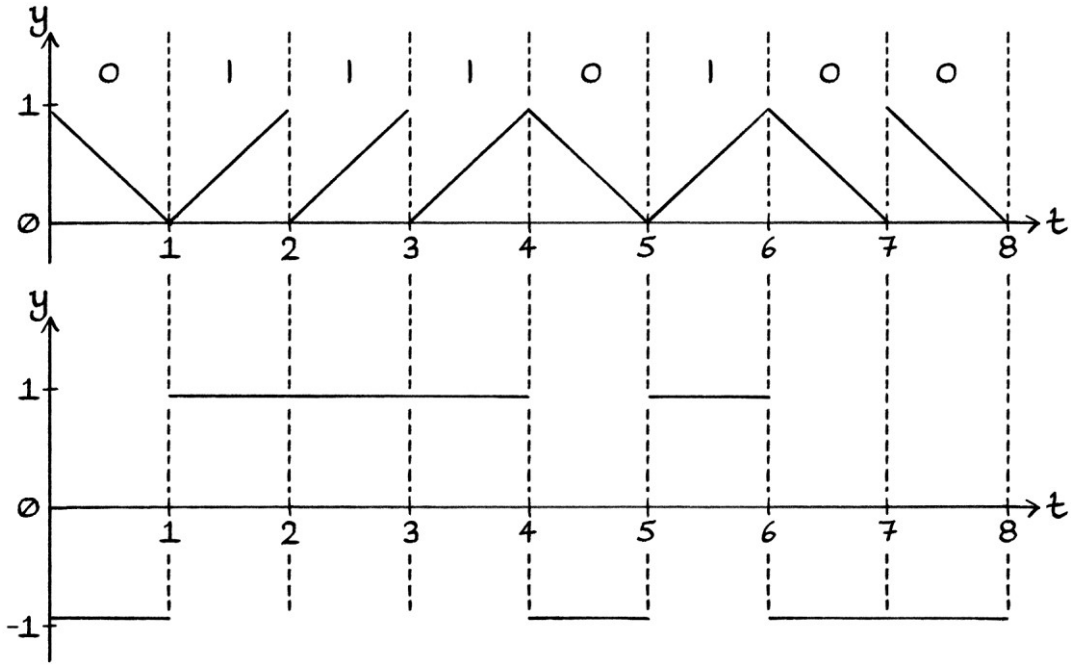
The difference now is that the frequency shift keying square wave will be exactly the derivative signal of our diagonal wave – we do not need to divide by any amount before we calculate the derivative. When the diagonal signal is falling from just under one to zero, it has a gradient of just under 1, and therefore, the square

wave will be a horizontal straight line at just under y = 1. When the diagonal signal is rising from zero to just under one, it has a gradient of just above −1, and therefore, the square wave will be a horizontal straight line at just above y = −1. The square wave is the derivative of the diagonal signal, and the diagonal signal is *one* of the anti-derivatives of the square wave.

The diagonal signal and the square signal shown together, with the y-axis and t-axis drawn to the same scale as each other, are as so:



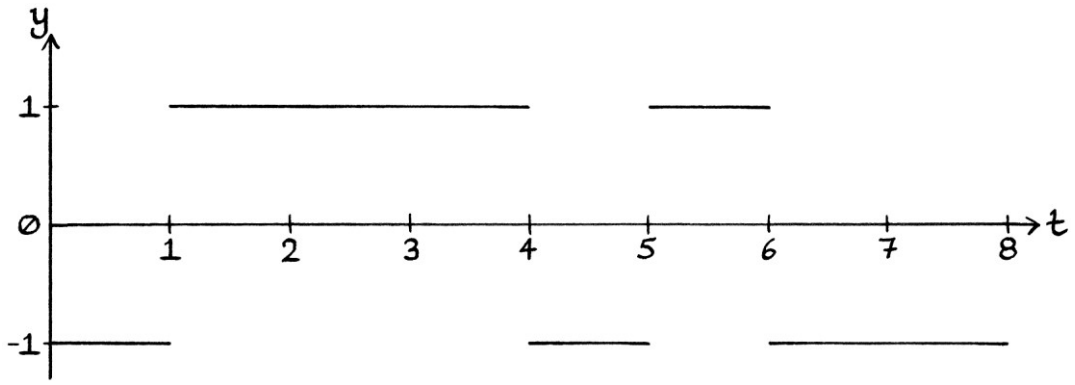
With more details, the graphs are so:



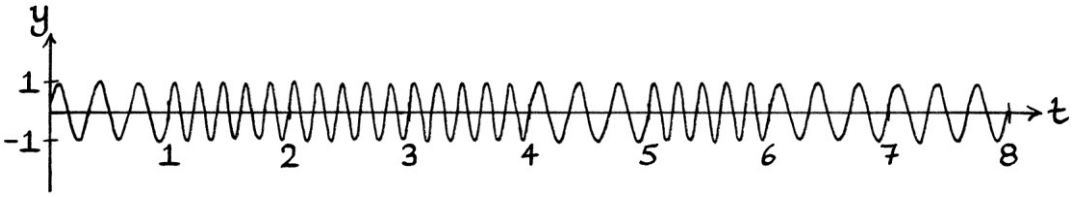
If the diagonal signal had moved between 0 and exactly 1, the corresponding square wave would switch between exactly +1 and −1. The phases in the modulated carrier wave would have varied between 0 and 2π, and the frequencies in the resulting signal would have varied between exactly 3 cycles per second and 5 cycles per second. In the above example, where the diagonal signal moves between 0 and just under +1, the frequencies in the resulting signal vary between just over 3 cycles per second and just under 5 cycles per second, but the actual difference is negligible.

FSK and integration

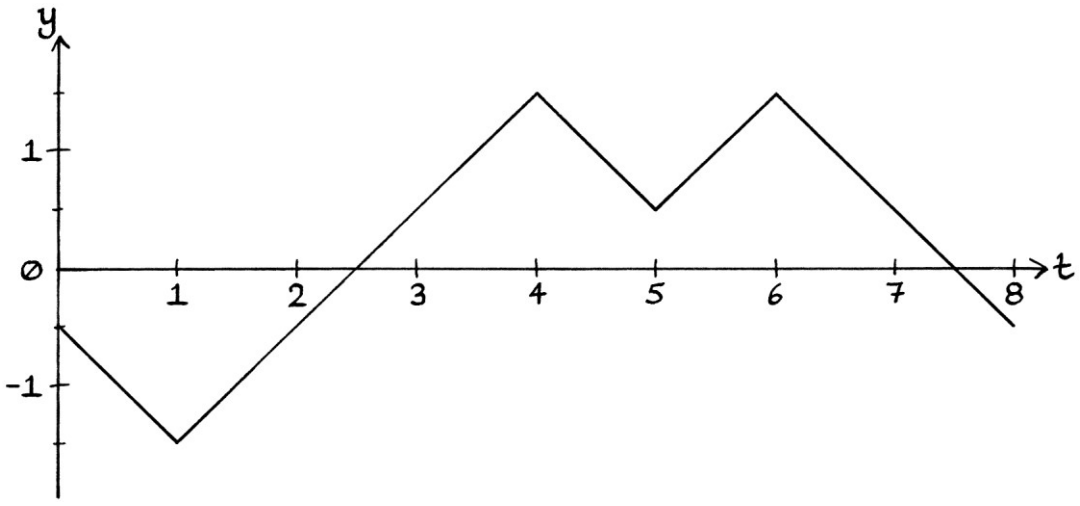
We can try to work backwards from the square wave to the diagonal signal. Doing this will reveal more about the process. To keep everything simpler, we will have a square wave that switches between exactly +1 and −1:



When the square wave is used to perform FSK, it produces this modulated signal:

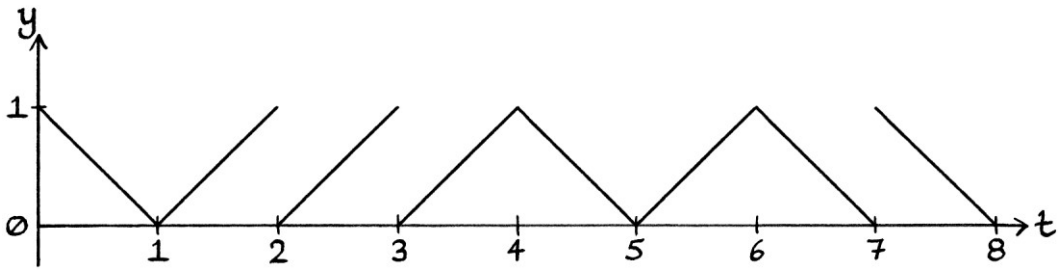


One of the anti-derivatives of the square wave is the following signal, which is centred around y = 0. We will call this signal “Diagonal Signal B”.

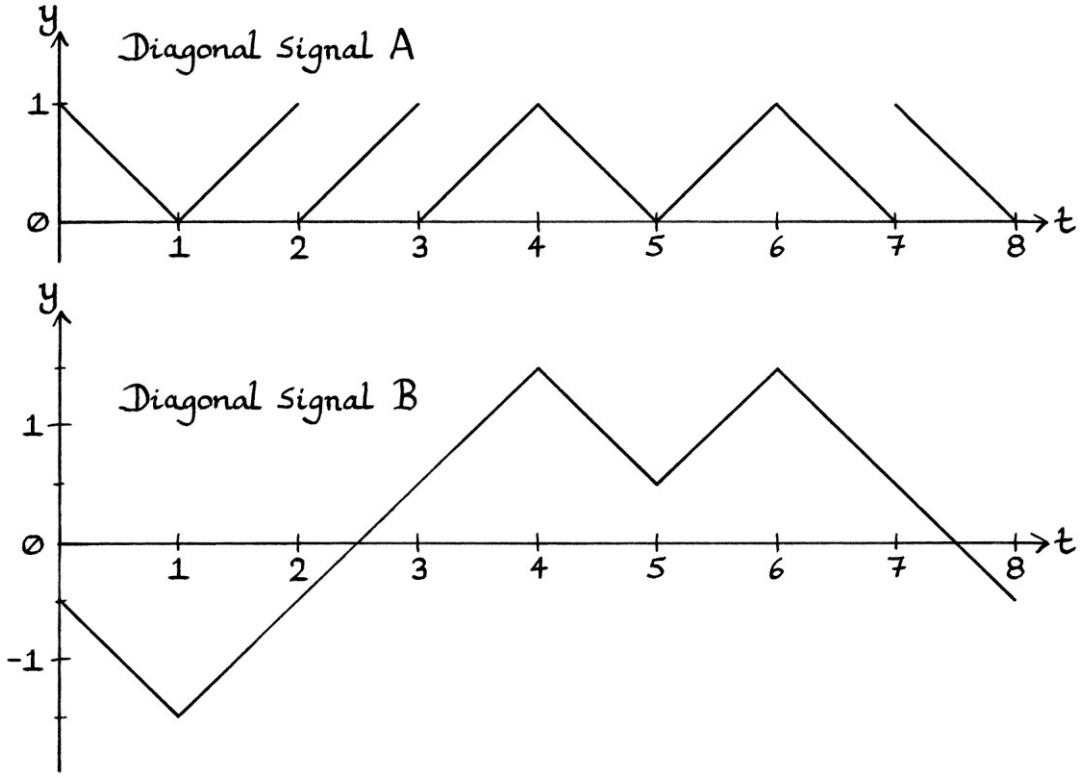


[How to calculate anti-derivative signals was explained in Chapter 30.]

This signal does not look like the diagonal signal that we might be expecting, which would look like this:



[This differs from the diagonal signal in the previous examples because this moves from 0 to 1 instead of from 0 to just under 1.] We will call the above signal "Diagonal Signal A". It is the most straightforward signal for performing frequency shift keying using phase modulation. The values in our anti-derivative signal ("Diagonal Signal B") do not move between 0 units and 1 unit. The two signals together for comparison are as so:



We will think about the new diagonal signal ("Diagonal Signal B"). At t = 0, the y-axis is at −0.5 units. This will result in an instantaneous phase in the carrier wave of −0.5 * 2π = −π radians, which is the same as +π radians (180 degrees). Over the first second, the y-axis values fall to −1.5 units, which means that the instantaneous phases will fall to −1.5 * 2π = −3π radians. The angle of −3π radians is the same as −π radians, which is the same as +π radians (180 degrees). Putting this in terms of

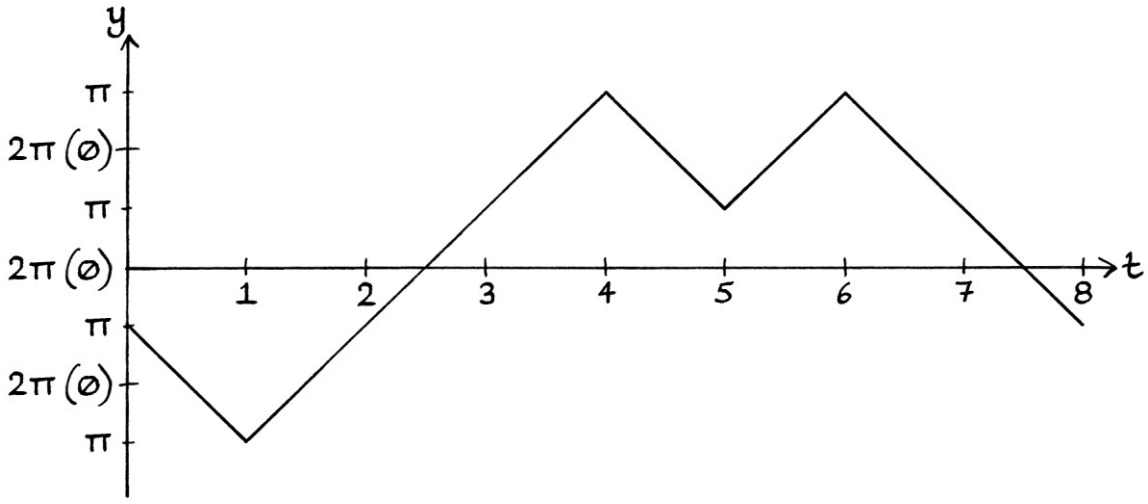
degrees to make it clearer, this means that the instantaneous phases fall downwards from 180 degrees to zero degrees (which is also 360 degrees), and then continue falling downwards from 360 degrees down to 180 degrees. The graph does not make it clear that the angles are rolling around.

From 1 second to 4 seconds, the y-axis values increase from −1.5 units up to +1.5 units. This is an increase of 3 units in 3 seconds, or one unit per second. The increase in instantaneous phases is as so:

- At t = 1, the phase will be −3π radians, which is π radians (180 degrees).
- At t = 1.5, the phase will be −1 * 2π radians = −2π radians = 0 radians.
- At t= 2.5, the phase will be 2π radians or 0 radians.
- At t = 3.5, the phase will be +1 * 2π radians = +2π radians = 0 radians.
- At t = 4, the phase will be +1.5 * 2π radians = 3π radians = π radians.

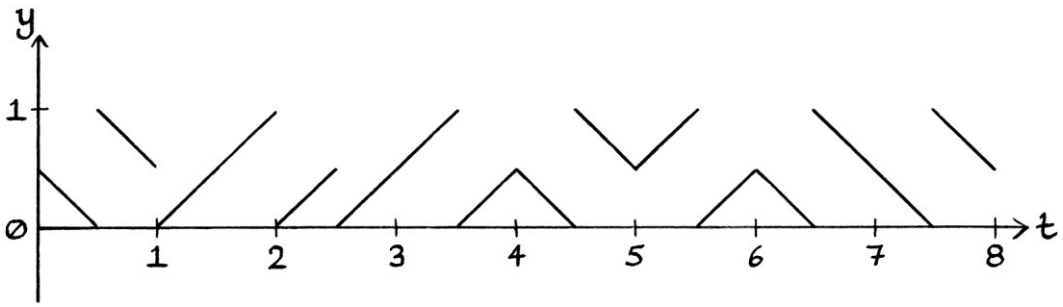
Therefore, the instantaneous phases roll over 2π radians (360 degrees) three times during this time.

We can change the numbering along the y-axis of “Diagonal Signal B” to indicate to what phase that y-axis value will set the carrier wave (in radians):



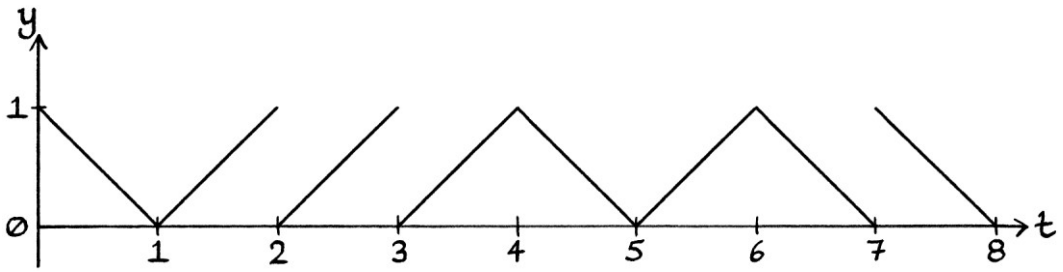
Perhaps more usefully, we can redraw “Diagonal Signal B” so that the y-axis values all fit between 0 and 1 units. To do this, we slide sections of the diagonal lines upwards or downwards.

The graph looks like so:

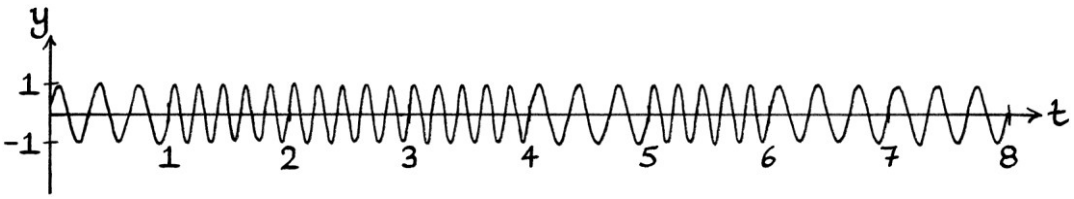


The phases that the shifted-around signal will produce are identical to those of "Diagonal signal B".

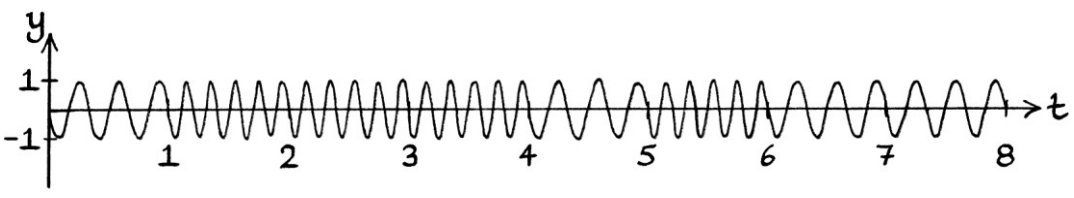
Our "Diagonal Signal A" looked like this:



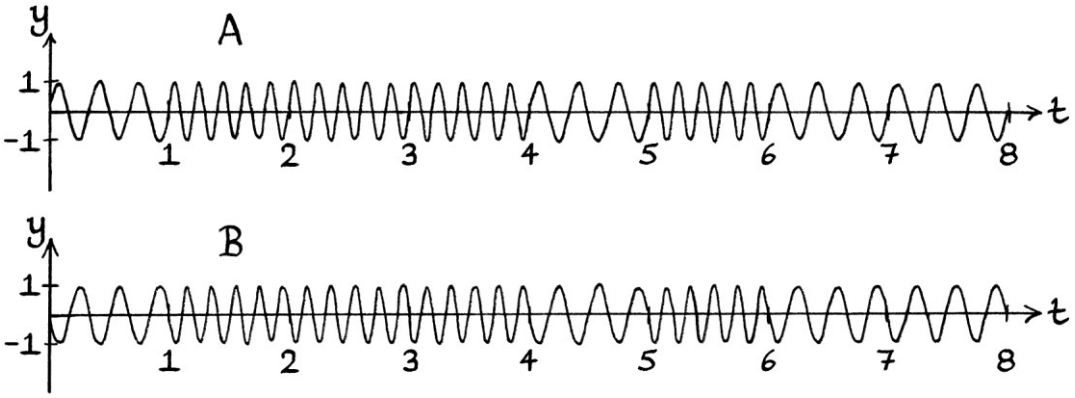
When we use it to phase modulate a 4 cycle-per-second carrier wave, we end up with the following signal, which is identical to that created by the square wave from the start of this section:



However we draw, or think of, "Diagonal Signal B", the actual y-axis values and the phases that they represent are different from those in "Diagonal Signal A". If we use "Diagonal Signal B" to modulate a 4 cycle-per-second carrier wave, we will end up with this signal:

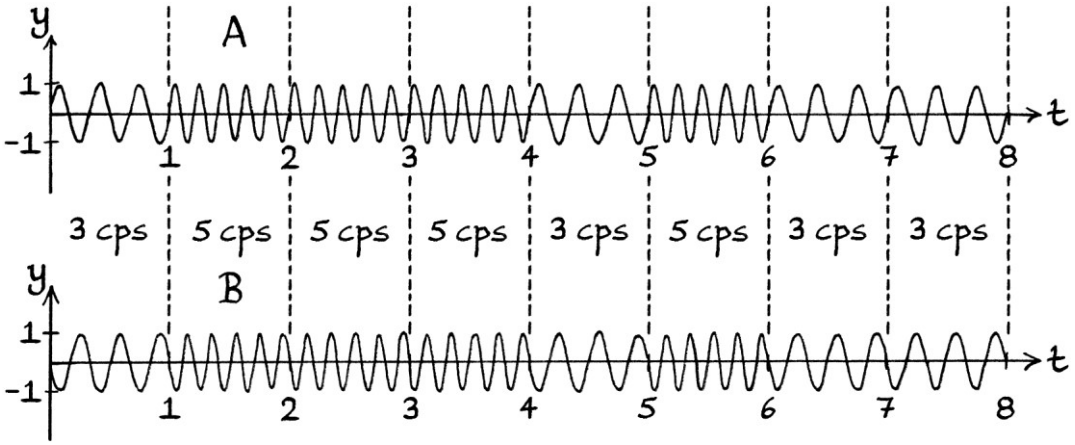


This is actually a different signal. We will view it next to the modulated signal from "Diagonal Signal A" to see the differences:



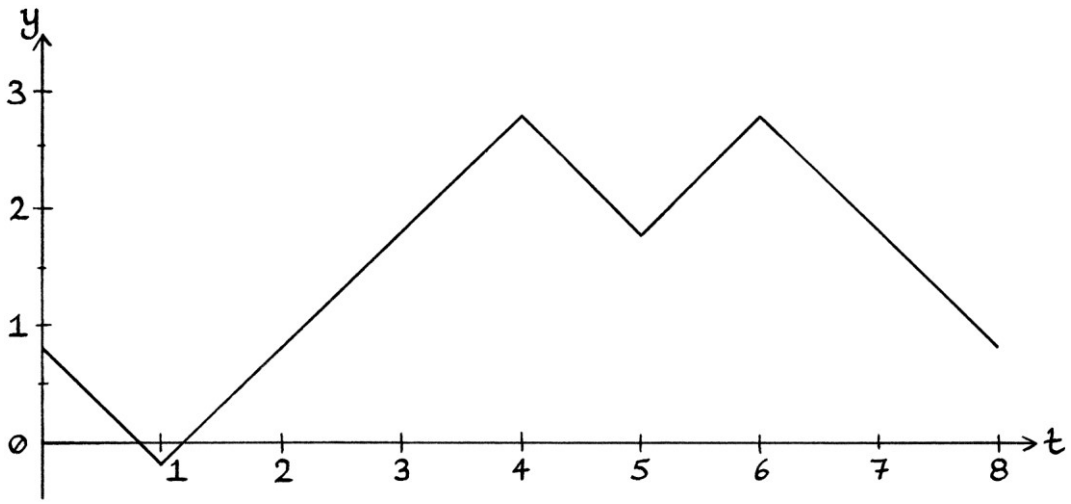
The most obvious difference is that the start of the signal created from "Diagonal Signal B" is inverted. The initial phase of "Diagonal Signal A" is zero radians. The initial phase of "Diagonal Signal B" is π radians (180 degrees).

Although the modulated signal from "Diagonal Signal B" is different, it still contains the same *frequencies* at the same times as the modulated signal from "Diagonal Signal A":

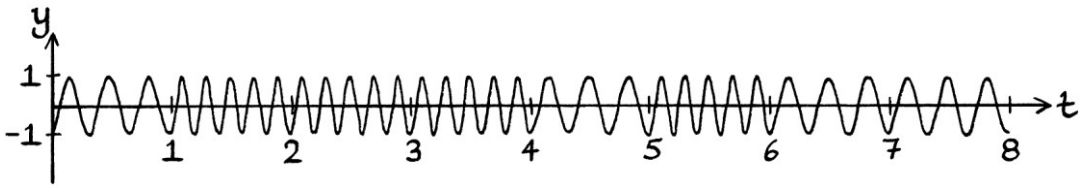


The instantaneous *phases* are different, but the instantaneous *frequencies* are the same. This is because the effect on frequency of phase modulation is not from the *actual* phases but from the rate at which those phases are changing. The y-axis values of "Diagonal Signal A" and "Diagonal Signal B" are different, but the rates of change of the lines are identical. Therefore, the frequencies in both modulated signals are identical.

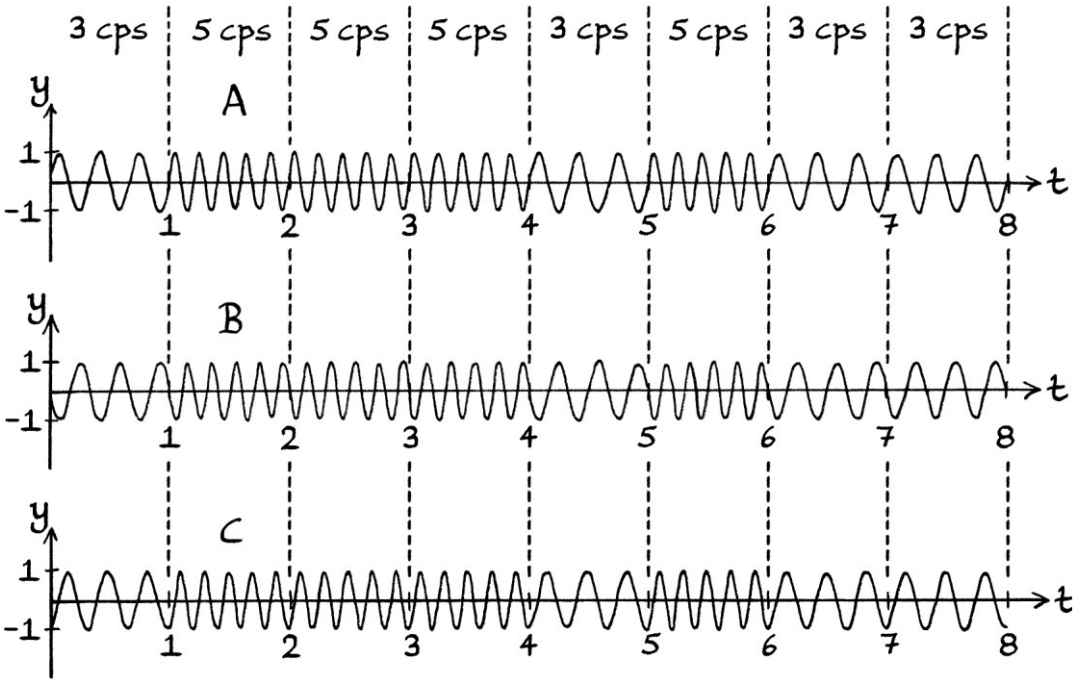
To illustrate again that it is the rates of change of the y-axis units (and therefore, the instantaneous phases), and not the actual values that affect the phases, we will shift our “Diagonal Signal B” up the y-axis by the arbitrary amount of 1.3 units:



We will call this signal, “Diagonal Signal C”. This signal starts at y = 0.8. This value would produce an initial instantaneous phase of 0.8 * 2π = 1.6π radians (288 degrees). We will use “Diagonal Signal C” to phase modulate a 4 cycle per second carrier wave. We end up with this signal:



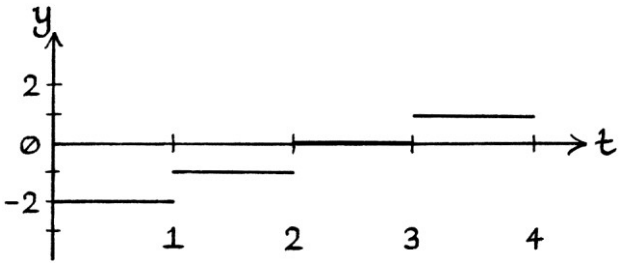
The modulated signals created from “Diagonal Signal A”, “Diagonal Signal B”, and “Diagonal Signal C” all have different phases, but the instantaneous frequencies are all identical:



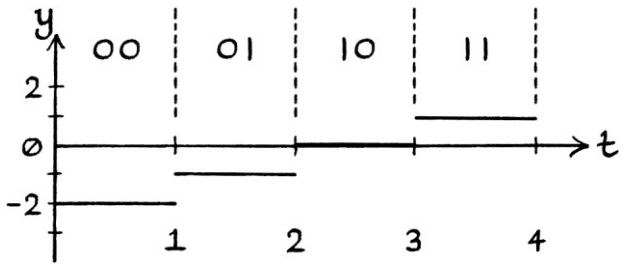
If we are using the diagonal signals to perform frequency shift keying, we are only interested in the frequencies in the modulated signals that they produce. In that case, the phases are irrelevant, and each resulting signal is equally good at conveying the message.

More complicated square waves

As a more complicated example, we will start with a 4-FSK example. We will use a 4-state square wave to encode the binary digits “00011011”. Each state will encode a pair of bits. The square wave switches from −2 to +1. The graph looks like this, with the t-axis numbering underneath to keep the graph clearer:

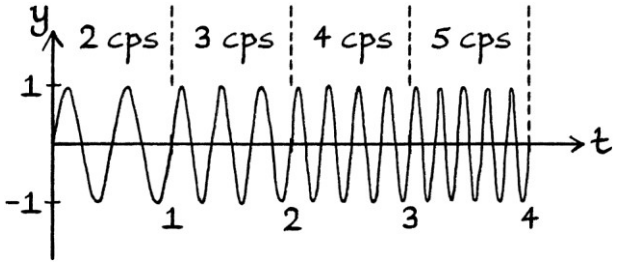


The same graph with the digits marked is as so:



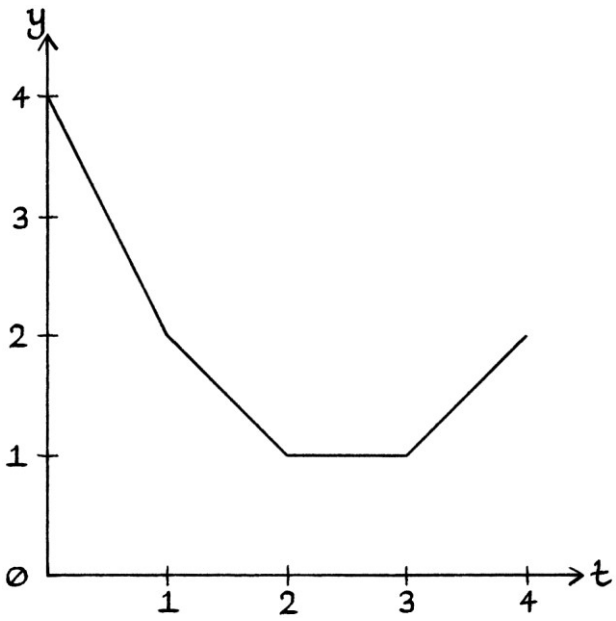
We will use a carrier wave with a frequency of 4 cycles per second. After being encoded with frequency shift keying, our bits will be portrayed by the following frequencies:
The bits “00” are encoded with a frequency of 4 – 2 = 2 cycles per second.
The bits “01” are encoded with a frequency of 4 – 1 = 3 cycles per second.
The bits “10” are encoded with a frequency of 4 + 0 = 4 cycles per second.
The bits “11” are encoded with a frequency of 4 + 1 = 5 cycles per second.

The frequency-shift-keyed signal looks like this:

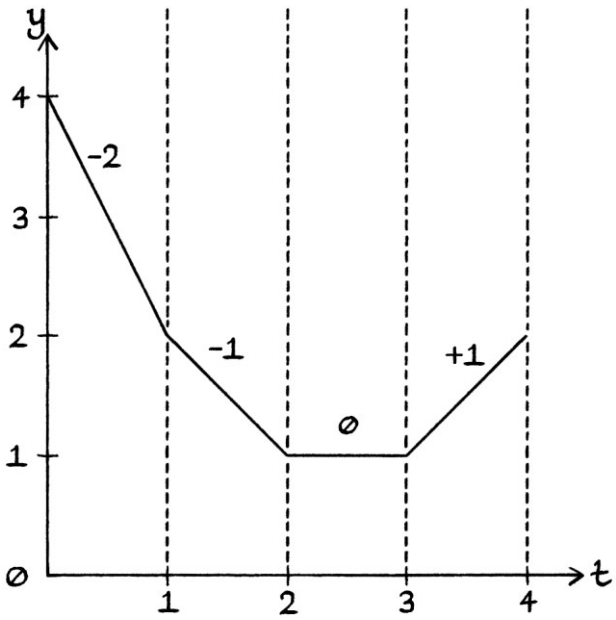


Now we will use phase modulation to achieve the same “frequency result” [in the sense that the frequencies will be the same, but the phases might not be the same, depending on which anti-derivative signal we use.]

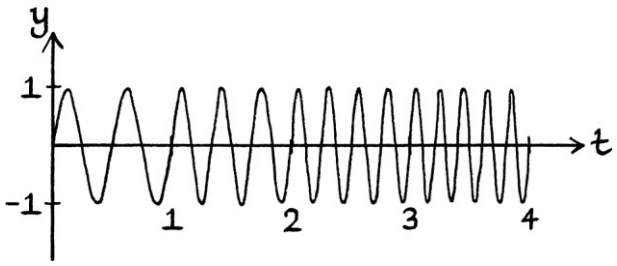
One of the possible anti-derivative signals of our square wave is as follows, with the axes drawn to the same scale as each other to show the gradients of the lines more clearly:



The same drawing with the gradients marked is as so:

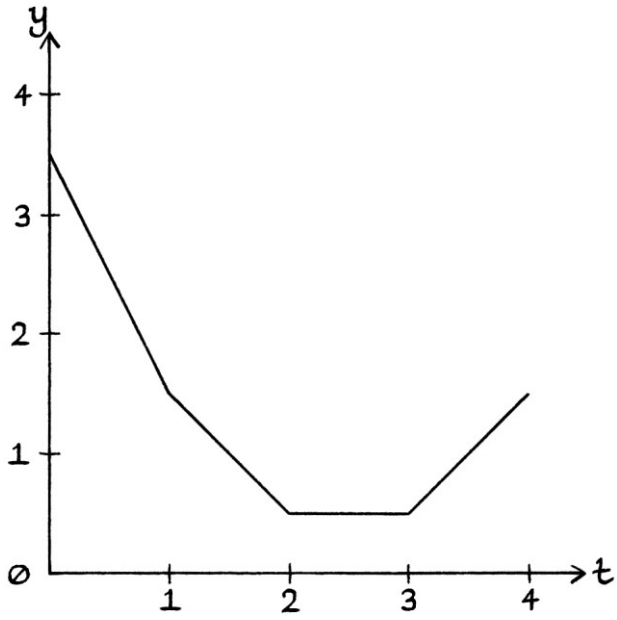


If we perform phase modulation using the above signal (with addition and multiplying each y-axis value by 2π) and a carrier wave with a frequency of 4 cycles per second, we will end up with this modulated signal:

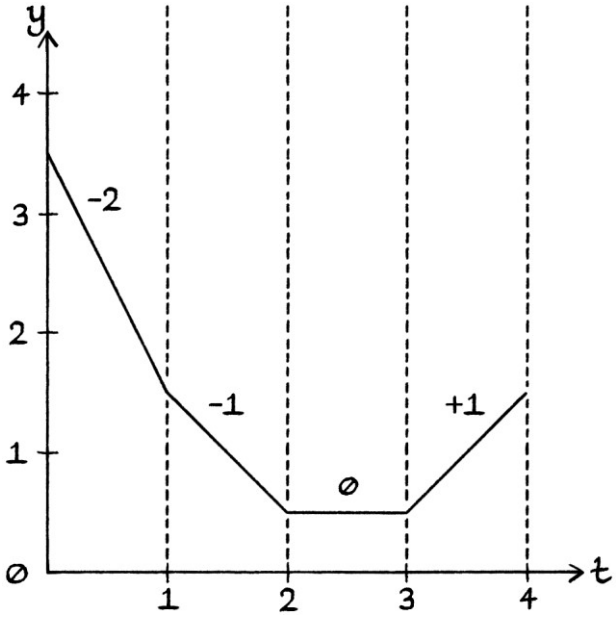


This is exactly the same signal as the modulated signal from the 4-FSK square signal.

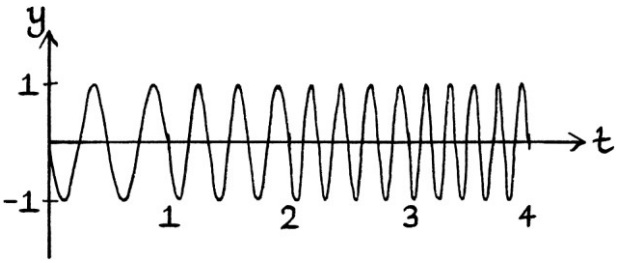
We will now subtract the arbitrary amount of 0.5 units from every y-axis value of our anti-derivative signal to produce the following signal, which is also a valid anti-derivative of our square wave:



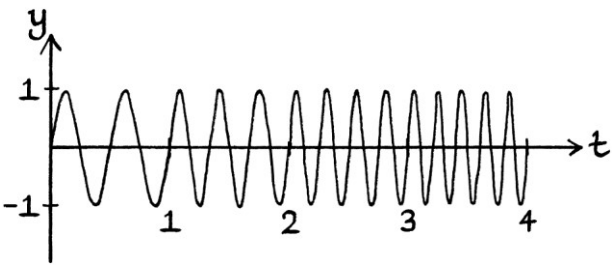
With the gradients marked, it looks like this:



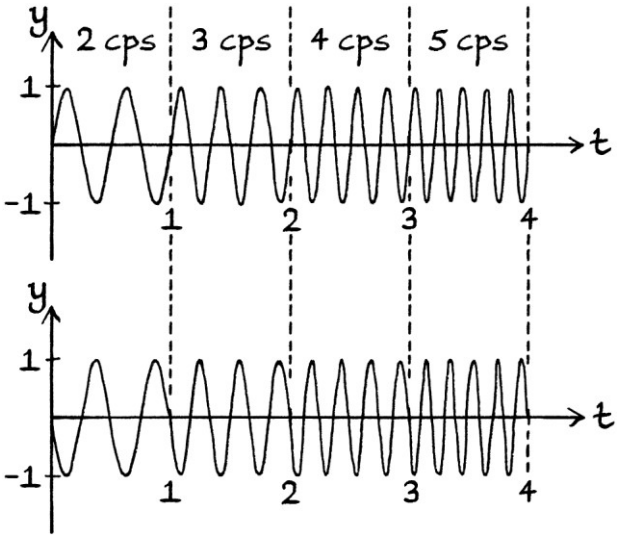
The gradients of this new anti-derivative signal are still the ones portrayed in our square wave, as they should be, or else it would not be an anti-derivative signal. When we use our new raised up anti-derivative signal to perform phase modulation, we will end up with this modulated signal:



This as inverted version of our previous modulated signal, which looked like this:



The instantaneous phases of the entire new signal are all π radians (180 degrees) different from those of the old signal. This makes the new signal appear upside down. However, the frequencies for each moment in time in both signals are identical:



Given that, this signal still encodes our original pairs of binary digits using frequency shift keying achieved by phase modulation. If we cared about the resulting phases, then our lowered modulating signal would not be of any use, but for frequency shift keying, we only care about the frequency, and the phase is irrelevant. Therefore, any anti-derivative of our 4-FSK square wave could have been used.

[In this particular example, if the anti-derivative signal starts at an integer on the y-axis (such as y = 0, y = 1, y = 2 and so on), the instantaneous phases of the result will match those of the modulated signal from the original square wave. If the anti-derivative signal starts elsewhere, the instantaneous phases will be shifted by between over 0 and under 2π radians.]

Summary

From all of the above, we can see that we can perform frequency shift keying using phase modulation by:

- Taking a square wave intended to be used with frequency shift keying.
- Finding any one of its anti-derivatives.
- Using phase modulation on one of the anti-derivative signals. [We need to use phase modulation with addition, where we multiply the y-axis values of the signal by 2π.]

The instantaneous *phases* of the modulated signal will depend on which anti-derivative we choose, but the instantaneous *frequencies* will be identical to if we had created the signal using frequency shift keying using the square wave and frequency modulation.

This means that we can now perform frequency shift keying with phase modulation. What is more, the method works with non-digital frequency modulation too. In other words, we can perform frequency modulation with any signal by using phase modulation on one of the anti-derivatives of that signal. We will explore this more in Chapter 38.

Phase modulation with multiplication

For interest's sake, in this section, we will look at phase modulation using multiplication. When we used addition, the values of the modulating signal had to fit between 0 and 2π units, or between 0 and 1 unit depending on how we were performing it. When we are using multiplication, the values of the modulating signal need to have a maximum related to the phase of the carrier wave that we are using.

The basic formula for phase modulation with multiplication is:
"$y = \sin((2\pi * ft) + (\phi * X))$"
... where "X" is the instantaneous amplitude of the modulating signal at any one moment in time.

In these, examples, we will give the carrier wave a phase of 0.5π radians (90 degrees). This means that the y-axis values of the modulating signal need to vary from 0 units up to just under 4 units.

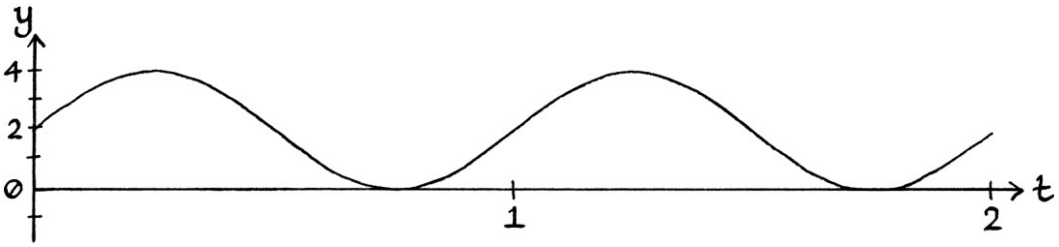
- When the y-axis value is 0 units, we will have a phase of 0.5 * 0 = 0 radians.
- When the y-axis value is 1, we will have a phase of 0.5π * 1 = 0.5π radians.
- When the y-axis value is 2, we will have a phase of 0.5π * 2 = π radians.
- When the y-axis value is 3, we will have a phase of 0.5π * 3 = 1.5π radians.
- If the y-axis rose to 4 units, we would have a phase of 0.5π * 4 = 2π radians, which is the same as 0 radians.

When the y-axis values are any other value between 0 and 4 units, the resulting phase will still be that y-axis value multiplied by 0.5π radians. In this way, we can have a continuous run of available phases. [If we used a carrier wave with a non-zero phase other than 0.5π radians, we would need to have the range of

amplitudes in the modulating signal vary between different values. If we used a carrier wave with a phase of zero radians, we would not be able to use multiplication at all – every instantaneous amplitude multiplied by zero would result in an instantaneous phase of 0 radians.]

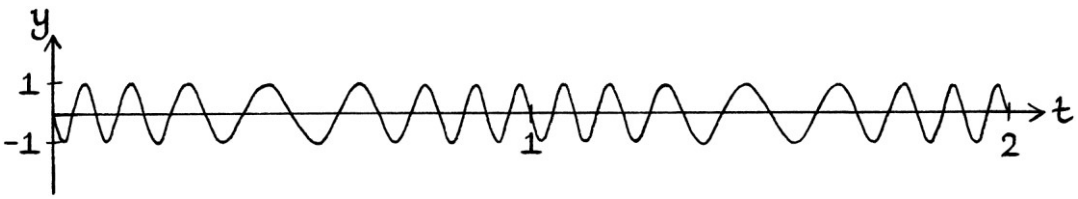
Example 1

As an example of using multiplication, we will start with the following modulating wave, which fluctuates between 0 units and 3.99 units. [For future reference, note that the value of 3.99 is 0.9975 * 4]. The wave's formula is:
"y = 1.995 + 1.995 sin (2π * 1t)"



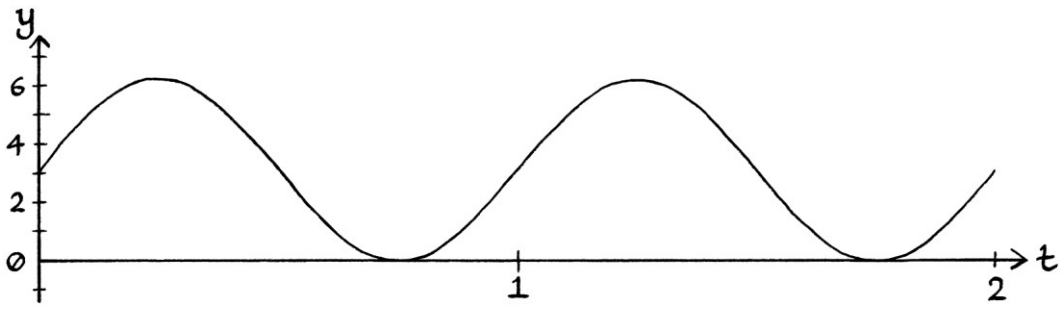
We will use a carrier wave with the formula:
"y = sin ((2π * 8t) + 0.5π)"

The resulting signal looks like this:



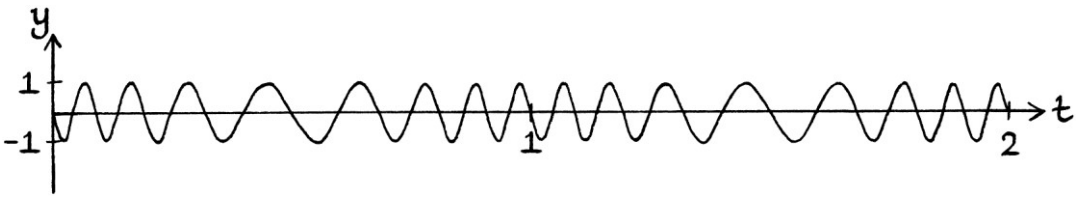
We will now use phase modulation with *addition* (with values from 0 to just under 2π) to encode a y-axis-scaled version of the same modulating signal. This scaled signal has the formula:
"y = 3.1337 + 3.1337 sin (2π * 1t)"
This formula means that the wave fluctuates from 0 to 6.2674 units. [The value of 6.2674 is 0.9975 * 2π.]

The wave looks like this:



We will use a carrier wave with the formula:
"y = sin (2π * 8t)"

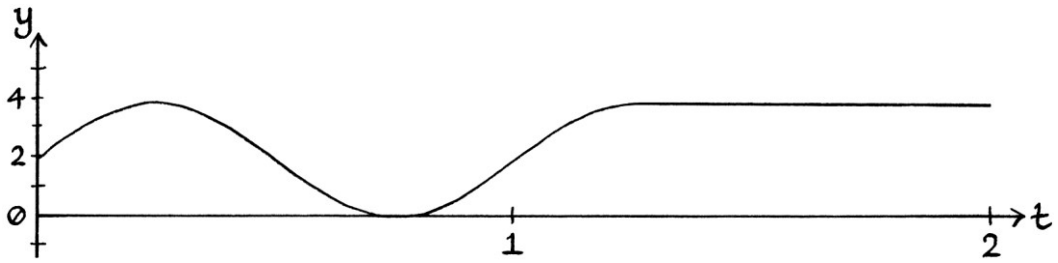
The resulting signal looks like this:



This is identical in every way to the signal we created with multiplication, which shows that multiplication works just as well as addition.

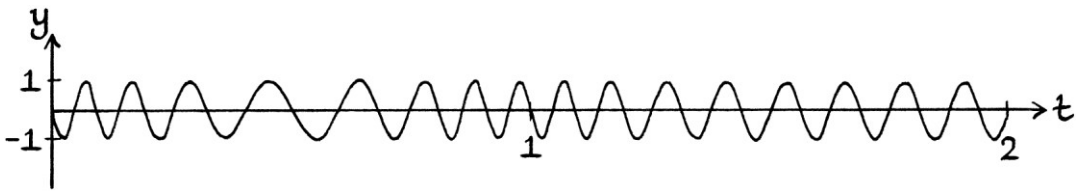
Example 2

As another example, we will use phase modulation with multiplication to encode the following signal:

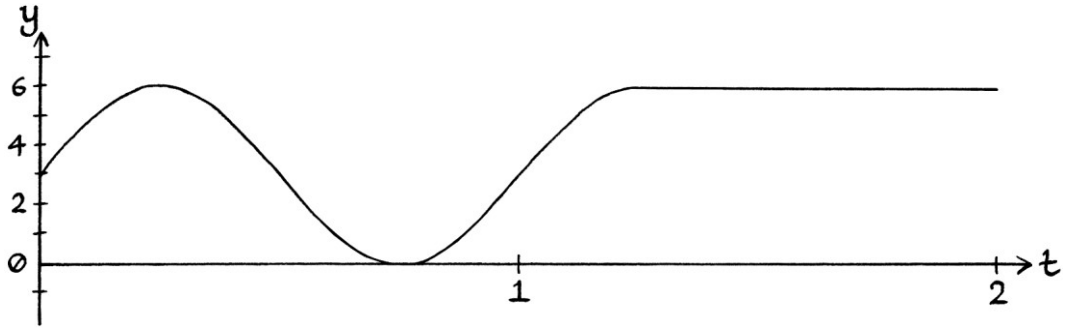


The signal is a Sine wave that ends up as a horizontal line at 3.8197 units. We will use a carrier wave with a frequency of 8 cycles per second and a phase of 0.5π radians (90 degrees).

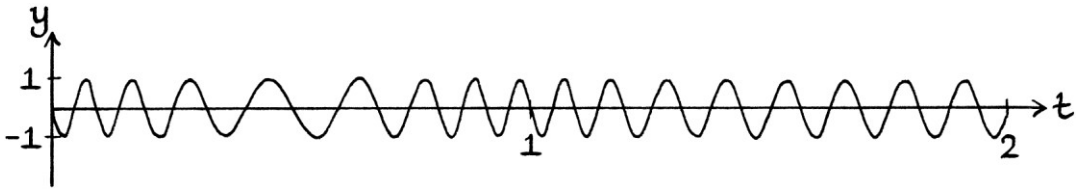
The resulting signal looks like this:



We will now do the same process using phase modulation with *addition* with values from 0 to just under 2π units. Our modulating signal is identical to the multiplication one, except that it has been scaled to fit between 0 and 2π units instead of 0 and 4 units. The signal is a Sine wave that ends up as a horizontal line at 6 units.



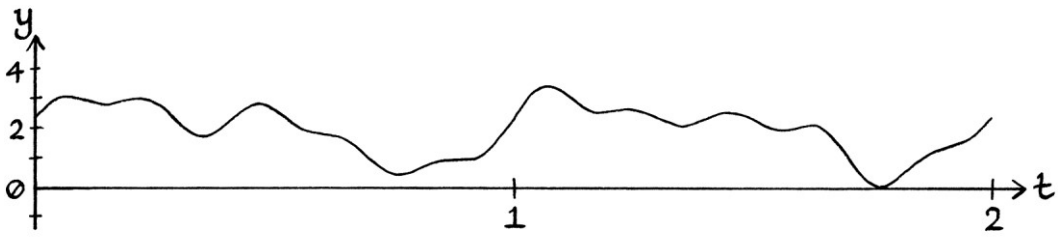
We will use a carrier wave with a frequency of 8 cycles per second and a phase of 0 radians. The resulting signal looks like this:



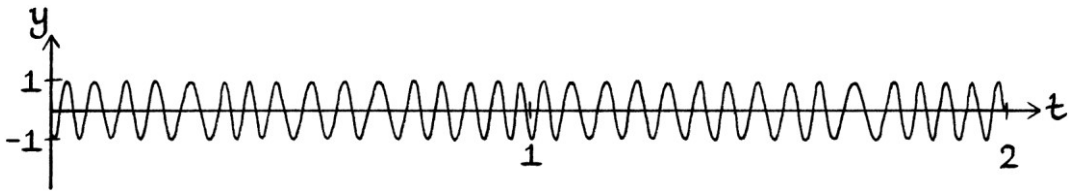
This is exactly the same result as when we used addition.

Example 3

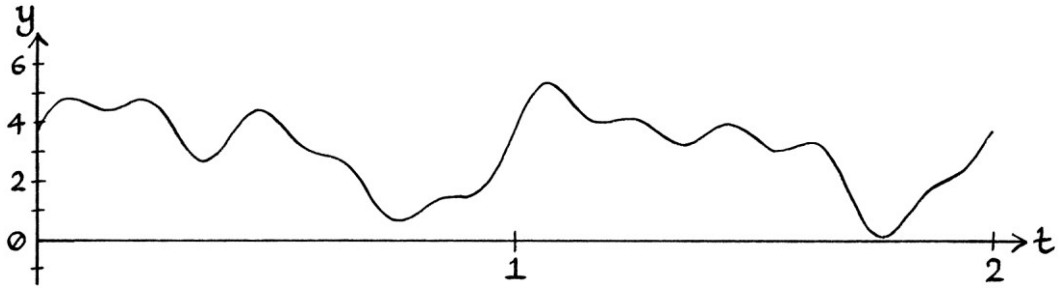
As a more complicated example, we will use this signal:



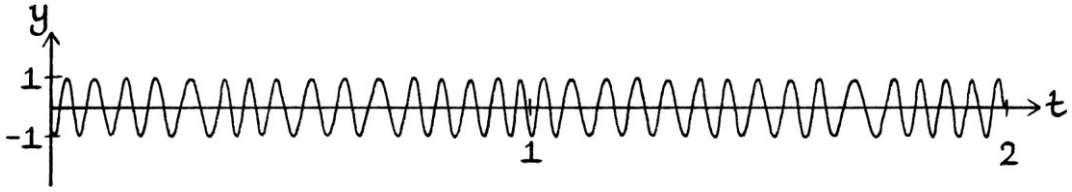
This signal moves between y = 0.09 units and y = 3.44 units. When using phase modulation with multiplication with a carrier wave with a frequency of 16 cycles per second and a phase of 0.5π radians, we end up with this signal:



The following graph shows the same signal that has been scaled to fit between 0 and 2π units. The signal's y-axis units vary between y = 0.14 units and 5.42 units.

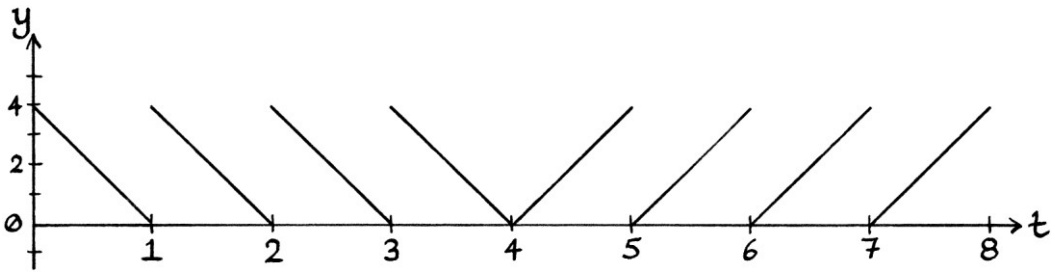


When we use phase modulation with *addition*, we end up with the same modulated signal as before:

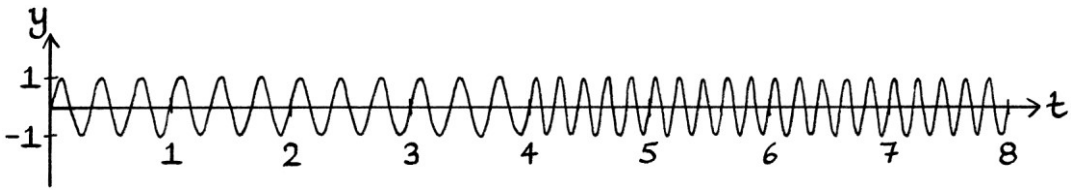


Example 4

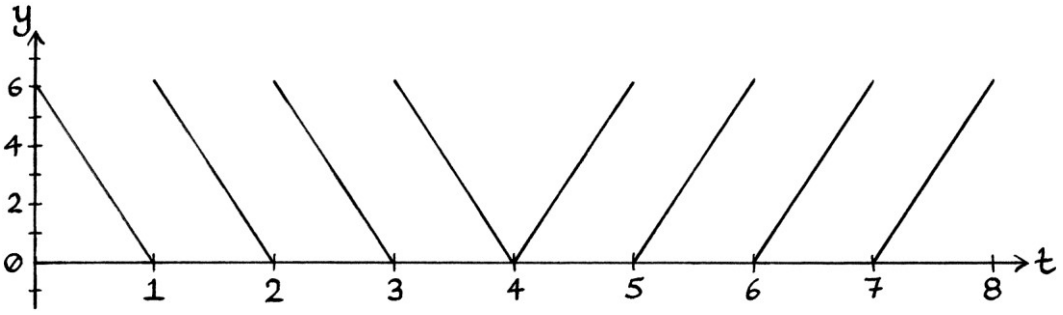
As another example, we will use this signal that moves from y = 0 to y = 3.98:



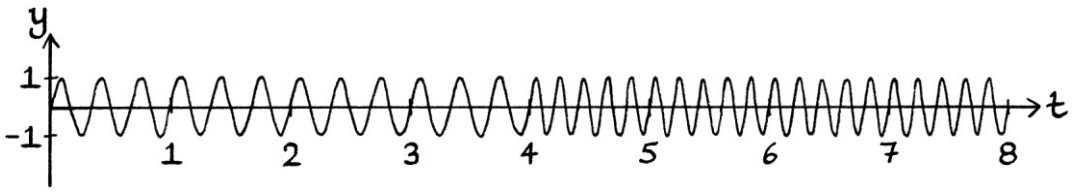
We will perform phase modulation with multiplication on a carrier wave with a frequency of 4 cycles per second and a phase of 0.5π radians. The result looks like this:



Now, we will perform phase modulation with *addition* using the following signal that varies between y = 0 and y = 6.26 units. [Remember that 2π is equal to 6.2832.]



We will use a carrier wave with a frequency of 4 cycles per second and a phase of 0 radians. The result is identical to when we used addition. It looks like this:



Thoughts

All of the above examples show that multiplication works just as well as addition when using phase modulation. Which is best depends on what it is we want to do and how we want to do it.

Chapter 38: Frequency modulation

Frequency modulation alters a carrier wave's frequency according to the instantaneous amplitude of a given signal. This is exactly what we were doing with frequency shift keying when we used square waves to control the frequency of a carrier wave. Frequency shift keying is just a form of frequency modulation with square modulating waves.

Whereas the frequency in a FSK signal jumps abruptly from one state to another, the frequency in non-digital frequency modulation usually changes more gradually. This is mainly because frequency modulation is more likely to be used with audio signals, and audio signals seldom have instant jumps of amplitude. It is much harder to discern the pattern of frequencies in a frequency-modulated signal than in an amplitude-modulated signal. To see the frequency changes, it is usually necessary to view the signal in a frequency domain graph.

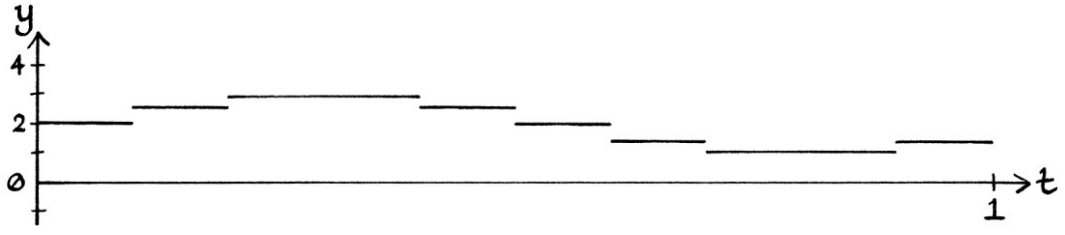
Without having seen how frequency shift keying works, the concept of frequency modulation would be much harder to visualise.

Non-digital frequency modulation is not a progression from frequency shift keying. The addition and multiplication methods of achieving frequency shift keying by altering the frequency begin to fail when the number of state changes gets too high.

Attempting FM

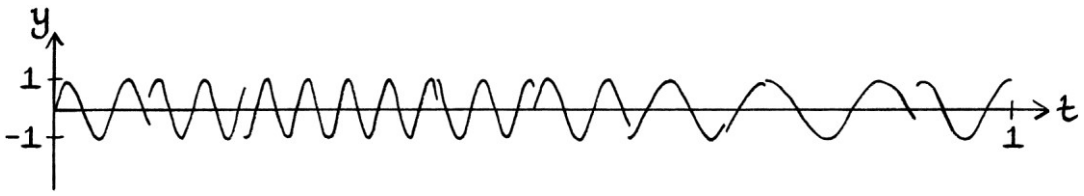
Multiplication

We will look at how non-digital frequency modulation cannot be achieved using multiplication of the frequency of the carrier wave. We will start with the following signal, which is essentially a multi-level square wave version of:
"y = 2 + sin (2π * 1t)":

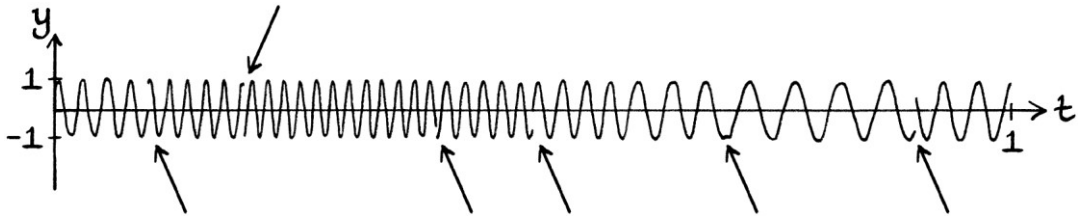


If “X” is the instantaneous amplitude of this square wave at any moment in time, and we are encoding the square wave using FSK with multiplication, we would use this formula:
“y = A sin (2π * X * ft)”

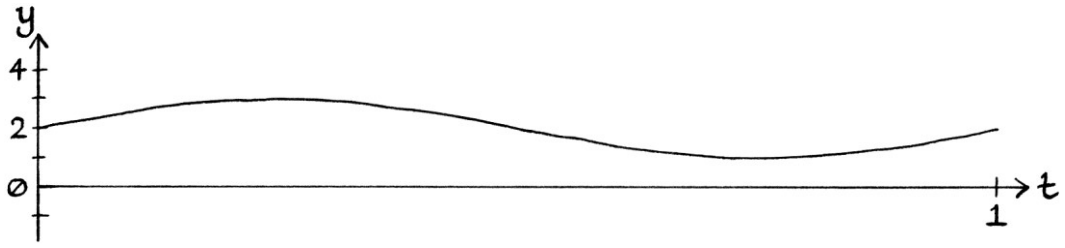
We will use a carrier wave with the formula “y = sin (2π * 8t)”. The resulting signal looks like this:



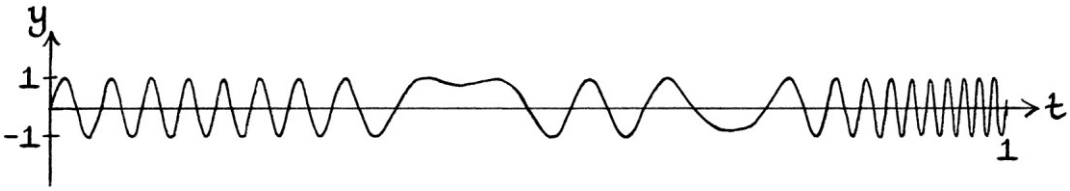
In the resulting signal, there are vertical jumps in the curve because the changes in the state of the carrier wave do not align with the cycles of the carrier wave. If we increase the frequency of the carrier wave to 20 cycles per second, we still get breaks in the curve:



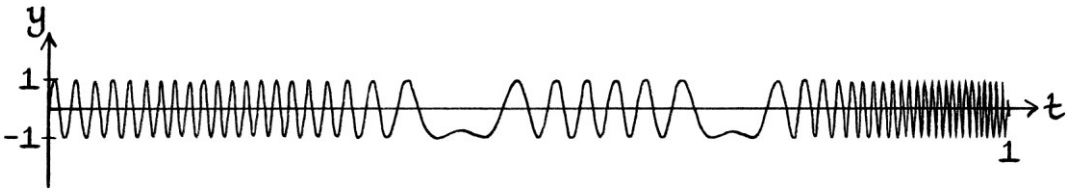
Even if we use a carrier wave with a frequency of 100 cycles per second, there will still be breaks in the curve. A similar problem occurs when we use a proper Sine wave to modulate the carrier wave, such as the following, which has the formula:
“y = 2 + sin (2π * 1t)”



If the carrier wave has a frequency of 8 cycles per second, the resulting modulated signal is this:



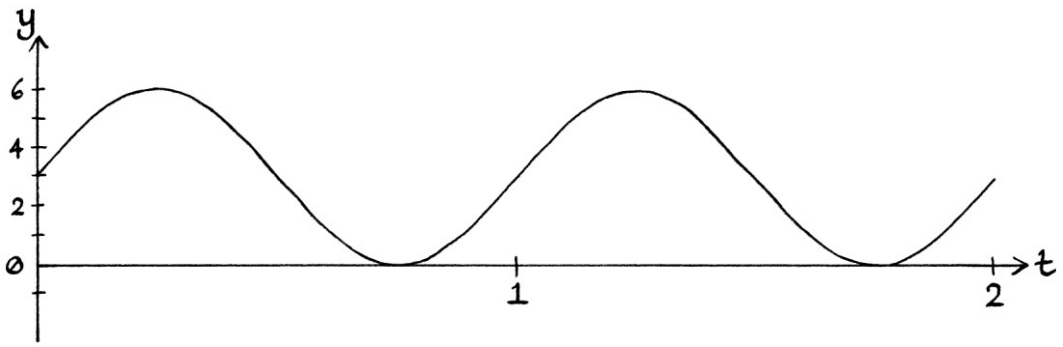
If the carrier wave has a frequency of 20 cycles per second, the result is this:



In the above signal, the instantaneous frequencies are not consistent with the instantaneous amplitudes of the modulating signal. This is because the process ends up creating fractions of cycles, which at times combine to produce apparent real cycles, but at other times produce a distorted signal. The problem is most obvious in the middle of the signal where the curve does not reach up to its maximum level. It does not matter how fast the carrier wave is, the resulting signal will always be corrupted.

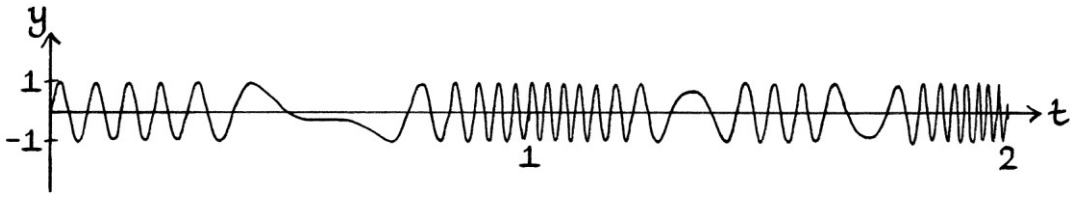
Addition

A similar problem occurs when we use addition in an attempt to achieve frequency modulation. As an example, we will use this Sine wave, which has the formula: "y = 3 + 3 sin (2π * 1t)":



We will use a carrier wave with the formula "y = sin (2π * 8t)". If "X" is the instantaneous amplitude of the modulating Sine wave at any moment in time, then the formula of the result will be:
"y = sin (2π * (8 + X) * t)"

The result looks like this:



This signal does not represent the instantaneous amplitudes of the modulating wave in its instantaneous frequencies. We have the problem that sometimes the curve does not reach to the maximum or minimum. The most obvious problem is that the first second of the result is not the same as the second second. The *modulating* wave repeats once every second, but the resulting signal does not repeat every second. In fact, the resulting signal is becoming faster and faster, which is not what we want.

The problems are related to how we are constantly altering the frequency of the carrier wave. As we know, if "X" is the instantaneous amplitude of the modulating Sine wave at any moment in time, then the frequency part of the resulting formula at that time will be:
"f + X"

If "X" were the same for all time, there would be a fixed frequency, and so the behaviour of the resulting signal would repeat after a set amount of time. For example, if "X" were always 1, then because "f" is "8", the resulting signal would always have a frequency of 9 cycles per second. It would repeat after 1 ÷ 9 = 0.1111 seconds. If "X" were always −1, then the resulting signal would always have a frequency of 7 cycles per second. It would repeat after 1 ÷ 7 = 0.1429 seconds. The problem in the above frequency modulation example is that because "X" is always changing, there is no set time in which it repeats. Despite the *sum* of "f + X" itself repeating every second, the cycles of the resulting signal *do not* repeat every second. By adding to the frequency with continuously changing values, we are continuously changing the frequency, and therefore, we are constantly changing where the cycles might repeat. This means that we do not achieve frequency modulation, but instead, we end up with a messy signal. This is also one of the reasons that frequency modulation with *multiplication* does not work, but it is harder to tell this when doing multiplication.

Frequency modulation in practice

If we want to perform frequency modulation correctly with multiplication or addition, we have to do it by using phase modulation. The basis of this idea was explained in the previous chapter, when we used phase modulation to perform frequency shift keying.

If we want to perform *frequency shift keying* with phase modulation, we take a square wave that represents binary digits, then we perform phase modulation (using addition and a multiplication by 2π) on one of its anti-derivatives. Depending on which anti-derivative we choose, the instantaneous *phases* of the result might be different, but the instantaneous *frequencies* that contain the message will still be the same as if we had used frequency shift keying with the original square wave. As the message is contained within the frequencies, the potential differences in phase do not matter.

We can use the same method to encode a non-digital signal. We take the signal on which we want to perform frequency modulation, and then we perform phase modulation (using addition and a multiplication by 2π) on one of its anti-derivatives. Different anti-derivatives will produce signals with different instantaneous *phases*, but all the signals will contain the same instantaneous *frequencies*. The message is contained in the instantaneous frequencies, so the instantaneous phases are irrelevant.

A reminder of basic calculus

We looked at integration of time-based waves and the sums of time-based waves in Chapter 30. To make the examples in this chapter clearer, we will briefly look at basic calculus with waves again.

Differentiation

The basic rule for *differentiation* of a time-based wave is that if we have this wave in radians:

$$y = h + A\ sin\ ((x * t) +\ \phi)$$

... where "x" is a fixed value scaling the time, then its derivative will be:

$$y = (x * A)\ sin\ ((x * t) +\ \phi + 0.5\pi)$$

In other words, to obtain the derivative:

- We remove the mean level. [Mean levels are irrelevant to gradients.]
- We multiply the amplitude by the value scaling the time. This value will usually be 2π multiplied by the frequency.
- We add quarter of a circle (0.5π radians) to the phase of the wave.

A more specific but less clear version of the rule is that if we have this wave:

$$y = h + A\ sin\ ((2\pi * f * t) +\ \phi)$$

... then its derivative will be:

$$y = (2\pi * f * A)\ sin\ ((2\pi * f * t) +\ \phi + 0.5\pi)$$

A consequence of this rule is that if the frequency of a wave is greater than $\frac{1}{2\pi}$ then the amplitude of the derivative wave will be higher than that of the original wave.

If we have a sum of waves, then the derivative of the whole sum will be the derivative of each individual wave added together. For example, the derivative of:
"Wave A" + "Wave B" + "Wave C"
... is:
"derivative of Wave A" + "derivative of Wave B" + "derivative of Wave C".

Integration

The basic rule for integration for time-based waves is that if we have this wave in radians:

$$y = A\ sin\ ((x * t) + \phi)$$

... where "x" is a fixed value scaling the time, then its integral is:

$$C\ +\ \frac{A}{x}\ sin\ ((x * t) + \phi - 0.5\pi)$$

In other words, to obtain the integral:

- We add on the constant "C" to indicate that there are countless answers all with different mean levels.
- We divide the amplitude by the value scaling the time. This value will usually be 2π multiplied by the frequency.
- We subtract 0.5π from the phase.

A more specific version of the rule is that if we have this wave:

$$y = A\ sin\ ((2\pi * f * t) + \phi)$$

... then the integral will be:

$$C\ +\ \frac{A}{2\pi * f}\ sin\ ((2\pi * f * t) + \phi - 0.5\pi)$$

A consequence of this rule is that if the frequency of a wave is greater than $\frac{1}{2\pi}$ then the amplitude of the integral wave will be *lower* than that of the original signal.

If we have a sum of waves, then the integral of the sum will be "C" added to the sum of the integral of each individual wave.

Anti-derivatives

An anti-derivative is just a specific form of an integral. Therefore, it will be the same as an integral, but with an actual mean level (instead of having the general value "C"). All the anti-derivatives of a formula have the same derivative.

Calculus without calculus

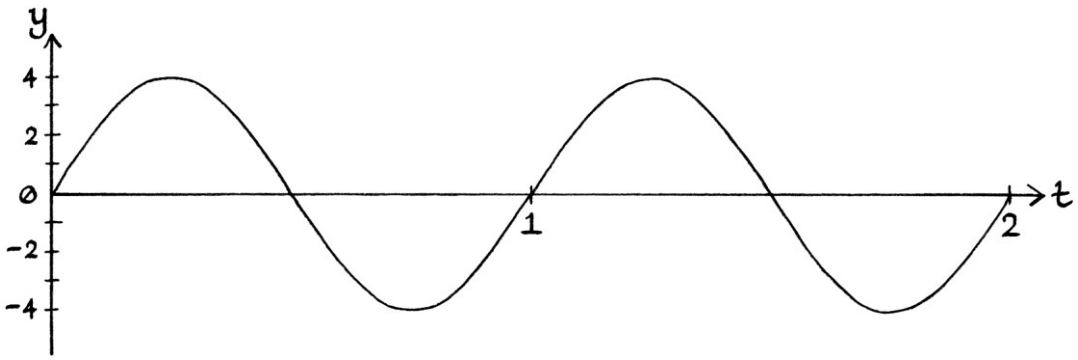
It is possible to perform differentiation and integration manually by calculating the change in y-axis values over tiny amounts of time. We can do this if we do not know the formula of a signal, or we have the signal as a discrete signal (where it is stored as a list of y-axis values at evenly spaced moments in time). How to do this was explained in Chapter 30.

Frequency modulation examples

We will now look at some simple examples of non-digital frequency modulation performed using phase modulation.

Example 1

As a simple example, we will say that we want to use frequency modulation to encode this wave:
"y = 4 sin (2π * 1t)"



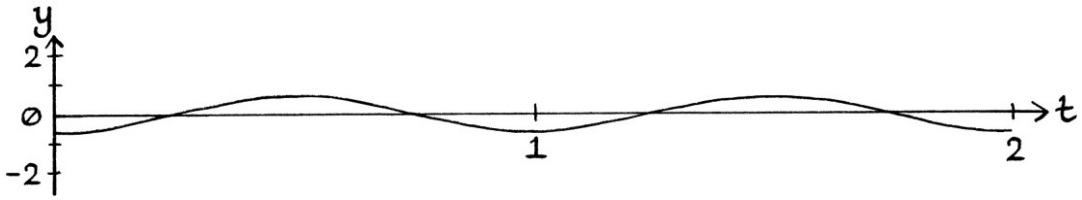
The first step is to find one of the anti-derivatives. For various reasons, it is generally easiest to calculate the anti-derivative that is centred around y = 0.

Our indefinite integral will be:
"y = C + (4 ÷ 2π) sin ((2π * 1t) – 0.5π)"
... which is:
"y = C + 0.6366 sin (2πt – 0.5π)"
... which, if we give it a positive phase, is:
"y = C + 0.6366 sin (2πt + 1.5π)"

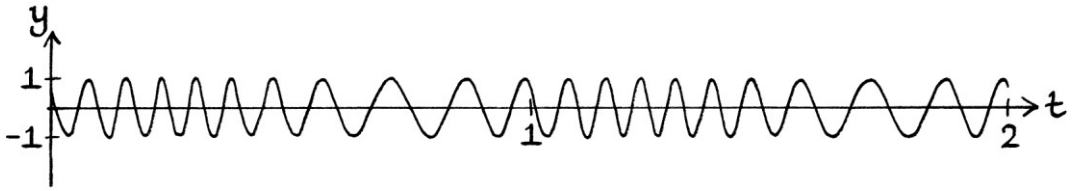
We will use our indefinite integral to choose an anti-derivative that has a mean level of zero. Therefore, we will pick this wave:
"y = 0 + 0.6366 sin (2πt + 1.5π)"
... which is:
"y = 0.6366 sin (2πt + 1.5π)"

We could have calculated the curve of this anti-derivative without using calculus, but by using calculus, we have the exact formula. [We could also have found the exact formula by using Fourier series analysis on an anti-derivative wave that had not been calculated with calculus.]

Our anti-derivative looks like this:

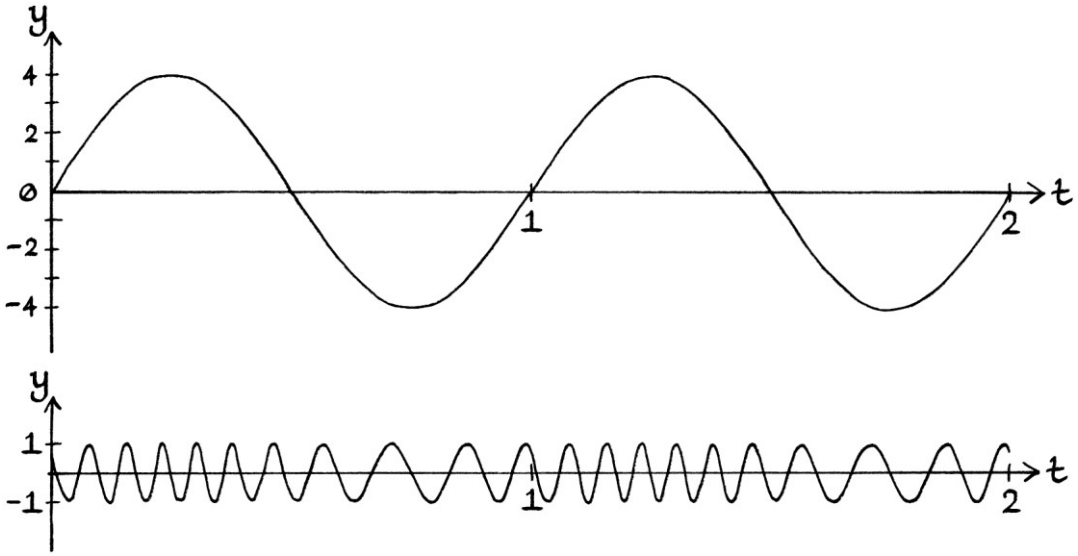


We can now perform phase modulation using addition (and a multiplication by 2π) with this wave. For every moment in time, we take the instantaneous amplitude of the wave, multiply it by 2π, and then set the phase of the carrier wave to the result. We will use a carrier wave with the formula "y = sin (2π * 10t)". The result of the phase modulation looks like this:



The above signal is a frequency-modulated version of the original wave:
"y = 4 sin (2π * 1t)"
... but we used phase modulation to achieve it by using an anti-derivative of the wave.

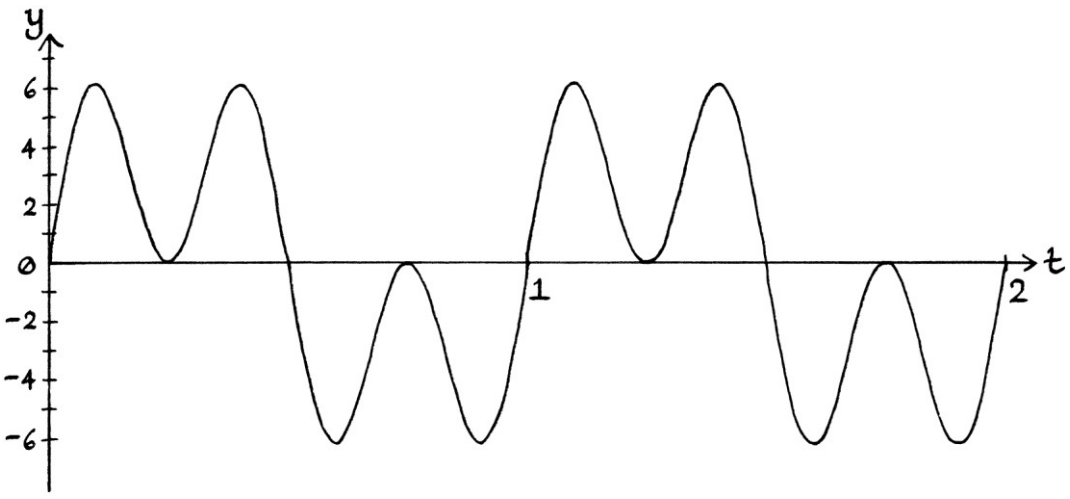
We will examine the resulting signal by comparing it with the original wave. The original wave and the resulting signal together look like this:



When the instantaneous amplitude of the Sine wave is high, the frequency of the modulated signal is faster. When the instantaneous amplitude of the Sine wave is low, the frequency of the modulated signal is slower. The resulting signal is a frequency-modulated version of the original wave. We had to use phase modulation to achieve frequency modulation, but the reasons why, and the method we used, should be reasonably straightforward if you have read this far.

Example 2

As a more complicated example, we will use the following signal, which has this formula: "y = 4 sin (2π * 1t) + 4 sin (2π * 3t)":

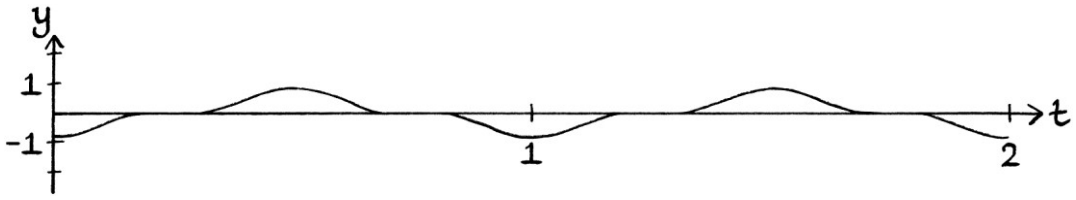


We need to find an anti-derivative of this signal, and to makes things simpler, we will find the one that has a zero mean level. To do this, we can either calculate the indefinite integral of this signal using calculus (in which case, we find the integral of each part of the sum), or we can go through the signal y-axis value by y-axis value and calculate an anti-derivative manually. For this example, we will use calculus. As we have the addition of two waves in the formula, we just find the integral of each wave in turn:
"y = C + (4 ÷ 2π) sin ((2π * 1t) – 0.5π) + (4 ÷ (2π * 3)) sin ((2π * 3t) – 0.5π)"
... which ends up as:
"y = C + 0.6366 sin (2πt – 0.5π) + 0.2122 sin ((2π * 3t) – 0.5π)"
... which, if we make the phases positive is:
"y = C + 0.6366 sin (2πt + 1.5π) + 0.2122 sin ((2π * 3t) + 1.5π)"

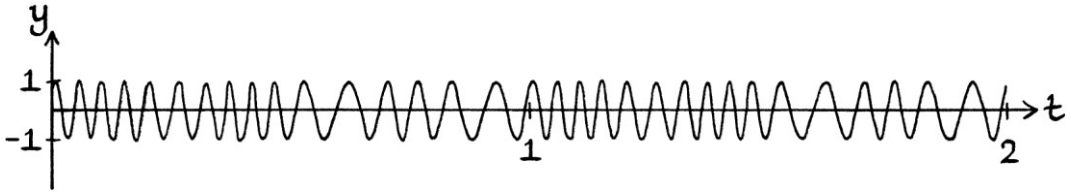
As we want the anti-derivative that has a zero mean level, we replace the "C" with zero, and we have:
"y = 0 + 0.6366 sin (2πt + 1.5π) + 0.2122 sin ((2π * 3t) + 1.5π)"
... which is:
"y = 0.6366 sin (2πt + 1.5π) + 0.2122 sin ((2π * 3t) + 1.5π)"

[We do not *have* to have an anti-derivative signal with a zero mean level, but for these examples, it makes things simpler.]

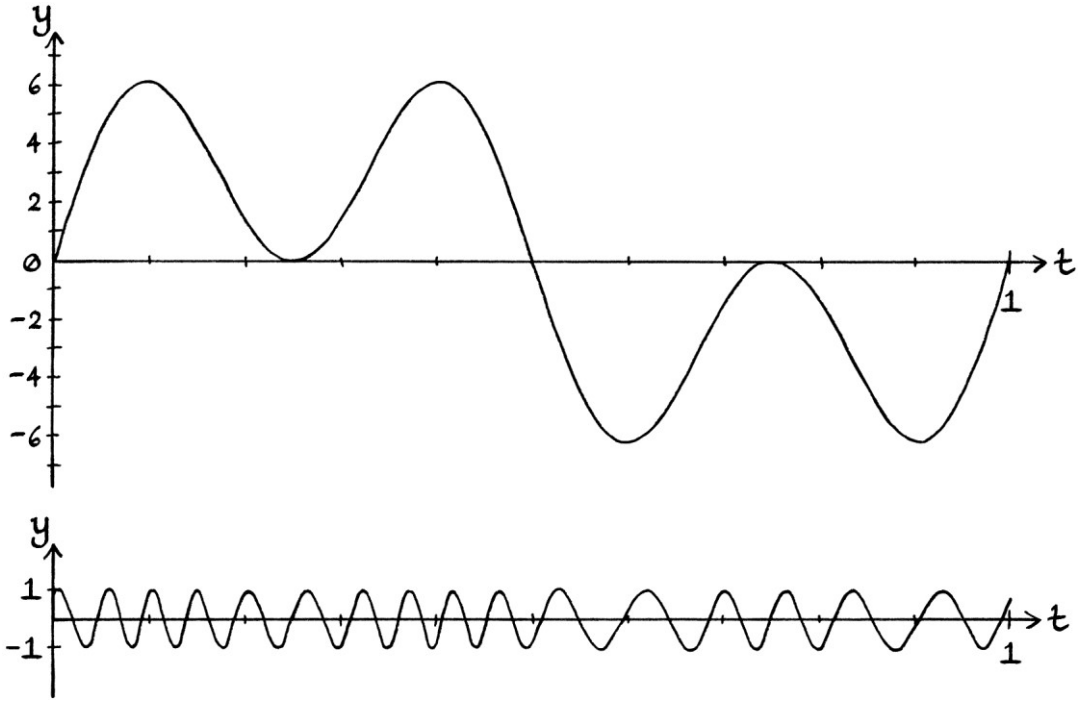
Our anti-derivative signal looks like this:



We will use a carrier wave with the formula "y = sin (2π * 16)". After using phase modulation with addition, we end up with this signal:



The way that the finished signal is a frequency-modulated version of the original signal is more apparent when we see the two signals together over just one cycle:



When the original signal is at its highest, the frequency of the result is at its highest. When the original signal is at its lowest, the frequency of the result is at its lowest. For this particular example, a carrier wave with a higher frequency would make a signal that better contained the characteristics of the modulating signal. However, it is harder to see the differences in frequency in a drawing if we use a faster carrier wave.

Faster carrier waves

As the modulating signal becomes more complicated, a faster carrier wave is needed to incorporate all the changes. The faster the carrier wave is, the harder it is to notice *visually* that there is any difference between the carrier wave and the modulated signal. For this reason, there is not much point in going through examples that are more complicated than those we have already seen in this and the previous chapter – we would not be able to make any new observations from them.

More about frequency modulation

Angle modulation

Phase modulation is often referred to as a type of "angle modulation" because it involves altering the phase of the carrier wave, and the term "phase" is really just a slightly more nuanced alternative to the word "angle". Because frequency modulation can be carried out with phase modulation, it, too, is often referred to as a type of "angle modulation". Therefore, in books on waves, it is common to see statements such as "phase modulation and frequency modulation are both types of angle modulation." As we saw in Chapter 35, frequency shift keying can be achieved without using phase modulation. This means that not all frequency modulation is strictly speaking a form of angle modulation. However, the fact that frequency shift keying *can* be performed with phase modulation if desired, means that this distinction is usually overlooked.

Because non-digital frequency modulation uses phase modulation, frequency modulation and phase modulation can be thought of as variations of the same idea.

Frequency modulation in the real world

Frequency modulation has advantages over amplitude modulation in real-world radio broadcasting because the message is encoded into the frequency. The frequency of a radio signal rarely changes over its journey. The amplitude of any signal will always decrease as the signal travels from its source, and the amplitude of a signal is much more prone to interference than the frequency.

FM radio stations produce higher quality broadcasts partly because they are using frequency modulation, and partly because they are allowed to use more bandwidth to transmit their signals. The fact that they use more bandwidth means that FM radio stations are usually legally required to broadcast between about 87 MHz and 108 MHz. The nature of how radio waves travel means a signal broadcast at these frequencies cannot travel as far as a signal broadcast at, say, 1 MHz.

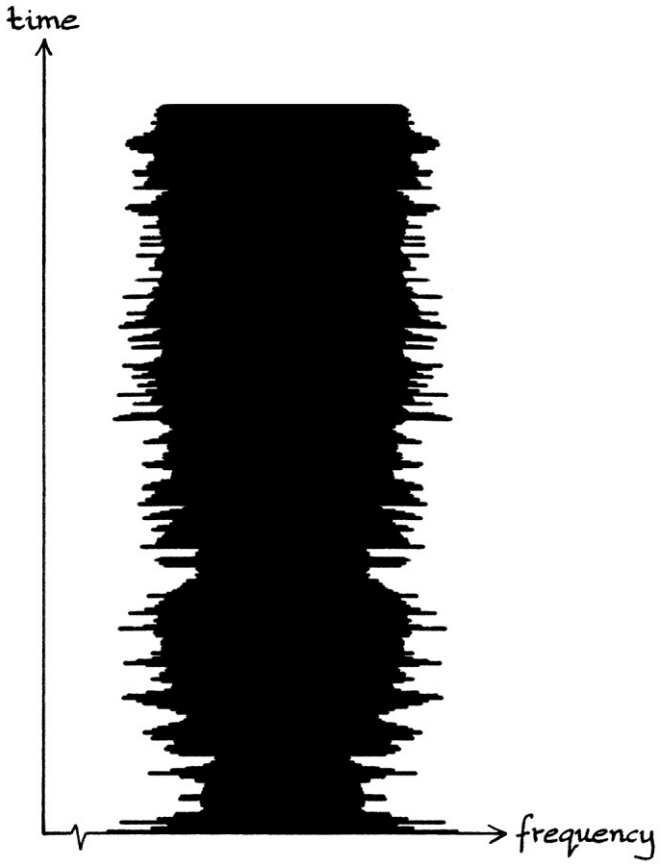
Frequency modulation requires more sophisticated encoding and decoding equipment than amplitude modulation does. This is not as much of a problem as it used to be, and nowadays, FM can be performed with cheap integrated circuits.

Carrier waves

When we use *amplitude* modulation, the signal as it appears in the envelope of the modulated signal is made up of individual points at the peaks of the signal. It is really a discrete signal. [A discrete signal is one made up of individual y-axis values at evenly spaced moments in time, as will be discussed in the next few chapters.] The faster the carrier wave, the closer the peaks will be, and the more accurately the modulating signal will be portrayed in the envelope of the modulated signal. With frequency modulation, the same is true. To decode a frequency-modulated signal, we need to identify the frequencies at regular intervals along the signal. The closer the intervals at which we identify the frequencies, the more accurately we will recover the original modulating signal. However, if the carrier wave had been a relatively slow one, the frequencies we measure will not differ enough to get an accurate portrayal. If we use a very slow carrier wave, we will have signals that make for clearer drawings, but they will not be able to portray the modulating signal particularly well. The faster the carrier wave, the better the recovered signal will be.

Recognising frequency modulation

FM radio broadcasts are easy to recognise on a real-time frequency domain plot on a computer screen. The signal will generally be a wide symmetrical column with a solid centre and jagged edges. Typical FM broadcasts are much wider than AM broadcasts.

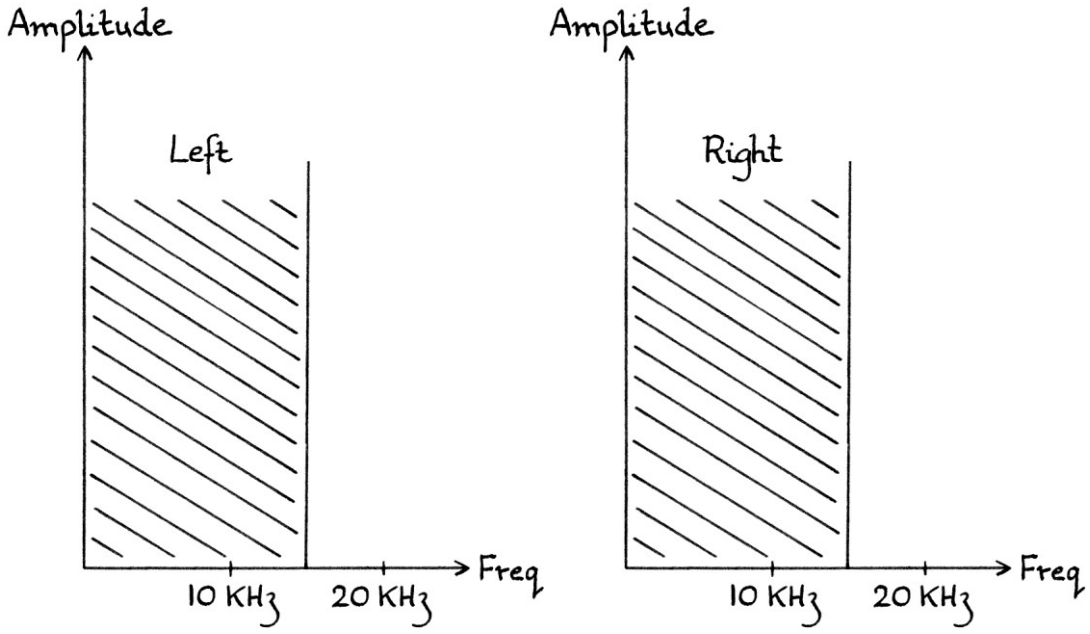


FM stereo

Only one signal can be encoded at a time with modulation. This does not really need saying because modulation is the alteration of a carrier wave by one signal. An obvious consequence of this is that any audio signal directly encoded using modulation will be a mono audio signal. Many radio stations use a method of encoding *stereo* audio into one signal, and then use that signal to frequency modulate a carrier wave. This is "FM stereo".

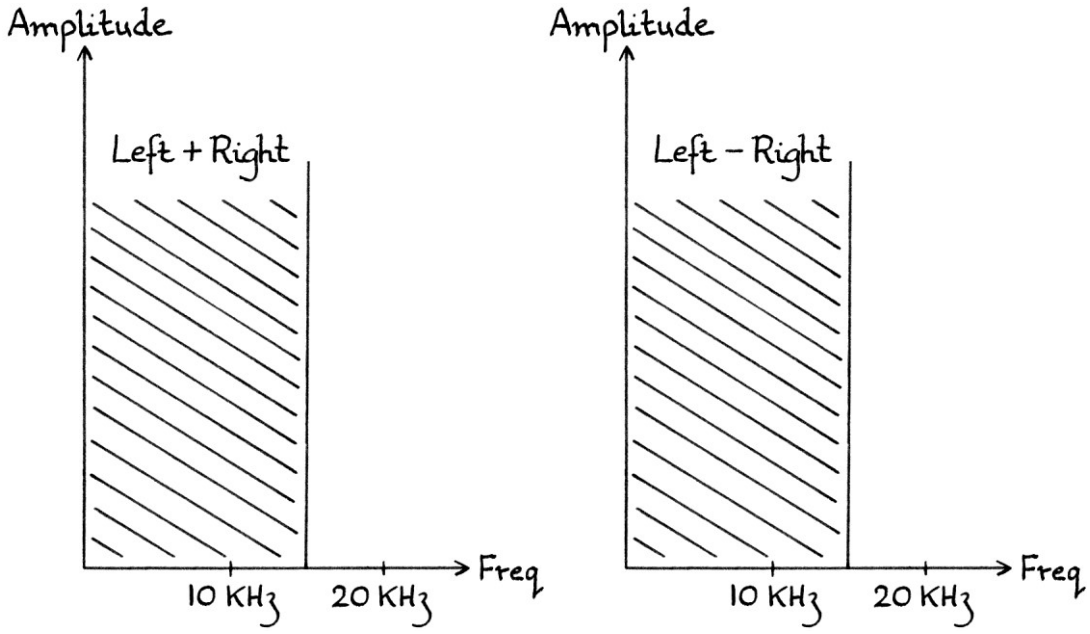
The method to achieve stereo FM broadcasting is clever, yet reasonably simple. The idea relies on the assumption that the original audio signal will not contain frequencies above 15 KHz. As humans cannot hear sounds with frequencies over about 20 KHz, radio stations can filter out any frequencies above 15 KHz without making much difference to the sound. [For comparison, the highest key on a typical full-size (88 key) piano has a frequency of 4,186 Hz.]

The process is as follows. First, we start with two audio signals. One will be the "left" audio signal and the other will be the "right" audio signal.



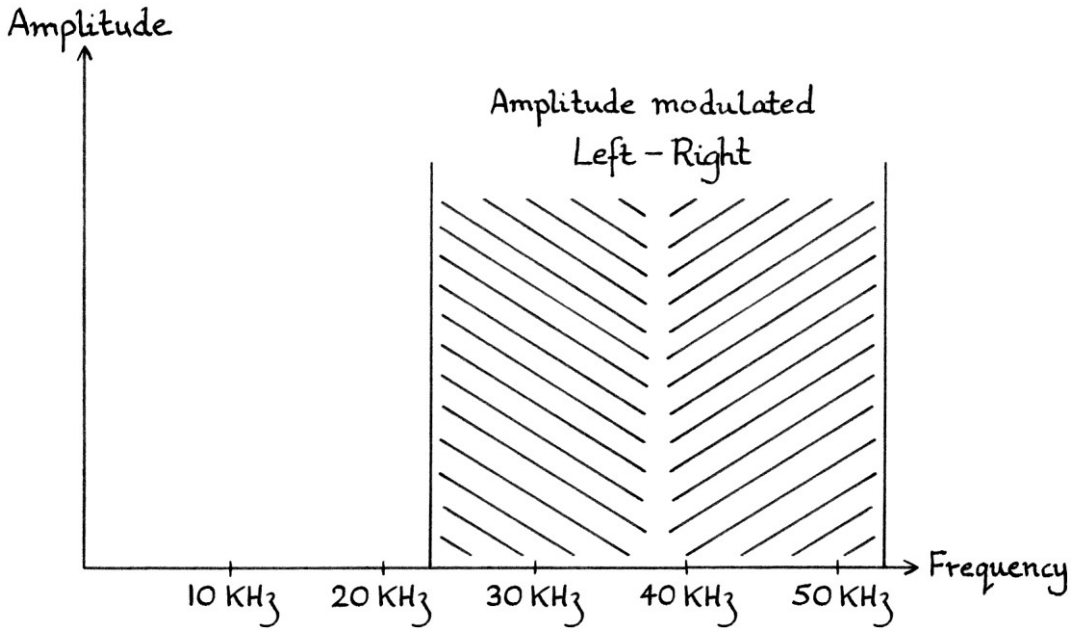
Before any actual frequency modulation takes place, the two signals are altered as follows. From the two signals, two new signals are created that are, first, the sum of the left and right signals, and second, the right signal subtracted from the left signal. To put this more succinctly, they are:

“Left + Right”
... and:
“Left – Right”.

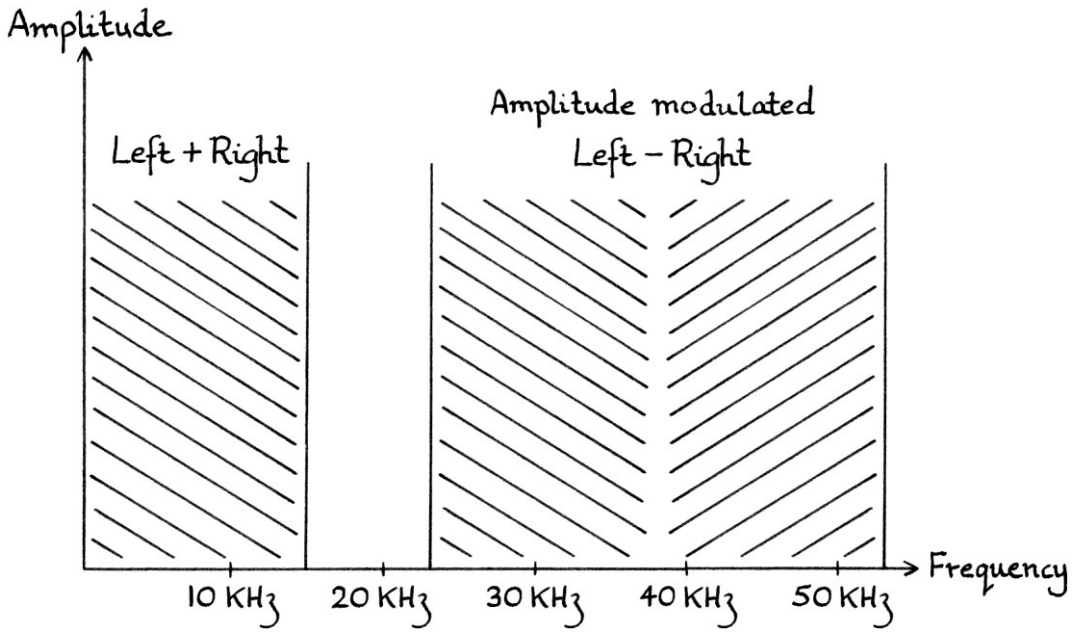


The “left + Right” signal is kept as it is.

The “Left – Right” signal is used to *amplitude*-modulate a 38 KHz carrier wave. The carrier wave itself is then removed from the result. The result of this amplitude modulation will be a signal that only contains frequencies between 23 KHz and 53 KHz.

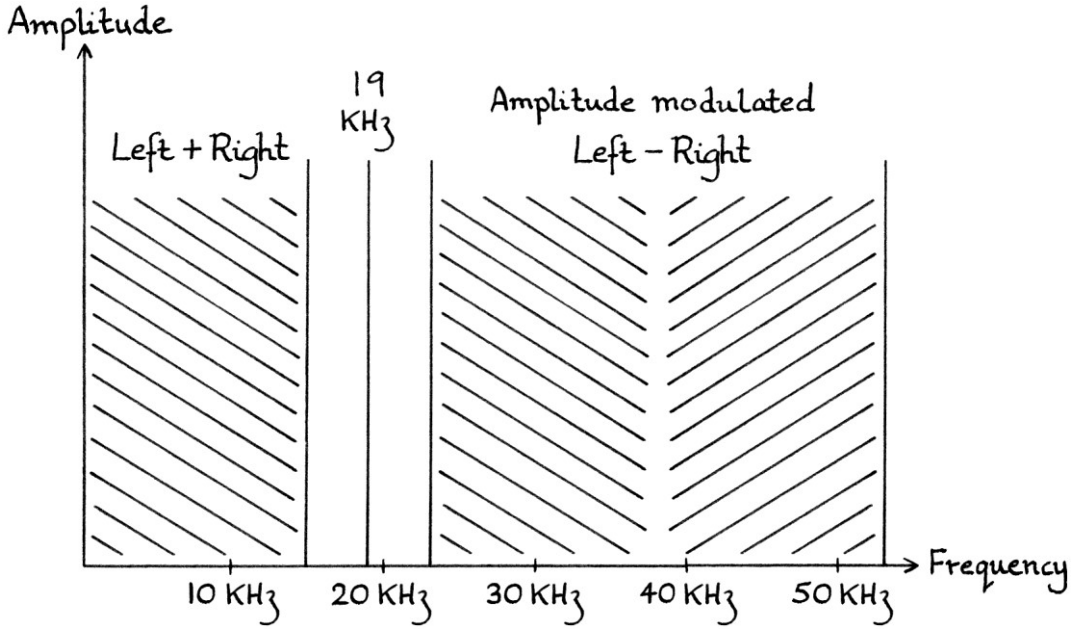


The result of the amplitude modulation is then added to the “Left + Right” signal. As different frequencies do not mix together, this means that we end up with a signal that contains the lower frequencies of the “Left + Right” signal, and the higher frequencies of the 38 KHz amplitude modulation.



The two separate signals do not affect each other in the sum because they are far enough apart frequency-wise that there are no matching frequencies. If we were to analyse this combined signal to finds its constituent frequencies, we would be able to discover the constituent waves of the two separate signals perfectly without any confusion. [If the two signals shared some frequencies, then the constituent waves with shared frequencies would become added together, and be irretrievable. As the two original "Left" and "Right" audio signals are not expected to have frequencies over about 15 KHz, the "Left + Right" signal will not have frequencies over 15 KHz, and the amplitude modulated signal will not have frequencies below 23 KHz (or above 53 KHz).]

The combined signal then has a 19 KHz wave added to it. This wave acts as an indicator to radio receivers that the signal is a stereo FM signal. As the wave is 19 KHz, when it is added to the combined signal, it will not conflict with either the lower frequency part of the signal (which will have a maximum frequency of 15 KHz) or the higher frequency part of the signal (which will have a minimum frequency of 23 KHz).



The whole combined signal is then transmitted using frequency modulation.

FM stereo decoding

When the signal is received, it is first demodulated using whichever method of decoding frequency modulation is desired. If the receiver does not understand FM *stereo* broadcasts, it does not matter, as it will still be able to decode the lower frequency half of the signal. The lower frequency part of the signal was originally the left audio stream added to the right audio stream. This sum sounds like a normal mono audio sound. The higher frequency half of the signal will be of too high a frequency for a mono FM decoder to decode, and even if it could decode it, the sound would have such a high frequency that it would not pass through a typical loud speaker (and no human would be able to hear it if it could).

If the receiver *does* understand FM stereo broadcasts, it will look for the 19 KHz wave, and if it can see it, it will know it can decode the signal as a stereo signal. First, it removes the 19 KHz wave. Then, it splits the signal into two parts by filtering it. It creates one signal by applying a low pass filter that blocks any frequencies *over* 19 KHz – this leaves the "Left + Right" part of the signal. It then creates a second signal by applying a high pass filter that blocks any frequencies *below* 19 KHz – this leaves the amplitude-modulated part of the signal.

The amplitude modulated part is decoded (for example, by multiplication and filtering) to produce the "Left – Right" part of the signal.

Now that the decoder has the "Left + Right" part and the "Left – Right" part, it can add them to produce just the left audio signal (but at twice the amplitude):
Left + Right + Left – Right = 2 * Left
... and it can subtract the second from the first to produce the right audio signal at twice the amplitude:
Left + Right – (Left – Right) = Left + Right – Left + Right = 2 * Right

It will then have the original left and right audio signals at twice the original amplitude. As the amplitudes of the signals will be altered by the volume control on the radio by the user, the exact amplitudes are not important as long as the relative amplitudes are correct.

FM Stereo extra signals

In the above system, the "Left + Right" signal takes up the frequency range from 0 to 15 KHz. The amplitude modulated "Left – Right" part of the signal takes up the frequency range from 23 to 53 KHz. It is also possible for more signals to be encoded into the main signal as long as they are altered to have constituent frequencies that do not interfere with the rest of the signal. For example, as long as a signal consisted solely of frequencies above 53 KHz, it could be added to the main signal, and it would not corrupt the existing information. Any number of extra signals could be added, but with the drawback that every extra signal increases the bandwidth of the final frequency-modulated signal. This is a problem for several reasons, among which are:

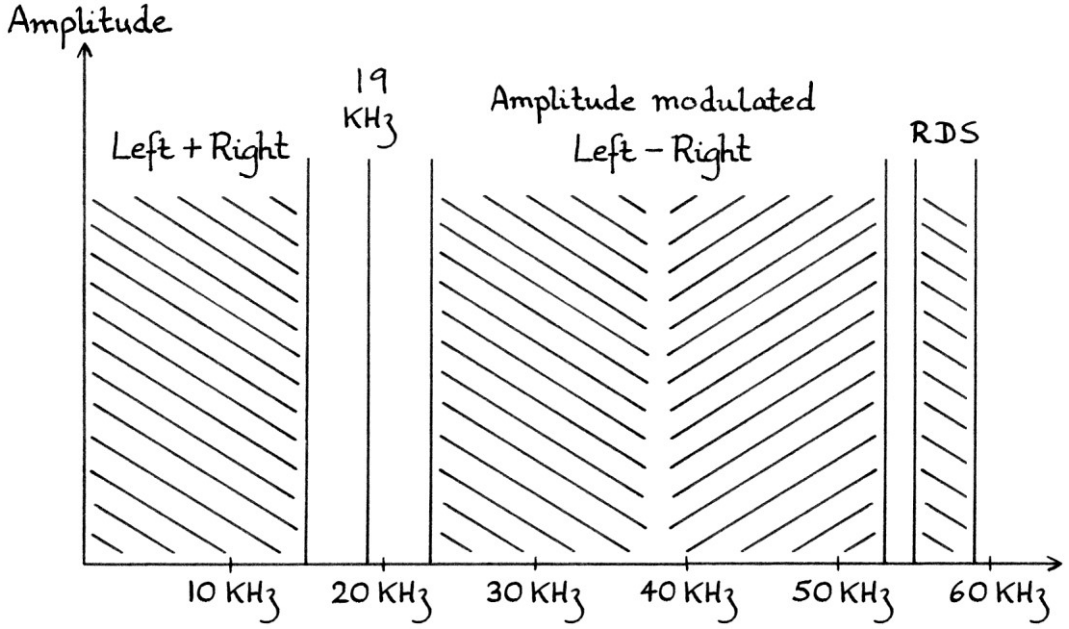
- Licensed FM radio stations are legally limited to broadcasting within a certain bandwidth. This can vary from 100 KHz to 200 KHz. This places a limit on the number of extra signals that can be transmitted.

- It costs money to transmit a radio signal, so any extra transmitted bandwidth either needs to be a negligible extra cost or to serve a useful purpose.

- There would need to be radio receivers capable of decoding the extra signals. This would involve an agreement between broadcasters and radio manufacturers. Manufacturers would need to believe that the extra ability was worth the cost of designing and making the receivers.

In some countries, FM broadcasts might contain extra signals that are used by specialist receivers to convey hidden music streams or traffic information, among other things.

In practice, the most common extra signal above the "Left – Right" part of the FM signal will be the Radio Data System signal, or "RDS" for short. This is a digital signal commonly decoded by vehicle radio receivers where it is used to show the name of the radio station on the display, among other messages. RDS is a 2-PSK signal that is added to "Left + Right, Left – Right" signal before the whole signal is frequency modulated. The RDS signal sends binary data at a rate of 1,187.5 bits per second. [Note that it never actually sends half a bit because that would be impossible and meaningless. The number of sent bits *averages* 1,187.5 bits every second.] The signal has frequencies reaching up to 2 KHz either side of 57 KHz.



Radio stations incorporate RDS using commercially made hardware. The broadcasting hardware is so simple to use that even pirate radio stations transmit RDS signals.

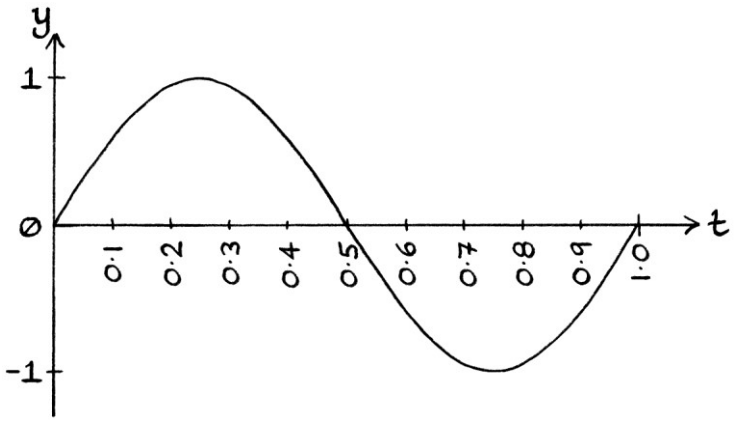
The future of FM

As broadcast radio stations move from analogue to digital, non-digital broadcast FM will eventually become obsolete. However, there might still be uses for non-digital FM for other purposes. Regardless of what happens in the future, it is good to know how non-digital frequency modulation works to get a better understanding of waves.

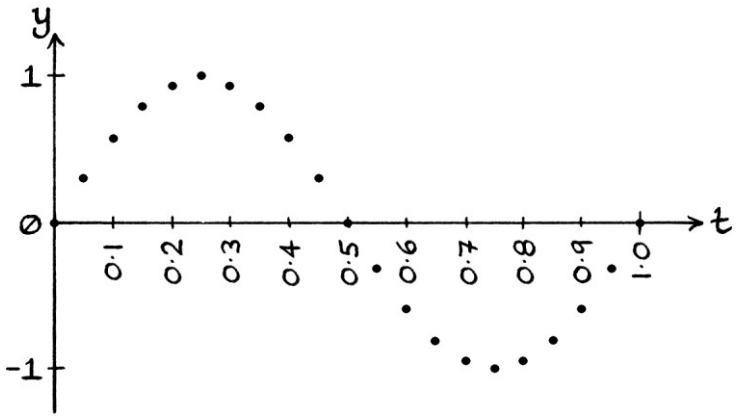
Chapter 39: Discrete signals

There have been many times in this book when we have needed to perform maths on signals, but we did not use the signals themselves, but instead evenly spaced readings from the signals. This made what we were doing easier to visualise, and easier to do. For example, if we want to add 1 to a wave's mean level, it is easier to take evenly spaced points from the wave, add 1 to each of them, and then join up the points afterwards. In Chapter 3 of the first part of this book, when we were first introduced to Sine waves, we took evenly spaced points from around a circle, and plotted their heights on a graph with the angle as the x-axis. The graph we created consisted of a series of individual points, but it had the form of a Sine wave. If we had taken hundreds of readings from the circle, the graph of the Sine wave we created would have been nearly indistinguishable from the graph of an actual Sine wave. If we had taken an infinite number of readings, the Sine wave we created would have been perfect. From all of the above, we can see that having a series of evenly spaced readings from a signal is sufficient both for portraying the signal and for performing calculations on the signal.

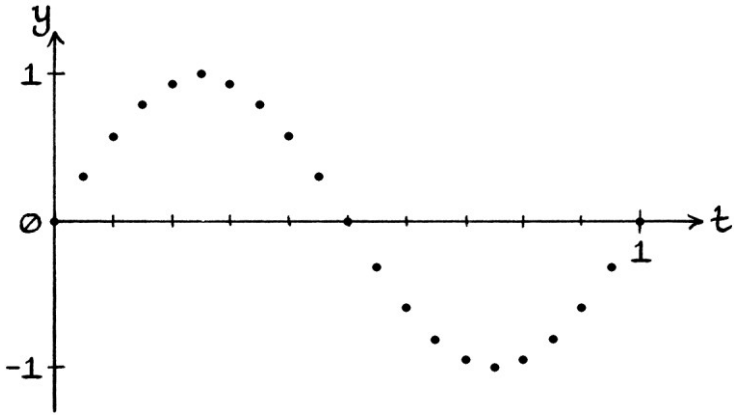
A Sine wave has an uninterrupted curve with no breaks in it. We can call such a wave "a continuous wave" or a "continuous signal" because its curve is continuous:



If we had a wave made up of evenly spaced readings from that Sine wave, it would be called a “discrete wave” or a “discrete signal”. If we had 20 readings per second of wave, it would look like this over one second:



The same graph with clearer t-axis numbering is as so:



The y-axis values in the above signal are as so:

Time (seconds)	**y-axis value (units)**
t = 0	y = 0
t = 0.05	y = 0.3090
t = 0.1	y = 0.5878
t = 0.15	y = 0.8090
t = 0.2	y = 0.9511
t = 0.25	y = 1
t = 0.3	y = 0.9511
t = 0.35	y = 0.8090
t = 0.4	y = 0.5878
t = 0.45	y = 0.3090

t = 0.5	y = 0
t = 0.55	y = −0.3090
t = 0.6	y = −0.5878
t = 0.65	y = −0.8090
t = 0.7	y = −0.9511
t = 0.75	y = −1
t = 0.8	y = −0.9511
t = 0.85	y = −0.8090
t = 0.9	y = −0.5878
t = 0.95	y = −0.3090
t = 1	y = 0 [This is the start of the second cycle.]

A discrete signal is one that is being portrayed as a series of evenly spaced y-axis values. These y-axis values might be readings from a continuous signal, or they might have been created from scratch. Whatever the source of a discrete signal's values, the values are intended to be a representation of a normal continuous signal.

Having a signal in a discrete form is useful for several reasons, including:

- It allows the signal to be stored as a list of numbers. We could write the values down on a piece of paper, or more commonly, they would be stored on a computer. This would be impossible to do with a continuous signal, mainly because the nature of a continuous signal makes it incompatible with how modern electronic computers work.

- It allows maths to be performed on the signal as a whole by dealing with each individual value in turn. This means that computers can operate on the values.

- It allows types of calculations to be performed that cannot be done as simply otherwise.

- It makes understanding waves easier. Seeing a wave as a series of values helps in visualising processes on the wave.

There are also some disadvantages:

- A discrete signal can only ever be an approximation of a continuous signal.
- If there are not enough readings per second of signal, then the discrete signal will not be a good representation, and any mathematical procedures will produce erroneous results.

Definitions

We will now look at the definitions of continuous and discrete signals in more depth.

Continuous signals

A continuous signal is what might be called a "natural" signal. We could zoom in forever, and never see gaps in the curve. There are an infinite number of points in any part of it. A continuous signal might jump upwards or downwards, or have areas that are zero, but there will be no areas where there is no value at all. For example, if we have a Sine wave, we can zoom in on any particular point of the wave and there will always be a continuous curve. Another way of thinking about this is that, given the formula for a Sine wave such as "$y = \sin 2\pi t$", there is no value of "t" for which there is no result. Even if "t" is 3.0000000000000000000001, "y" will still have a value. A Sine wave is continuous. Similarly, for a square wave that fluctuates between 1 and 0, we can zoom in on any part, and we will always see values that are either 1 or 0. There are no gaps where there is no value.

Another name for a continuous signal is an "analogue" signal. The term "analogue" has come to mean "not digital" in the sense of not relating to digital electronics. However, in its strictest sense, "analogue" (when used with signals) means "analogous to nature". In other words, it means a signal that is the same as we would see "naturally".

Discrete signals

A discrete signal is made up of distinct, separate, values. In fact, the word “discrete” means distinct and separate. A discrete signal is really a list of values that are, or could be, readings from a continuous signal.

A discrete signal is a list of values. The values can be plotted on a graph or be subjected to maths, but the important thing to know is that a discrete signal is a list of numbers. When supplied with a discrete signal, we might just be given a list of values such as this:
0, 0.3090, 0.5878, 0.8090, 0.9511, 1, 0.9511, 0.8090, 0.5878, 0.3090 and so on.

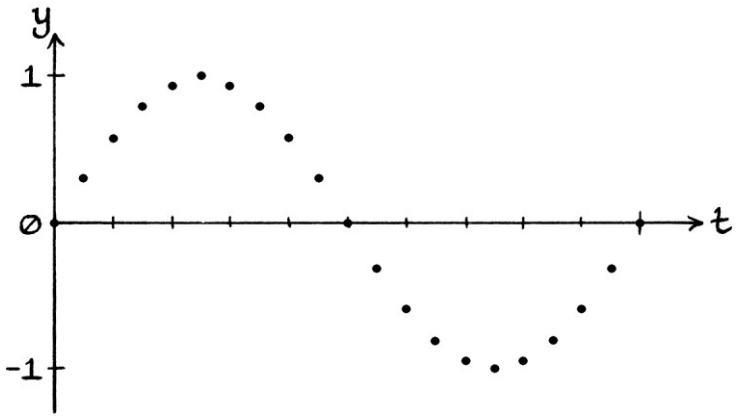
If we know how many y-axis values there are per second, then we know the timings that go with each of the values in the list. If we do not, then it is still possible for the list of values to be useful. It is helpful to know the timing, but not always essential.

With a discrete signal, there is nothing between each value. By this, I do not mean that the signal is zero between each value, but that the signal is literally undefined between each value. For example at the start of our earlier list of values from a Sine wave, we had the y-axis values:

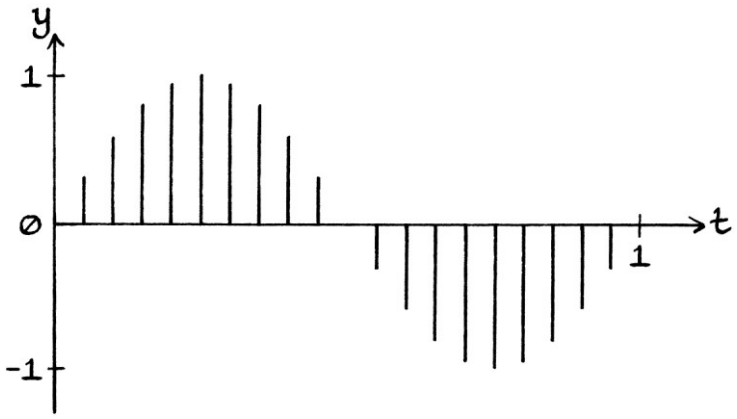
0
0.3090
0.5878
0.8090

There are no values between 0 and 0.3090, just as there are no values between 0.3090 and 0.5878, or between 0.5878 and 0.8090. We can make no assumptions about any hypothetical y-axis values between the given values of the list. We only know the values that we have been given, and anything in the gaps is considered undefined.

Drawings of discrete signals generally take two forms. Either we can mark the values with crosses or dots as so:



... or we can draw vertical lines from the t-axis up to, or down to, the values:



When portraying discrete signals on a graph, we should not join up the dots to make a curve because doing so implies that we know the values between the points. It implies that the signal is continuous. Therefore, if we are being "academically correct", a discrete signal should only be drawn as separate values. The only times when we should join up the dots are when we are turning the discrete signal graph into a continuous signal graph, or as part of an explanation that requires it. Note that previously in this book, we have joined up the dots, but this was done to demonstrate ideas and help with learning. If we are working with signals that are supposed to be discrete, then we should not do this. In the previous examples in which we joined up the dots, we knew that the values in the gaps were between the values either side. In a proper discrete signal, we do not know this and cannot make that assumption. [I will sometimes join up the dots of discrete graphs to make a clearer picture of how the samples are situated – it will not be to imply that the signal continues between the samples.]

When dealing with discrete signals, the only information we are given is the list of readings, and perhaps, but not necessarily, the time that those readings were taken.

One big distinction between continuous signals and discrete signals is that discrete signals are a list of numbers, while continuous signals *cannot* be portrayed by a list of numbers. *Approximations* of continuous signals can be portrayed with a list of numbers, but that approximation will be a discrete signal.

Previously in this book

Throughout this book, we have essentially used the concepts of both continuous signals and discrete signals interchangeably, but without acknowledging the fact. For most of this book, we were really making a temporary step from continuous to discrete signals and back for the purposes of maths or an explanation. With discrete signals, as long as there are enough readings per second, most maths performed on the readings will be identical to the same maths performed on the original continuous signal.

Spelling

It is easy to confuse the spelling of the word “discrete”, which means distinct or separate, with the spelling of the word “discreet”, which means careful in being secretive. One way to remember the spelling of “discrete” is to notice that each “e” is separated by the letter “t”. The letters are distinct. Each letter “e” is discrete in the word “discrete”.

Creating a discrete signal

It is possible to create a discrete signal from scratch, but the potential obstacles in doing so are the same as creating one with readings from a continuous signal. Therefore, in this section, we will look at reading values from a continuous signal to make a discrete signal.

Sampling

An individual y-axis value in a discrete signal is called a "sample". We can think of the continuous signal that the discrete signal is based on as having been "sampled" in the same way that one might sample food – a piece is taken from the whole. The following list of y-axis values can be called a list of samples:

0, 0.3090, 0.5878, 0.8090, 0.9511, 1, 0.9511, 0.8090, 0.5878, 0.3090

"Sampling" is the name of the process of reading y-axis values from a signal. It is the process that collects samples. When we collect samples from a signal, we need to take them at evenly spaced moments in time. If we do not, then we would have to record the time that they were taken, which would require more information to be stored. There is a general presumption that any discrete signal consists of values that exist at evenly spaced moments in time. To create a discrete signal otherwise would make everything unnecessarily confusing.

The number of samples taken from a continuous signal per second is called the "sample rate" – it is the rate of sample taking. If we take 10 y-axis readings from a signal every second, we would have a sample rate of 10 samples per second. If we have a sample rate of 100 samples per second, then in every second, we would have 100 samples, which is the same as saying 100 y-axis values.

The term "samples per second" is often abbreviated to "sps". Do not confuse this "sps" with the "sps" that means "*symbols* per second". The context will usually indicate which is meant. [The term "symbols per second" refers to how many groups of binary bits are sent in one go when transmitting binary data; the term "samples per second" refers to how many y-axis values are recorded per second when creating a discrete signal.] The sample rate is also sometimes called the "sampling frequency" because it refers to how often a sample is taken per second. It can be slightly confusing to have the term "sampling *frequency*" when we are discussing waves and signals that have their own definition of "frequency", but you will get used to the idea.

The amount of time between each sample is the reciprocal of the sample rate, and vice versa. In other words:

1 ÷ sample rate = interval between each sample in seconds
1 ÷ interval between each sample = the sample rate.

If we have a sample rate of 1000 samples per second, there will be 0.001 seconds between each sample. If we have a sample rate of 5 samples per second, there will be 0.2 seconds between each sample.

The same list of values can have different meanings depending on the sample rate. For example, if we have this list of samples:

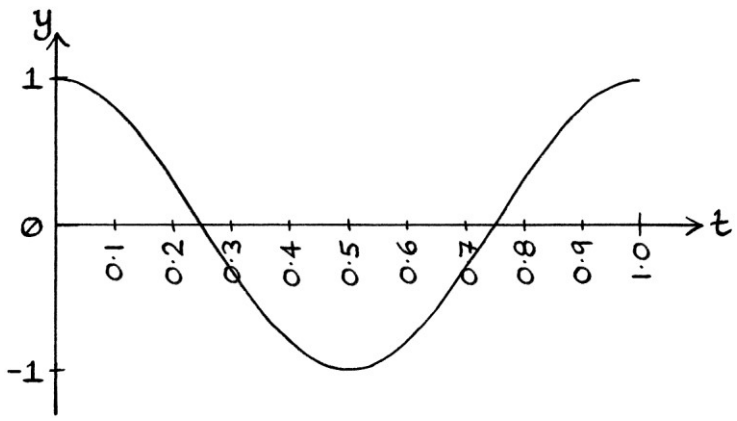
0, 0.3090, 0.5878, 0.8090, 0.9511, 1, 0.9511, 0.8090, 0.5878, 0.3090

... and the sample rate is 10 samples per second, then there are 0.1 seconds between each sample. However, the same list of samples with a sample rate of 1000 samples per second would mean that there are only 0.001 seconds between each sample, so the wave represented by the samples is a much faster one. Knowing the sample rate gives meaning to the list of samples.

The amount of time between each sample is called the “sampling period”. It is the period of time between the samples.

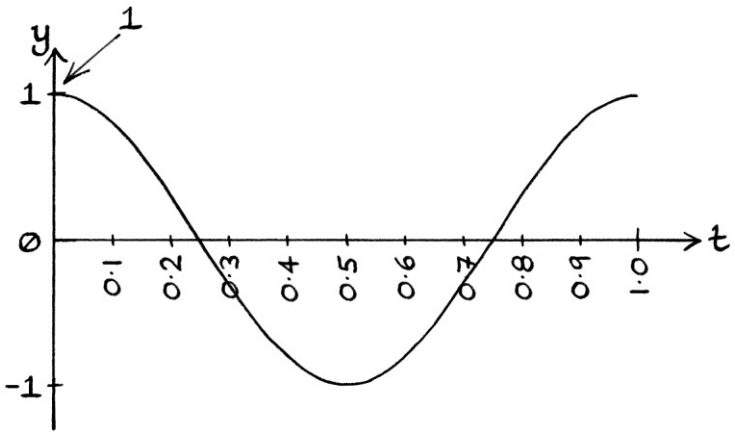
An example

As a simple example of sampling, we will take samples from this wave, which has the formula “y = sin ((2π * 1t) + 0.5π)” in radians:

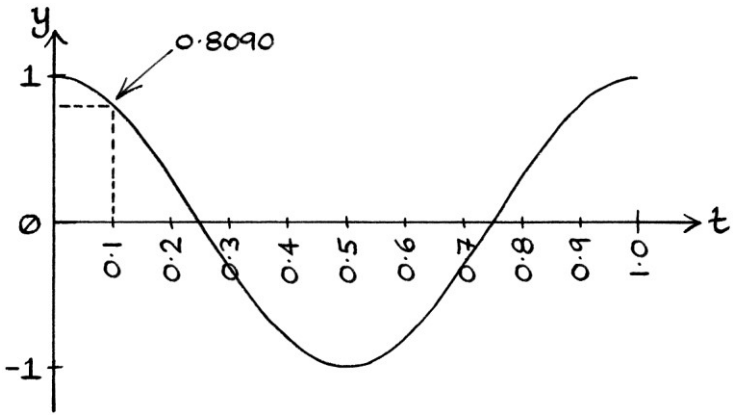


We will have a sample rate of 10 samples per second and we will take samples for one cycle. As each cycle lasts one second and our sample rate is 10 samples per second, this means that we will be taking 10 readings, and we will end up with 10 samples. Each sample will be 0.1 seconds apart – the sampling period will be 0.1 seconds. The process is very straightforward, and is essentially one that we have been using since the start of this book.

Our first reading is the y-axis value at t = 0:



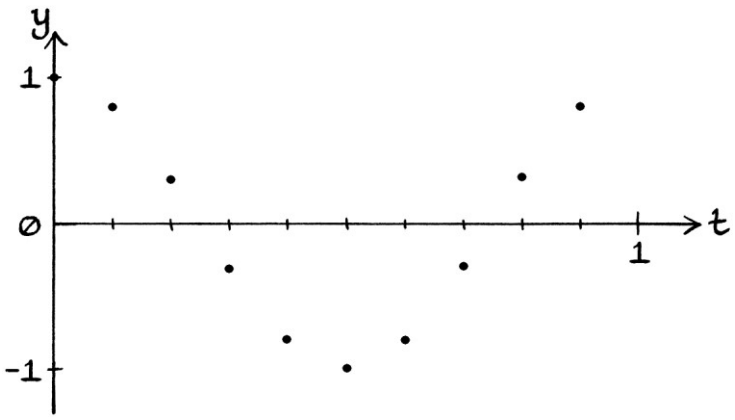
This is 1. Therefore, our first sample is 1. Our second reading is the y-axis value at t = 0.1 seconds, which is 0.8090 units. Therefore, our second sample is 0.8090.



Our third reading is at t = 0.2 seconds, which is 0.3090
Our fourth reading is at t = 0.3 seconds, which is −0.3090
The fifth reading is at t = 0.4 seconds, which is −0.8090
The sixth reading is at t = 0.5 seconds, which is −1
The seventh reading is at t = 0.6 seconds, which is −0.8090
The eighth reading is at t = 0.7 seconds, which is −0.3090
The ninth reading is at t = 0.8 seconds, which is 0.3090
The tenth reading is at t = 0.9 seconds, which is 0.8090

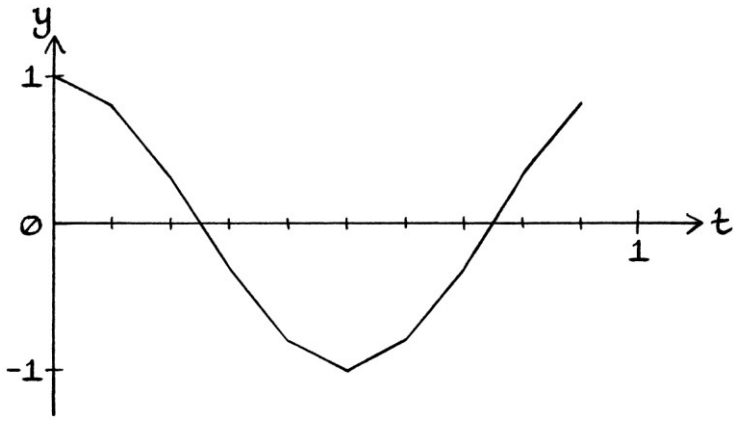
If we were to take another reading, it would be the eleventh reading at t = 1 second, which is the start of the next cycle and the start of the next second. As we are taking only one cycle's worth of readings, we will stop at the tenth reading. Our final list of samples is as so:
1, 0.8090, 0.3090, −0.3090, −0.8090, −1, −0.8090, −0.3090, 0.3090, 0.8090

If we plot these points on a graph, we end up with this picture:



We could give our list of values to anyone, and as long as they knew the sample rate, they would be able to recreate our *discrete* wave graph. However, without other information, they would not be able to recreate our original *continuous* wave from the list. If they joined up the points on a graph, they would have an approximation to the original wave. The approximation might be sufficient for many purposes, but it would not be the original wave.

We can get an idea of how well the samples represent our continuous wave by joining up the points on the graph. Note that we are joining up the points only to see the layout of the samples – we are not suggesting that we know the values between the samples. The joining up is merely a temporary aid to understanding the positions of the samples.



We can improve the accuracy of our discrete wave by increasing the sample rate. Therefore, we will choose a sample rate of 20 samples per second. This means that our readings will be 1 ÷ 20 = 0.05 seconds apart – the sampling period will be 0.05 seconds. We proceed in the same way as when we had a sample rate of 10 samples per second, but we take more readings.

Our first reading will be at t = 0, where y is 1. Therefore, our first sample is 1.
Our second reading will be at t = 0.05 seconds, where y is 0.9511 units. Therefore, our second sample is 0.9511. Our third sample is at t = 0.1 seconds, which is 0.8090.

Our full list of y-axis values and the times they came from is as so:

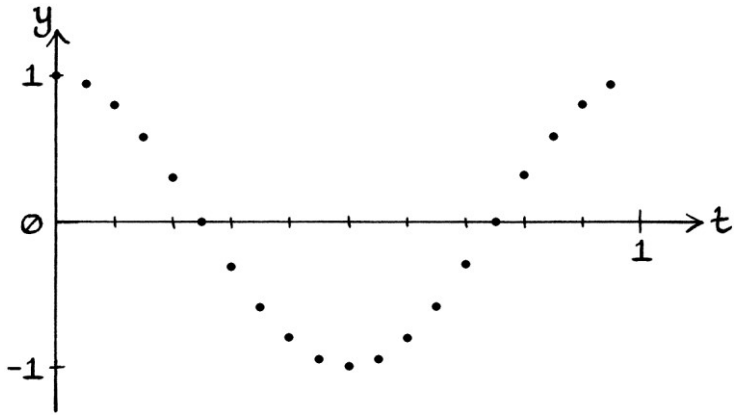
Time (seconds)	**y-axis value or in other words, the sample value**
t = 0	y = 1
t = 0.05	y = 0.9511
t = 0.10	y = 0.8090
t = 0.15	y = 0.5878
t = 0.20	y = 0.3090
t = 0.25	y = 0
t = 0.30	y = −0.3090
t = 0.35	y = −0.5878
t = 0.40	y = −0.8090
t = 0.45	y = −0.9511
t = 0.50	y = −1
t = 0.55	y = −0.9511
t = 0.60	y = −0.8090
t = 0.65	y = −0.5878
t = 0.70	y = −0.3090
t = 0.75	y = 0
t = 0.80	y = 0.3090
t = 0.85	y = 0.5878
t = 0.90	y = 0.8090
t = 0.95	y = 0.9511

We can portray our discrete wave as a list of samples as so:

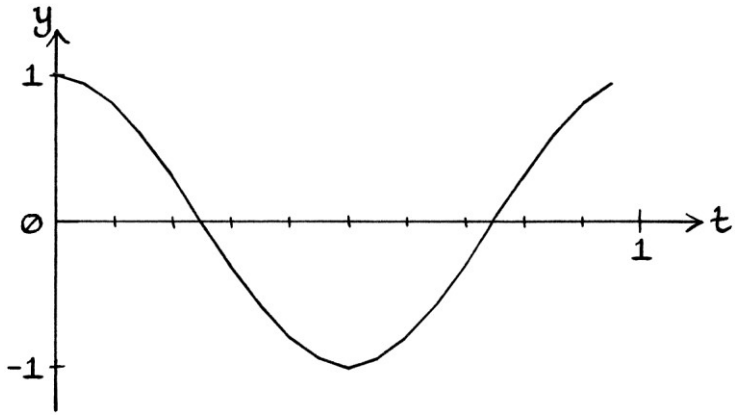
1, 0.9511, 0.8090, 0.5878, 0.3090, 0, −0.3090, −0.5878, −0.8090, −0.9511, −1, −0.9511, −0.8090, −0.5878, −0.3090, 0, 0.3090, 0.5878, 0.8090, 0.9511

As before, we could give the list to anyone, and, as long as they knew the sample rate, they would be able to recreate our discrete wave. The discrete wave is a better approximation of the original wave than when we used 10 samples per second, but it is still an approximation, as all discrete waves are.

We can plot the points on a graph:



If we join up the points of our discrete wave, we can see how the resulting signal is closer to our original wave: [Again, we are joining the points up to see the layout of the samples on the graph better – we are not intending to imply that we know the values between each sample.]



The higher the sample rate (the more samples we take per second), the more accurately the discrete wave will represent the original continuous wave. Conversely, the higher the sample rate, the more effort it is to create and use the discrete signal. If, for example, we wanted to add 1 to every value of our discrete signal as we have just sampled it, we would only need to perform 20 additions. We could also write out the list of samples reasonably easily. If we had a sample rate of ten million samples per second, we would have a more accurate representation of the original wave, but adding 1 to every value would require ten million additions, and it would be much more effort to write out the list of samples. The work

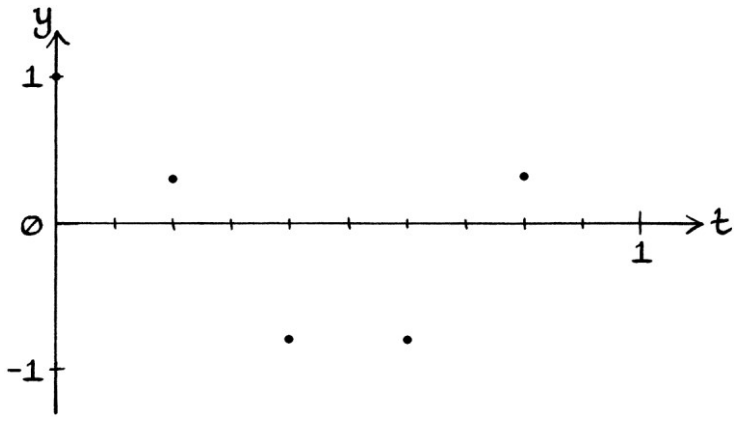
required in using ten million samples might take too long to be useful, especially if the signal is being analysed in real time. There is always a compromise between accuracy and usability. Depending on what we want to do with the discrete signal, there is a maximum number of samples per second above which there is no real benefit. For example, if we want to know how many peaks a four-cycle-per-second wave has in one second, then a sample rate of 20 samples per second is perfectly adequate. If we want to perform more complicated procedures, we might need a higher sample rate.

Lower limits

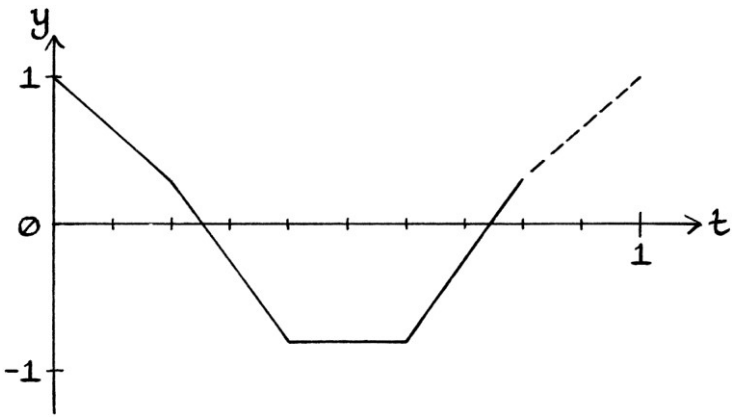
If we have the ability to achieve it, there is no limit to how high a sample rate can be. [Whether there is any point in having the highest sample rate possible is another matter.] However, there is a limit to how *low* a sample rate we can have without the discrete signal becoming useless. Earlier in this chapter, we used a sample rate of 10 samples per second to sample "y = sin ((2π * 1t) + 0.5π)", and we were still able to see the outline of the wave in the samples. If we only used a sample rate of 5 samples per second, one cycle of our continuous wave would be reduced to just this list of values:

1, 0.3090, −0.8090, −0.8090, 0.3090

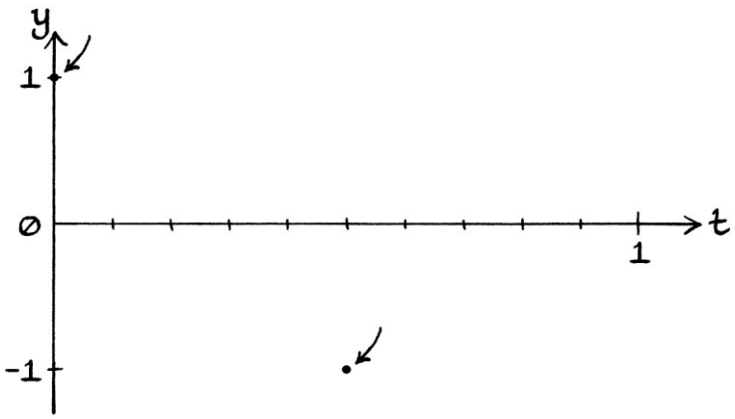
These points on a graph look like this:



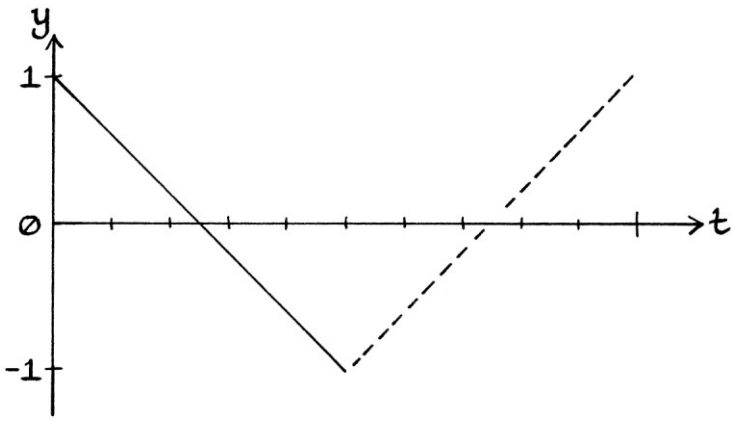
If we join up these points, we can see that our points are no longer a good approximation of the wave. [The dotted lines connect the samples we have to the sample from the next cycle, which is not included in our list.]



If we reduced the sample rate to just 2 samples per second, we would end up with just these values for the first cycle of our wave: +1, −1. These points are as so:



If we joined the points up, we would have the following graph: [The dotted lines connect the samples we have to the sample from the next cycle, which is not in our list.]



Such a sample rate might be adequate for counting the peaks and dips of our particular wave, but it would not be useful for other calculations. We cannot tell how the shape of the curve progresses from such a signal. We would not be able to distinguish this signal from one that literally consisted of two straight lines for each cycle. Most significantly, such a sample rate would be useless if our wave had:

- The same frequency, but a phase of zero cycles per second.
- A frequency faster than 1 cycle per second.

If we sampled the wave "y = sin (2π * 1t)" with a sample rate of 2 samples per second, the samples for each cycle would be: 0, 0. These two samples do not make a record of the continuous wave.

If we sampled *any* wave with a frequency of 2 cycles per second using a sample rate of 2 samples per second, there would only be 1 sample to describe each cycle. For example, the wave "y = sin ((2π * 2t) + 0.5π)" sampled with a sample rate of 2 samples per second would just have this sample for each cycle: 1

One sample per cycle is insufficient to record the wave, so our record of the wave will be useless. The problem here is similar to one discussed in Chapter 18. In that chapter, we calculated which waves were added to create a signal in a picture. We used readings spaced at intervals of a twentieth of a second. If the constituent frequencies in the signal had been slightly faster, we would have ended up with spurious results because our readings were too sparse.

The basic rule for choosing a sample rate is that it must be fast enough that nothing of interest can happen between the sample readings. Whether something is of interest depends on what it is we are doing. If a peak occurs between two samples, then we will have no record of it. If there is a change in a curve between two samples, of which we would like to be aware, then again, we will have no record of it. Therefore, the sample rate must be fast enough that we do not miss such things.

There is a more specific rule for sampling a wave that ensures that the recorded samples cannot be confused with those of a slower frequency wave – the sample rate must be more than twice the frequency of the wave. Therefore, if we were sampling a wave with a frequency of 5 cycles per second, we would have to sample it at a sample rate *above* 10 samples per second. The rule applies to sums of waves too, in which case, the sample rate must be more than twice the fastest constituent frequency within the sum of waves. If we were sampling a signal that contained frequencies of 1, 3, 5, and 9 cycles per second, we would need to sample it at a sample rate *above* 2 * 9 = 18 samples per second. We will learn more about this in Chapter 42.

The rule is best as a guide to a sample rate below which the samples will definitely represent the samples from other signals. In practice, we would need to sample at a higher sample rate, so that we can perform maths on the samples without corrupting the representation of the signal. For example, if we used the minimum sample rate to record a wave, we would not be able to add a faster frequency wave to it without the record of the waves becoming incorrect.

There are still useful attributes that can be missed even if the rule is followed, and outside of signal processing, the rule may or may not be useful. If we were using the curve of a wave as a template for a curve in architecture, then we would need a much higher sample rate.

Music

The most common place to see mentions of sample rates in everyday life is in digital audio files such as MP3s and WAV files. Digital music files tend to have sample rates such as:

- 22,040 samples per second
- 44,100 samples per second
- 48,000 samples per second
- 192,000 samples per second

  ... and so on.

A WAV file with a sample rate of 44,100 samples per second has, as expected, 44,100 samples every second. The original sound is sampled and reduced to 44,100 values every second. Such a sample rate is high enough that constituent frequencies below 22,050 cycles per second are recorded correctly in the discrete signal.

Often, with audio files, the sample rate is called the “sampling frequency”. In which case, instead of “samples per second”, the number of samples per second will be given in “hertz”. This can be slightly confusing to start with, but you will get used to it.

Accuracy

In the previous examples in this chapter, the samples were given to 4 decimal places. Doing this means that each sample is an approximation of the actual reading from the continuous signal. There will be samples where the sampled value is exactly the same as the relevant point on the continuous signal, but there will be more points where the value is an approximation. This is unavoidable, as we will now see.

We will consider this wave:
"y = sin (2π * 1t)"
... sampled with a sample rate of 8 samples per second.

At t = 0, the reading from the curve will be:
sin (2π * 0) = 0.

The sample will be exactly zero, which is completely correct.

However, at t = 0.125, the reading from the curve will be:
sin (2π * 0.125) = 0.707106781186548 to 15 decimal places.

This is the square root of 0.5, which is an irrational number. The decimal places continue forever. We cannot express this number without an infinite number of decimal places, which means that any time we write it down, we are using an approximation. Therefore, if we are using a set number of decimal places for our samples, it is impossible to have this value as an accurate sample. This might seem a contrived example, but unless we have an unusual signal such as a square wave, most of the samples will be approximations of the y-axis values of the curve.

The approximation of the readings from the curve is something that can distort the discrete signal away from the continuous signal that it represents. Ideally, we would have as much accuracy as possible, but the more accuracy we have, the more effort is involved and the more space is taken up with digits. Accuracy, like the choice of a sample rate, is always a compromise.

There are limits to the amount of accuracy that we can have. First, there is a limit to how accurately we can measure a point on a curve. Second, even if we could measure very accurately, there is a limit to the number of decimal places that we can use. As an extreme example, we could not easily write down a 10-sample-per-second signal with samples with a million decimal places each. There is also the fact that discrete signals are generally used by computers, and computers can only

work efficiently with a limited number of decimal places. [Strictly speaking, a computer could work with huge numbers of decimal places in software, but their hardware limits them to working *quickly* with a limited number.]

Although there are limits to the amount of accuracy we can have, in practice, we need only as much accuracy as allows us to do a particular task. If we were just drawing pictures of graphs to fit on a piece of paper, we would not need as much accuracy as if we were adding thousands of signals together. Similarly, there would be very few situations where we would actually need to have dozens of digits after the decimal point.

We will look at what happens when we use less accurate readings. As we saw earlier, the samples for our previous "y = sin ((2π * 1t) + 0.5π)" wave example with a sample rate of 20 samples per second, given to 4 decimal places were as so:

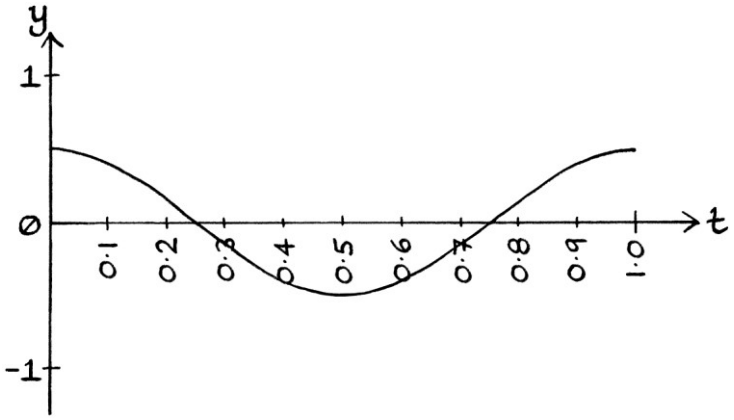
1, 0.9511, 0.8090, 0.5878, 0.3090, 0, −0.3090, −0.5878, −0.8090, −0.9511, −1, −0.9511, −0.8090, −0.5878, −0.3090, 0, 0.3090, 0.5878, 0.8090, 0.9511

Supposing we could not make readings any more accurate than to 1 decimal place (or we used a computer that could only work with values to 1 decimal place), the samples would be as so:

1.0, 1.0, 0.8, 0.6, 0.3, 0, −0.3, −0.6, −0.8, −1.0, −1.0, −1.0, −0.8, −0.6, −0.3, 0, 0.3, 0.6, 0.8, 1.0

In this list, the value of 0.9511 has become rounded up to 1.0. Therefore, the second sample is now identical to the first sample. Similarly, the tenth and twelfth samples are the same as the eleventh sample. This level of accuracy has given the samples slightly misleading values. By using only one decimal place, we are now describing the wave with just a choice of 10 different values.

We will look at a continuous wave that has half the amplitude:
"y = 0.5 sin (2π * 1t) + 0.5π)"

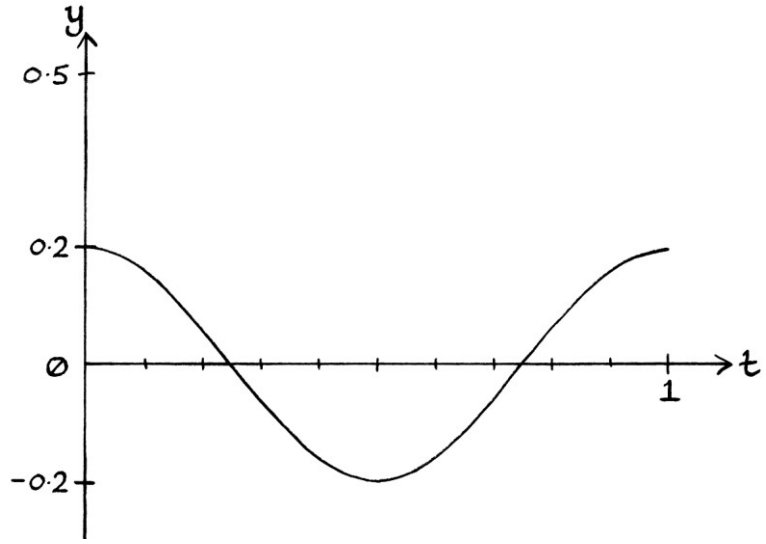


If we take samples at 20 samples per second, measured to 4 decimal places, we will end up with this list:
0.5, 0.4755, 0.4045, 0.2939, 0.1545, 0, −0.1545, −0.2939, −0.4045, −0.4755, −0.5, −0.4755, −0.4045, −0.2939, −0.1545, 0, 0.1545, 0.2939, 0.4045, 0.4755

The same samples rounded up to just one decimal place are as so:
0.5, 0.5, 0.4, 0.3, 0.2, 0, −0.2, −0.3, −0.4, −0.5, −0.5, −0.5, −0.4, −0.3, −0.2, 0, 0.2, 0.3, 0.4, 0.5

The reduced accuracy again means that the samples do not represent the curve particularly well.

We will look at a wave that has an amplitude of 0.2 units:
"y = 0.2 sin (2π * 1t) + 0.5π)"

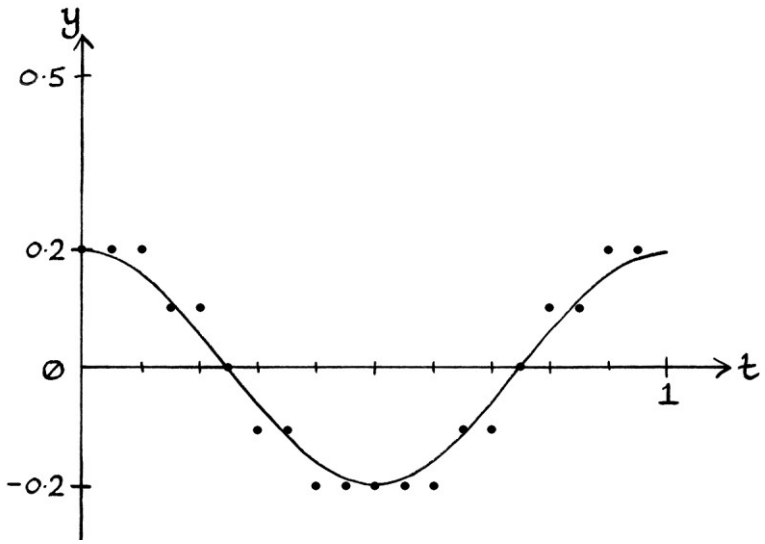


If we take 20 samples per second and give them to 4 decimal places, we end up with these samples:

0.2, 0.1902, 0.1618, 0.1176, 0.0618, 0, −0.0618, −0.1176, −0.1618, −0.1902, −0.2, −0.1902, −0.1618, −0.1176, −0.0618, 0, 0.0618, 0.1176, 0.1618, 0.1902

If the samples only have 1 decimal place, the list appears as so:
0.2, 0.2, 0.2, 0.1, 0.1, 0, −0.1, −0.1, −0.2, −0.2, −0.2, −0.2, −0.2, −0.1, −0.1, 0, 0.1, 0.1, 0.2, 0.2

As we can see, the accuracy we are using is inadequate for the amplitude of the wave. Samples that should be different end up with the same value. There are five consecutive values of −0.2. If we plot the samples on top of the original wave, we can see how inaccurate our list of samples is:



When we have low accuracy, we end up forcing the true values upwards or downwards, and we distort our record of the signal.

Scaling

A solution to having a limited amount of accuracy is to scale a signal so that it fits with the accuracy that we do have. For example, if the possible values that we can use to describe each sample are integers between −9 and +9, and our wave reaches up to +1 and down to −1, then we can scale the wave by 9 so that +1 becomes +9 and −1 becomes −9. In this way, we make the best use of the available values.

[Although this is a contrived example, such an idea would be useful if, for some reason, we could only portray the samples using one decimal digit each. Maybe they were being typed into a very basic spreadsheet or something.]

As an example, we will start with the wave "y = sin (2π * 1t)", a sample rate of 20 samples per second, and an accuracy of 4 decimal places. We would end up with these samples:

0, 0.3090, 0.5878, 0.8090, 0.9511, 1, 0.9511, 0.8090, 0.5878, 0.3090, 0, −0.3090, −0.5878, −0.8090, −0.9511, −1, −0.9511, −0.8090, −0.5878, −0.3090

Now, we will say that we can only use the integers from −9 up to +9 to identify the values of a sample. Every reading from a continuous signal can only be one of the following integers:

−9, −8, −7, −6, −5, −4, −3, −2, −1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

This means that there are only 19 different values to portray all the possible instantaneous amplitudes of our wave.

For our wave "y = sin (2π * 1t)" with a sample rate of 20 samples per second, and an accuracy limited by the results being rounded up to integers between −9 and +9, we end up with these samples:

0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, −1, −1, −1, −1, −1, −1, −1, 0

These samples are obviously useless. We can tell when the curve of the continuous wave was nearer −1, 0 or +1, but otherwise, the samples do not represent the curve of the wave in any useful way. However, we can scale the readings from our continuous wave so that the maximum value becomes +9 and the minimum value becomes −9. As the maximum value of our original wave is 1, and the minimum value is −1, we just multiply all the readings by 9, and then round them up to be integers. In this way, we will be able to have more variety of values in our samples.

We can portray our continuous wave using these values, and we will be able to see its changes.

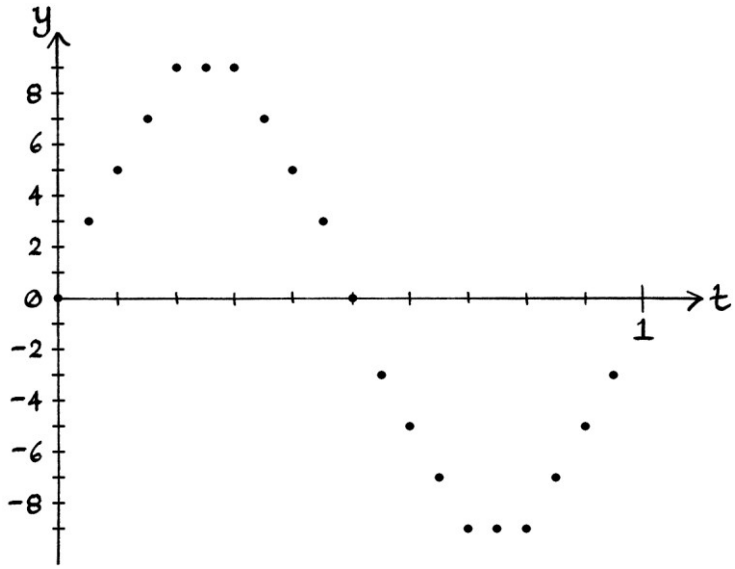
If we scale all our readings by 9, while keeping the results to 4 decimal places, we end up with these samples (to 4 decimal places):

0, 2.7812, 5.2901, 7.2812, 8.5595, 9, 8.5595, 7.2812, 5.2901, 2.7812, 0, −2.7812, −5.2901, −7.2812, −8.5595, −9, −8.5595, −7.2812, −5.2901, −2.7812

If we then turn these samples into integers between −9 up to +9, we end up with this list of samples:

0, 3, 5, 7, 9, 9, 9, 7, 5, 3, 0, −3, −5, −7, −9, −9, −9, −7, −5, −3

These samples look like this on a graph:



This is not an ideal way to sample a continuous wave, but it is the best we can do given the limited number of values that we can use. If we wanted to recreate an approximation of the original samples, we would need to divide each value by 9.

Thoughts on scaling

If our wave were a radio signal that we had received, we may not be as bothered by the *actual* values as we are by their *relative* proportions to each other. In other words, as long as the shape of the wave is kept intact, we would still have all the information that is required to interpret and deal with it. Therefore, for many situations, scaling a signal does not effect the information adversely [as long as we have a wide range of numbers into which the samples are scaled.]

With a radio or sound wave, the actual amplitude that is received is not the same as the amplitude that was originally produced. First, the strength of the radio signal or sound fades as it travels over a distance. Therefore, the received amplitude is less than when it was created. Second, the form of how the sound or radio signal is received might be different from how it was created. For example, a sound wave is a change in air pressure, but a received sound might be in the form of current in an electronic circuit. We cannot really say that the amplitude as measured in pascals matches the amplitude as measured in amps, especially as the equipment to convert one to the other will vary. For received radio and sound waves, amplitude is really a *relative* measure, and not an *exact* measure. Therefore, whether we scale a received signal to have a higher or lower amplitude is really irrelevant as long as we are consistent – this is because we cannot actually know what the original amplitude was. Any record we have of the amplitude of such a received wave is a relative value anyway, so scaling it some more will not make a difference.

Obviously, scaling the amplitude will affect the results of any calculations, in that any calculation that depends on the amplitude will not produce the correct result. Therefore, if the actual instantaneous amplitude of a wave *is* important, scaling it will render any calculations meaningless. When it comes to received radio and sound waves, the actual instantaneous amplitude cannot be important because it would vary depending on how far away the receiver is from the source.

If the actual instantaneous amplitude is important, we can still use scaling for *storing* the signal. Then, if we want to perform any calculations, we can scale it back first.

More scaling

In the previous example, we scaled our wave so that it would make the best use of our range of possible integer values. We can, of course, use other ranges of integers. As an example, we will sample our "y = sin (2π * 1t)" wave with a sample rate of 20 samples per second, and with only integer values from −100 and +100. We will end up with the following samples:

0, 31, 59, 81, 95, 100, 95, 81, 59, 31, 0, −31, −59, −81, −95, −100, −95, −81, −59, −31

Although these are still integers, because we have 201 different possible values, we have a much better range of sample values. [When we used the numbers from −9 to +9, we had just 19 possible different values.]

The resulting list of samples has exactly the same shape and accuracy as if we had used the values −10 to +10, and had an accuracy of 1 decimal place. If we had done that, we would have ended up with this list of samples:

0, 3.1, 5.9, 8.1, 9.5, 10, 9.5, 8.1, 5.9, 3.1, 0, −3.1, −5.9, −8.1, −9.5, −10, −9.5, −8.1, −5.9, −3.1

Possible values

If we allowed each sample to be of any value with any number of decimal places, there would be an infinite number of possible values for each sample.

If we only allowed numbers from −10 to +10 *with one decimal place*, we would have 201 different possible values for each sample.

If we only allowed integers from −100 to +100, we would have the same number of possible values for each sample, which is 201.

If we only allowed integers from −10 to +10, we would have 21 different possible values for each sample.

The general rule is that the more possible values we can choose from to describe each sample, the more accurately we can portray the original continuous signal.

We might want to restrict the samples to a particular range of values to make them compatible with the way that computers store numbers.

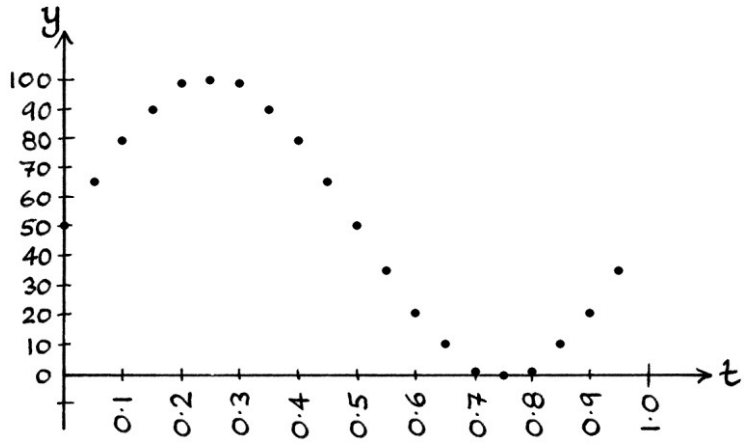
Only positive values

As well as limiting our samples to only integers, we can also limit the samples to only *positive* integers.

We will sample "y = sin (2π * 1t)" with a sample rate of 20 samples per second, and with only integer values from 0 up to 100. This means that as well as scaling our samples, we have to shift them up so that the minimum value is zero and the maximum value is 100. We are effectively scaling the amplitude to 50 units, and giving the wave a mean level of 50 units. The samples will essentially be integers from the wave: "y = 50 + 50 sin (2π * 1t)". The samples are as follows:

50, 65, 79, 90, 98, 100, 98, 90, 79, 65, 50, 35, 21, 10, 2, 0, 2, 10, 21, 35

On a graph, they look like this:



These values still have the same shape as our Sine wave, as they should do, but they are centred around 50 instead of zero. There are 100 different possible values for these samples.

When we constrain the samples in this way, it is mainly useful for storage purposes. We could write out the samples on a piece of paper, or store them in a computer file as integers. We could represent the wave using piles of pebbles on the ground. However, to perform maths on the values, we would need to convert them back into both positive and negative values by re-centring them around zero.

Quantization

The name for the process of taking readings from a continuous signal and adjusting them to fit in with a set of values is called “quantization”. We can think of “quantization” as meaning “turning into a specific quantity”.

Converting readings to fit in with a particular range and type of number is necessary for computers to store and work with the readings. Strictly speaking, any series of readings from a curved continuous signal will likely have been adjusted, whether the values are being used by computers or not. If we are counting in decimal, an obvious example is the value of a third. It cannot be written with a finite number of decimal places, so unless we want to spend the rest of time writing it down, we will have to truncate it at some point. In which case, it will have been quantized. Other examples include irrational numbers, which similarly, cannot be expressed with a finite number of digits. If any reading has been rounded up to a particular number of decimal points, then we can say it has been “quantized”.

By definition, a discrete signal does not have to have quantized samples. A discrete signal can contain numbers such as a third, and it can contain irrational numbers such as “π”. However, in practice, nearly all discrete signals will consist of quantized samples because it makes everything easier.

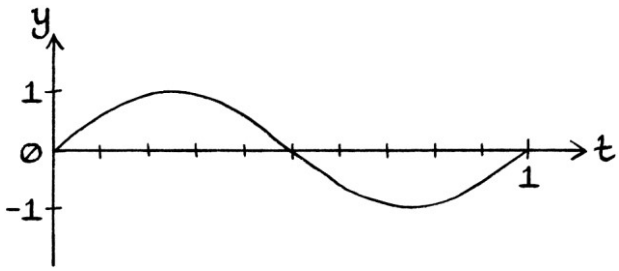
If a discrete signal contains quantized samples, then there will be two reasons that the signal can only be an approximation of a continuous signal. First, we are only taking readings from individual moments in time. Second, all those readings are approximations of the true values.

Problems

So far, when we have scaled a signal, we have made its maximum y-axis value become equal to the maximum possible value for our range and type of numbers. If we are using integers from −100 to +100, and our signal has a minimum value of −1 and a maximum value of +1, then we multiply every y-axis reading by 100. This makes the best use of the numbers that we have to portray the signal. Supposing that the signal fluctuated between −2 and +2, then we could scale all the readings by 50. If the signal fluctuated between −0.1 and +0.1, then we could scale the readings by 1000. This makes the best use of the numbers we have, but suffers from two potential problems that concern “clipping” and “dynamic range”.

Clipping

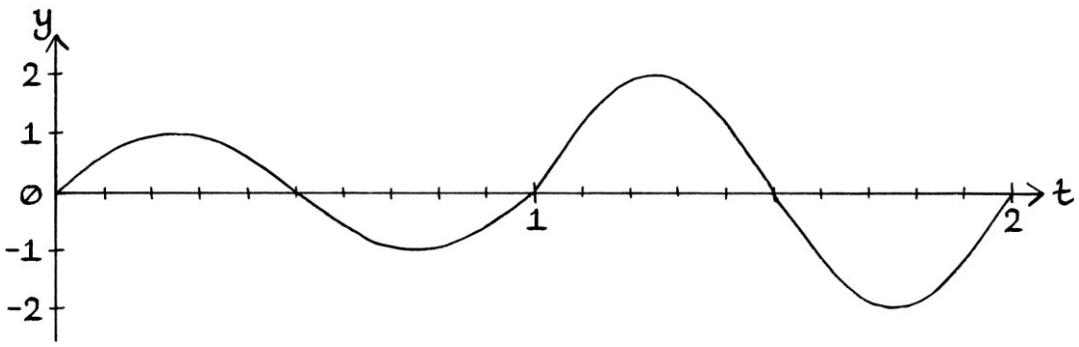
We will imagine that we are making readings from a radio signal while it is being received, such as this:



We will take 20 samples per second from the signal. We will scale all the y-axis readings so that they become integers from −100 to +100. The signal fluctuates from −1 to +1, so we multiply the samples by 100. A table of our readings is as follows. The rightmost column contains our final integer samples:

Time (seconds)	**Actual reading**	**Reading multiplied by 100**	**Turned into an integer between −100 and +100**
0.00	0.0000	0	0
0.05	0.3090	30.90	31
0.10	0.5878	58.78	59
0.15	0.8090	80.90	81
0.20	0.9511	95.11	95
0.25	1.000	100.00	100
0.30	0.9511	95.11	95
0.35	0.8090	80.90	81
0.40	0.5878	58.78	59
0.45	0.3090	30.90	31
0.50	0.0000	0.00	0
0.55	−0.3090	−30.90	−31
0.60	−0.5878	−58.78	−59
0.65	−0.8090	−80.90	−81
0.70	−0.9511	−95.110	−95
0.75	−1.0000	−100.00	−100
0.80	−0.9511	−95.11	−95
0.85	−0.8090	−80.90	−81
0.90	−0.5878	−58.78	−59
0.95	−0.3090	−30.90	−31

Storing the samples in the above way is fine for the signal that we have. However, a problem arises if the instantaneous amplitude of the signal later increases as shown here:



As our scaling is already at the maximum possible value, when the signal's instantaneous amplitude increases, we will end up trying to have numbers higher than +100 and lower than −100. As we only have the integers from −100 to +100 to use, this means that anything over +100 will end up as +100, and anything under −100 will end up as −100.

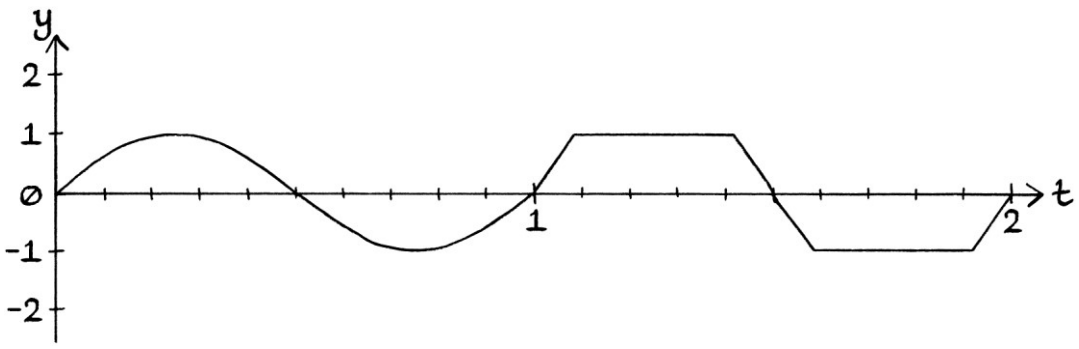
The next 20 samples will be as so:

Time (seconds)	**Actual reading**	**Reading multiplied by 100**	**Turned into an integer between −100 and +100**
1.00	0.0000	0.000	0
1.05	0.6180	61.80	62
1.10	1.1756	117.56	100
1.15	1.6180	161.80	100
1.20	1.9021	190.21	100
1.25	2.0000	200.00	100
1.30	1.9021	190.21	100
1.35	1.6180	161.80	100
1.40	1.1756	117.56	100
1.45	0.6180	61.80	62
1.50	0.0000	0.0000	0
1.55	−0.6180	−61.80	−62
1.60	−1.1756	−117.56	−100
1.65	−1.6180	−161.80	−100
1.70	−1.9021	−190.21	−100
1.75	−2.0000	−200.00	−100

1.80	−1.9021	−190.21	−100
1.85	−1.6180	−161.80	−100
1.90	−1.1756	−117.56	−100
1.95	−0.6180	−61.80	−62

Many of the values for the second part of the signal are too high or too low, so end up being limited to the maximum and minimum values that we can have: +100 and −100. This problem is called “clipping”. The signal’s height has been “clipped” in a similar way to how a hedge that has grown too high becomes clipped. Clipping is the problem that arises when a signal’s instantaneous amplitude exceeds the boundaries of the number range we are using to record it.

Our collected samples are representing the following signal, instead of the one we intended:



The way to avoid clipping is to try to make room for any likely future maximum and minimum values before the reading process starts. This will reduce the number of possible integers available to store the signal, but with the benefit that none of the future signals will become too high or too low to be recorded. We could have scaled the signal so it reached −50 and +50 to start with, in which case the later part of the signal would fit in with the sampling process.

Our sampling method failed in this case because we did not scale the wave to take into account future higher peaks. This problem occurs in the real world when someone sings too loudly into a microphone. The sound becomes distorted because the microphone and equipment can only cope with a certain range of volume. Anything louder, as in anything outside that range, ends up being given the same top value. With audio recording equipment, there is often a “gain” adjustment that essentially reduces the scale of the instantaneous amplitude of the sound so that it will fit in with the abilities of the recording system.

Another problem with storing any given signal to fit *exactly* within the range of possible values is that any calculations on the signal will push the values out of the range. For example, we would not be able to add anything to each sample of a signal if we had scaled it to use the maximum possible value. For this reason, it can be better to use a scaling method that reaches to the maximum or minimum just as a way of *storing* the data. If we want to perform calculations, it is better to rescale the data into a system that gives us more leeway.

Clipping is not a problem that is exclusive to discrete signals. It can also occur when recording continuous signals.

A consequence of clipping is that any analysis of the clipped signal will find a different range of constituent waves from those found by analysing the original signal.

Dynamic Range

We will imagine that we were using scaling to fit two different signals into the integer numbers from −100 to +100. We will say that the first signal is:
"y = 1 sin (2π * 1t)"
... and the second signal is:
"y = 100 sin (2π * 1t)"

The first signal fluctuates from −1 to +1; the second signal fluctuates from −100 to +100. If we scaled each signal *separately* to fit the available range of numbers, then we would multiply the first signal's readings by 100 and turn them into integers. We can turn the second signal's readings directly into integers.

A problem arises when we want to store both signals together. If the second signal is stored in the same way as the first, the readings will be multiplied by 100 and then turned into integers. This will make most of the second signal's values above +100 or under −100, in which case, they will end up as being +100 or −100 because we cannot use numbers outside the +100 to −100 range. On the other hand, if the first signal is stored in the same way as the second, the readings will be turned directly into integers, so will all have values of +1, 0, or −1. These problems mean that our range of numbers is not sufficient to encode both signals at the same time.

One solution is to use a larger range of numbers, for example, integers from +1000 to −1000. With this range, we could multiply the readings from both signals by 10. The first signal's readings would end up as integers from +10 to −10, and the

second signal's readings would end up as integers from +1000 to −1000. The first signal's readings would still not have much variety, but would be better than before. The second signal's readings would not go out of range.

The ability for a range of numbers to represent a wide range of readings adequately is called "dynamic range". Our system of integers from +100 to −100 has a relatively "low dynamic range" – it is incapable of adequately representing readings if the readings vary a lot in size, and if the small readings are as important as the high readings. A system of integers from +1000 to −1000 would have a relatively "high dynamic range". It would be much better at representing a variety of sizes of readings where the small readings are as important as the high readings.

One solution to increase the dynamic range of a system of numbers is to use a logarithmic scale. In this way, the larger values become scaled at an ever-decreasing rate. This distorts the readings, but means that very high values can be stored alongside very small readings. The readings would need to be decoded back to their proper non-logarithmic values before any calculations were performed on them.

The concept of dynamic range is noticeable in certain situations in real life. Recording equipment can be set up to record a particular level of sound, but will suffer clipping if the sound becomes too loud, or will hear nothing if the sound becomes too quiet. It would be rare to find recording equipment that could record the sound of a mouse's footsteps and a jet engine without needing adjustment between recordings. The dynamic range would not be sufficient.

As a more abstract example, the human eye can see reasonably well in bright light or in dim light. It can adapt to the brightness. Another way of thinking about this is that it can scale its "brightness reception". The limits of the dynamic range of the human eye become obvious when we are confronted with a scene that is both bright and dim. For example, if a person is standing in front of us with the sun shining behind them, they will appear as a silhouette, and we will be unable to see their features. If a cloud should pass in front of the sun, or we block out the background around the person with our hands, we would be able to see the person perfectly well. The human eye can cope with a range of different levels of brightness well, but it cannot cope with significantly different levels *at the same time*.

Another example of the problems of achieving a good dynamic range is how weighing scales tend to be able to weigh light things to great accuracy, or heavy things to much less accuracy. Scales that have a high dynamic range and can weigh very heavy things and very light things equally well are rare and expensive.

Non-digital thermometers generally either measure within a small range of temperatures accurately, or measure a high range of temperatures inaccurately.

The average micrometer can measure distances of up to 15 centimetres or so to an accuracy of a fraction of a millimetre, but measuring devices that can measure distances longer than a metre have much lower accuracy.

Vocabulary so far

We will look at some of the vocabulary we have learnt so far in this chapter.

Sample

A "sample" is a value that is, or could be, a reading from a continuous signal. It is the same as a y-axis value from a graph, and therefore, it is the same as a measurement of the instantaneous amplitude of a signal at a particular time. A sample does not have to have a limited number of decimal places – we could have an irrational number as a sample. In practice though, most of the time, samples will be given to a particular level of accuracy.

Sampling

Sampling is the process of creating samples by reading values from a signal.

Sample rate

The "sample rate" is the number of readings taken from a continuous signal every second – it is the number of samples that appear in a discrete signal every second. The sample rate is sometimes called the "sampling frequency", in which case, we can think of it as meaning the number of times per second that a sample is taken. The sample rate is measured in "samples per second", which is abbreviated to

"sps". For people who like to use the term "sampling frequency", the sample rate is sometimes given in hertz. Therefore, a sample rate of 4 samples per second is the same as a sampling frequency of 4 hertz. It might seem confusing to use hertz to refer to both the frequency of a wave and the sample rate that is used to sample it, but the context will usually clarify what is meant. The sample rate is sometimes represented by the symbol "f_s", where the "f" stands for "frequency" and the subscripted "s" stands for "sampling". In this way, it means "the frequency of sampling" or "sampling frequency". Abbreviations such as "f_s" can be confusing when you first see them, but you will get used to them quickly.

Sampling period

The period of time between samples is called the "sampling period". The sampling period is the reciprocal of the sample rate. In other words, 1 divided by the sample rate is the sampling period, and 1 divided by the sampling period is the sample rate. In the same way that we can use the letter "T" to represent the period of a wave, so can we use "T_s" to represent the sampling period. This is consistent with using "f_s" to represent the sampling frequency (the sample rate).

Quantization

Quantization is the process of adjusting samples so that they fit within the range and precision of numbers that we are able to use. We might be constrained to use a certain range and precision of numbers by the computer architecture we are using. Quantization involves scaling samples to fit into a particular range, and rounding the values up to a certain number of decimal places (or binary places if we are working in binary).

Clipping

Clipping is the problem that occurs when the value of a sample is too high to fit within the range of numbers that we are using to store our samples. Depending on how the sampling process is being executed, if a value is higher than the maximum possible value, it might become set to the maximum value. Alternatively, it might roll around to become a low value in the same way that an old-fashioned milometer [odometer] in a vehicle rolls around from 999999 kilometres to 000000 kilometres. Clipping means that the record of the signal will be incorrect.

Dynamic range

Dynamic range is a relative term that refers to how well the range of numbers we are using to store our samples can record both small values and large values at the same time.

Digital signals

There are three definitions of "digital signal" that are in general use.

Signal processing definition

As we know, a *discrete* signal is made up of individual values that are, or could have been, taken from a continuous signal at equally spaced moments in time. A discrete signal does not necessarily need to have the values given to a specific accuracy. There does not need to be a limit to the number of decimal places for each value. Therefore, we could have a discrete signal that contained irrational numbers or fractions with infinite numbers of decimal places. For example, this list of numbers makes a perfectly valid discrete signal:
0.1
1/3
2
π
7
9
$\sqrt{2}$
e
4/17

In the academic world of signal processing, a "digital signal" is a discrete signal where every sample is given to a particular level of accuracy. In other words, the samples are rounded up to a particular number of decimal places (or binary places, as we shall see in Chapter 40). We could say that there is a limited number of *digits* after the decimal point.

If the above discrete signal containing irrational numbers were to be a digital signal, it might appear as so:

0.1000
0.3333
2.0000
3.1416
7.0000
9.0000
1.4142
2.7183
0.2353

In this sense, a digital signal is a discrete signal with a constrained accuracy. A digital signal is one that has been both sampled *and quantized*. Some people describe such an idea by saying that a digital signal is discrete in both amplitude and time – all the values have a particular precision and come from particular moments in time. Such a digital signal does not have to have any relationship with digital electronics – if the values were all integers, we could portray the signal with groups of pebbles, and it would still be discrete and digital.

For the "signal processing" definition, we can say that all digital signals are discrete signals, but not all discrete signals are digital signals. "Signal processing" people often confuse the terms "discrete" and "digital" because, within this definition, nearly all discrete signals will also be digital signals, so most of the time, there is no difference.

Digital electronics definition

In the world of electronics, a digital signal is one that is being used in digital electronics, and that switches between particular values. The square waves from the "automation" section of Chapter 35 were digital signals in this sense. A digital signal might consist of voltages that switch between 0 volts and 5 volts to convey the binary digits 0 and 1. It might be a theoretical signal that switches between 0 units and 1 unit to do the same. It might switch between several states.

Such digital signals might or might not also be continuous. We could have a continuous square wave that switched between 0 and 1 units – we could zoom into the graph forever and we would always see a value that was either 0 or 1. There

would never be a time when there was no value. Such a square wave would be a "digital signal" in this sense.

To complicate matters, we could create a discrete version of that continuous 0-or-1 square wave, in which case it would switch between particular values, but exist only at particular moments in time. It would be "digital" in the digital electronics sense, and also "discrete and digital" in the signal processing sense. For digital-electronics digital signals that switch between set values that are given to a particular number of decimal places, a discrete version of such a signal will also be a digital signal in the signal processing sense. If on the other hand, it should switch between irrational numbers or particular non-integers, which is unlikely in practice, it would just be a discrete signal, and not digital in the signal processing sense.

In this sense of "digital signal", we can say that some digital signals are discrete signals, and some are not. Most digital signals, in this sense, when made discrete, would be digital in the signal processing sense too.

Although one might think that a "signal processing" digital signal is a "digital electronics" digital signal with a great number of switching levels, the nuance is different. A digital electronics digital signal might be continuous or discrete, but its main purpose is to *switch* between values. A signal-processing digital signal will always be discrete, and might be used for many different purposes.

General definition

Outside of signal processing and digital electronics, any signal involved in digital electronics might be described vaguely as being a "digital signal". For example, people might say that digital television broadcasting uses digital signals, but without specifying exactly which aspect uses digital signals. This general meaning of "digital" is part of what confuses people when understanding the meaning of "digital signal" in signal processing – people might assume, incorrectly, that because a discrete signal was being used in digital electronics, it was a digital signal.

Thoughts

Given that the term “digital signal processing” is a form of signal processing, the “digital” in the name refers to signals that are discrete and digital, as opposed to those that are switching between particular values, or those that are in any way related to digital electronics. Therefore, if we were being pedantic, we could perform digital signal processing with piles of pebbles. In practice, most digital signal processing would be performed with digital electronics, which is why everyone’s definitions of “digital” are becoming more muddled.

Thoughts on discrete signals

For relatively high sample rates and a good range of values, discrete signals are analogous to continuous waves, and basic maths on them will produce the same results.

If we were performing maths on a continuous signal, we would essentially be performing maths on every single point, and there are really an infinite number of points. When we perform maths on a discrete signal, we are performing maths on the values that we are given. There are a set number of values. The accuracy of any results (being how representative the results are, compared with if we had done the maths with the continuous version of the signal) is limited to the accuracy of the samples and how many samples we have per second.

Chapter 40: Binary and hexadecimal

Before we continue looking at discrete signals, we will look at how computers store and use numbers. This will enable us to connect everything so far in this book with the world of computers. For a computer to use a number, the number must fit in with how computers count. For a computer to deal with a discrete signal, each sample must be one that a computer can use.

This chapter will be most useful if you want to write computer programs that deal with waves, or if you want to understand the limitations of programs that do. For the purposes of learning about waves, it does not particularly matter if you do not understand or remember everything in this chapter. However, it will be helpful to read it anyway just so you can have a better understanding of discrete signals.

Binary

Computers count in binary, and therefore, their methods of storing numbers are based around binary. Binary is a counting system where every digit is either zero or one. Normally, we count in decimal, where every digit is 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9. As we know, the progression of counting in decimal is as so:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 and so on.

The basic rule of decimal is that when the rightmost digit is going to pass 9, we reset the digit to 0, and add 1 to the digit to the left. If *that* digit is going to pass 9, then we set that to 0, and add 1 to the digit to the left, and so on.

Counting in binary works in a similar way, but as binary has only two different digits (zero and one), counting progresses as so:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100 and so on.

When the rightmost digit of a number is going to be above 1, that digit becomes 0, and the digit to the left is raised by 1. If *that* digit is going to be above 1, then it becomes 1, and the digit to the left is raised by 1, and so on. Counting in binary is based on identical ideas as counting in decimal.

When we have a number in *decimal*, we can think of each digit as being in a "column" representing, from right to left, ones, tens, hundreds, thousands, ten thousands and so on. Each column is ten times the one before it. For example, the number "245" has 2 in the hundreds column, 4 in the tens column, and 5 in the ones column. The number "245" is really:
2 * 100, added to 4 * 10, added to 5 * 1

We can think in the same way about binary. From right to left in binary, the columns are ones, twos, fours, eights, sixteens, thirty-twos, sixty-fours, one hundred and twenty eights, and so on. Each column is twice the one before it. As an example, the binary number "1100" has 1 in the eights column, 1 in the fours column, 0 in the twos column, and 0 in the ones column. The binary number "1100" is really:
1 * 8
... added to:
1 * 4
... added to:
0 * 2
... added to:
0 * 1

The decimal numbers 0 to 15 and their binary equivalents are as so:

Decimal	**Binary**
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

A binary digit is called a “bit”. Therefore, we can say that the number “111001” contains 6 bits. This is the same as saying that the number “111001” has 6 digits.

Electronics can use the binary counting system because the system works well with switches, whether they are real or in circuitry. We can portray a binary number by having a series of switches set to on or off. For example, we could portray the number “1010” by having four switches, with the first and third switched on, and the second and fourth switched off. In digital electronics, we can portray binary numbers in a similar way by having parallel lines with different voltages. We can have a voltage being set to, say, 5 volts to represent a one, and a voltage set to 0 volts to represent a zero. By using just two different states, we avoid any confusion because a state represents either a one or a zero. There is nothing between the states.

Groups of bits

Although digital electronics can use binary numbers of any chosen length, when it comes to computers, binary numbers are grouped into particularly sized sets of digits. Computer processors are designed to operate most quickly when dealing with a particular sized group of bits. The size of the group depends on the processor. [Away from computers, you can use any grouping that you want.]

[The following size names are those used by Intel when referring to 80x86 and 64-bit computer processors. These are common modern definitions of the terms, but some other, especially older, computer architectures define things differently. If you ask someone how big a “word” is, an Intel assembly language programmer will say 16 bits. Someone else might say, “It depends...”]

Bytes

A “byte” is a group of 8 bits. In other words, a byte is a binary number that is 8 digits long. To put this another way, a number that is stored as a byte will always be 8 bits long, even if the higher bits, or even all the bits, are zero. When a computer reads one byte, it will always read 8 bits. It will never read fewer or more. Given that, if we are writing out a binary number as a byte, it makes sense to include any preceding zeroes. If a decimal number is converted to a byte, the result will end up as an 8-bit number.

As a byte is always 8 bits long, it means that the binary numbers that can be contained within a byte are as follows:

Binary number	**Decimal Equivalent**
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
00000101	5
00000110	6
00000111	7
00001000	8
00001001	9
00001010	10
00001011	11
00001100	12
00001101	13
00001110	14
00001111	15
00010000	16
00010001	17
... and so on until:	
11111100	252
11111101	253
11111110	254
11111111	255

The highest number that can be stored as a byte is 11111111 in binary, which is 255 in decimal. As we start at 00000000 in binary, this means that we can actually store 256 different values as a byte. [If we were using a byte to indicate the position of something in a list, it makes sense to treat the first item in the list as item 0, instead of item 1, as that means we can distinguish between 256 different items instead of 255.]

Supposing we were dealing with binary numbers on a piece of paper, if we were to add 1 to the binary number 11111111, we would end up with the number 100000000, which is 9 digits long. However, if we are dealing with binary numbers on a computer, and dealing with bytes, if we added 1 to 11111111, we would end

up with 00000000, which is zero. This is because we cannot have more than 8 digits in a byte. The 1 that would be the ninth digit becomes lost, and the whole number rolls over to zero. This is the same effect as when a milometer [odometer] in a vehicle, or a meter on a fuel pump, reaches its maximum value and rolls over to zero.

By only using numbers with a set number of digits, computers can work more efficiently. A byte is generally the smallest grouping of bits with which a computer processor can work quickly. It is usually possible for a processor to read individual bits or half bytes, but to do so, it would first have to load the number into a byte or larger number grouping, and the process is slower than dealing with bytes.

Half bytes

A "half byte" is a 4-bit number – it is a grouping of 4 bits. A half byte is always 4 bits long. It is half the length of a byte. A half byte is sometimes called a "nibble". The binary numbers that can fit into a half byte are as follows:

Binary number	**Decimal Equivalent**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

A half byte can contain the binary digits 0000 to 1111. The maximum number it can hold is the binary equivalent of the decimal number 15. As it can hold the number 0, it is capable of holding 16 different values.

On Intel 32-bit and 64-bit processors, it is possible to access a half byte easily, but the number must first be placed into a byte or a larger number type. It is slower to use half bytes than it is to use bytes.

Words

A "word" is a grouping of 16 bits as one entity. A word is twice as long as a byte. The binary numbers that can be contained in a word are as follows:

Binary number	**Decimal Equivalent**
0000000000000000	0
0000000000000001	1
0000000000000010	2
0000000000000011	3
0000000000000100	4
0000000000000101	5
0000000000000110	6
0000000000000111	7
0000000000001000	8
0000000000001001	9
0000000000001010	10
0000000000001011	11
... and so on until:	
1111111111111100	65,532
1111111111111101	65,533
1111111111111110	65,534
1111111111111111	65,535

A word can hold values from 0 up to 65,535. This means that it can hold 65,536 different values.

[Note that a few people use the term "word" to describe *any* grouping of bits. You might see this in older books. Intel have been using the term "word" to mean a grouping of 16 bits for at least thirty years.]

The rule for finding out how many different values can be stored in a binary number, based on the bit size, is to raise 2 to the power of the bit size. For example, a 16-bit binary number can hold: 2^{16} = 65,536 different values. Note, however, that this is not the maximum value because we start at zero. The maximum value is always 1 less than the number of available values. For example, an 8-bit binary number can hold 2^8 = 256 values, but the maximum value will be $2^8 - 1 = 255$.

Computer processors can be categorised according to the maximum number of bits they can deal with in one go. For example, if we had a processor that could only operate with bytes (8-bit numbers), it would be referred to as an "8-bit processor". The longest number an 8-bit processor can read in one go is 8 bits long. Generally, an 8-bit processor will be fastest at dealing with 8-bit numbers. It will be able to read 8 bits in one go, store 8 bits in one go, and perform calculations on 8 bits in one go. If a processor is a 16-bit processor, the largest and optimal size of number it can deal with is 16 bits long. The processor will be best at operating with 16-bit numbers. It will be able to read and work with numbers most efficiently if they are 16 bits long. Note that a 16-bit processor will still be able to work with shorter numbers, but it will take slightly longer to do so. It will also be able to work with longer numbers, but it will require being told how to do so via a computer program, and it will take much longer. [This is analogous to using a calculator that can only count up to one hundred to perform maths with numbers over one hundred. We would have to split calculations into parts, and write down partial calculations on a piece of paper. We could do it, but it would be more effort.]

A common mistake that programmers used to make was to use of one of the binary number types, either to count with or as an indicator to a place in a list, and to forget about the limits of the number type they were using. For example, if we are using a 16-bit word to count the number of letters in a piece of text, and we count more than 65,535 letters, the number will roll around to zero, and carry on from there. This means that the total will be wrong. Similarly, if we had a list of names and we used a 16-bit word to identify the different entries, with the first entry being entry 0, we would only be able to have 65,536 different entries. If we tried to access the 65,537th entry, we would end up accessing the very first entry [entry 0] instead because we cannot count that high with 16-bit numbers. As modern computers more commonly use 64-bit processors, where the highest number is considerably larger, this is now a rarer problem. However, it still happens if the programmer, for some reason, chooses a smaller number type.

Doublewords

A “doubleword” or “dword” is 32 bits long. A dword can hold the binary numbers from:
00000000000000000000000000000000
... to:
11111111111111111111111111111111
... in binary.

The number of different values that can be expressed by a dword is:
2^{32} = 4,294,967,296

The highest number that can be expressed by a dword is $2^{32} - 1$ = 4,294,967,295

The binary numbers that can be held in a dword are as follows:

Binary number	**Decimal Equivalent**
00000000000000000000000000000000	0
00000000000000000000000000000001	1
00000000000000000000000000000010	2
00000000000000000000000000000011	3
... and so on until:	
11111111111111111111111111111100	4,294,967,292
11111111111111111111111111111101	4,294,967,293
11111111111111111111111111111110	4,294,967,294
11111111111111111111111111111111	4,294,967,295

An Intel 32-bit 80x86 processor works most efficiently with 32-bit numbers. It is fastest when reading, working on, and storing 32-bit numbers. Although the processor has commands to deal with words (16-bit numbers), bytes (8-bit numbers) and half bytes (4-bit numbers), and it can test if individual bits are one or zero, it is most efficient when handling 32-bit numbers. [Other 32-bit processors are likely to be similar.] If we wanted a 32-bit processor to work with a 64-bit number, we would have to do it in a slower, more contrived way by splitting the number into two 32-bit numbers and then using slightly more awkward maths.

Quadwords

A “quadword” or “qword” is 64 bits long. A qword can hold the binary numbers from:
0000000000000000000000000000000000000000000000000000000000000000
... to:
1111111111111111111111111111111111111111111111111111111111111111
... in binary.

The number of values that can be expressed by a qword is:
2^{64} = 18,446,744,073,709,551,616

The highest number that can be expressed by a qword is:
$2^{64} - 1$ = 18,446,744,073,709,551,615
This is roughly 18.4 million million million, which is $18.4 * 10^{18}$.

A 64-bit processor works most efficiently with 64-bit binary numbers. Intel 64-bit processors have commands that work with 64-bit qwords, 32-bit dwords, 16-bit words, and 8-bit bytes, and can test whether individual bits are 1 or 0, but they work best with 64-bit qwords. [Intel 64-bit processors can work with 4-bit half bytes but those half bytes must already be within a byte, word, dword or qword.]

Although a 64-bit processor might seem best because it works with large numbers, a consequence is that the storage of numbers requires more space. If we wanted to store the number “3” as a 64-bit qword, it would require 64 bits. If we wanted to store it as a 32-bit dword, it would require 32 bits. If we wanted to store it as a 16-bit word, it would require 16 bits. If we wanted to store it as an 8-bit byte, it would require just 8 bits. Storing smaller numbers as 64-bit values uses 8 times as much space as using 8-bit bytes. On the other hand, we cannot store a number larger than 255 in a byte. [If we wanted to store the number “3” with the minimum size possible, it would require only 2 bits because the decimal number “3” is “11” in binary. However, it is rare that a computer processor would have a grouping that was only two bits long. We could write the 2-bit number on a piece of paper, though, as then we would be unconstrained by the groupings of bits.]

Double quadwords

It is also possible to have a 128-bit number, which is called a “double quadword” or “dqword”. Such numbers work in the same way as other numbers, but just contain more bits. To keep things simple, in this book, we will focus on bytes, words, dwords and qwords.

Programming

In higher-level programming languages such as C, C++, Java, Python and so on, the actual bit group sizes are usually hidden from the programmer. An advantage of this is that the programmer can use the same code on different types of processors. A disadvantage is that useful underlying aspects of the processor are hidden from the programmer, so software might not run as efficiently as it could. It is possible to be reasonably good at programming, but without knowing anything about bytes, words, dwords, qwords, or even binary.

Converting binary to decimal

It is rare that you will ever need to convert long binary numbers to decimal. You are much more likely to need to convert binary numbers to hexadecimal, which we will look at later in this chapter. It is a lot of effort to convert any remotely long binary number directly to decimal, and it is much easier to convert it to hexadecimal first, and then to decimal. Therefore, there is not much point in learning how to convert binary to decimal. Having said that, if you want to become proficient in using binary and hexadecimal, it is very helpful to know every 4-bit binary number’s decimal equivalent. Learning them is easy with practice, but whether you *need* to learn them depends on what you want to do. If you frequently use binary, you will end up learning the numbers anyway from seeing them so often.

The numbers are as follows:

Binary	Decimal	How to remember
0000	0	This is easy to remember.
0001	1	This is easy to remember.
0010	2	This is a 1 in the twos column.
0011	3	This is two 1s to the right.
0100	4	This is a 1 in the fours column.
0101	5	This is 4 + 1
0110	6	This is two 1s in the middle.
0111	7	This is three 1s, all to the right.
1000	8	This is a 1 in the eights column.
1001	9	This is 1 more than 8.
1010	10	This is 8 + 2
1011	11	This is 8 + 3
1100	12	This is 8 + 4, or two 1s to the left.
1101	13	This is 12 + 1
1110	14	This is three 1s, all to the left.
1111	15	This is four 1s.

If you can remember 0100 (four), 0110 (six), and 1100 (twelve), then it is easy to calculate the others in your head. When you are completely used to binary, it will become irrelevant to you whether a 4-bit binary number is written in binary or its decimal equivalent. You will see them as the same thing.

We will see how to convert from binary to decimal and back in the “Binary and hexadecimal” section later in this chapter.

Hexadecimal

Decimal is a counting system based around the number ten. Binary is a counting system based around the number two. Now we will look at hexadecimal, which is a number system that is based around the number sixteen. Each digit of a hexadecimal number can have 16 different values. The first ten, representing the decimal numbers from 0 to 9, are also the numbers 0 to 9. The next digits are “a”, “b”, “c”, “d”, “e” and “f”. Therefore, the letter “a” represents the decimal number ten, but in the form of only one digit. The letter “b” is the digit that represents the decimal number eleven. The letter “c” is the digit that represents the decimal number twelve, and so on.

The term “hexadecimal” is often abbreviated to the word “hex”.

The hexadecimal numbers from the equivalent of the decimal number 0 up to the decimal number 15 are as so:

Hexadecimal number	**Equivalent decimal number**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
a	10
b	11
c	12
d	13
e	14
f	15

When we count in *decimal*, we have a ones column, a tens column, a hundreds column, a thousands column and so on. Each new column is ten times the previous one. When we count in binary, we have a ones column, a twos column, a fours column, an eights column and so on. Each new column is twice the previous one. When we count in hexadecimal, we have a ones column, a sixteens column, a 256s column, a 4096s column and so on. Each new column is 16 times the previous one. When the ones column in hexadecimal gets past "f", we set the ones column to zero, and add 1 to the sixteens column. If that makes the sixteens column go past "f", then that is set to zero, and 1 is added to the 256s column and so on.

Hexadecimal is easier to use than to explain. Here is a long list of hexadecimal numbers and their decimal equivalents so we can see how hexadecimal works. [It is worth reading this, but it is not worth trying to remember which hexadecimal value is equal to which decimal value]:

Hexadecimal number	**Equivalent decimal number**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
a	10
b	11
c	12
d	13
e	14
f	15
10	16 [This is 1 times 16]
11	17 [This is 1 times 16, with 1 added on]
12	18
13	19
14	20
15	21
16	22

Hexadecimal number	**Equivalent decimal number**
17	23
18	24
19	25
1a	26
1b	27
1c	28
1d	29
1e	30
1f	31
20	32 [This is 2 times 16]
21	33
22	34
23	35
24	36
25	37
26	38
27	39
28	40
29	41
2a	42
2b	43
2c	44
2d	45
2e	46
2f	47
30	48 [This is 3 times 16]
31	49
... and so on, until:	
99	153
9a	154
9b	155
9c	156
9d	157
9e	158
9f	159
a0	160 [This is ten times 16]

Hexadecimal number	Equivalent decimal number
a1	161
a2	162
... and so on, until:	
f9	249
fa	250
fb	251
fc	252
fd	253
fe	254
ff	255
100	256 [This is 16 times 16]
101	257
102	258

Notation

Using decimal, binary and hexadecimal numbers together can be confusing if we do not identify in which number system a sequence of digits belongs. For example, without any context, the sequence of digits “111” could mean the decimal number “one hundred and eleven”, the binary number equivalent to the decimal number “seven”, or the hexadecimal number equivalent to the decimal number “273”.

There are three main ways to avoid such confusion. The first way is the most common, and is used in higher-level programming languages such as C. For this way, a hexadecimal number is prefixed with “0x”, a binary number is prefixed with “0b”, and a decimal number is left alone. Therefore, the hexadecimal number 111 would be written as so:
0x111

The binary number 111 would be written as so:
0b111

The decimal number 111 would be written as:
111

The second way is used in assembly language programming, specifically in programming for Intel and AMD processors, but probably for other processors too. This involves putting the letter “h” after a hexadecimal number, and if that number starts with a letter, putting a zero in front. [The zero stops it being confused with the name of a variable.] If we have a binary number, we put the letter “b” at the end. If we have a decimal number, we put the letter “d” at the end. Therefore, the hexadecimal number 111 is written as:
111h

The hexadecimal number “ff” becomes:
0ffh
... because the number starts with a letter, so we prefix it with a zero.

The binary number 111 becomes:
111b
[This cannot be confused with the hex number “111b” because, if we were using this system, all hex bytes would have an “h” after them. The hexadecimal number “111b” would be written as “111bh”.]

The decimal number 111 becomes:
111d
[Again, this cannot be confused with the hex number “111d” because the hex number would be “111dh”.]

The third way of distinguishing between the types of numbers is to put a subscript on each number to indicate the base. For example, 1011_2 for binary, $f5_{16}$ for hex, 23_{10} for decimal. The downsides to this method are:

- It is an effort to type subscripted text on computers.
- Sometimes the subscript can be misread as part of the number.
- Copying and pasting subscripted text sometimes loses the attribute of being subscripted, thus turning, say, 1011_2 into 10112.

Most people use the “0x” and “0b” method. In this book, I will use the 0x method for hex numbers, and rely on the context for binary and decimal numbers.

The first few hexadecimal numbers written with the “0x” prefix are as so:

0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xa
0xb
0xc
0xd
0xe
0xf
0x10
0x11
... and so on.

Sometimes, if the context is clear, it is tidier and easier to have the values without the prefix. For example, if we know that a list of values is in hexadecimal, then there is no need to distinguish each value with a prefix.

Binary and hexadecimal

There is a connection between binary and hexadecimal that makes hexadecimal useful. A group of 4 binary bits can be represented by exactly one hexadecimal digit. Therefore, instead of dealing with very long sequences of binary digits, we can group them into fours, and deal with shorter sequences of hexadecimal digits.

The way that hexadecimal can be used to represent binary numbers more succinctly is clearer in the following table.

Decimal	Binary	Hexadecimal
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xa
11	1011	0xb
12	1100	0xc
13	1101	0xd
14	1110	0xe
15	1111	0xf

When we convert from binary to hexadecimal, we turn 4 binary digits into just 1 hexadecimal digit.

The usefulness of hexadecimal is more obvious if we have a long sequence of binary such as this:
10110010000011010011111110100110

To convert this into hexadecimal, we first split it into 4-bit sections:
1011 0010 0000 1101 0011 1111 1010 0110

Then, we convert each 4-bit section into a hexadecimal digit:
b 2 0 d 3 f a 6

Then, we join up the hexadecimal digits to create the final value:
0xb20d3fa6

We have turned a binary number with 32 digits into a hexadecimal number with 8 digits. To put this another way, we have turned a 32-bit binary number into a 32-bit hexadecimal number. By calling the hex number "32-bit", despite it containing 8 digits, we are still acknowledging that it represents 32 binary bits. We are really just using hexadecimal to make long binary numbers more palatable. Hexadecimal is easier to read and remember than binary, but its ultimate meaning is the same.

Converting between hex and decimal

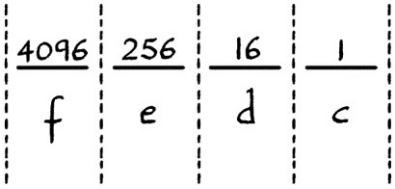
Depending on what it is we are trying to do, generally, there is less need to convert from hexadecimal to decimal and back, than there is to convert from hexadecimal to binary and back. In most situations, it is easier to remain in the world of hexadecimal than it is to swap to decimal and back. It is easy to know if one hexadecimal number is higher than another, which is what is usually needed. If you need to convert a long hexadecimal number to decimal, it is easiest to use a calculator to do it, than to bother trying to do it in your head. However, even if you always intend to use calculators, it is good to know the hexadecimal values 0x0 to 0xf by heart.

Converting between decimal and hexadecimal is not in any way difficult, but it takes more time than using a hex calculator. Before we do the conversion, we first have to be aware of the digit columns for a hexadecimal number, which from right to left are:

- 1s
- 16s
- 256s
- 4096s
- 65,536s

... and so on. Each column is 16 times the one before it.

If we had the number 0xfedc, it would have 0xf in the 4096s column, 0xe in the 256s column, 0xd in the 16s column, and 0xc in the 1s column.



The simplest way to convert from *hexadecimal to decimal* is to read each digit from each column, convert it to decimal, multiply it by the column it is in (1s, 16s, 256s, 4096s and so on), and then add up all the multiplications. For the number 0xfedc, we would proceed as follows:

The digit "f" is in the 4096s column. 0xf in hex is 15 in decimal. Therefore, we multiply 15 by 4096 to get 61,440.

The digit "e" is in the 256s column. 0xe in hex is 14 in decimal. Therefore, we multiply 14 by 256 to get 3,584.

The digit "d" is in the 16s column. 0xd in hex is 13 in decimal. Therefore, we multiply 13 by 16 to get 208.

The digit "c" is in the 1s column. 0xc in hex is 12 in decimal. Therefore, we multiply 12 by 1 to get 12.

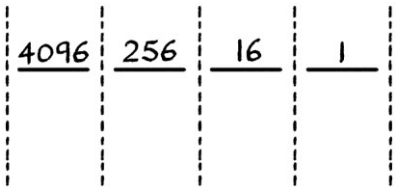
We then add up the results of the four multiplications:
61,440 + 3,584 + 208 + 12 = 65,244

We now know that 0xfedc is equal to 65,244 in decimal.

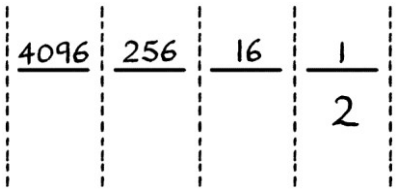
Converting from *decimal to hexadecimal* is easiest using a different type of method. The process is as follows:

- We divide the number by 16. We take the remainder, convert it into a hex digit, and put it into the 1s column.
- We then divide the integer part of the previous result by 16. We take the remainder, convert it into a hex digit, and put it into the 16s column.
- We then divide the integer part of the previous result by 16. We take the remainder, convert it into a hex digit, and put it into the 256s column.
- We continue in this way until we reach a time when the integer part of the previous division was zero.

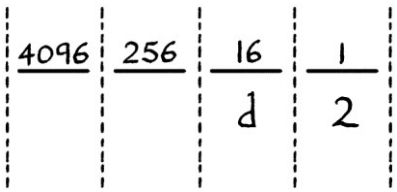
As an example, we will convert the decimal number 1234 to hexadecimal. We start with our empty hexadecimal number columns:



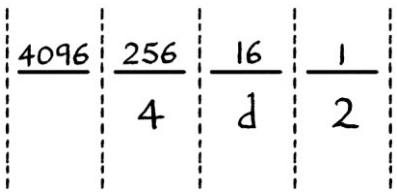
We divide 1234 by 16, and we get 77.125. This is 77 with a remainder of 2. [To calculate the remainder, we multiply the part after the decimal point by 16]. We convert the remainder to hex – it is 0x2 – and put it into the 1s column:



Then, we divide the integer part of the result from the last division (77 in decimal) by 16. This gives us 4.8125, which is 4 with a remainder of 13. We convert the remainder to a hex digit – it is 0xd – and put it in the 16s column:



Then we divide the integer part of the result from the last division (4) by 16. This gives us 0.25, which is 0 with a remainder of 4. We convert the 4 to a hex digit – it stays as 4 – and put it in the 256s column:



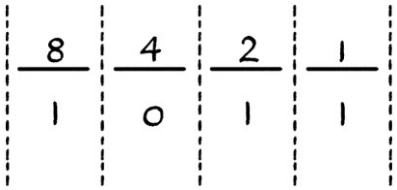
As the integer part of the result of the previous division was zero, we have finished. Our converted number is 0x4d2.

Converting between binary and decimal

Converting between *binary* and decimal uses a similar method as that for converting between hex and decimal. To convert from binary to decimal, we imagine the number columns for binary, which are, from right to left, ones, twos, fours, eights, sixteens, 32s, 64s, 128s and so on.

We go through the binary number, and add up all the number column values for whenever the bit is 1. [For example, if the number column is eights and the bit in that column is 1, then we add eight to our total.] The final total will be the binary number in decimal.

As an example, we will convert the binary number 1011 to decimal. [This is actually a number that, with practice, you will be able to do in your head instantly.] The number shown with the binary number columns is as so:

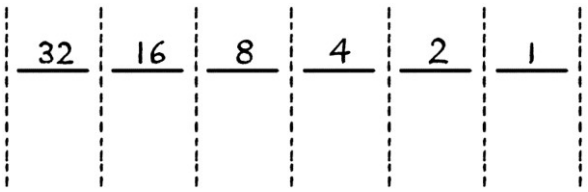


We will arbitrarily start at the leftmost bit, but it does not matter where we start. The leftmost bit is 1, which is in the eights column. Therefore, our running total will start with 8 in it. The next 1 is in the twos column. Therefore, we add 2 to our running total. The next 1 is in the ones column. Therefore, we add 1 to our running total. Our full sum is 8 + 2 + 1 = 11 in decimal.

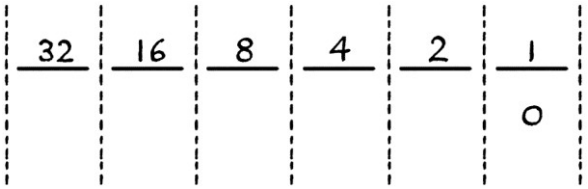
To convert from decimal to binary, we use a similar method to the one for converting from decimal to hex. If we have a decimal number to convert, we proceed as follows:

- We divide the number by 2. We take the remainder, which will be 0 or 1, and put it into the ones column.
- We then divide the integer part of the previous result by 2. We take the remainder, which again, will be 0 or 1, and put it into the twos column.
- We then divide the integer part of the last result by 2. We take the remainder, which again, will be either be 0 or 1, and put it into the fours column.
- We continue in this way until we reach a time when the integer part of the previous division was zero.

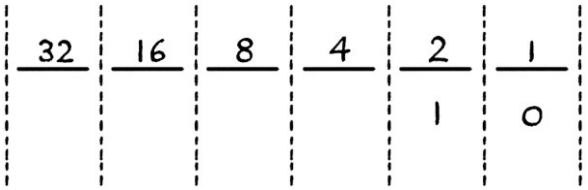
As an example, we will convert the decimal number 26 to binary. We will start with the blank number columns, which for this example, we will extend up to the 32s column. [We will only know how many columns we will need after we have performed the conversion.]



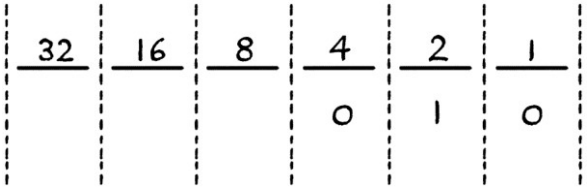
We divide our number by 2, and we end up with 13 and no remainder. Another way of saying this is that the remainder is 0. We put the remainder in the ones column:



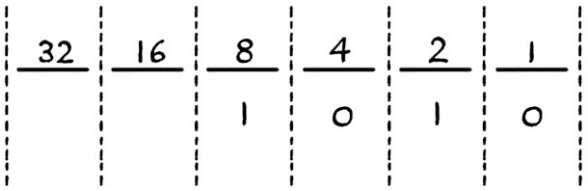
We then divide 13 by 2, and end up with 6.5, which is the same as 6 and a remainder of 1. We put the 1 in the twos column:



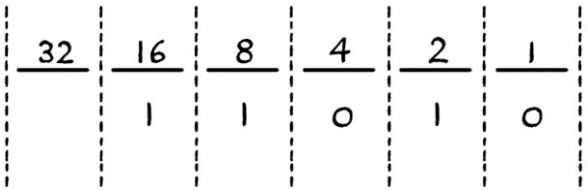
We then divide 6 by 2 and get 3 with a remainder of 0. We put the 0 in the fours column.



We then divide 3 by 2 and get 1.5, which is 1 with a remainder of 1. We put the one in the eights column.



We then divide 1 by 2, and we get 0 with a remainder of 1. We put the 1 in the sixteens column:



As our integer result was zero, we can stop here. Our final binary number (remembering to read from the sixteens column to the right) is 11010. In this example, we did not need to use the 32s column (or the 64s column, the 128s column and so on). If we wished, we could prefix our resulting binary number with zeroes to make it into a byte (00011010), word (0000000000011010), dword or qword. Alternatively, we can choose to leave it as it is.

When converting long binary numbers to decimal, it is easiest to convert the number to hex first, then and convert that to decimal. Conversely, if it looks like a conversion from decimal to binary will produce a long binary number, it is easier to convert from decimal to hex, and then from hex to binary.

Converting between hex and binary

Converting from hexadecimal to binary and back is easy because we know that 4 binary digits will end up as 1 hex digit, and that 1 hex digit will end up as 4 binary digits. A sequence of binary digits can be split up into fours (starting from the right-hand side), and each group of four can be converted into a hex digit. [If a binary number has too few digits to be split into groups of 4, we can prefix it with one or more zeroes to make it the correct length.] Similarly, a sequence of hex digits can be converted digit by digit into four binary bits.

Given all of that, if you want to become proficient at converting between hexadecimal and binary, it pays to learn the following table. The table is easiest to learn with practice – if you frequently need to convert between binary and hexadecimal, you will remember it quickly. If you seldom need to convert, it will be harder to learn, but, on the other hand, there will be less reason to learn it.

Binary	**Hexadecimal equivalent**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

If we have the binary number 1011110000000001, we first separate it into 4-bit sections to make it easier to read:
1011 1100 0000 0001
... then, we convert each 4-bit section to the single hex digit it represents:
b c 0 1
... which means that our hex number is:
0xbc01

If we have the hex number 0xfc23, we convert each digit to the 4-bit binary number it represents:
1111 1100 0010 0011
... and then we remove the spaces (which should only really exist to make the number easier to read):
1111110000100011

Groups of bits

As we saw earlier in this chapter, when dealing in binary, there are predefined groups of bits. These are half bytes, bytes, words, dwords and qwords. These groups are the same as for hexadecimal, but are easier to read.

A half byte, being 4 bits, is represented by just one hexadecimal digit. For example, the binary number 1101 is 0xb in hexadecimal.

A byte, being 8 bits, is represented by two hexadecimal digits. Despite this, it will still be referred to as an 8-bit number. As it will always be two hexadecimal digits, if the number is less than 16 (in decimal), it will have a preceding zero in front. Examples of hexadecimal bytes are:

```
0x00
0x01
0x02
0x03
0x04
0x05
0x0a
0x0b
0x0c
0x0d
0x0e
0x0f
0x10
0x11
0x21
0xa0
0xb7
0xdd
0xfe
0xff
```

If we have the hexadecimal *byte* 0xff, and add 1 to it, we will end up with the byte 0x00. This is because 0xff is the highest possible number in a byte. It is 255 in decimal or 11111111 in binary. Adding 1 to 0xff makes it roll over to zero. If we added 2 to the byte 0xff, we would end up with the byte 0x01. If we added 3 to the byte 0xff, we would end up with the byte 0x02. [Remember that the rolling over at 0xff only occurs when we are using *bytes* to store a number. For other number groupings, or away from computers, adding 1 to 0xff would result in 0x100.]

As we know, a word is 16 bits long. This means that it has 16 binary digits if we are portraying it in binary. A word, written as hexadecimal, has 4 hexadecimal digits. Despite this, it will still be referred to as a 16-bit number. Some examples of words in hexadecimal are as follows:

```
0x0000
0x0001
0x0002
0x000f
0x0010
0x0011
0x0012
0x056a
0x1679
0x2acd
0x9abb
0xde01
0xdfff
0xe000
0xfffe
0xffff
```

If we are working with words, and we added 1 to 0xffff, we would end up with 0x0000. The number 0xffff is the highest number that we can represent with a word (16 bits).

If we had a word in binary as 1111000001100110, we could write it in hexadecimal as 0xf066.

A doubleword, or dword, is 32 bits long. When we write a dword as binary, we have 32 binary digits in the number. When we write it in hexadecimal, we have 8 hexadecimal digits. Despite this, it would still be referred to as a 32-bit number. Examples of hex dwords are as follows:

```
0x00000000
0x00000001
0x0000000f
0x00000010
0x0000ffff
0x12345678
0xe0000000
0xefffffff
0xf0000000
0xfffffffd
```

A quadword, or qword, is 64 bits long. In binary, such a number would require 64 digits. In hexadecimal, the same number would require 16 hexadecimal digits. Despite this, it would still be referred to as a 64-bit number.

Whereas a dword in hexadecimal is reasonably easy to read and remember, a qword is an awkward size. Therefore, for the purposes of this explanation, I will put a space in the middle of the number. You can think of this as similar to putting a comma in a long decimal number such as 1,000,000. If we were writing a qword in a computer program, we would not be able to put a space in the middle.

Some examples of qwords in hexadecimal, with a space in the middle for clarity, are as follows:

```
0x00000000 00000000
0x00000000 00000001
0x00000000 00000002
0x00000000 00000003
0x00000000 00000004
0x00000000 00000005
0x00000000 0000000f
0x00000000 00000010
0x00000000 0000ffff
0x12345678 9abcdef0
0x2fffffff ffffffff
0x30000000 00000000
0xcccccccc cccccccc
0xf0000000 00000000
0xfffffffd fffffffd
0xfffffffe fffffffe
0xffffffff ffffffff
```

The last number in the list is the highest number that we can represent with a qword.

More on hexadecimal

Hexadecimal should really be thought of as another way of writing binary. It pays to think of binary and hexadecimal as being interchangeable, or if they were really the same thing in different forms. A number, whether written as binary or hexadecimal, is still the same number. Although computers ultimately work in binary, it is easier to think of them as working in hexadecimal. Hexadecimal is a way of making endless sequences of binary digits have a recognisable meaning. For example, it is easier to think of this 32-bit binary number:
11110001011101110110000100100011
... as being:
0xf1776123
... than it is to think of it as binary. It is much easier to read and remember hexadecimal numbers, and it is much easier to recognise patterns in hexadecimal numbers. Hexadecimal's close connection with binary is the reason that it is usually easier not to convert hexadecimal to binary, but stay in the world of hexadecimal. If we know a hexadecimal number, we know the exact position of every bit in its binary equivalent.

Hexadecimal also has a slight advantage over decimal in that it is better for larger numbers. The most extreme example is how the maximum qword value (0xffffffffffffffff) takes 16 hexadecimal digits, but its decimal equivalent is 20 digits long.

Hex editors

On a computer, every file, whether a text file, an executable, a picture, an MP3, a Microsoft Word document, or anything, is really a sequence of binary digits. When an MP3 file, for example, is opened by an MP3 player program, the sequence of bits in the file is interpreted as music. If a text file is opened by a text editor, the sequence of bits in the file is interpreted as text. When an executable is run, the operating system interprets the sequence of bits as instructions to be executed. If you open an executable file in a text editor, you will see seemingly random characters. If you try to play a text file in an MP3 player, the player will complain that it does not understand the file. To keep things straightforward, programs generally hide the underlying binary content of files from users.

It is possible to see the hexadecimal equivalent of the binary bits that make up a file by opening the file in a hex editor. A hex editor is a program that opens any file, without interpreting its contents in any way. When you open a file in a hex editor, you will see the contents of a file as a sequence of hexadecimal bytes. [It would be possible to have a program that showed the binary bits that made up a file, but it would be much harder to make sense of the data in that way. It is better to view the binary in the form of hexadecimal.] Using a hex editor is the best way to become accustomed to how computers store data. It is worth downloading a free hex editor, so you can improve your understanding of hexadecimal and files in general.

As an example of what we might see in a hex editor, here are the first 64 bytes from a text file containing this very sentence:

```
41 73 20 61 6e 20 65 78 61 6d 70 6c 65 20 6f 66
20 77 68 61 74 20 77 65 20 6d 69 67 68 74 20 73
65 65 20 69 6e 20 61 20 68 65 78 20 65 64 69 74
6f 72 2c 20 68 65 72 65 20 61 72 65 20 74 68 65
```

Because we know that these bytes are hexadecimal, there is no need to prefix them all with “0x”, and doing so would make them harder to read.

Due to the size of a typical file and the width of a typical computer screen, only so many bytes will fit neatly on a line. The start and end points of a line of bytes is solely due to how a particular hex editor has decided to fit the bytes on a line, and is not related to the nature of the bytes.

The grouping into bytes is just to make the digits easier to read. The digits could just as easily be written as one continuous block as so:

```
417320616e206578616d706c65206f662077686174207765206d696768742073656520
696e2061206865782065646974f722c20686572652061726520746865
```

Although computers work with binary, a hex editor shows the data as hexadecimal to make it easier to read and interpret. If it were to show the data in binary, we might expect to see it as so:

```
00100001 01110011 00100000 01100001 01101110 00100000 01100101
01111000
```

... and so on.

Or, if there were no arbitrary spaces between the bits, we would see it as so:

```
00100001011100110010000001100001011011100010000001100101011110000
```

... and so on.

We can see that it is much easier to read the data as hexadecimal.

Hex editors usually show where in the file each line of hex bytes starts by prefixing each line with the "offset" of the line. The "offset" is the position of a particular byte with respect to the start of the file. Our 64 bytes of text would look like this with the starts of each line written to the left of the data:

```
0000: 41 73 20 61 6e 20 65 78 61 6d 70 6c 65 20 6f 66
0010: 20 77 68 61 74 20 77 65 20 6d 69 67 68 74 20 73
0020: 65 65 20 69 6e 20 61 20 68 65 78 20 65 64 69 74
0030: 6f 72 2c 20 68 65 72 65 20 61 72 65 20 74 68 65
```

The offsets themselves are given in hexadecimal. The file starts at byte number 0x0000. To put this another way, the first byte is at offset 0x0000 in the file. The first line is 16 (in decimal) bytes long, which is 0x10 (in hex) bytes long. This means that the first byte of the *next* line is at offset 0x0010 in the file. Byte number sixteen (in decimal) in the file has the value 0x20. The second line is 0x10 bytes long. Therefore, the third line as seen in the hex editor's display shows data from byte number 0x30 onwards in the file. Different hex editors vary in how many bytes they show on a line, but they will usually show the offset of where the bytes are. In the example above, I have given the offset as 4 hex digits to save space on the page. Most hex editors will give the offset as 8 hex digits so that they can work with very large files.

To make recognising patterns in hexadecimal easier, hex editors usually have a section to the right of each line of hex that shows if any of the bytes would be valid ASCII characters, and if so what they would be. [I will explain ASCII characters shortly, but for now, it is enough to know that ASCII is a generally accepted way of encoding letters of the alphabet with 8-bit numbers (bytes).] The bytes from our text file would look like this in a hex editor that showed the ASCII part:

```
0000: 41 73 20 61 6e 20 65 78 61 6d 70 6c 65 20 6f 66 As an example of
0010: 20 77 68 61 74 20 77 65 20 6d 69 67 68 74 20 73  what we might s
0020: 65 65 20 69 6e 20 61 20 68 65 78 20 65 64 69 74 ee in a hex edit
0030: 6f 72 2c 20 68 65 72 65 20 61 72 65 20 74 68 65 or, here are the
```

As the data in our example is all text, every byte represents a letter of the alphabet, so the whole of the right hand column is text. The very first byte of our file is 0x41, which is also the ASCII number for the letter “A”. The next byte is 0x73, which is the ASCII letter “s”. The next byte is 0x20, which is the ASCII number for a space.

ASCII

The abbreviation “ASCII” is short for the “American Standard Code for Information Interchange”. ASCII is a system that assigns a number to each letter of the alphabet, to each decimal digit, and to some punctuation. ASCII is a generally accepted standard for encoding basic text characters. By following the ASCII system of assigning particular numbers to characters, different computer programs can store and read text in the same way. It makes everything much easier to have a standard system. It is worth noting that the ASCII method of assigning numbers to characters is completely arbitrary, and its creation and its use is independent of how the processors of computers work, except for how each ASCII value is stored as one byte.

Having a general understanding of ASCII is useful in understanding what you might see in a hex editor.

In the ASCII system, each character is identified by 8 bits. Therefore, each character can fit in exactly one byte. Whether we choose to treat the value in that byte as binary, hexadecimal or decimal is a matter of choice. In some situations, it is easier to think of them as binary; in some situations, particularly when viewing them in a hex editor, it is easier to think of them as hexadecimal. Some people prefer to think of them as decimal numbers, but doing that is generally less useful.

Being an American system, ASCII prioritises characters from a subset of the modern English alphabet. As well as letters, numbers and punctuation, there are some obsolete “control” characters, which were used to control early printers among other things.

Note that a byte only represents an ASCII character if the computer program that is dealing with it chooses to interpret it as one. If a program is not specifically working with ASCII text (or it does not know that it should be), a byte will just be treated as a number.

The first 128 characters of ASCII, being the characters from 0x00 to 0x7f, are as follows. I have ignored the meanings of most of the control symbols:

ASCII number as binary	ASCII number as hex	Character or meaning
00000000	0x00	This is often used, arbitrarily, in programming languages to mark the end of a piece of text.
00000001	0x01	
00000010	0x02	
00000011	0x03	
00000100	0x04	
00000101	0x05	
00000110	0x06	
00000111	0x07	On older computers, reading this character would make the computer beep.
00001000	0x08	Backspace
00001001	0x09	Tab
00001010	0x0a	In Linux, this is treated as a new line character. It means any text starts on a new line after this appears. In Windows, this is treated as a new line character when it is preceded by 0x0d.
00001011	0x0b	
00001100	0x0c	In some text editors, this is indicates a new page.
00001101	0x0d	In Windows, 0x0d followed by 0x0a is treated as a new line.
00001110	0x0e	
00001111	0x0f	
00010000	0x10	
00010001	0x11	
00010010	0x12	
00010011	0x13	
00010100	0x14	
00010101	0x15	
00010110	0x16	
00010111	0x17	
00011000	0x18	
00011001	0x19	
00011010	0x1a	
00011011	0x1b	

ASCII number as binary	ASCII number as hex	Character or meaning
00011100	0x1c	
00011101	0x1d	
00011110	0x1e	
00011111	0x1f	
00100000	0x20	Space: “ ”
00100001	0x21	!
00100010	0x22	"
00100011	0x23	#
00100100	0x24	$
00100101	0x25	%
00100110	0x26	&
00100111	0x27	'
00101000	0x28	(
00101001	0x29	)
00101010	0x2a	*
00101011	0x2b	+
00101100	0x2c	,
00101101	0x2d	-
00101110	0x2e	.
00101111	0x2f	/
00110000	0x30	0
00110001	0x31	1
00110010	0x32	2
00110011	0x33	3
00110100	0x34	4
00110101	0x35	5
00110110	0x36	6
00110111	0x37	7
00111000	0x38	8
00111001	0x39	9

[To convert the ASCII code of a number to the actual number that it represents, we can just subtract 0x30.]

ASCII number as binary	ASCII number as hex	Character or meaning
00111010	0x3a	:
00111011	0x3b	;
00111100	0x3c	<
00111101	0x3d	=
00111110	0x3e	>
00111111	0x3f	?
01000000	0x40	@
01000001	0x41	A
01000010	0x42	B
01000011	0x43	C
01000100	0x44	D
01000101	0x45	E
01000110	0x46	F
01000111	0x47	G
01001000	0x48	H
01001001	0x49	I
01001010	0x4a	J
01001011	0x4b	K
01001100	0x4c	L
01001101	0x4d	M
01001110	0x4e	N
01001111	0x4f	O
01010000	0x50	P
01010001	0x51	Q
01010010	0x52	R
01010011	0x53	S
01010100	0x54	T
01010101	0x55	U
01010110	0x56	V
01010111	0x57	W
01011000	0x58	X
01011001	0x59	Y
01011010	0x5a	Z

ASCII number as binary	ASCII number as hex	Character or meaning
01011011	0x5b	[
01011100	0x5c	\
01011101	0x5d	]
01011110	0x5e	^
01011111	0x5f	_
01100000	0x60	`
01100001	0x61	a
01100010	0x62	b
01100011	0x63	c
01100100	0x64	d
01100101	0x65	e
01100110	0x66	f
01100111	0x67	g
01101000	0x68	h
01101001	0x69	i
01101010	0x6a	j
01101011	0x6b	k
01101100	0x6c	l
01101101	0x6d	m
01101110	0x6e	n
01101111	0x6f	o
01110000	0x70	p
01110001	0x71	q
01110010	0x72	r
01110011	0x73	s
01110100	0x74	t
01110101	0x75	u
01110110	0x76	v
01110111	0x77	w
01111000	0x78	x
01111001	0x79	y
01111010	0x7a	z

[To convert a lower-case letter to upper case, we can just subtract 0x20 or set the third bit from the left to zero.]

ASCII number as binary	ASCII number as hex	Character or meaning
01111011	0x7b	{
01111100	0x7c	|
01111101	0x7d	}
01111110	0x7e	~
01111111	0x7f	

The ASCII characters from 0x80 to 0xff are called the "Extended ASCII" characters because they are an extension to the original ASCII system, which used only 7 of the available 8 bits of a byte. There are numerous variations for the meanings of the Extended ASCII bytes, depending on the encoding and language being used. These variations are called "code pages". The most commonly used of these were defined by Microsoft, and have the names "Windows-1250", "Windows-1251", "Windows-1252" and so on. Among these are the following:

- "Windows-1250" uses the bytes to represent symbols and accented letters from Eastern and Central European alphabets that are based on the Latin alphabet (such as those used with Polish and Czech).
- "Windows-1251" uses the bytes to represent symbols and letters from a subset of the Cyrillic alphabet (as used in Russian and Ukrainian).
- "Windows-1252" uses the bytes to represent symbols and accented letters from Western European alphabets that are based on the Latin alphabet (such as French).
- "Windows-1254" is for accented Turkish letters.
- "Windows-1255" is for Hebrew letters.
- "Windows-1256" is for Arabic letters.

In each system, the bytes from 0x00 to 0x7f still have the same meaning as in the long list from before.

The different interpretations of the bytes from 0x80 onwards mean that text that is supposed to be from one particular language will be displayed incorrectly if the wrong interpretation is used. If someone writes Russian text in an ASCII text file on a Russian version of Microsoft Windows, that file will appear as random accented Latin letters and symbols on an English version of Microsoft Windows. It will appear as random Arabic letters and symbols in an Arabic version of Windows. This is one of the big flaws of ASCII – it is not particularly good for non-English alphabets.

Now that memory is cheaper, and computers and networks are faster, ASCII has generally been replaced with Unicode. Unicode is a similar system, which, depending on its implementation, can use one, two, or more bytes to represent a character. Alphabets that have many characters might use three bytes or more for each character. The "0x00 to 0x7f" ASCII set of Latin characters is portrayed with one byte in the Unicode version called UTF-8, or two bytes in the Unicode version called UTF-16. As there is no limit to the number of bytes that can be used per character, each Unicode value represents a unique character or symbol, and there can never be any confusion between languages.

More hex editor examples

Here is the very start of a Microsoft Windows executable (that is to say, a program or application) as opened with a hex editor:

```
0000: 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0010: b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0030: 00 00 00 00 00 00 00 00 00 00 00 00 c8 00 00 00 ................
0040: 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0050: 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
0060: 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
```

The first two bytes are 0x4d, 0x5a, which are the ASCII letters "MZ". This is what is called the "signature" of a file. It gives clues to the operating system, or any program trying to open it, as to what sort of file this is. Generally, Microsoft Windows executables will start with these letters or the operating system will complain. The dots on the right hand side indicate that if the associated byte on the left hand side were interpreted as ASCII, it would not be a letter, number or punctuation. Due to the ambiguity of ASCII characters over 0x80, most hex editors do not give equivalent letters for those bytes, and just have a "." instead. Therefore, if there were Cyrillic ASCII text in a file, we would have to recognise it by the bytes themselves, and the hex editor would not help.

The "@" symbol, corresponding to the byte 0x40 is a coincidence. Hex editors cannot know if a value is intended to be a letter, number or punctuation, so they presume, usually incorrectly, that anything that could be, will be.

The part of the file that we see above is part of what is called the “header”. The header of a file is information that indicates to the operating system, or the program that opens it, certain attributes of the data within the file. For example, the full header for an executable will list where in the file the commands are, where the data is, where any resources such as icons and dialog layouts are stored, and so on. Not all files have headers – for example, a basic ASCII text file does not need one.

The following is what the start of a particular TIFF image file looks like when opened in a hex editor:

```
0000: 49 49 2a 00 08 00 00 00 0a 00 00 01 03 00 01 00 II*.............
0010: 00 00 00 08 00 00 01 01 03 00 01 00 00 00 00 02 ................
0020: 00 00 03 01 03 00 01 00 00 00 01 00 00 00 06 01 ................
0030: 03 00 01 00 00 00 01 00 00 00 11 01 04 00 01 00 ................
0040: 00 00 96 00 00 00 16 01 04 00 01 00 00 00 00 02 ................
0050: 00 00 17 01 04 00 01 00 00 00 00 00 02 00 1a 01 ................
0060: 05 00 01 00 00 00 86 00 00 00 1b 01 05 00 01 00 ................
0070: 00 00 8e 00 00 00 28 01 03 00 01 00 00 00 03 00 ......(.........
0080: 00 00 00 00 00 00 40 00 00 00 01 00 00 00 40 00 ......@.......@.
0090: 00 00 01 00 00 00 ff ff ff ff ff ff ff ff ff ff ................
00a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
```

The first two bytes are 0x49, 0x49. These indicate to any program opening the TIFF file that, first, this is a TIFF file, and second, how the data within it is stored. The bytes 0x2a, 0x00 tell the program which version of the TIFF image format is being used. The following bytes up to the byte at offset 0x96 tell the program how many rows and columns of pixels there are in the TIFF file, whether the data is compressed, what the resolution is, and where the actual image data starts. This particular file is an uncompressed black and white bitmapped TIFF file, which means that it assigns one bit to every pixel. That bit will be either a 1 to indicate a white pixel, or a 0 to indicate a black pixel. The actual image data starts at offset 0x96. The data starts as a sequence of 0xff bytes, where each byte represents 8 consecutive white pixels. The “*” sign, the bracket and the “@” sign on the right-hand side are coincidences where a hexadecimal value happens also to be a readable ASCII character.

Memory

When programmers run programs they have written, it can be useful to know exactly what is going on in the computer, so they can find mistakes or improve their code. To do this they will use a program that steps through each command in turn, while showing what is happening inside the processor, and what is stored in the relevant part of memory. Such a program is called a "debugger" because it is used to debug software. When viewing what is stored in memory, a debugger will show a very similar view to what we might see in a hex editor. The main difference will be that the offsets will not generally start at zero, but instead at an offset from the start of the computer's memory being assigned to the program. For a Windows program intended for a 32-bit Intel processor, we might see something such as this:

```
040ca100: c3 eb 12 90 90 90 90 00 48 33 C0 48 8B 74 24 40 ........H3.H.t$@
```

In this example, the ASCII characters on the right-hand side are just coincidences where the bytes happen to have values equal to valid ASCII characters.

Other information

Here is some more information about hexadecimal and binary.

Bit number

When identifying entries in a list using binary or hexadecimal, it makes sense to treat the first entry as item zero, instead of item one. This is because, by doing so, we are able to count higher. For example, a list of names might start as so:

Name number 0: Gertrude
Name number 1: Gladys
Name number 2: Gwen
Name number 3: Gwynne
... and so on.

If we are constrained by using bytes, we can distinguish 256 different names if we start at zero. We would have name number 0x00 up to name number 0xff. If we start at 1, we can only distinguish 255 different names – we would have name number 0x01 up to name number 0xff.

This way of counting from zero is also used when identifying a particular bit in a binary number. When identifying the bits of a binary number, we count from the right hand side. Bit 0 is the rightmost bit. With a byte, bit 7 is the leftmost bit. With a word, bit 15 is the leftmost bit. With a dword, bit 31 is the leftmost bit. With a qword, bit 63 is the leftmost bit.

In the following binary number as a byte:
00000001
... bit 0 has the value “1”. All the other bits have the value “0”. Bit 7 (the leftmost bit) is “0”.

With this binary number:
10000000
... bit 0 (the rightmost bit) has the value “0”, and bit 7 (the leftmost bit) has the value “1”.

With the binary number:
01001110

- bit 7 is 0
- bit 6 is 1
- bit 5 is 0
- bit 4 is 0
- bit 3 is 1
- bit 2 is 1
- bit 1 is 1
- bit 0 is 0

As well as identifying bits in binary numbers, we can take advantage of how there is a one-to-one relationship between binary and hexadecimal, and identify the bits within hexadecimal numbers. For example, in the hex byte 0xf1:

- bit 7 is 1
- bit 6 is 1
- bit 5 is 1
- bit 4 is 1
- bit 3 is 0
- bit 2 is 0
- bit 1 is 0
- bit 0 is 1

This is all because 0xf1 is 11110001 in binary. Despite speaking about a hexadecimal number and not a binary number, we can still treat it as consisting of bits.

Significant bits

When dealing in binary and hexadecimal, it is common to see the phrases “most significant bit” and “least significant bit”.

The most significant bit is the leftmost bit of a byte, word, dword or qword. It is called this because the leftmost bit is the bit whose presence or absence has the greatest effect on the overall *size* of a number. For example, the leftmost bit in the binary number 10000001 makes the difference between the number being 0x81 or 0x01. This is a difference of 128 in decimal. The term “most significant bit” is often abbreviated to “msb”.

The least significant bit is the rightmost bit. This is because the presence of the rightmost bit has the least effect on the overall *size* of a number. The rightmost bit of the binary number 11111111 is the difference between the number being 0xff and 0xfe (255 and 254 in decimal). There is only a difference of 1. The term “least significant bit” is often abbreviated to “lsb”.

It is obviously quicker and more descriptive to say “leftmost bit” and “rightmost bit”, but some people prefer the terms “most significant” and “least significant”. One problem with the term “significant” is that it only makes sense if the byte, word, dword or qword is being used to count. In an ASCII byte, for example, every bit is just as significant as any other because the byte is not being used as a number, but as an index to an item in a table of characters.

Storing data in memory or files

We will imagine a computer processor is dealing with the 32-bit dword, 0x87654321. When it has finished working with the dword and it wants to store it for later, perhaps in memory or to disk, there are two ways that it can lay down the number.

The first way is for the processor to place the number down in the order of the digits. This is the most intuitive way, and is how most people would expect it to do it. This means that if we were looking at a file in a hex editor, with the dword placed at the very start, we would see this:

```
0000: 87 65 43 21 00 00 00 00 00 00 00 00 00 00 00 00 .eC!............
```

[Note that each 0x00 is just filler that I have put in to make up a full line. The letters “eC!” are there because, by chance, the bytes 0x65, 0x43, and 0x21 are also valid readable ASCII characters.]

If the computer processor then wanted to read the dword from the file or memory, it would read the whole dword in one go. The reason it would work in this way hardly needs explaining, as it is exactly what one would expect it to do.

The second way a computer processor might place a dword into memory or to a file is by splitting the dword into bytes and putting them into *reverse* order. As we are starting with the dword 0x87654321, this means that the dword becomes the bytes 0x87, 0x65, 0x43, 0x21, and each byte is put down in reverse order. A file containing the result of such a thing as the very first few bytes would look like this:

```
0000: 21 43 65 87 00 00 00 00 00 00 00 00 00 00 00 00 !Ce............
```

[Again, the zeroes are filler to make up a full line, and the letters “!Ce” are there because the bytes are, by coincidence, valid readable ASCII characters.]

Although this method seems to overly complicate matters, it can be useful. If we want to find the first byte of the stored dword 0x87654321, we look at the offset in the file of where we put the whole dword. The first byte of 0x87654321 is 0x21, and it is at offset 0x0000 in the file. Similarly, if we wanted to find the first *word* of 0x87654321, we would look at offset 0x0000 in the file, and read the two bytes there (0x21 and 0x43), which because we are using this backwards ordering system, we rearrange to be 0x4321. Therefore, this backwards method allows us easily to retrieve the lower bytes of words, dwords and qwords without having to know whether we are using words, dwords and qwords. It is a useful method, but it is mostly hidden from anyone who is not programming in assembly language, examining files with hex editors, or debugging software.

As with a dword, if a computer processor were working in this system and storing a qword or a word in memory or a file, it would similarly put its bytes in reverse order. If the processor were storing a byte, then the byte would be the same within the computer processor as it would be stored in a file or memory.

If a processor stored the qword, 0x8070605040302010 into memory or a file, it would appear as so:

```
0000: 10 20 30 40 50 60 70 80 00 00 00 00 00 00 00 00 ."3DUfw.........
```

[Note that the offset would vary depending on where in the file or memory the number was placed.]

If a processor stored the dword, 0x40302010 it would appear as so:

```
0000: 10 20 30 40 00 00 00 00 00 00 00 00 00 00 00 00 ."3D............
```

If a processor stored the word, 0x2010 it would appear as so:

```
0000: 10 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ."..............
```

If a processor stored the byte, 0x10 it would appear as so:

```
0000: 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

If we wanted to store a half byte (4 bits), we would first have to turn it into a whole byte, perhaps with the first 4 bits set to zero. The result of storing it would be the same as if we were storing a byte.

This “backwards” method of storing data is called the “little-endian” method of storing data. The “lowest” bytes are placed first, which we can rephrase as the “littlest end of the word, dword or qword is placed first”. Given that Intel processors use this method, at the time of writing, this is probably as common as the ostensibly more sensible “forwards” method, which is called “big-endian”. We can think of big-endian as placing the “highest” or “biggest” bytes first. Although little-endian seems unintuitive, the more you are exposed to it, the less puzzling it will seem.

When something is stored in *memory*, it is stored using the system that the processor uses. Therefore, (at the time this is being written) an Intel or AMD processor will always use the “backwards” little-endian system when putting a number in memory. When something is stored in a *file*, the system that is used depends on the program that is storing the data. However, it is much easier and quicker to store data in a way that is consistent with the processor. Doing so means that something can be copied directly from memory into a file, without needing rearranging. Other processors might use the “forwards” big-endian system, and some processors are able to use both systems.

In a TIFF image file, the first two bytes of the header tell the image-viewing software whether the data has been stored with the big-endian or little-endian system. If the first two bytes are the ASCII letters “II”, then the data in the file is little-endian; if the bytes are “MM”, the data is big-endian. This allows TIFF files to be viewed on computers that operate in either the big-endian system or the little-endian system, without reading the data the wrong way around. If a program is going to be reading data in group sizes larger than bytes, it needs to have a way of knowing which system is being used, or it might misinterpret the data.

When we store wave data in files, which we will do later in this chapter, we will similarly have the choice of whether to use the big-endian method or the little-endian method.

If we have several numbers in sequence, then we store them all in either the little-endian way or the big-endian way. As an example, we will look at this list of dwords:

```
0x0000ffff
0x11223344
0x55667788
0xfefeffff
```

Stored in the little-endian way, they would look like this in a file viewed in a hex editor:

```
0000: ff ff 00 00 44 33 22 11 88 77 66 55 ff ff fe fe ....D3"..wfU....
```

Stored in the big-endian way, they would look like this:

```
0000: 00 00 ff ff 11 22 33 44 55 66 77 88 fe fe ff ff .....”3DUfw.....
```

When it comes to storing ASCII text, each letter is represented by a single byte, so the bytes are laid down in their original order.

```
0000: 48 65 72 65 27 73 20 73 6f 6d 65 20 74 65 78 74 Here's some text
```

When it comes to storing Unicode text, and when the characters are being portrayed by 16-bit words each, the 16-bit words are stored in either the little-endian way or the big-endian way.

Octal

Octal is a counting system that is based around the number 8. If we were to count in octal, we would proceed as follows:

0
1
2
3
4
5
6
7
10
11
12
13
14
15
16
17
20
21
22
23
24
25
26
27
30

... and so on.

For some reason, octal is often taught as if it were as useful to computers as hexadecimal or binary. Computers do not natively use octal, and it is much easier to be using hexadecimal, binary and decimal. I only mention octal here in case you should see it mentioned elsewhere.

Negative numbers

Away from a computer, we can portray negative numbers and fractions in binary and hexadecimal in the same way that we do with decimal numbers. For example, the number −12 in decimal could be written as −1100 in binary or −c [or −0xc] in hexadecimal. The decimal number 2.5 could be written as 10.1 in binary or 2.8 in hexadecimal. The number −1011.0011001 is a perfectly valid way to write a number in binary. When it comes to computers, we cannot write numbers in this way because the default system of counting in binary or hexadecimal for computers only allows positive integers. There is no place for a decimal point or a negative sign in one of the group-types for bits.

To allow computers to work with negative numbers and fractions, slightly awkward workarounds were invented. These involve *interpreting* seemingly normal hexadecimal or binary values in a different way. In other words, we still use bytes, words, dwords and qwords, and these still contain binary bits. However, we alter the bits of the number in a special way before we put the number into a byte, word, dword or qword. Someone reading the byte, word, dword or qword must know that they need to interpret it differently, or else the value will be misinterpreted as a positive integer.

In this section, we will look at how computers manage negative numbers.

Two's complement

The standard way that computers express negative numbers in binary or hexadecimal is called "two's complement". The general idea is that the leftmost bit (the highest bit) is set to 0 if the number is positive, and it is set to 1 if the rest of the number is negative. This means that we have fewer bits with which to portray values, but it allows us to have both positive and negative numbers – *as long as we and anyone else using the number know that it should be interpreted in this way*. The actual method is slightly more complicated than just setting the leftmost bit, as we will see shortly.

Two's complement allows one byte to portray the numbers from −0x80 to +0x7f (−128 to +127 in decimal). It allows a word to hold the numbers from −0x8000 to +0x7fff (−32,768 to +32,767 in decimal). It allows a dword to hold the numbers from −0x80000000 to +0x7fffffff (−2,147,483,648 to +2,147,483,647 in decimal). It allows a qword to hold values from −0x8000000000000000 to +0x7fffffffffffffff (−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 in decimal).

All of this means that if we only wanted to store positive integers, using two's complement would only allow us to store 1 less than half the highest value that we could store normally.

Trying to store a value outside of the available ranges will result in the number being stored incorrectly, and later, being interpreted incorrectly.

The way to format a number as a two's complement number is fairly easy. We start by making sure that our number is not too high or too low to fit in the byte, word, dword or qword that we are using.

If the number we have is negative, then:

- We make it positive.
- We subtract 1.
- We "flip" all the bits apart from the leftmost bit. In other words, if a bit is 0, we change it to 1; if a bit is 1, we change it to 0.
- We set the leftmost bit to 1 to indicate that it is a negative number.

If the number we have is positive, then:

- We set the leftmost bit to 0 to indicate that it is a positive number.
  [Strictly speaking, it should be 0 already, because if it is not, then the number is too large to be encoded as a two's complement number.]

When we have a negative number, it will always be the case that once we have subtracted 1, the leftmost bit will be zero. Therefore, we can speed up the method by flipping all of the bits including the leftmost bit in one go – we do not need to set the leftmost bit to 1 to indicate that it is a negative number because the flipping will do that for us. Therefore, a quicker method to store a negative number is:

- We make it positive.
- We subtract 1.
- We flip all the bits. In other words, if a bit is 0, we change it to 1; if a bit is 1, we change it to 0.

The "complement" of a bit is its opposite. The complement of 0 is 1, and the complement of 1 is 0. When flipping a bit from 0 to 1, or 1 to 0, we are finding its complement. For negative numbers, we find the complement of every bit. As we are using binary and complementing the binary digits, the system is called "two's complement".

When reading a converted value, as long as the processor or software (or a person reading it) is aware that the value is a two's complement number, it will be known what the value represents. On the other hand, if a negative two's complement number is read as a normal number, it will be misinterpreted as a different value. There is no way of knowing whether a given hex or binary value is intended to be a two's complement number or not by looking at it. It needs context to be decoded correctly. Two's complement is really just a slightly awkward compromise that reuses normal binary and hex in a different way.

Examples

We will look at some simple examples involving bytes. The first thing to realise is that any *positive* number under and including 0x7f will be the same whether it is in two's complement or not. A value above 0x7f cannot be converted to a two's complement byte, as the maximum possible positive value is 0x7f. The byte 0x7f is 01111111 in binary. Any number higher than this would have a 1 as its leftmost digit. The number 0x23 as a two's complement byte will still be 0x23. The number 0x70 will still be 0x70 as a two's complement byte. The number 0x00 will still be 0x00.

If we want to express −1 as a negative two's complement byte, we first make it positive. It becomes:
00000001

We subtract 1 to end up with:
00000000

We then flip all the bits, which also has the effect of setting the leftmost bit to 1 to indicate that it is a negative number:
11111111

This is 0xff in hex, so we can say that −1 as a two's complement byte is 0xff.

A useful pattern to recognise is that, *in two's complement*, −1 is 0xff as a byte, 0xffff as a word, 0xffffffff as a dword, and 0xffffffffffffffff as a qword. It pays to remember that if we were not interpreting the bytes as two's complement numbers, the values would have their normal hexadecimal meanings, with 0xff being 255 in decimal, 0xffff being 65,535, and so on. There is no way of knowing by looking at a number whether it is intended to be a two's complement number or not – we have to know the context in which it is being used.

If we want −2 as a two's complement byte, we first make it positive. It becomes:
00000010

We subtract 1 to end up with:
00000001

We then flip all the bits:
11111110

This is 0xfe in hex. The number −2, *when using two's complement*, is 0xfe as a byte, 0xfffe as a word, 0xfffffffe as a dword, and 0xfffffffffffffffe as a qword. If we were not using two's complement, then we could not express −2 in hex on a computer, and the values 0xfe, 0xfffe, 0xfffffffe and 0xfffffffffffffffe would have their normal meanings.

Now, we will encode the largest possible negative number for a byte, which is −128. To do this, we first make it positive, so it becomes +128 in decimal, which, in binary is:
1000000

We might think this number is too large to be encoded because it uses the leftmost bit. However, the next step is to subtract 1, so we end up with:
01111111

Now the number does not use the leftmost bit. This is how the negative numbers for a byte can reach down to −128, but the positive values can only reach up to +127.

We then flip the bits:
10000000

This is 0x80, so we can say that the decimal number −128 as a two's complement number is 0x80.

Thoughts

Any hexadecimal two's complement number that starts with 8, 9, a, b, c, d, e or f must be a negative number – for it to start with one of those digits, it must have the leftmost bit set to 1:

8 is 1000
9 is 1001
a is 1010
b is 1011
c is 1100
d is 1101
e is 1110
f is 1111

Any two's complement number that starts with 0, 1, 2, 3, 4, 5, 6, or 7 must be a positive number – for it to start with one of those digits, it must have the leftmost bit set to 0:

0 is 0000
1 is 0001
2 is 0010
3 is 0011
4 is 0100
5 is 0101
6 is 0110
7 is 0111

From this we know that 0xa4, for example, must be a negative number and that 0x74, for example, must be a positive number *when those numbers are intended to be, and being interpreted as, two's complement numbers.* If the numbers are not intended to be two's complement numbers, or they are not being interpreted as two's complement numbers, then both 0xa4 and 0x74 will be positive numbers, as all non-two's complement numbers are.

More examples

Now we will look at −7 as a two's complement *dword*. First, we make it positive, and write it as a binary dword: [I have put in spaces to make it easier to read]
00000000 00000000 00000000 00000111

We subtract 1:
00000000 00000000 00000000 00000110

Then we flip the bits:
11111111 11111111 11111111 11111001

This is 0xfffffff9 in hex. Therefore, −7 in decimal is 0xfffffff9 in hexadecimal as a two's complement dword. [As a byte, it would be 0xf9. As a word, it would be 0xfff9.]

Another method

Another way to calculate a two's complement number is to take the negative number we want to express, make it positive, subtract 1, and then subtract the number from 0xff (if we are using bytes), from 0xffff (if we are using words), from 0xffffffff (if we are using dwords), or from 0xffffffffffffffff (if we are using qwords).

As an example, we will say that we want to portray the hex number −0x67 as a two's complement byte. We make it positive as +0x67, then subtract 1 to get 0x66, and then subtract that from 0xff. We end up with 0x99.

This method works because subtracting a binary number from a second binary number that is all 1s has the same effect as flipping the bits of the first number. For −0x67 as a byte, we would be calculating 0xff minus 0x66, which is:

11111111
... minus:
01100110
... which is:
10011001
... which is:
0x99 in hex.

Whether this method is better than the bit flipping method depends on what it is we are doing. If we have a hexadecimal calculator, and one that works only with positive numbers, this method might be quicker than the bit flipping method. [However, if we have a hexadecimal calculator, it is likely to be able to convert negative numbers into two's complement numbers anyway.]

Converting away from two's complement

Converting from a two's complement number to a normal number is easy. If the two's complement number is positive (it has the leftmost bit set to 0), then we leave it as it is. If the two's complement number is negative (it has the leftmost bit set to 1), we proceed as follows:

- We flip all the bits
- We add 1
- We put a minus sign in front of the number

As an example, we will convert 0xd5 to a non-two's complement number. As the hexadecimal digit "d" is 1101 in binary, we can tell that this is a negative number – the leftmost bit is 1. The number in full is 11010101. We flip all the bits to produce:
00101010
... then we add 1:
00101011
... then we put a minus sign in front of the number:
−00101011

This is the same as −0x2b in hexadecimal, which is −43 in decimal.

Why the system is how it is

If we are given a two's complement number *and we know that it is supposed to be a two's complement number*, we can tell if it is a negative number or not by whether the leftmost bit is set to 1 or not. A negative number always has the leftmost bit set to 1, and a positive number always has the leftmost bit set to 0.

When converting negative numbers to two's complement, we start by making the number positive and then subtracting 1. This is how the generally agreed system works. You might think that an easier system would keep the number the same and set the leftmost bit to 1 for a negative number, and set it to 0 for a positive

number. However, doing this would not make full use of the available numbers. For an 8-bit byte, we would be able to count down to 1111111, which in the new system would be −127, and we would be able to count up to 01111111, which in the new system (and in two's complement) would be +127. However, we would have two numbers being used for zero: 10000000 and 00000000, which would be negative zero and positive zero. The two's complement system makes the best use of the available space by not having two zeroes, and by having one extra negative number. We can count from −128 up to +127. The system is ever so slightly more complicated than it could be, but it enables us to store one more number. It also allows us to perform addition with negative numbers more easily, as we will see shortly.

Flipping the bits

The simplest way to flip the bits of a hexadecimal number is to convert it to binary [each hex digit is turned into 4 binary digits], then flip the bits, and then convert the result back to hex [each group of 4 binary bits are turned into 1 hex digit.] Alternatively, it can be quicker to remember which hex digit becomes which hex digit when the bits are flipped, as shown in this table:

Hex digit	**Hex digit after the bits have been flipped**
0	f
1	e
2	d
3	c
4	b
5	a
6	9
7	8
8	7
9	6
a	5
b	4
c	3
d	2
e	1
f	0

A rule for remembering the table is that 0x7 becomes 0x8, and 0x8 becomes 0x7. The values either side become turned into the values the other side of 7 or 8. We only really need to know half the table as the top half is mirrored in the bottom half. If we learn the table, we will be able to know instantly, for example, that if the bits are flipped in 0x053e2076, we will end up with 0xfac1df89. If you were going to be converting a lot of numbers without a calculator, it might be useful to learn the table.

Maths

One reason that two's complement is a useful system is that we can perform addition and subtraction on two two's complement numbers without needing to know if they are two's complement or not, and the results will still be correct. As a simple example, we will add −1 (0xff) and +1 (0x01). As binary, these are:
11111111
... added to:
00000001

If we were dealing with *words, dwords or qwords*, the above addition would result in the nine-digit number:
100000000
However, as we are dealing in *bytes,* and bytes only have 8 bits, the whole number rolls over to zero. Therefore, the result is 00000000 in binary, which is 0x00 in hex.

We will add the bytes −3 (0xfd) and −4 (0xfc). This addition is:
11111101
... added to:
11111100

The result is 9 bits long:
111111001
... but because we are dealing with bytes, this is truncated to 8 bits by ignoring the ninth bit, as so:
11111001
... which is 0xf9 in hex, which, *as a two's complement number*, is −7 in decimal.

We will add the two's complement numbers 0xf7 and 0x23. *As two's complement numbers*, these are −9 and +35 in decimal. In binary, the calculation is as so:

11110111
... added to:
00100011
... which is the 9-digit number:
100011010
... but as we are dealing in bytes, we truncate it to 8 bits, and have:
00011010

We can instantly tell that this is a positive number because the leftmost bit is zero. This result is 0x1a in hex, which, whether we treat it as a two's complement number or not, is +26 in decimal.

More on negative numbers

In programming, a common term for a two's complement number is a "signed integer", where the word "signed" means that the decimal equivalent would be portrayed with a plus sign or a minus sign. A normal number would be called an "unsigned integer". A signed integer might be positive or negative, and will always be portrayed using the two's complement system. On the other hand, an unsigned integer will always be positive, and will always be portrayed normally with binary or hex numbers.

In higher-level programming languages, the details of what a two's complement number is, or how it is encoded, are generally hidden from the programmer. If you are dealing with files containing data stored as two's complement numbers, it is helpful to have at least a passing understanding of how they work.

Although two's complement seems like a slightly contrived way of encoding negative integers, computer processors have specific commands that work with them, and the system is generally accepted.

There are two likely ways in which using two's complement numbers will lead to mistakes:

- The first is not knowing whether a byte, word, dword or qword should be interpreted as being a two's complement number or not. A value that should be interpreted as a negative number represents a completely different number if it is treated normally. The most extreme example of this is 0xffffffffffffffff, which, as a two's complement qword, is billions of times smaller in magnitude than it is as a normal number.

  Whether a byte, word, dword or qword is acting as a two's complement number relates only to whether someone wants to treat it as such. For a byte sitting in memory, there is literally no difference between 0xff meaning "255" in decimal, and 0xff meaning "−1" in decimal. They are both 0xff. You cannot tell if a byte is meant to be a signed integer or not by looking at it. The processor itself does not even know unless it is in the middle of operating on such a value. However, if you know that the byte is a signed integer because you created it yourself, for example, or because you know the context of how it is being used, you can perform maths on it that uses this knowledge. Similarly, once a computer processor is told to act as if the byte is a signed integer, it can treat it accordingly.

- The second way of making mistakes is to forget that the highest possible positive number in a two's complement number is 1 less than half of what it would be normally. For example, a byte can normally count up to 0xff (255 in decimal), but if it is being used to hold a two's complement number, the maximum is only 0x7f (127 in decimal).

  If 1 were added to 0x7f as a *normal* byte, we would get 0x80, which is +128 in decimal.

  If we add 1 to 0x7f, *when 0x7f is being treated as a two's complement number*, *and whatever it is that is doing the adding knows this*, we will get 0x00 because we would have gone past the maximum possible positive value, and rolled over to zero. A computer processor would keep the number positive and not let it go above 0x7f.

  If we add 1 to 0x7f, *when it is being treated as a two's complement number*, *but whatever is doing the adding does *not* know this*, we would end up with 0x80, which, in two's complement, is −128. Instead of the number increasing by 1, it has fallen by 255.

The largest positive number we can have in a two's complement signed byte is:
+127 in decimal, which is 0x7f in hex and 01111111 in binary.

The largest negative number we can have in a two's complement signed byte is:
−128 in decimal, which is 0x80 in hex and 10000000 in binary.

If we ignore zero, the *smallest* positive number we can have in a two's complement signed byte is:
+1 in decimal, which is 0x01 in hex and 00000001 in binary.

The smallest negative number we can have in a two's complement signed byte is:
−1 in decimal, which is 0xff in hex and 11111111 in binary.

Non-integers

We have just seen how computers store negative numbers in hexadecimal and binary. The method is essentially a contrived workaround that redefines how hexadecimal and binary normally work. Now we will look at the standard way that computers store non-integers using hex and binary. This also redefines how hex and binary normally work.

A computer stores a non-integer using binary or hex by treating the number as an exponential. More specifically, it rephrases the number so that it becomes a single digit value followed by a decimal point and any number of digits afterwards, multiplied by "2 raised to a particular power". [Strictly speaking, it is not a decimal point, but a *binary* point because we will be working in binary.]

Throughout this explanation, we will get closer and closer to the actual implementation used by computer processors, however to keep the explanation simple, we will advance step by step.

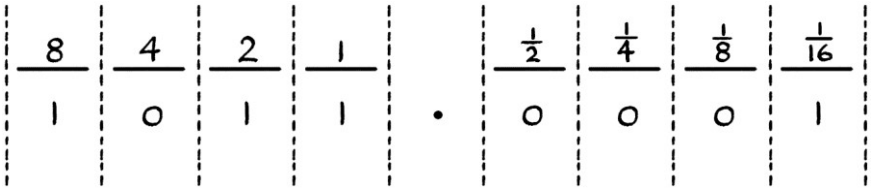
The concept of binary non-integers

Although a computer processor has to use a contrived way to deal with binary non-integers, away from computers, we can portray them in the same way that we would portray decimal non-integers – as an integer followed by a *binary* point followed by the binary digits of the fraction part. [A decimal point is used in decimal numbers and a binary point is used in binary numbers.] For example,

away from a computer, “1011.0001” would be a perfectly valid way to express a binary number. However, a computer processor would be unable to use this method because the processor works with bytes, words, dwords and qwords, and there is no way to place a binary point in the middle of one of these bit groupings.

When thinking about a number such as “1011.0001”, it helps to remember the binary number columns. We know about the integer columns that extend to the left, becoming ever higher: ones, twos, fours, eights, sixteens and so on. The “fraction” columns extend to the right, becoming ever smaller: halves, quarters, eighths, sixteenths, thirty-seconds, sixty-fourths and so on. The number columns to the right of the binary point are all 1 divided by a power of 2.

The binary number “1011.0001” can be thought of as it appears in the following picture:



When we think of a *decimal* number with digits after the decimal point, we are really saying that its fraction part is the sum of so many units of one column, added to so many units of another column, added to so many units of another column, and so on. Given that the number columns after the decimal point are tenths, hundredths, thousandths, ten thousandths and so on, the decimal number 0.7903, for example, really means the sum of 7 lots of tenths added to 9 lots of hundredths, added to zero lots of thousandths, added to 3 lots of ten thousandths. We could express the sum as so:
7 * 0.1
... added to:
9 * 0.01
... added to:
0 * 0.001
... added to:
3 * 0.0001

The nature of storing fractions in this way means that if the fraction part of a number cannot be created by adding multiples of tenths, hundredths, thousandths, ten thousandths and so on, then it cannot be expressed by a finite number of decimal places. For example, we cannot express the number created by dividing 1 by 3 using a finite number of decimal places. This is because 1 divided by 3 cannot

be expressed as a sum of tenths, hundredths, thousandths and so on. Expressing 1 divided by 3 ends up with an infinite number of decimal places: 0.3333333333333... and so on forever.

Storing *binary* numbers with fractions is similar to storing decimal numbers with fractions. If we have the binary number 0.111111, then it is really the sum of:
1 * 0.5
... added to:
1 * 0.25
... added to:
1 * 0.125
... added to:
1 * 0.0625
... added to:
1 * 0.03125
... added to:
1 * 0.015625

If we try to encode any non-integer in binary, then it must be possible to express the part after the binary point as the sum of one or more of the following:

0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.00390625
0.001953125
0.0009765625
... and so on.

If the fraction part cannot be expressed as the sum of one or more of those values, then it will require an infinite number of digits after the binary point. Even if it can be expressed as the sum of one or more of those values, it might need many more digits after the binary point than the same number expressed in decimal would require after the decimal point.

Converting decimal non-integers to binary

For this section, we need to remember how to convert a decimal *integer* into a binary *integer*, as mentioned earlier in this chapter. The method is as follows:

- We divide the integer by 2. We take the remainder, which will be 0 or 1, and put it into the ones column.
- If the integer part of the division was not zero, we divide it by 2. We take the remainder, which again, will be 0 or 1, and we put it into the twos column.
- If the integer part of the previous division was not zero, we divide it by 2. We take the remainder, which again, will be either be 0 or 1, and we put it into the fours column.
- We continue in this way until we reach a time when the integer part of the previous division was zero.

[If it seems as if the resulting binary number will be very long, it is easier to convert a decimal number into hex, and then convert the hex number into binary.]

The simplest way to convert a decimal non-integer to binary is to keep multiplying the decimal number by 2 until it becomes an integer, and then to convert that integer into a binary integer. We then divide the binary integer by the amount by which we scaled the original number. This division will always be a power of 2.

Dividing a binary number by a power of 2 is easier than it sounds – we just slide the binary point one digit to the left for each power of 2. For example, if we needed to multiply our decimal non-integer by 2 ten times to create an integer, we would need to divide the equivalent binary integer by 2 ten times, which we would do by sliding the binary point to the left by ten digits.

As an example, we will convert 34.25 to binary. First, we turn it into an integer, by multiplying it by 2 two times. We end up with 137. We then convert 137 into binary. It is 10001001. As we multiplied our original number by 2 twice to make it into an integer, we need to *divide* this number by 2 twice. This means we just move the binary point two digits to the left. We end up with 100010.01 as our result.

Although it is simple to do, the "multiplying by powers of two to turn a number into an integer, then dividing the binary equivalent by powers of two" method reveals a particular problem with converting non-integers into binary in general. For example, if we had the decimal number 34.55, it would require multiplying by two 42 times before it became an integer. It would become 151,952,506,958,643. We would have to convert that into a binary number, and then shift the binary

point 42 digits to the left. This instantly tells us that the binary number would have 42 digits after the binary point.

The seemingly simple *decimal* number 0.1 requires multiplying by 2 fifty times before it becomes an integer. This means that the binary number would need to be divided by 2 fifty times, and so the result would have 50 digits after the binary point.

Another problem is that it might not always be possible to convert a number into an integer by multiplying by 2. As a simple example, no matter how many times we multiply a third by 2, it will never become an integer.

These problems are not related to this method of converting from decimal to binary. Instead, they relate to how it is sometimes difficult or impossible to represent a non-integer in binary correctly, in the same way that it is sometimes difficult or impossible to represent a non-integer in decimal correctly.

A good example of a number that cannot be portrayed in any (sensible) number system is π. To portray the fraction part of π would require an infinite number of digits, no matter whether we used decimal, binary, hexadecimal, or any number system based on integers. [We could use a number system where the number columns were somehow based on multiples of π, but such a numbering system would be inaccurate when portraying any numbers that were not related to π.]

These problems are sometimes misinterpreted as a flaw in the way that computers store numbers, when in reality, they are just a consequence of using binary. Problems of this type would occur no matter which number system we used.

If it turns out that we need to multiply our decimal non-integer by 2 a huge number of times to turn it into an integer (or an infinite number of times), then we have to make the choice of where to truncate the digits after the binary point. This is always an arbitrary choice and depends on what we are doing. The rule for rounding up a *decimal* number is that if the digit to the right of where we are truncating it is 5 or more, the digit to the left has 1 added to it. Otherwise, the digit is left alone. For example:

7.00015 becomes 7.0002 to four decimal places.
7.00014 becomes 7.0001 to four decimal places.

The rule for binary is that if the digit to the right of where we are truncating is 1, we add 1 to the digit to the left. Otherwise, we leave the digit to the left alone. For example:

1.00001 becomes 1.0001 to four binary places.
1.00000 becomes 1.0000 to four binary places.
1.00011 becomes 1.0010 to four binary places.
1.00010 becomes 1.0001 to four binary places.

Non-integers for computers

We will now move on to how computer processors store and work with binary non-integers. The exact details are easiest to understand if we first go through the method with *decimal* non-integers.

When thinking in decimal, any number except zero can be rephrased to be a non-zero single digit followed by a decimal point and any number of digits afterwards, multiplied by ten raised to a particular power. [We will think about how to encode zero later.] For example:

300 can be rephrased to be $3 * 10^2$
7000 is $7 * 10^3$
20 is $2 * 10^1$
330 is $3.3 * 10^2$
7078 is $7.078 * 10^3$
22 is $2.2 * 10^1$

1234 is $1.234 * 10^3$
12.34 is $1.234 * 10^1$
1.234 is $1.234 * 10^0$
0.1234 is $1.234 * 10^{-1}$
0.01234 is $1.234 * 10^{-2}$
0.000000001234 is $1.234 * 10^{-9}$
12,340,000 is $1.234 * 10^7$

−77 is $-7.7 * 10^2$
−314,159 is $-3.14159 * 10^5$
−11.0000001 is $-1.10000001 * 10^1$

From all of these examples, we can see that whether or not a decimal number is an integer, once we have rephrased it, there are four identifying characteristics:

- Whether it is positive or negative.
- The single digit before the decimal point.
- The digits after the decimal point.
- The exponent of the number 10.

These four attributes are enough to identify any possible number except zero.

For example, if we have the number −0.1234, it can be rephrased to be: $-1.234 * 10^{-1}$. The identifying characteristics are:

- It is negative. We could say that its “sign” is negative.
- The value of the digit before the decimal point is 1.
- The digits after the decimal point are 234.
- The exponent of the number 10 is −1.

The number 0.00000007954 can be rephrased to be $7.954 * 10^{-8}$. The identifying characteristics are:

- It is positive. We could say that its “sign” is positive.
- The value of the digit before the decimal point is 7.
- The digits after the decimal point are 954.
- The exponent of the number 10 is −8.

The number 2,000,007 can be rephrased to be $2.000007 * 10^{6}$. The identifying characteristics are:

- It is positive.
- The value of the digit before the decimal point is 2.
- The digits after the decimal point are 000007.
- The exponent of the number ten is 6.

The number −99,999.12 is $-9.999912 * 10^{4}$. The identifying characteristics are:

- It is negative.
- The value of the digit before the decimal point is 9.
- The digits after the decimal point are 999912.
- The exponent of the number ten is 4.

For each of these examples, we can write out the identifying characteristics as a line of text:

−0.1234: negative, 1, 234, −1
0.00000007954: positive, 7, 954, −8
2,000,007: positive, 2, 000007, 6
−99,999.12: negative, 9, 999912, 4

Knowing just "negative, 1, 234, −1" is enough to know that we are talking about the number −0.1234. There is no other number for which these four characteristics apply.

The system in binary

We can use the same idea in binary. When thinking in binary, any number except zero can be rephrased to be a single non-zero digit followed by a *binary* point and a number of *binary* digits afterwards, multiplied by 2 raised to a particular power.

[It always helps to know that we can multiply a binary number by 2 by shifting the binary point one digit to the right. We can divide a binary number by 2 by shifting the binary point one digit to the left.]

To make the idea easier to understand, we will first look at some examples with the values as binary numbers, *but with the exponentials kept as decimal numbers*. This makes it easier to see how far the binary point has been moved. [Mathematically, it could be confusing to have calculations containing both binary and decimal values, but for the sake of the explanation, it makes things easier to understand.]

1100 can be rephrased to be $1.1 * 2^{3}$
10001111 can be rephrased to be $1.0001111 * 2^{7}$
1011 can be rephrased to be $1.011 * 2^{3}$
−11001100 can be rephrased to be $-1.1001100 * 2^{7}$

1100.001 can be rephrased to be $1.100001 * 2^{3}$
−0.0111 can be rephrased to be $-1.11 * 2^{-2}$
−0.00000001 can be rephrased to be $-1 * 2^{-8}$
11.1111 can be rephrased to be $1.11111 * 2^{1}$

We can identify any binary number (except zero) using just 4 attributes:

- Whether it is positive or negative.
- The value of the digit before the binary point.
- The binary digits after the binary point.
- The exponent of the number 2.

[We will see how to encode zero later in this chapter.]

For example, −0.0111 can be rephrased as $-1.11 * 2^{-2}$. It can be identified with these four attributes:

- It is negative.
- The digit before the binary point is 1.
- The digits after the binary point are 11.
- The exponent of the number 2 is −2.

The number 11.1111 can be rephrased to be $1.11111 * 2^{1}$. It can therefore be identified by these four attributes:

- It is positive.
- The digit before the binary point is 1.
- The digits after the binary point are 11111.
- The exponent of the number 2 is 1.

One interesting consequence of doing this in binary is that the single digit before the binary point will *always* be 1. This is because we always slide the number to have one non-zero digit before the binary point, and binary only has one type of non-zero digit, which is 1. As the digit before the binary point will always be 1, we do not need to bother making a note of it. Its existence is implied. Therefore, we can encode any binary number using just three attributes:

- Whether it is positive or negative.
- The binary digits after the binary point.
- The exponent of the number 2.

For example, 1011 is also $1.011 * 2^3$. The three attributes of this are:

- It is positive.
- The digits after the binary points are 011.
- The exponent of the number 2 is 3.

We will now make this explanation one step more complicated by giving the exponentials in binary too. First, this means the base of the exponential changes from being "2" in decimal to being "10" in binary. Second, the exponent becomes a binary number too. The previous binary examples rephrased to have binary exponentials are:

1100 is $1.1 * (10)^{11}$
10001111 is $1.0001111 * (10)^{111}$
1011 is $1.011 * (10)^{11}$
−11001100 is $-1.1001100 * (10)^{111}$
1100.001 is $1.100001 * (10)^{11}$
−0.0111 is $-1.11 * (10)^{-10}$
−0.00000001 is $-1 * (10)^{-1000}$
11.1111 is $1.11111 * (10)^{1}$

[Note that because we are writing the binary numbers and exponentials out by hand, we can still use negative signs with them. A computer would not be able to do this, and would have to use two's complement. This is a potential source of confusion. Away from computers, we have much more leeway in writing binary – we can use binary points and negative signs. When a computer deals with non-integers and negative values, it has to use the contrived alternatives.]

We will look at one of these examples. The binary number 1100.001 can also be portrayed as $1.100001 * (10)^{11}$, where all the values are in binary. The three attributes that distinguish this number from any other number are therefore:

- It is positive.
- The binary digits after the binary point are 100001.
- The exponent of the binary number 10 is 11 (in binary), which is 3 in decimal.

We can summarise all of the examples with their sign (whether they are positive or negative), the digits after the binary point, and the exponent:

1100:	positive, 1, 11
10001111:	positive, 0001111, 111
1011:	positive, 011, 11
−11001100:	negative, 1001100, 111
1100.001:	positive, 100001, 11
−0.0111:	negative, 11, −10
−0.00000001:	negative, 0, −1000
11.1111:	positive, 11111, 1

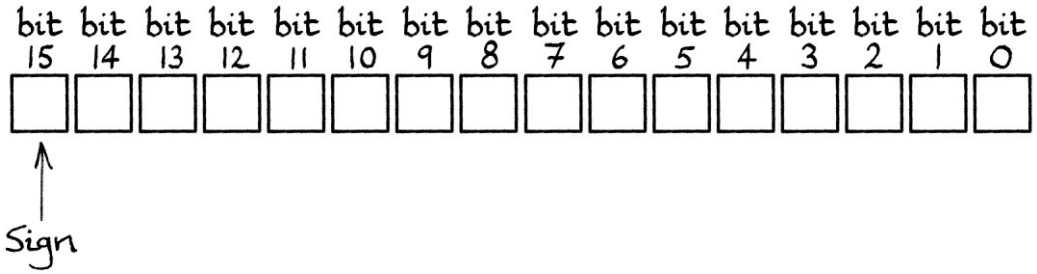
Floating-point numbers

The method of identifying any number apart from zero by the three attributes is the basis of how computer processors store and work with non-integers. Processors store the three attributes in what is called a "floating-point" number. We can think of the process of obtaining the three attributes as *sliding* the binary point so that we have one digit before the binary point. We could also think of the binary point as *floating* left or right. A floating-point number can indicate any number at all, whether it is an integer or a non-integer, and whether it is positive or negative.

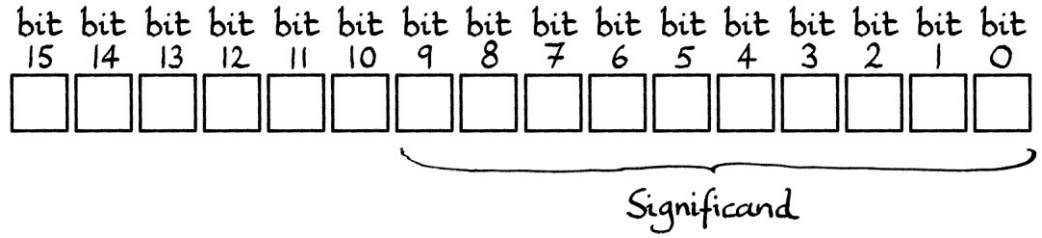
When a value is encoded as a floating-point number, the three attributes are joined together so that they fit within particularly sized groups of bits. At the time of writing, for Intel and AMD 64-bit processors, the minimum sized group of bits is a 16-bit word. As this is the shortest size to contain a floating-point number, we will look at these numbers first.

Of our three attributes of a number, the first refers to whether the number is positive or negative. This is called "the sign", because it relates to the "plus sign" or "minus sign" that would normally be used to prefix a number. The sign is represented by the leftmost bit of the 16-bit word. If there is a plus sign, then this is set to zero, if there is a minus sign, it is set to one. Thinking of this the other way around, if the leftmost bit of the word is zero, the number is positive; if the leftmost bit is one, the number is negative.

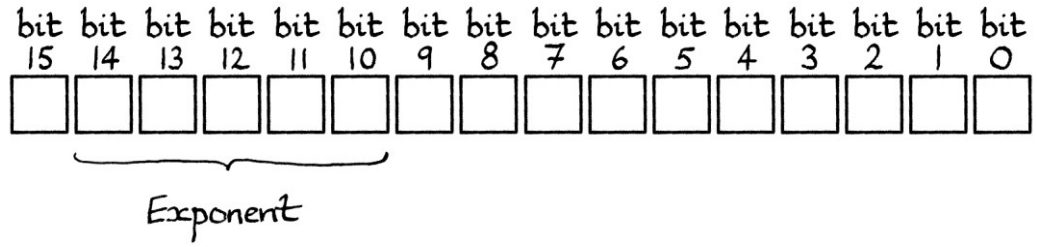
In the following picture, each of the boxes represents a binary digit. The sign is the leftmost bit (which is bit 15):



The second attribute refers to the digits after the binary point. It is called the "significand" on account of it being considered the "significant" part of the number in this system. In a floating-point number contained within a 16-bit word, this is placed in the rightmost part from bit 9 to bit 0 [Remember that we number the bits from right to left, and count from zero upwards]. This means that it will be ten bits long. To fit the significand into the space available, it might have zeroes placed after it, or it might be truncated or rounded up.

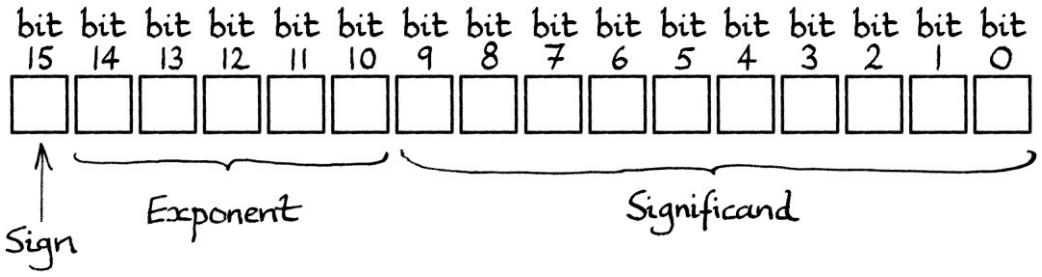


The third attribute refers to the exponent of the exponential in binary. In a floating-point number contained within a 16-bit word, this is placed in the middle, from bit 14 to bit 10. This means that it will be 5 bits long. In practice, the actual exponent is adjusted before it is stored, as we will see shortly.



The full layout of a floating-point number contains the following, in this order:

- the sign
- the exponent
- the significand (in other words, the value after the binary point).



The exponent

Among the possible problems we can see with this system so far is that the exponent might need to be negative. Therefore, we have to decide how to encode the exponent as a negative binary number. One way would be to use two's complement. In practice, however, a fixed value is added to the exponent so that it always becomes positive. [That value is subtracted when the floating-point number is decoded.] The value added to the exponent varies depending on how many bits we are using to store the floating-point number. For a 16-bit word, where the exponent is contained within 5 bits, this number will be 15 (in decimal), which is 01111 in binary. [Note that I am writing it as 01111 instead of 1111 so that it takes up 5 bits, and so removes any confusion as to what happens if we end up with just 1111, which is what would happen if the exponent were zero.]

For 16-bit words, whatever the exponent, it will have 15 added to it to make it into a positive number between, and including, 00001 and 11110 in binary [which are 0x01 and 0x1e in hex, or 1 and 30 in decimal.] It is important that the exponent ends up as a number between 00001 and 11110 and not between 00000 and 11111. This is because exponents that are all zeroes or all ones have special meanings, as we shall see later. This restriction means that the original exponent before the addition can only be between −14 and +15. Therefore, we can only encode numbers that have an exponential between 2^{-14} and 2^{15}. In binary, we would say that they can only be between $(10)^{-1110}$ and $(10)^{1111}$.

The value that is always added to the exponent is called the "bias". We can say that the "exponent is biased", where "bias" in this sense ultimately means a constant distortion in one direction.

Here are some examples of storing the exponent as a "biased" value – in other words, they are examples of storing the exponent with the addition of the fixed value of 15. When using 16-bit words to encode floating-point numbers, there are 5 available bits for the biased exponent.

Original number and exponential (in binary)	**Exponent (in binary)**	**Exponent after the addition of 01111 (in binary) [The biased exponent]**
$-1.1010101 * (10)^{1}$	1	10000
$1.11101 * (10)^{101}$	101	10100
$1.001 * (10)^{1000}$	1000	10111
$1.11 * (10)^{-1000}$	−1000	00111
$1.1 * (10)^{-1110}$	−1110	00001
$1.1 * (10)^{1111}$	1111	11110

The full layout

For a floating-point 16-bit word, the first bit holds the sign, the next 5 bits hold the exponent, and the next 10 bits hold the significand. We can portray this with a template that indicates the 16 bits with letters standing in for the bits:

sEEEEEnnnnnnnnnn

This is a template that we can use to fill in the bits. The "s" (for "sign") will be replaced with the sign bit. The five letters "EEEEE" (for "Exponent") will be replaced by the biased exponent. The ten letters "nnnnnnnnnn" (for "significand") will be replaced by the significand. The choice of letters is arbitrary, but these are easy to distinguish from each other. The template is just to make this explanation easier to understand, and normally such a thing would not be used.

Using the template

We will look at a complete example. We start with our empty 16-bit word template:
sEEEEEnnnnnnnnnn

We will look at the first example in the previous table: −1.1010101 * $(10)^1$. The sign is negative, so we set the leftmost bit to 1:
1EEEEEnnnnnnnnnn

The exponent of our number is 1. Therefore, we add it to 01111 (in binary), and we end up with the binary number 10000. This then goes into the exponent section ["EEEEE"] of our 16-bit word, which is from bit 14 to bit 9. Our word so far will be:
110000nnnnnnnnnn

The significand (containing the digits after the binary point) is: 1010101. There are only 7 digits, but the section for the significand for a 16-bit word is ten digits long. Therefore, we extend our binary digits to ten digits by putting zeroes *at the end*, as so: 1010101000. These extra digits do not affect the number that we are encoding in any way. These digits are then placed into the remaining bits as so:
1100001010101000

[If there had been more than ten digits after the binary point, we would have had to round up or truncate the digits.]

We now have our finished number:
1100001010101000
... which, given spaces to make it easier to read, is:
1100 0010 1010 1000
... and this is:
0xc2a8
... in hexadecimal.

Our final 16-bit floating-point number is 0xc2a8. As with two's complement numbers, this is only a floating-point number *if we treat it as such*. There is no way of knowing that this 16-bit word is a floating-point number by looking at it. Floating-point numbers require context to be interpreted correctly. With no context, we might treat this as a normal unsigned integer word, in which case, it would be 49,832 in decimal. If we mistakenly thought it was a signed two's complement integer, it would have the value −0x3d58, which is −15,704 in decimal. From the point of view of computers, 0xc2a8 only means −1.1010101 * $(10)^1$ if the computer processor is told that 0xc2a8 is a floating-point number when it is supplied with the number.

16-bit example 1

We will now look at several examples.

We will say that we want to convert the binary number:
1100.0011
... into a 16-bit floating-point number. First, we arrange it to be a single digit preceding the binary point multiplied by an exponent of 2. If we give the exponential entirely in binary, we will have:
$1.1000011 * (10)^{11}$

We split this into the three parts that make up a floating-point number:

- The sign is positive.
- The significand (the digits after the binary point) is 1000011.
- The exponent is 11 in binary (which is 3 in decimal).

We will start with our 16-bit template:
sEEEEEnnnnnnnnnn

As the sign is positive, the leftmost bit is set to zero:
0EEEEEnnnnnnnnnn

Our exponent is 11, but we need to turn this into a biased exponent by adding 01111. This gives us 10010. We can put this into the exponent section:
010010nnnnnnnnnn

Our significand is 1000011. This is 7 digits long, but we have space for 10 digits. Therefore, we extend it to ten digits by putting zeroes *at the end*. We end up with this: 1000011000. We can then place this into the significand section:
0100101000011000

We now have our finished binary word. We will convert it into hexadecimal – first, we split it into 4-bit groups to make it easier to read:
0100 1010 0001 1000
... then we convert each 4-bit group into the relevant hex digit:
4 a 1 8
... and our final hex word is:
0x4a18

16-bit example 2

Next, we will convert −1111000011 to a floating-point number. Rephrased to have a single digit integer part and to be multiplied by an exponential, we have:
$-1.111000011 * (10)^{1001}$

- The sign is negative
- The significand is 111000011. This is 9 digits long, but we need it to be 10 digits long, so we put a zero at the end: 1110000110.
- The exponent is 1001. Therefore, the biased exponent will be:
  1001 + 01111 = 11000.

We will start with our 16-bit template:
sEEEEEnnnnnnnnnn

As the sign is negative, we set the leftmost bit to 1:
1EEEEEnnnnnnnnnn

We then fill in the exponent part with our biased exponent 11000:
111000nnnnnnnnnn

We then fill in the significand:
1110001110000110

This is our final floating-point number, but we will convert it to hex. First, we split it into 4-bit sections:
1110 0011 1000 0110
... then we convert each 4-bit section into one hex digit:
e 3 8 6
... which is:
0xe386

16-bit example 3

Now we will convert the *decimal* number 123.25 into a 16-bit binary or hex floating-point number. First, we have to convert the decimal number into a binary number with a binary point. To do this, we keep multiplying the decimal number by 2 until it becomes an integer:
123.25 * 2 = 246.5
246.5 * 2 = 493

Therefore, we have to convert 493 into binary. As this looks as if it will produce a long binary number, we will convert it into hex first, and then convert the result into binary. Doing this is quicker and simpler than converting directly to binary. First, we divide the number by 16. This is 30.8125, which is 30 and a remainder of 13. We convert 13 to hex – it is 0xd – and put it into the 1s column. We then divide 30 by 16, and we get 1.875, which is 1 and a remainder of 14. We convert 14 to hex – it is 0xe – and put it in the 16s column. We then divide 1 by 16, and get 0 and a remainder of 1. We put the 1 into the 256s column, and we have finished the conversion into hex. The result is:
0x1ed
... which is:
0001 1110 1101 in binary (with spaces to make it easier to read)
... or:
000111101101 (without the spaces)
... or:
111101101 (without the preceding zeroes).

As we multiplied our original number (123.25) by two twice to turn it into an integer, we need to divide our binary number by two twice. We do this by moving the decimal point two digits to the left to produce:
1111011.01

We now have our binary number ready for conversion to a floating-point number. We alter it so the integer part is only one digit, and it is multiplied by an exponential with 2 as the base. Entirely in binary, this is:
$1.11101101 * (10)^{110}$
[where the binary number 110 is equal to the decimal number 6.]

The three attributes of this number are:

- The sign is positive.
- The significand is 11101101
- The exponent is 110

We take our 16-bit template:
sEEEEEnnnnnnnnnn

The sign bit will be 0 because the number is positive:
0EEEEEnnnnnnnnnn

The exponent needs biasing, so we add 01111 to it:
01111 + 110 = 10101

We then put that into our template:
010101nnnnnnnnnn

The significand is 11101101, which is 8 bits long. As it needs to be ten digits long, we put two zeroes onto the end to produce 1110110100. We then put this into our template, and we have the finished floating-point number:
0101011110110100

We will split this into 4-bit sections to make it easier to read:
0101 0111 1011 0100
... and then we convert each 4-bit section to hex:
5 7 b 4
... and our final floating-point number as a hex word is:
0x57b4

16-bit example 4

We will convert the *binary* number −1.1 into a floating-point number. Portraying this as a single digit integer, followed by a fraction and an exponential, it becomes:
$-1.1 * (10)^0$

- The sign is negative
- The significand is 1, which we will extend to 10 digits by putting nine zeroes after it: 1000000000
- The exponent is zero, to which we will add 01111 to become 01111

We take our 16-bit template:
sEEEEEnnnnnnnnnn

The sign is negative, so the leftmost bit is set to 1:
1EEEEEnnnnnnnnnn

We put the biased exponent in:
101111nnnnnnnnnn

We put the significand in, and we have our finished floating-point number in binary:
1011111000000000

To convert this into hex, we split the number into 4-bit sections:
1011 1110 0000 0000
... then we convert each 4-bit section to a hex digit:
b e 0 0
... and our floating-point number in hex is:
0xbe00

16-bit example 5

Now we will convert 1001.111100001010101 into a floating-point number. As a value multiplied by an exponential, this is:
$1.001111100001010101 * (10)^{11}$
... where the binary exponent 11 is in 3 in decimal.

We start with our template:
sEEEEEnnnnnnnnnn

- The sign is positive, so the sign bit will be zero:
  0EEEEEnnnnnnnnnn

- The exponent is 11 in binary. We add this to 01111, and we get 10100, which we put into the template:
  010100nnnnnnnnnn

- The significand is 0011111000010101010. The maximum number of digits we can have for the significand is ten. Our significand is eighteen digits long. Therefore, we have to round it to ten digits, which means that we will lose some accuracy. This is an unavoidable consequence of floating-point numbers, but is more likely to happen when we are using 16-bit floating-point numbers as we are doing here. A 32-bit, 64-bit, or 80-bit floating-point number can contain more digits in the significand. Our significand rounded up to ten digits is 0011111000. We put this into the template:
  0101000011111000

Our final floating-point number in binary is:
0101000011111000

To convert this into hex, we split it into 4-bit sections:
0101 0000 1111 1000
... and convert each one to a single hex digit:
5 0 f 8
... so our final floating-point number in hex is:
0x50f8

16-bit example 6

We will now encode the following binary number as a floating-point number [shown with spaces to make it easier to read]:
0.0000 0000 0000 01

This has 14 decimal places. As a value multiplied by an exponential, it becomes:
$1 * (10)^{-1110}$

As significands always show the digits to the right of the number 1, we will write it with some zeroes after the binary point to make it clearer and easier to work with:
$1.0000 * (10)^{-1110}$

- The sign is positive, so the sign bit will be zero.

- The significand is 0000 (although it would be equally valid to say that it is 0 or 00 or 000, or 000000, or any number of zeroes). We will extend it to ten digits as 0000000000.

- The exponent is −1110, to which we must add 01111. The addition is easiest if we think of it as 01111 – 1110 = 1. [If we were confused by binary maths, we could convert the numbers to decimal, then add them, then convert the result back to binary. We would be adding −14 and +15, which results in +1 in decimal, which is +1 in binary.] As the biased exponent needs to be 5 digits long, we will write this as 00001.

Out template is:
sEEEEEnnnnnnnnnn

We fill in the sign:
0EEEEEnnnnnnnnnn

We fill in the biased exponent:
000001nnnnnnnnnn

Then we fill in the significand, which is all zeroes, and our completed floating-point number is:
0000010000000000

We split this into 4-bit sections:
0000 0100 0000 0000
... and convert each one to a hex digit:
0 4 0 0
... and our floating-point number as a hex word is:
0x0400

Capacity

Example 5 showed how there is a limit to the number of digits after the binary point that we can store – the limit is ten digits. However, in example 6, we managed to store a number with 14 digits after the binary point. The difference is that in example 6, all except the last digit was zero.

It is difficult to say exactly how large or small a number we can store in a floating-point word. This is because the number of digits we can store depends on the nature of the bits in the number we have.

It is possible to store a 16 digit binary *integer* – we would actually be storing only 15 digits because the first 1 is implied. However, we would only be able to fit in the first ten digits after the first 1. The digits after the first 1 are rounded up to ten digits. If the digits after the first ten digits were already zeroes, this will not make any difference – we will be storing the number we have completely accurately. However, if the digits were not zeroes, we would be storing an approximation. This approximation can still be useful as the size of the stored number will be very similar to the number we started with – only the lower digits will be zeroed out. The idea is analogous to having, say, the decimal number 1,300,021 but only being able to store it as 1,300,000.

As an example, if we tried to store this 16 digit binary integer:
1111 0000 1111 1111
... as a 16-bit floating-point word, we would only be able to store the first ten digits after the first 1. It would end up being converted to:
1111 0000 1110 0000
This is because it would be rounded up to ten significant digits after the first 1. The first number is 61,695 in decimal. The second number is 61,664 in decimal. We still have a similarly high number, but it is not exactly the same.

We could store a 15 digit binary number if it is entirely a fraction less than 1. We would actually only need to store 14 digits because the first 1 is implied. However, we would only be able to store the first ten digits after the first 1 because the digits after the first 1 would be rounded up to ten decimal places. Again, depending on the number we are storing, this might not matter.

If we wanted to have the binary number:
0.1111 0000 1111 1111
... as a 16-bit floating-point word, it would end up being changed to:
0.1111 0000 1110 0000
... because it would be rounded up after the first ten decimal places after the first 1.

16-bit example 7

We will now decode a 16-bit floating-point number to find out the number it represents. We will decode: 0x8e23. In binary this is:
1000 1110 0010 0011 (with spaces to make it easier to read)
... or:
1000111000100011 (without spaces)

By putting the number against our template, we can see which bits make up the sign, exponent and significand:

```
sEEEEEnnnnnnnnnn
1000111000100011
```

... or to make it clearer:

```
s EEEEE nnnnnnnnnn
1 00011 1000100011
```

- The sign bit is 1, so the number is negative.

- The exponent bits are 00011. [As this is less than 01111, it means that the actual exponent must have been negative.] We have to subtract 01111 to find the actual exponent. We calculate: 00011 – 01111 = −1100. If you are still getting used to binary, this can be easier to calculate in decimal: 3 – 15 = −12.

- The significand is 1000100011. This means that the original number that was multiplied by the exponential was 1.1000100011 [We prefix the significand with the implied "1"].

Our decoded binary number is:
$1.1000100011 * (10)^{-1100}$

We need to shift the binary point of 1.1000100011 by 1100 (in binary) digits to the left. This is 12 (in decimal) digits to the left. Each shift of the binary point to the left results in another zero being placed at the start of the number. We end up with the binary number:
0.000000000011000100011

Zero and infinity

So far, we have not been able to store the number zero. This is because zero cannot be expressed as a binary number multiplied by a power of 2 if that binary number has 1 as the integer part. For this reason, the floating-point number system sets aside a special combination of significand and exponent that represents a zero. The exponent and significand are both set to zero, so the exponent becomes 00000, and the significand becomes 0000000000. The sign bit can be either 1 or 0, which means we can have positive zero or negative zero. These ultimately mean the same thing.

Positive zero in a 16-bit word looks like this in binary (with spaces added to make it easier to read):
0000 0000 0000 0000
... which is:
0x0000 in hex.

Negative zero is this in binary:
1000 0000 0000 0000
... which is:
0x8000 in hex.

We can also represent infinity, in which case, the exponent is set to all ones: 11111, and the significand is set to zeroes: 0000000000. The sign bit can be either 1 or 0, which means we can represent positive infinity and negative infinity.

Positive infinity in binary is:
0111 1100 0000 0000
... which is:
0x7c00 in hex.

Negative infinity in binary is:
1111 1100 0000 0000
... which is:
0xfc00 in hex.

32-bit floating-point numbers

Although a 16-bit floating-point word is good for explanations, on Intel and AMD 64-bit processors, a floating-point number as a 16-bit word is only useable in certain situations, and then the processor immediately converts it into a 32-bit floating-point number. Intel and AMD processors mainly use larger sizes, which have lengths of 32 bits, 64 bits and 80 bits.

A 32-bit floating-point number fits into a dword. Its formal name is “a single-precision” floating-point number. [A 16-bit floating-point number is called “a half-precision” floating-point number.] In programming languages such as C, a 32-bit floating-point number is called a “float”. Away from programming, the term “32-bit float” is more descriptive.

A 32-bit floating-point number has a similar layout to a 16-bit one, but it has more room for the exponent and the significand. Specifically, there are 8 bits for the exponent, and 23 bits for the significand. The exponent has to be “biased” by adding it to 127 (in decimal), which is 01111111 in binary or 0x7f in hex. The bits available for the exponent allow it to store the exponents of exponentials from 2^{-126} to 2^{127}. This means that a dword can store 126 binary places or 127 integer digits, but the digits will be rounded up to the first 23.

To make the dword easier to visualise, a 32-bit template would look like this:

sEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnn

... or shown with spaces to make it clearer:

s EEEEEEEE nnnnnnnnnnnnnnnnnnnnnnn

Zero is represented by setting the exponent and significand to all zeroes. Depending on the sign bit, we can have positive zero or negative zero. Positive zero is this in binary (with spaces to make it easier to read):
0000 0000 0000 0000 0000 0000 0000 0000
In hex, it is:
0x00000000

Negative zero is:
1000 0000 0000 0000 0000 0000 0000 0000
... which, in hex, is:
0x80000000

Infinity is represented by setting the exponent to all ones, and the significand to all zeroes. We can represent positive or negative infinity. Positive infinity in binary is:
0111 1111 1000 0000 0000 0000 0000 0000
In hex, it is:
0x7f800000

Negative infinity in binary is:
1111 1111 1000 0000 0000 0000 0000 0000
In hex, it is:
0xff800000

As an example of a 32-bit floating-point number, we will store the binary number:
−1101.000001111

As before, this involves turning it into a number with a single digit integer multiplied by an exponent of 2. It becomes:
$-1.101000001111 * (10)^{11}$

The sign is negative, so we will set the sign bit in our template to 1:
1EEEEEEEEnnnnnnnnnnnnnnnnnnnnnnn

The exponent is 11. We have to turn it into a biased exponent, so we add 01111111 to it, which produces 10000010. We put this into the exponent section:
110000010nnnnnnnnnnnnnnnnnnnnn

The significand is 101000001111. This is twelve digits long, but there are 23 available bits. Therefore, we put eleven zero digits after it to make it the correct length. It becomes 10100000111100000000000. We put it into the significand section, and we have our finished 32-bit floating-point number in binary:
11000001010100000111100000000000

This is a good example of a number that is much easier to read in hex. We split it into 4-bit sections:
1100 0001 0101 0000 0111 1000 0000 0000
... and we convert each 4-bit section to its relevant hex digit:
c 1 5 0 7 8 0 0
... and our number as a hex dword is:
0xc1507800

64-bit floating-point numbers

64-bit floating-point numbers are called “double-precision” floating-point numbers. They are also called “double floats” or even just “doubles”. A more descriptive short name for them is “64-bit floats”.

A 64-bit floating-point number has eleven digits for the exponent and 52 digits for the significand. The exponent of a number that is about to be stored needs first to be added to 1023 in decimal, which is 0x3ff in hex and 01111111111 in binary (this is eleven digits long). The bits available for the exponent allow it to store the exponents of exponentials from 2^{-1022} to 2^{1023}.

Although it does not really make the qword easier to visualise, a 64-bit template looks like this:

```
sEEEEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

... or like this with the parts drawn on different lines:

```
s
EEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

As with 16-bit and 32-bit floating-point numbers, zero is represented by setting the exponent and significand to zeroes. Positive zero looks like this *in hexadecimal*:
0x0000000000000000
... while negative zero looks like this *in hexadecimal*:
0x8000000000000000

Infinity is represented by setting the exponent to all ones and the significand to all zeroes. Positive infinity looks like this *in hexadecimal*:
0x7ff0000000000000
... while negative infinity looks like this *in hexadecimal*:
0xfff0000000000000

As an example of a 64-bit floating-point number, we will store the following binary number, which I have put into 4-bit sections to make it easier to read:
0.0000 0000 0000 1100 1111 0001 0000 1111

As a value multiplied by an exponential this is:
$1.1001\ 1110\ 0010\ 0001\ 111 * (10)^{1101}$
... where the binary exponent 1101 is 13 in decimal.

To make things simpler, we will use our 64-bit template:
`sEEEEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn`

The number is positive, so the leftmost bit is set to zero:
`0EEEEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn`

Our exponent is 1101. For 64-bit numbers, we have to add the exponent to 01111111111 before we can store it. We calculate:
01111111111 + 1101 = 10000001100

We put this into our 64-bit template:

`010000001100nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn`

Our significand is 100 1111 0001 0000 1111. This is 19 digits, but we have space for 52 digits. Therefore, we put extra zeroes after it. We can then put it into our 64-bit number, and we have the finished number, which looks like this in binary:

`0100000011001001111000100001111000000000000000000000000000000000`

To make this more palatable, we will convert it into hexadecimal. We start by splitting it into 4-bit sections:

```
0100 0000 1100 1001 1110 0010 0001 1110 0000 0000 0000 0000 0000
0000 0000 0000
```

Then we convert each 4-bit section to the relevant hex digit:
4 0 c 9 e 2 1 e 0 0 0 0 0 0 0 0
... and we end up with our 64-bit floating-point number in hex as:
0x40c9e21e00000000

80-bit floating-point numbers

The next type of floating-point number uses 80 bits. Unlike 16-bit, 32-bit and 64-bit floating-point numbers, 80-bit floating-point numbers do not fit into a standard grouping of bits. We cannot store an 80-bit number in a byte (8 bits), word (16 bits), dword (32 bits), or qword (64 bits). However, we can still store them in computer memory or in a file.

The formal name for an 80-bit floating-point number is a "double-extended-precision" floating-point number. They are also called "long doubles", but a more descriptive name for them would be "80-bit floats".

80-bit floating-point numbers are slightly different from 64-bit, 32-bit and 16-bit floating-point numbers. On Intel and AMD 32-bit and 64-bit processors, they can only be used in what is called the floating-point unit. To explain this most simply, the floating-point unit is the part of the processor that is dedicated solely to dealing with floating-point numbers one instruction at a time. A modern Intel or AMD processor has one floating-point unit, but it also has other parts that can use commands to operate on floating-point numbers in parallel. [Adding in parallel, for example, means that the values in two lists of numbers can be added together more quickly than if they were added one value at a time.] The set of parallel commands are called the "Single Instruction Multiple Data" instructions, or "SIMD" instructions. The commands in this group that operate on floating-point numbers in parallel are called the "Streaming SIMD *extensions*" (or "SSE" for short) because the original SIMD commands only worked on integers. The SSE instructions can work on integers or floating-point numbers, but when working on floating-point numbers, they can only work on 32-bit and 64-bit floating-point numbers. As new processor designs were released, the set of SSE instruction set was added to, and the new additions are called "SSE2", "SSE3" and so on. 16-bit half-precision

floating-point numbers can only be used with SSE instructions, and then they are immediately converted to 32-bit single-precision floating-point numbers.

Although 80-bit floating-point numbers can store bigger and more accurate numbers, they are less versatile than 32-bit or 64-bit floating-point numbers. An Intel or AMD computer processor can only work with 80-bit floats one at a time in the floating-point unit. It is quicker not to use 80-bit floats if we need to perform the same calculations on several floating-point numbers at the same time. A computer game, for example, would be faster using 32-bit or 64-bit floats with SSE instructions than using 80-bit floats in the floating-point unit. On the other hand, a maths program would be more accurate when using 80-bit floats. Which is the best size depends on what it is we want to do, and how we want to do it. When programming in higher-level languages, if asked to use 80-bit floats, some compilers will ignore the request and use 64-bit floats instead. [A compiler is the program that takes the text of a program's source code and converts it into the finished executable program.]

When a 32-bit or 64-bit float is loaded into an Intel or AMD processor's floating-point unit, it becomes converted into an 80-bit float. When the floating-point unit stores the 80-bit float to memory, it can store it as an 80-bit float, a 64-bit float or a 32-bit float, depending on what it is asked to do. [It can also store it as a rounded up signed integer in the form of a two's complement 16-bit word, 32-bit dword or 64-bit qword.]

80-bit floating-point numbers have 15 bits for the exponent and 64 bits for the significand. What makes them different is that the significand will contain the preceding "1" that is implied with the other sizes of float. Therefore, the leftmost bit of the significand will *generally* be set to 1. [It is set to 0 to indicate zero and "denormalised" numbers, which I will explain later in this chapter.] Given that the leftmost bit is used, there are actually only 63 bits for the significand.

The 15 bits for the exponent mean that the exponent can go from −16,382 to +16,383 in decimal. These are −0x3ffe to +0x3fff in hex, or:
−11 1111 1111 1110
... to:
+11 1111 1111 1110
... in binary (with spaces to make the numbers easier to read).

The exponent needs to be biased by having 16,383 in decimal added to it. This is 0x3fff in hex or 011 1111 1111 1111 in binary

A template for an 80-bit floating-point number is as follows. It consists of 1 bit for the sign, 15 bits for the exponent and 63 bits for the significand. The number is so long that it is easier to read when split over three lines:

```
s
EEEEEEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

If we want to portray zero, it would have an exponent of zero, a significand of zero, and a sign of one or zero, depending on whether we want it to be positive or negative zero. In binary, positive zero is:
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
... which is:
0x00000000000000000000
... in hex, or with the lower dwords separated by spaces to make it easier to read:
0x0000 00000000 00000000

In binary, negative zero is:
1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
... which is:
0x80000000000000000000
... in hex, or with the lower dwords separated by spaces to make it easier to read:
0x8000 00000000 00000000

With 16-bit, 32-bit and 64-bit floating-point numbers, infinity was represented by setting the exponent all to ones and the significand all to zeroes. With 80-bit floating-point numbers, the exponent is similarly set to all ones, but the significand is set to all zeroes *except for the leftmost bit,* which is set to 1. Therefore, in binary, positive infinity is:

0111 1111 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

In hex, this is:
0x7fff 80000000 00000000
... with spaces separating the lower dwords to make it easier to read.

In binary, negative infinity is:

1111 1111 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

In hex, this is:
0xffff 80000000 00000000

It is worth noting that apart from when encoding zero and certain rare exceptions (denormal numbers, which I mention later), the leftmost bit of the significand will always be set to 1. This is because the implied 1 that is ignored in 16-bit, 32-bit and 64-bit floating-point numbers is kept in 80-bit floating-point numbers.

We will convert a binary number to an 80-bit floating-point number to see how it works. We will convert the following binary number:
0.0000 0000 0000 0000 0000 0000 1100 1100 0000 1100 1100 1111 1111

This can be rephrased to be:
$1.1001\ 1000\ 0001\ 1001\ 1001\ 1111\ 111 * (10)^{-11001}$
... all in binary, where −11001 is −0x19 in hex and −25 in decimal.

The attributes of this number are:

- The sign is positive.

- The exponent is −11001. We have to add this to 011111111111111 for it to become our biased exponent. This is slightly easier to do by working in decimal, in which case we add −25 to 16,383, which is 16,358. This is 0x3fe6 in hex, and 011 1111 1110 0110 in binary.

- For 16-bit, 32-bit and 64-bit floating-point numbers, we would say that the significand is 1001 1000 0001 1001 1001 1111 111. However, for 80-bit floating-point numbers, we have to include the normally implied “1” as the leftmost bit. Therefore, our significand is actually all of that number preceded by 1, which is: 1100 1100 0000 1100 1100 1111 1111. We also have to extend the number to 64 digits by putting zeroes at the end.

Our template looks like this:

```
s
EEEEEEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

We set the sign to 0:

```
0
EEEEEEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

We set the exponent to 011111111100110:

```
0
011111111100110
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

We set the significand to 11001100000011001100111111111 followed by enough zeroes to make it 64 digits long.

```
0
011111111100110
1100110000001100110011111111100000000000000000000000000000000000000000
```

We then join up the pieces, and our final binary number (with spaces to make it easier to read) is:
0011 1111 1110 0110 1100 1100 0000 1100 1100 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000
... which, in hex, is:
0x3fe6 cc0ccff0 00000000
... with spaces separating the lower dwords to make it easier to read.

Denormalised numbers

The method of encoding numbers with the floating-point system described in this chapter refers to what are called “normalised” or “normal” numbers. This is sufficient for nearly all use of floating-point numbers. Normalised numbers are so-called mainly to distinguish them from “denormalised” or “denormal” numbers. We can think of “normalised” numbers as just being numbers as encoded with the floating-point system already described where:

- With 16-bit, 32-bit, and 64-bit floats, there is an implied initial 1 digit.
- With 80-bit floats, the 1 digit is included.

“Denormalised” or “denormal” numbers are ones that are so small that they are likely to be treated as zero because of the way that the floating-point system encodes numbers – the exponent cannot go high enough to accommodate them. This can be a problem if a calculation produces a very small number and the program needs to know that the result is not exactly zero. To avoid treating tiny numbers as zero, the way they are stored is temporarily altered to accommodate

them. This involves removing the idea of an implied 1 (or an included 1 for 80-bit floats) and having an implied *zero* instead (which is an included zero for 80-bit floats). On Intel processors, the switch to a denormalised number is flagged with a warning to tell a program that it has happened. As we have seen, 80-bit floating-point *normal* numbers have the usually implied 1 kept as the leftmost bit of the significand. If a number becomes too small for the exponential to encode it, the number becomes *denormal* and the leftmost bit of an 80-bit number becomes set to zero. In 16-bit, 32-bit and 64-bit floats, the zero becomes implied instead of the usual 1 being implied.

You do not need to understand denormalised numbers for the purposes of understanding anything in this book, or even for most programming. I only mention them here because other explanations of floating-point numbers might discuss them.

More on floating-point numbers

Here are some more facts about floating-point numbers.

There is an internationally agreed system for the encoding of binary numbers with the floating-point method. It is called “IEEE 754”. Intel and AMD processors follow this system.

Values stored as floating-point numbers are limited to those that have significant digits that can fit into the significand, and also those that are of a size that can be represented by the exponent. For most purposes, the floating-point number system is perfectly suitable. As long as a number has the basic layout of:
11100000000
... or:
0.0000000111
... it can be encoded without any trouble Problems can arise if we have a number that is both large and requires accuracy, such as this one:
1110000000000.0000000000011
[Using signal processing language, we could say that we are limited by the dynamic range of the floating-point system.]

Floating-point numbers have an unavoidable limit to their accuracy. As more calculations are performed with a number, this can become a problem and rounding errors will start to produce increasingly inaccurate results. A consequence of this is that a calculation performed with an 80-bit float might

produce slightly different results from the same calculation performed with a 64-bit or 32-bit float. An 80-bit float will be slightly more accurate. For 80-bit, 64-bit and 32-bit floats, any accumulated inaccuracies will be different, depending on which we are using.

The significand is sometimes called the “mantissa”. “Mantissa” is a Latin word that means a small amount of something used to top up a quantity to achieve a certain weight. For example, if we added a small amount of flour to a bag of flour to make it up to a 1 kilogramme, then, in the Latin sense, that small amount of flour would be a mantissa. The word has a related sense as “a worthless addition”. It is used in maths with the idea that the part after the binary (or decimal) point is not as important as the part before. Etymologically, the term “significand” really means the opposite of “mantissa” because it implies significance. The significand in a floating-point number is the most significant part. Some people prefer the word “significand”; some people prefer the word “mantissa”. In Intel’s “Intel 64 and IA-32 Architectures Software Developer’s Manual” (which is the definitive description of Intel’s processors), Intel uses the term “significand”.

As we have seen, floating-point numbers can store integers as well as non-integers. However, if we were only dealing in integers, it would be much easier not to use floating-point numbers and to use standard words, dwords and qwords instead.

Storing in memory

Floating-point numbers are stored in memory in the same way as bytes, words, dwords and qwords. In other words, they can be placed in the “backwards” little-endian way, or the more straightforward big-endian way. It is easiest to store them in files in the same way as they are stored in a computer’s memory.

A 16-bit half-precision floating-point number is a word, so it is stored as two bytes. If we store the 16-bit float 0x4a18 in the little-endian way, it would look like this in memory when viewed in a debugger or in a file as viewed in a hex editor: [I have arbitrarily added zeroes afterwards to fill a full line.]

```
0000: 18 4a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .J..............
```

In the big-endian way, it would look like this:

```
0000: 4a 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 J...............
```

A 32-bit single-precision floating-point number is a dword, so it is stored as four bytes. If we stored the 32-bit float 0xc1507800 in the little-endian way, it would look like this in memory or a file:

```
0000: 00 78 50 c1 00 00 00 00 00 00 00 00 00 00 00 00 .xP.............
```

In the big-endian way, it would look like this:

```
0000: c1 50 78 00 00 00 00 00 00 00 00 00 00 00 00 00 .Px.............
```

A 64-bit double-precision floating-point number is a qword, so is stored as eight bytes. If we stored the 64-bit float 0x40c9e21e00000000 in the little-endian way, it would look like this in memory or a file:

```
0000: 00 00 00 00 1e e2 c9 40 00 00 00 00 00 00 00 00 .......@........
```

If we stored it in the big-endian way, it would look like this:

```
0000: 40 c9 e2 1e 00 00 00 00 00 00 00 00 00 00 00 00 @...............
```

An 80-bit double-extended-precision floating-point number is 80 bits long, so it takes up ten bytes. If we stored the 80-bit float 0x3fe6cc0ccff000000000 in the little-endian way, it would look like this in memory or a file:

```
0000: 00 00 00 00 f0 cf 0c cc e6 3f 00 00 00 00 00 00 .........?......
```

In the big-endian way, it would look like this:

```
0000: 3f e6 cc 0c cf f0 00 00 00 00 00 00 00 00 00 00 ?...............
```

Intel 64-bit processors (among other 64-bit processors) work fastest when reading numbers from memory if the numbers are aligned to an offset that is on a 64-bit (8-byte) boundary. In other words, they can read a number more quickly if the offset of the number in memory ends in 0 or 8. For example, it is quicker for the processor to read the 64-bit qword 0xffffffffffffffff in this bit of memory, where it starts on a 64-bit offset:

```
000000014000D000: ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00
```

... than it is to read it from this bit of memory, where it starts 1 byte after a 64-bit offset:

```
000000014000D000: 00 ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00
```

For this reason, in memory, numbers are frequently stored at 64-bit offsets, and if a number does not take up the full 64 bits, the bytes afterwards until the next 64-bit boundary become unused. They might be set to zero, they might be set to 0xcc, or they might be left as whatever they were before. [Setting them to 0xcc can be helpful for programmers trying to detect problems when debugging software – if the number 0xcc, 0xcc, 0xcccc or similar is read by the program when it should not be, then it is a sign that the program is reading from the wrong parts of memory. Similarly, if the processor tries to execute the command 0xcc, it will instantly pass control of the program to a debugger.] Aligning numbers to 64-bit boundaries means that in memory, the consecutive 32-bit unsigned integer dwords 0xffffffff and 0x22222222 would appear as so:

```
000000014000D000: ff ff ff ff 00 00 00 00 22 22 22 22 00 00 00 00
```

If speed is not of importance, then the numbers might appear normally as so:

```
000000014000D000: ff ff ff ff 22 22 22 22 00 00 00 00 00 00 00 00
```

The 64-bit alignment idea means that an 80-bit floating-point number, which takes up ten bytes, will often be followed by padding bytes so that the next number starts on a 64-bit offset. There will be 6 bytes of padding. Therefore, two consecutive 80-bit floating-point numbers might be stored in memory as follows: [Shown here with the padding bytes set to 0xcc to make it clearer as to what is a number and what is padding.]

```
000000014000d000: 00 30 1d df 27 58 c5 c5 00 40 cc cc cc cc cc cc
000000014000d010: 00 40 c7 f7 09 56 71 81 02 c0 cc cc cc cc cc cc
```

When storing the data into a *file*, there is less of a need to have padding, as the file will need to be loaded into memory before the processor can read it. Reading the data from a file into memory is so many times slower than reading from memory that there are no real *speed* advantages in having padding in a file. Therefore, in a file, the above numbers would be much more likely to appear consecutively as so:

```
0000: 00 30 1d df 27 58 c5 c5 00 40 00 40 c7 f7 09 56
0010: 71 81 02 c0 00 00 00 00 00 00 00 00 00 00 00 00
```

Sometimes, it is less effort from a programming point of view just to copy data directly from memory into a file, without looking at the data while it is being done. In such situations, a file will unnecessarily contain the padding. This is why sometimes when you look at a file in a hex editor, the numbers might appear differently from how you would expect.

Programming terms

In this chapter, we have looked at the different ways of storing binary and hex in a way that emphasises the number of bits in each number. Someone from Intel or an assembly language programmer would speak of the numbers in this way. Generally, in programming languages, the actual details of the numbers being used are hidden from the programmer. Therefore, the names that a C programmer, for example, would use can be different from those that an assembly language programmer would use. The terms used in higher-level programming tend to be much more vague in their meaning, and can even refer to different things depending on the type of processor on which the program is going to be run. When higher-level programmers discuss the types of numbers that are stored in a file, they might use higher-level programming terms instead of saying bytes, words, dwords, 80-bit floats, and so on. Therefore, it pays to know what they mean.

In this section, we will look at the more common programming terms used in C programming.

Char

The term “char” is short for “character”. A “char” is 8 bits long and is so-called due to how an ASCII character is 8 bits in length. A “char” is specifically a *signed* byte. Therefore, it can store positive and negative integers, but it can only store a positive integer up to +127.

A “uchar” is an unsigned byte, or what I would call a normal byte. This means it can store positive integers from 0 up to 255.

Int

The term “int” is short for “integer”, and specifically means a “*signed* integer”. This is an ambiguous term because it can refer to different lengths of numbers depending on the processor for which the program is being written. On a 16-bit processor, an “int” will be a 16-bit *signed* integer. For a 32-bit processor, an “int” will *usually* be a 32-bit signed dword, but not always. Similarly, for a 64-bit processor, an “int” will *usually* be a 64-bit signed qword, but not always. All of this means that if someone mentions the term “int”, its meaning is ambiguous without further information. Having a slightly ambiguous definition of an unsigned integer allows fairly simple code to be used on different types of computers without

needing to be rewritten. [It is not good if you want to tailor your code to take advantage of a particular type of processor or operating system.] The ambiguity is a nuisance when it comes to numbers stored in a file, as saying “this file contains a list of numbers as ints” is as useless as saying “this file contains numbers”. It is better either to refer to signed or unsigned bytes, words, dwords and qwords, or otherwise to specify the bit size of the int.

To have an unsigned integer, there is the term “uint”.

There is also the term “long”, which refers to a signed integer that is bigger than an “int”. This can also be ambiguous.

Float

A “float” is a 32-bit single-precision floating-point number. Outside of programming, it would be more descriptive to refer to floats as “32-bit floats” to indicate how many bits they use.

Double

A “double” is a 64-bit double-precision floating-point number. Outside of programming, it is more descriptive to refer to such numbers as “64-bit floats”, “64-bit doubles” or even “64-bit double floats”.

Long double

A “long double” is an 80-bit double-extended-precision floating-point number. Outside of programming, it is more descriptive to refer to such numbers as “80-bit floats”, “80-bit long doubles” or even “80-bit long double floats”. Some modern compilers will ignore any requests to use 80-bit long doubles, and replace them with 64-bit doubles instead. This means that there will be less accuracy, but the program might run faster because it can use a wider range of instructions to deal with the numbers.

Conclusion

Understanding how computers store numbers is useful for understanding the different ways that discrete waves can be stored as files. It is easiest to understand and remember everything in this chapter if you have a need to do so. It is well worth downloading a free hex editor to get more used to binary and hexadecimal.

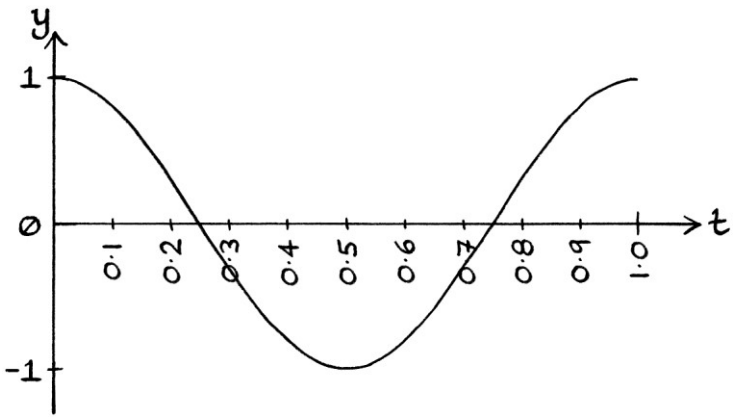
Even if you do not remember or understand everything in this chapter, just knowing that a number can be portrayed in different ways is useful for understanding discrete signals.

Chapter 41: More on discrete signals

Now that we know about hexadecimal and binary, we will look at sampling and quantization as they occur in the realm of computers.

Quantization with hexadecimal

In Chapter 39, we sampled the following wave for one second with a sample rate of 20 samples per second. The formula of the wave is "y = sin (2πt + 0.5π)".



A table showing our readings is as so:

Time (seconds)	**y-axis value or in other words, the sample value**
t = 0	y = 1
t = 0.05	y = 0.9511
t = 0.10	y = 0.8090
t = 0.15	y = 0.5878
t = 0.20	y = 0.3090
t = 0.25	y = 0
t = 0.30	y = −0.3090
t = 0.35	y = −0.5878
t = 0.40	y = −0.8090
t = 0.45	y = −0.9511
t = 0.50	y = −1
t = 0.55	y = −0.9511
t = 0.60	y = −0.8090

t = 0.65 y = −0.5878
t = 0.70 y = −0.3090
t = 0.75 y = 0
t = 0.80 y = 0.3090
t = 0.85 y = 0.5878
t = 0.90 y = 0.8090
t = 0.95 y = 0.9511

We can portray the sample values in a list as so:

1, 0.9511, 0.8090, 0.5878, 0.3090, 0, −0.3090, −0.5878, −0.8090, −0.9511, −1, −0.9511, −0.8090, −0.5878, −0.3090, 0, 0.3090, 0.5878, 0.8090, 0.9511.

As explained in Chapter 39, the list of samples makes up what is called a “discrete signal”. We could take the list of samples in this form and write it down or give to someone else, and the discrete signal could be recreated at a later date. We could also perform maths with the discrete signal because we have values with which to perform calculations. If we wanted to store the values so that a computer could use them, we would have to convert them to binary or hex, and then adjust them to fit in with one of the several ways that computers store numbers.

32-bit floating-point numbers

We will convert our list of decimal values into a list of 32-bit floating-point values. We will convert our *readings* of the y-axis values from the curve, and not the actual y-axis values. Our readings have 4 decimal places.

The first step is to convert each number into binary. To do this, we take each number in turn, and keep multiplying it by 2 until it becomes an integer. Then we convert the result to binary (or to hex then binary if that is easier). Then we divide that binary number by the multiple of 2 by which the decimal number was multiplied to make it an integer. The second step is to convert the result to a floating-point number.

For a 32-bit floating-point number, there are 8 available bits for the exponent. Before we store the exponent, we need to add 01111111 to it. There are 23 bits for the significand. The template for a 32-bit float is:
sEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnn

Our first decimal number is 1. We do not need to do any maths to turn this into binary. It is 1 in binary, which is best written as 1.0000 for the purposes of easily converting it to a floating-point number. The number can be rephrased to be a value multiplied by an exponent of 2 as:
$1.0000 * (10)^0$

The number is positive. The exponent is 0, to which we add 01111111 to get 01111111. The significand is 0000, which we will extend to 23 digits of zeroes. Our floating-point number is:
00111111100000000000000000000000

We split this into groups of 4:
0011 1111 1000 0000 0000 0000 0000 0000
... and convert it into the relevant hex digits:
3 f 8 0 0 0 0 0
... and we end up with:
0x3f800000

Our second sample is 0.9511. We multiply this by 2 until it becomes an integer. It requires 47 multiplications by 2 to become an integer. It becomes:
133,855,425,174,752. To save time, we will use a hex calculator to turn this into hex. [It is quicker to turn this into hex and then turn the hex number into binary, than it is to turn it into binary in one go. It is also easier to convert it using a calculator than by hand. If we were using a computer, it would do the entire conversion for us.] The number is 0x79bda5119ce0 in hex. In binary, this is:
0111 1001 1011 1101 1010 0101 0001 0001 1001 1100 1110 0000

We divide it by two 47 times by moving the binary point 47 digits to the left, and we get:
0.1111 0011 0111 1011 0100 1010 0010 0011 0011 1001 1100 000

We will now convert this to a floating-point number. As a number multiplied by an exponent of 2, it is:
$1.1110\ 0110\ 1111\ 0110\ 1001\ 0100\ 0100\ 0110\ 0111\ 0011\ 1000\ 00 * (10)^{-1}$

The sign is positive. The exponent is −1, to which we add 01111111 to get 01111110. The significand is 1110 0110 1111 0110 1001 0100 0100 0110 0111 0011 1000 00. As the significand can only be 23 digits long, we round this number up to be: 1110 0110 1111 0110 1001 010.

Our final floating-point number in binary is:
00111111011100110111101101001010

We split this into 4-bit sections:
0011 1111 0111 0011 0111 1011 0100 1010
... and we convert each 4-bit section to its relevant hex digit:
3 f 7 3 7 b 4 a
... and our floating-point number in hex is:
0x3f737b4a.

Our third sample is 0.8090. We will multiply this by 2 until it becomes an integer. It also requires 47 multiplications of 2 to become an integer, and it becomes: 113,856,628,079,460. We will use a hex calculator to convert this into hex. It is 0x678d4fdf 3b64 in hex. This is:
0110 0111 1000 1101 0100 1111 1101 1111 0011 1011 0110 0100
... in binary. We divide it by two 47 times by moving the binary point 47 digits to the left, and we get:
0.1100 1111 0001 1010 1001 1111 1011 1110 0111 0110 1100 100

We now convert that to a floating-point number. As a value multiplied by an exponent of 2, it is:
1.1001 1110 0011 0101 0011 1111 0111 1100 1110 1101 1001 00 * $(10)^{-1}$

The sign is positive. The exponent is −1, to which we add 01111111 to get 01111110. The significand is 1001 1110 0011 0101 0011 1111 0111 1100 1110 1101 1001 00. As the significand can only be 23 digits long, we will round this up to be: 1001 1110 0011 0101 0100 000. Our final floating-point number, in binary, will be:
00111111010011110001101010100000

We split this into 4-bit sections:
0011 1111 0100 1111 0001 1010 1010 0000
... and convert each 4-bit section to its relevant hex digit:
3 f 4 f 1 a a 0
... and our final hex 32-bit floating-point number is:
0x3f41aa0.

Converting the numbers by hand is straightforward but time consuming, and it is very easy to make mistakes. To save time, for the other values, we will just presume we have a calculator that converted all the values for us.

The full list of samples as 32-bit floating-point values is in the following table:

Time (seconds)	**y-axis value (sample value) in decimal**	**y-axis value (sample value) as a 32-bit floating-point number in hex**
t = 0	1	0x3f800000
t = 0.05	0.9511	0x3f737b4a
t = 0.10	0.8090	0x3f4f1aa0
t = 0.15	0.5878	0x3f167a10
t = 0.20	0.3090	0x3e9e353f
t = 0.25	0	0x00000000
t = 0.30	−0.3090	0xbe9e353f
t = 0.35	−0.5878	0xbf167a10
t = 0.40	−0.8090	0xbf4f1aa0
t = 0.45	−0.9511	0xbf737b4a
t = 0.50	−1	0xbf800000
t = 0.55	−0.9511	0xbf737b4a
t = 0.60	−0.8090	0xbf4f1aa0
t = 0.65	−0.5878	0xbf167a10
t = 0.70	−0.3090	0xbe9e353f
t = 0.75	0	0x00000000
t = 0.80	0.3090	0x3e9e353f
t = 0.85	0.5878	0x3f167a10
t = 0.90	0.8090	0x3f4f1aa0
t = 0.95	0.9511	0x3f737b4a

If we wrote the values as a list on a piece of paper, they would appear as so:

0x3f800000, 0x3f737b4a, 0x3f4f1aa0, 0x3f167a10, 0x3e9e353f, 0x00000000, 0xbe9e353f, 0xbf167a10, 0xbf4f1aa0, 0xbf737b4a, 0xbf800000, 0xbf737b4a, 0xbf4f1aa0, 0xbf167a10, 0xbe9e353f, 0x00000000, 0x3e9e353f, 0x3f167a10, 0x3f4f1aa0, 0x3f737b4a.

To make use of such a list, someone would need to know the sample rate, so they knew when the curve of the graph had those values. They would also need to know that the 32-bit values were 32-bit floating-point numbers and not 32-bit unsigned (as in positive) integers or 32-bit signed (as in two's complement) integers. Without any context, it is impossible to know.

If we had a file containing these values stored in the little-endian way (so in other words, with the bytes in a backwards order), and we viewed the file in a hex editor, we would see the following:

```
0000: 00 00 80 3f 4a 7b 73 3f a0 1a 4f 3f 10 7a 16 3f ...?J{s?..O?.z.?
0010: 3f 35 9e 3e 00 00 00 00 3f 35 9e be 10 7a 16 bf ?5.>....?5...z..
0020: a0 1a 4f bf 4a 7b 73 bf 00 00 80 bf 4a 7b 73 bf ..O.J{s.....J{s.
0030: a0 1a 4f bf 10 7a 16 bf 3f 35 9e be 00 00 00 00 ..O..z..?5......
0040: 3f 35 9e 3e 10 7a 16 3f a0 1a 4f 3f 4a 7b 73 3f ?5.>.z.?..O?J{s?
```

All of the characters on the right hand side are coincidences where the bytes happen also to be valid ASCII characters.

Although it might difficult at first to see patterns in the bytes, we can clearly see four consecutive zero bytes on the second and fourth lines. These are the zero samples of our signal. Being able to recognise certain samples makes it easier to know where in the file we are if we should want to remove or copy parts of the file.

If we had a file containing the values stored in the big-endian way (so in other words, with the bytes in their normal order), and we viewed the file in a hex editor, we would see the following:

```
0000: 3f 80 00 00 3f 73 7b 4a 3f 4f 1a a0 3f 16 7a 10 ?...?s{J?O..?.z.
0010: 3e 9e 35 3f 00 00 00 00 be 9e 35 3f bf 16 7a 10 >.5?......5?..z.
0020: bf 4f 1a a0 bf 73 7b 4a bf 80 00 00 bf 73 7b 4a .O...s{J.....s{J
0030: bf 4f 1a a0 bf 16 7a 10 be 9e 35 3f 00 00 00 00 .O....z...5?....
0040: 3e 9e 35 3f 3f 16 7a 10 3f 4f 1a a0 3f 73 7b 4a >.5??.z.?O..?s{J
```

Again, we can see the zero samples in the form of four consecutive zero bytes on the second and fourth lines.

8-bit signed bytes

Now we will convert our decimal samples to 8-bit two's complement integers ("signed integers"). The main thing to understand when doing this is that our original samples were all between −1 and +1. Therefore, if we were to round them up to be positive and negative integers, every value would end up being −1, 0 or +1. This would be of no use as a record of our signal. Therefore, we will scale all the values in size to make the best use of the available bits. An 8-bit two's complement byte has values from −128 to +127. As we are recording just the values from our signal, where we know the minimum and maximum are −1 and +1, we will multiply every value by 127. Therefore, the new minimum will be −127 and the

new maximum will be +127, and every value will be an integer. By scaling the values to become integers, and putting them into 8-bit two's complement bytes, we will lose a lot of accuracy, yet we will have simpler numbers to deal with.

A table showing the new values in *decimal* is as follows:

Time (seconds)	**Original sample (decimal)**	**Sample multiplied by 127**	**Sample multiplied by 127 and turned into an integer**
t = 0	1	+127	+127
t = 0.05	0.9511	+120.7897	+121
t = 0.10	0.8090	+102.743	+103
t = 0.15	0.5878	+74.6506	+75
t = 0.20	0.3090	+39.243	+39
t = 0.25	0	0	0
t = 0.30	−0.3090	−39.243	−39
t = 0.35	−0.5878	−74.6506	−75
t = 0.40	−0.8090	−102.743	−103
t = 0.45	−0.9511	−120.7897	−121
t = 0.50	−1	−127	−127
t = 0.55	−0.9511	−120.7897	−121
t = 0.60	−0.8090	−102.743	−103
t = 0.65	−0.5878	−74.6506	−75
t = 0.70	−0.3090	−39.243	−39
t = 0.75	0	0	0
t = 0.80	0.3090	+39.243	+39
t = 0.85	0.5878	+74.6506	+75
t = 0.90	0.8090	+102.743	+103
t = 0.95	0.9511	+120.7897	+121

Our scaled samples in decimal are:
127, 121, 103, 75, 39, 0, −39, −75, −103, −121, −127, −121, −103, −75, −39, 0, 39, 75, 103, 121.

We can now convert these into 8-bit two's complement bytes (signed bytes). It is easiest to convert these into hex in one go, without converting them into binary first. As a reminder of how two's complement works, a positive number is just converted into hex normally. A negative number is converted into hex, has 1 subtracted from it, and then has all its bits flipped.

The first number is +127. To convert this into a hex byte, we divide it by 16. The result is 7 and a remainder of 15. We convert the remainder into hex (it becomes "f"), and put it into the 1s column. Then we convert the result of the division into hex (it becomes "7") and put it into the sixteens column. Therefore, our number in hex is 0x7f. As this is a positive number, the two's complement number is the same and is 0x7f.

The second number is +121. We divide this by 16, and we get 7 and a remainder of 9. We put the remainder into the ones column and the 7 into the sixteens column. Our hex number is 0x79, and our two's complement number is also 0x79.

The third number is +103. We divide this by 16, and we get 6 and a remainder of 7. We put the remainder into the ones column and the 6 into the sixteens column. Our hex number is 0x67, and our two's complement number is also 0x67.

The next few positive numbers end up as: 0x4b, 0x27 and 0x00.

The first negative number is −39. We therefore convert +39 to hex, subtract 1 and flip the bits. The decimal number +39 is 0x27 in hex. We subtract 1 to get 0x26, which, for the sake of flipping the bits, is this in binary:
0010 0110

After flipping the bits, we get:
1101 1001
... which is 0xd9 in hex.

The next number is −75. We will convert +75 to hex, subtract 1, then flip the bits. The number +75 is 0x4b in hex. We subtract 1 to get 0x4a, which for the sake of flipping the bits is this in binary:
0100 1010

After flipping the bits, we get:
1011 0101
... which is 0xb5 in hex.

We continue in this way until we have converted all the samples into signed bytes. It is, of course, *much* easier to use a hexadecimal or binary calculator.

Our full list of samples is:

0x7f, 0x79, 0x67, 0x4b, 0x27, 0x00, 0xd9, 0xb5, 0x99, 0x87, 0x81, 0x87, 0x99, 0xb5, 0xd9, 0x00, 0x27, 0x4b, 0x67, 0x79

If the signed bytes were stored consecutively in a file, and we looked at the file in a hex editor, we would see this:

```
0000: 7f 79 67 4b 27 00 d9 b5 99 87 81 87 99 b5 d9 00 .ygK'...........
0010: 27 4b 67 79                                     'Kgy
```

[Remember that the "backwards" little-endian way of storing data and the "forwards" big-endian way of storing data make no difference to storing individual bytes.]

Thoughts on numbers

Our original samples were numbers between −1.0000 to +1.0000 to four decimal places. There were potentially 20,001 different values that we could have used to identify the different y-axis values from our curve. When we use two's complement bytes to hold scaled versions of our samples, the values are integers from −127 to +127. There are only 255 different values with which to identify y-axis values from our curve. [We could have used 256 different values if we had used a slightly more complicated way of using the available bits.]

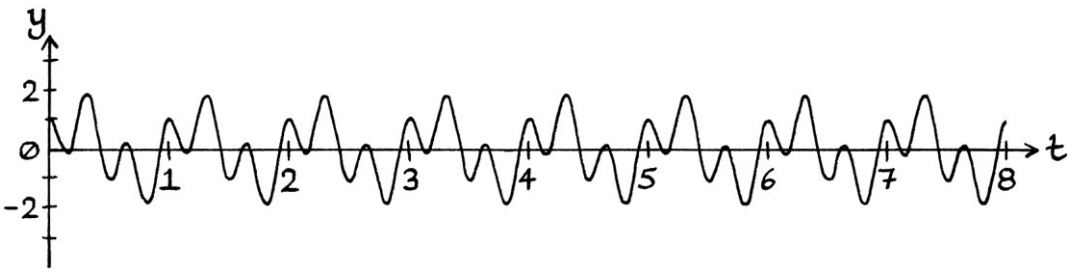
Strictly speaking, an original y-axis reading from the curve could have had an infinite number of decimal points of accuracy. However, the difficulties in making readings to that accuracy, and the practicalities of writing such numbers down, would make that impractical. Any y-axis reading is really a compromise between accuracy and usability. If we stored the data as 80-bit floating-point numbers, we would have a good level of accuracy, but a file containing the data would be ten times the size of one containing the data as scaled signed bytes. On the other hand, the accuracy lost by storing the data as scaled signed bytes means that any calculations will quickly suffer from rounding errors.

[Another problem with 80-bit floating-point numbers is that using them on current Intel 32-bit and 64-bit processors is considerably slower than using 64-bit floating-point numbers. 80-bit floating-point numbers cannot be processed in parallel, unlike 64-bit and 32-bit floating-point numbers.]

Depending on what we are doing, the accuracy contained in an 80-bit, 64-bit, or 32-bit floating-point number might be far more than we need. The best way of storing samples is the one that benefits what it is we want to do with them.

A longer example

As another example, we will look at the following signal, which has the formula: "y = sin (2π * 1t) + sin ((2π * 3t) + 0.5π)"



We will store 8 seconds' worth of the signal with a sample rate of 50 samples per second. We will presume we can take readings with perfect accuracy. We will scale the readings and turn them into two's complement bytes (signed bytes), so that the maximum and minimum y-axis readings become +127 and −127 respectively.

The maximum y-axis value of the actual signal is +1.87870685 and the minimum y-axis value is −1.87870685. Therefore, we might expect that every point will be multiplied by 127 ÷ 1.87870685 ≈ 67.5997 before it is turned into an integer and then a signed byte. However, because we are taking equally spaced *samples* from the curve, our maximum and minimum *samples* are actually +1.87341021 and −1.87341021. This is because the actual maximums and minimums of the curve lie between the points where we take our readings. A simplified idea of what is happening is shown in this drawing, where the dots mark from where we are reading the samples:

This is an unavoidable consequence of sampling. If we used a higher sampling rate, we would be more likely to have samples that were exactly at the peaks, or at least close enough to the peaks that the differences were negligible. In previous examples, by good luck, the maximums have coincided with the point where a sample was taken.

As the maximum and minimum sampled values are +1.87341021 and −1.87341021, we will be scaling by: 127 ÷ 1.87341021 = 67.7908. [If the maximum and minimum values were not the opposites of each other, we would have to choose the "absolute" maximum – we would choose whichever of the maximum and minimum, when both made positive, was largest.]

The following table shows the first few readings and how they are converted:

Time (seconds)	**y-axis value**	**That value scaled by 67.7908**	**That scaled value turned into an integer**	**That integer turned into a signed byte**
0.00	1	67.7908	68	0x44
0.02	1.0551	71.5261	72	0x48
0.04	0.9777	66.2791	66	0x42
0.06	0.7939	53.8191	54	0x36
0.08	0.5445	36.9121	37	0x25
0.10	0.2788	18.9000	19	0x13
0.12	0.04712	3.1943	3	0x03
0.14	−0.1058	−7.1723	−7	0xf9
0.16	−0.1478	−10.01948	−10	0xf6
0.18	−0.06376	−4.3223	−4	0xfc
0.20	0.1420	9.6263	10	0x0a

... and so on.

If we stored 8 seconds' worth of samples in a file as signed bytes, and looked at the file in a hex editor, we would see the following:

```
0000: 44 48 42 36 25 13 03 f9 f6 fc 0a 1e 37 50 67 77 DHB6%.......7Pgw
0010: 7f 7c 70 5a 3d 1c fc df c9 bc b8 be ca db ed fd .|pZ=...........
0020: 07 0a 04 f6 e2 c9 b0 99 89 81 84 90 a6 c3 e4 04 ................
0030: 21 37 44 48 42 36 25 13 03 f9 f6 fc 0a 1e 37 50 !7DHB6%.......7P
0040: 67 77 7f 7c 70 5a 3d 1c fc df c9 bc b8 be ca db gw.|pZ=.........
0050: ed fd 07 0a 04 f6 e2 c9 b0 99 89 81 84 90 a6 c3 ................
0060: e4 04 21 37 44 48 42 36 25 13 03 f9 f6 fc 0a 1e ..!7DHB6%.......
0070: 37 50 67 77 7f 7c 70 5a 3d 1c fc df c9 bc b8 be 7Pgw.|pZ=.......
0080: ca db ed fd 07 0a 04 f6 e2 c9 b0 99 89 81 84 90 ................
0090: a6 c3 e4 04 21 37 44 48 42 36 25 13 03 f9 f6 fc ....!7DHB6%.....
00a0: 0a 1e 37 50 67 77 7f 7c 70 5a 3d 1c fc df c9 bc ..7Pgw.|pZ=.....
00b0: b8 be ca db ed fd 07 0a 04 f6 e2 c9 b0 99 89 81 ................
00c0: 84 90 a6 c3 e4 04 21 37 44 48 42 36 25 13 03 f9 ......!7DHB6%...
00d0: f6 fc 0a 1e 37 50 67 77 7f 7c 70 5a 3d 1c fc df ....7Pgw.|pZ=...
00e0: c9 bc b8 be ca db ed fd 07 0a 04 f6 e2 c9 b0 99 ................
00f0: 89 81 84 90 a6 c3 e4 04 21 37 44 48 42 36 25 13 ........!7DHB6%.
```

```
0100: 03 f9 f6 fc 0a 1e 37 50 67 77 7f 7c 70 5a 3d 1c ......7Pgw.|pZ=.
0110: fc df c9 bc b8 be ca db ed fd 07 0a 04 f6 e2 c9 ................
0120: b0 99 89 81 84 90 a6 c3 e4 04 21 37 44 48 42 36 ..........!7DHB6
0130: 25 13 03 f9 f6 fc 0a 1e 37 50 67 77 7f 7c 70 5a %.......7Pgw.|pZ
0140: 3d 1c fc df c9 bc b8 be ca db ed fd 07 0a 04 f6 =...............
0150: e2 c9 b0 99 89 81 84 90 a6 c3 e4 04 21 37 44 48 ............!7DH
0160: 42 36 25 13 03 f9 f6 fc 0a 1e 37 50 67 77 7f 7c B6%.......7Pgw.|
0170: 70 5a 3d 1c fc df c9 bc b8 be ca db ed fd 07 0a pZ=.............
0180: 04 f6 e2 c9 b0 99 89 81 84 90 a6 c3 e4 04 21 37 ..............!7
```

Although the ASCII column is only showing coincidences where the signed bytes happen to be equal to valid ASCII characters, it is useful for quickly seeing that the bytes repeat. The bytes repeat because the signal they are describing repeats its shape – it is periodic. The ASCII pattern of "DHB6%" repeats eight times because the signal repeats eight times in the eight seconds of samples that we have.

The final line in our file starts at offset 0x0180 and ends at offset 0x018f. [The file starts at offset 0x0000, as all files do because the offset is how far into the file we are.] As the last byte of the file is byte number 0x018f, and we started with byte number 0x0000, there are 0x018f + 1 = 0x0190 bytes in our file, which is 400 bytes in decimal. The file is this size because we have 8 seconds' worth of samples with a sample rate of 50 samples per second, and each sample is contained within a byte. [50 * 8 = 400].

Recreating the signal

We will imagine that we have forgotten the details of the original signal, but we want to recreate it by reading the file. To do this, we *must* know:

- How many samples there are per second.
- In what form the data is stored.
- Whether the data is stored in the "backwards" little-endian way or the "forwards" big-endian way (if the data is stored in a form that uses more than one byte per sample).
- By how much the data was scaled (if the data was scaled to fit into a particular type of number).

If we do not know the sample rate, then we could recreate the signal's shape, but we would have no idea of the timing. However, we could create a signal with the x-axis as "sample number" instead of time, and for some purposes that would be sufficient.

If we do not know the form in which the data is encoded, we will not know how to decode the data. We will not know whether it is made up of unsigned or signed bytes, words, dwords or qwords, or whether it might be made up of 16-bit, 32-bit, 64-bit or 80-bit floating-point numbers. Similarly, if we do not know the "endianness", we will not know how to decode the data. [For simple signals where we know what the signal should look like, we could try different choices until we stumbled across something that looked right. For more complicated signals, we might not be able to do this.]

The first 16 bytes of the file are as so:

```
0000: 44 48 42 36 25 13 03 f9 f6 fc 0a 1e 37 50 67 77 DHB6%.......7Pgw
```

If we do not know how the data is actually encoded, we could misinterpret it in many different ways. Here are the many ways that we could interpret the start of the file:

Interpretation	**First number if we use this interpretation**
Unsigned bytes	0x44 (+68 in decimal)
Signed bytes	0x44 (+68 in decimal)
Unsigned words (little-endian)	0x4844 (+18,500 in decimal)
Unsigned words (big-endian)	0x4448 (+17,480 in decimal)
Signed words (little-endian)	0x4844 (+18,500 in decimal)
Signed words (big-endian)	0x4448 (+17,480 in decimal)
Unsigned dwords (little-endian)	0x36424844 (+910,313,540 in decimal)
Unsigned dwords (big-endian)	0x44484236 (+1,145,586,230 in decimal)
Signed dwords (little-endian)	0x36424844 (+910,313,540 in decimal)
Signed dwords (big-endian)	0x44484236 (+1,145,586,230 in decimal)
Unsigned qwords (little-endian)	0xf903132536424844 (a big number)
Unsigned qwords (big-endian)	0x44484236251303f9 (a big number)
Signed qwords (little-endian)	0xf903132536424844 (big negative)
Signed qwords (big-endian)	0x44484236251303f9 (big positive)
16-bit floats (little-endian)	0x4844 (8.53125 in decimal)
16-bit floats (big-endian)	0x4448 (4.28125 in decimal)
32-bit floats (little-endian)	0x36424844 (about 0.000003 in decimal)
32-bit floats (big-endian)	0x44484236 (about 801.0345 in decimal)
64-bit floats (little-endian)	0xf903132536424844 (big negative)
64-bit floats (big-endian)	0x44484236251303f9 (big positive)
80-bit floats (little-endian)	0xfcf6f903132536424844 (big negative)
80-bit floats (big-endian)	0x44484236251303f9f6fc (big positive)

Despite the many possibilities, it is possible to spot patterns in hex bytes that suggest what sort of numbers they might be. If we see some bytes such as this:

```
f7 00 f8 03 ef 33 fe 67
```

... then it might suggest that they were big-endian signed words – the presence of "f" and "e" [1111 and 1110 in binary] on alternate bytes would fit in with words where the leftmost bit is set to 1 to indicate it being negative. We could guess that these were signed words (as opposed to signed bytes or signed dwords). We might not be correct, but we could proceed with this assumption to see if everything fell into place. As you get more used to hex, you will find it easier to spot patterns. For simple signals such as we have seen in this book, we could look out for sequential zeroes (indicating zero-valued samples). If we were sure that the signal never stayed at zero for more than one sample, then sequential zeroes would tell us how many bytes were used for each sample.

We might presume that it is unlikely that someone would use 64-bit signed or unsigned integers to store wave samples because a 64-bit float would use the same number of bytes but also allow non-integers. It is relatively rare for 16-bit floating-point numbers to be used. It is more likely than someone would use 32-bit or 64-bit floating-point numbers than 80-bit floating-point numbers. If the samples were created by low cost analogue-to-digital converters (described later in this chapter), we might expect signed bytes or signed 16-bit words.

If we know the sample rate and how many seconds of data are in a file, then we can calculate how many bytes are used to store each sample. This can reduce the possibilities.

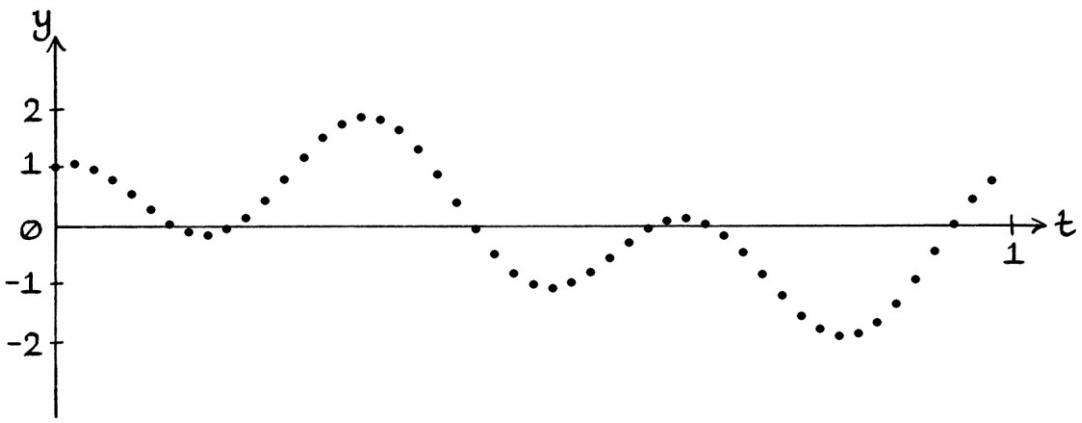
If the data had been scaled before being encoded into a number type, then we would need to know by how much it was scaled if we wanted to recreate the signal with its original values. However, if we do not know the scaling amount, the data is still useable because the decoded signal will still have the correct proportions no matter how it is scaled. If the data represented radio or sound waves, then the concept of exact instantaneous amplitudes is not particularly useful anyway – the instantaneous amplitudes would be dependent on how far the receiver was from the source, and the form in which they were recorded. The actual recorded values before they were scaled could only ever be thought of as being relative values, so any scaling would not remove anything useful from them.

For our example, we know that the sample rate is 50 samples per second. We know that the data is stored as signed bytes (two's complement bytes). We also know that the samples were scaled by multiplying them by 67.7908. If the sample rate is 50 samples per second, then there are 1 ÷ 50 = 0.02 seconds between each sample. In other words, the "sampling period" is 0.02 seconds.

We will take the first line of the file:

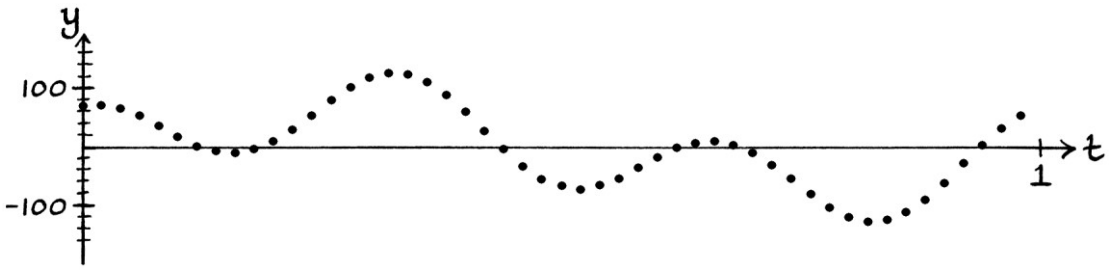
```
0000: 44 48 42 36 25 13 03 f9 f6 fc 0a 1e 37 50 67 77 DHB6%.......7Pgw
```

We know that the signed byte 0x44 will be divided by 67.7908, and then be placed at t = 0. The next signed byte, 0x48, will be divided by 67.7908, and placed at t = 0.02. The next signed byte, 0x42, will be divided by 67.7908, and placed at t = 0.04. We continue in this way until we have gone through the whole file. Our recreated *discrete* signal looks like this (with only the first second shown to make it clearer):



If we wanted to recreate an approximation to the original continuous signal, we could join these points up. However, the result could only ever be an approximation for two reasons. First, we only took measurements at evenly spaced moments in time – specifically, we used a sample rate of 50 samples per second. This means that we cannot know how the curve looked between these measurements. Anything that happened between these points is lost forever in the sampling process. By joining up the points, we are making a possibly false assumption that the curve maintained a similar shape during the unmeasured times. [There are two ways of joining up the points: either we can use straight lines, or we can use lines that curve between the points. Although the curved way more closely matches the original signal in this case, we cannot know that either way is correct because all we have are the individual samples.] The second reason why the recreated signal is only an approximation is that we had to fit our numbers into 8-bit signed bytes. This reduced the accuracy of every measurement.

If we had not “un-scaled” the data when we turned it back into a signal, then each byte would have remained between −127 and +127. The recreated signal would have kept the same shape, but would have remained scaled in size. The first second would look as so:



For many purposes, whether we resize the numbers to their original values or not is irrelevant. With recordings of radio and sound waves, the relative size of each point of the curve is more important than the actual original size.

Complex samples

So far, we have been storing just the y-axis values from signals in our samples. As we know, any pure wave can be thought of as being derived from a circle. For a time-based wave, we can be more specific and say that a pure wave can be thought of as being derived from an object moving around a circle. Similarly, any impure signal can be thought of as being derived from a “shape”. We can also think of circles as helices, and shapes as helix-like curves. A pure wave, circle and helix are the same thing viewed from different perspectives. Similarly, an impure signal, shape and helix-like curve are all the same thing viewed from different perspectives. All of this was explained in part one of this book.

Supposing we had an object moving around a circle or a shape, we could store its x-axis and y-axis coordinates at evenly spaced moments in time, in much the same way that we have been storing the y-axis values of waves and signals at evenly spaced moments in time. By doing this, we would be storing discrete versions of the vertically derived signal and the horizontally derived signal from that circle or shape. If we were storing the coordinates of an object rotating around a circle, we would be storing discrete versions of the derived Sine wave and the derived Cosine wave.

By storing the coordinates of an object rotating around a circle or a shape, we would be storing what are called “Complex samples”. The word “Complex” here refers to how we would be storing x and y-axis values from the Complex plane. To be more pedantic, we would be storing Real and Imaginary values from the Complex plane. An object’s position on the Complex plane at any one time would be recorded by measuring its Real (x-axis) position and its Imaginary (y-axis) position at evenly spaced moments in time. If we were to think of the Complex plane in signal processing terms, we would say that we are storing the In-Phase axis and Quadrature axis values. The initial letters of the words “In-Phase” and “Quadrature” are “I” and “Q”. Therefore, Complex samples are sometimes called “IQ” samples, which is a bit like calling them “xy” samples. When we do this, we will have two samples for each moment in time.

Complex samples can be used to store an object’s path along a circle or shape, but more commonly, they will be used as a more thorough way of storing the y-axis values from a pure wave or a signal. When using Complex samples to record a single Sine wave, the y-axis values of the Sine wave are recorded along with the y-axis values of the corresponding Cosine wave. In this way, we are actually recording the details of the circle (or helix) from which the Sine wave could have been derived. When using Complex samples to record an impure signal, the y-axis values of the signal are recorded along with the y-axis values of the corresponding sum of Cosine waves (the corresponding-by-sum signal). [It would be necessary to calculate the details of the corresponding signal so that it could be sampled.] In this way, we are recording the shape (or the helix-like shape) from which the signal could be said to be derived. When recording waves or signals in this way, we will have two samples for each moment in time – one for the actual wave or signal, and one for the calculated corresponding wave or signal. [If we treated the original signal as a sum of Sine waves, we would need to find the corresponding sum of Cosine waves. If we treated the original signal as a sum of Cosine waves, we would need to find the corresponding sum of Sine waves.]

Saying this all again in a different way, for a pure wave or an impure signal, we store the coordinates of an object moving around the Complex plane (or the circle chart), from which the wave or signal could be derived. As the y-axis coordinates of an object moving around the Complex plane describe the vertically derived signal, and the x-axis coordinates describe the horizontally derived signal, we are storing a given signal and its corresponding-by-sum twin.

By storing the coordinates from circles or shapes as Complex samples (in other words, as coordinates), we are still including readings from a given signal, but the extra information makes some signal processing calculations much easier.

We can use several methods to calculate the corresponding signal for a given signal. The most straightforward would be to find the constituent waves of our signal, and then add up the corresponding waves for those constituent waves. The result would be the corresponding-by-sum signal.

Pure waves

As an example of using Complex samples, if we wanted to store a Sine wave, we would store the coordinates of an object moving around a circle for which the y-axis values produced that Sine wave. The y-axis values would describe the Sine wave itself, and the x-axis values would describe the Sine wave's corresponding Cosine wave. Putting this in terms of Complex numbers, the Imaginary axis values would describe the Sine wave, and the Real axis values would describe the Sine wave's corresponding Cosine wave.

If we had the Sine wave, "y = 3 sin (2π * 7t)", the coordinates of the object rotating around the circle from which that wave was, or could have been, derived can be given by:
(3 cos (2π * 7t), 3 sin (2π * 7t))

To put this in terms of Complex numbers, the object's position at any moment in time would be:
3 cos (2π * 7t) + 3i sin (2π * 7t)

The x-axis (Real axis) coordinates are described by:
"3 cos (2π * 7t)"
... which is the corresponding wave to our:
"y = 3 sin (2π * 7t)"
... wave. Therefore, when using Complex samples to store:
"y = 3 sin (2π * 7t)"
... we store the samples from:
"y = 3 cos (2π * 7t)" and "y = 3 sin (2π * 7t)" together.

When it comes to recording the readings from each wave to a computer file, we would usually store one reading from the Cosine wave, followed by one reading from the Sine wave for the same time, then another reading from the Cosine wave, followed by another reading from the Sine wave for the same time, and so on. In this way, the samples from each wave are "interleaved" with each other.

Another way we could store the readings is to have all of one wave's readings in one group, and then have the other wave's readings in a second group afterwards. The downside to doing this is that we have to read from (or write to) two completely different places within the file to get one pair of coordinates. The other method that keeps the readings as pairs means that we only need to read two consecutive numbers for each coordinate.

If we were treating a given pure wave as a Cosine wave, then we would find its Sine twin, and store the readings in the same way.

Impure signals

If we wanted to store a signal that was being treated as a sum of Sine waves, we would imagine the shape from which that signal was, or could be said to be, derived. The shape would portray the path of an object moving around the x and y-axes (or the Complex plane). The original signal (the sum of Sine waves) would be the vertically derived signal, and the corresponding signal would be the horizontally derived signal (the sum of Cosine waves). Another way of saying this is that we store readings from the signal's corresponding-by-sum signal, as well as readings from the signal itself. By convention, we would store the samples with the horizontally derived signal first, and the vertically derived signal second.

We would use the same idea if we were storing a signal that was being treated as the sum of Cosine waves. We would imagine the shape from which the signal was, or could have been, derived, and store the horizontally derived signal and the vertically derived signal as before.

If we had the signal:
"y = 2 sin (2π * 3t) + 0.5 sin (2π * 11t)"
... then we would find its corresponding-by-sum signal, which is:
"y = 2 cos (2π * 3t) + 0.5 cos (2π * 11t)"
... and store readings from both signals.

If we had the signal:
"y = 2 cos (2π * 15t) + 12 cos (2π * 1000t)"
... then we would find its corresponding-by-sum signal, which is:
"y = 2 sin (2π * 15t) + 12 sin (2π * 1000t)"
... and store readings from both signals.

If we did not have a formula for a signal, then we would have to calculate the corresponding signal. If the given signal is periodic, we can use Fourier series analysis to find the signal's constituent waves, and then we would add up the corresponding waves to find the corresponding signal. If the signal is not periodic, we could use the Fourier transform to do this. There are also other more complicated methods to calculate the corresponding signal.

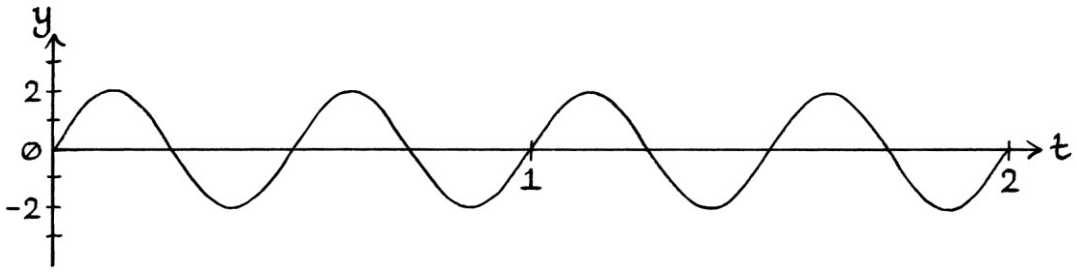
Example 1

We will say that we have been given the wave:
"y = 2 sin (2π * 2t)"
... and that we want to store it as Complex samples.

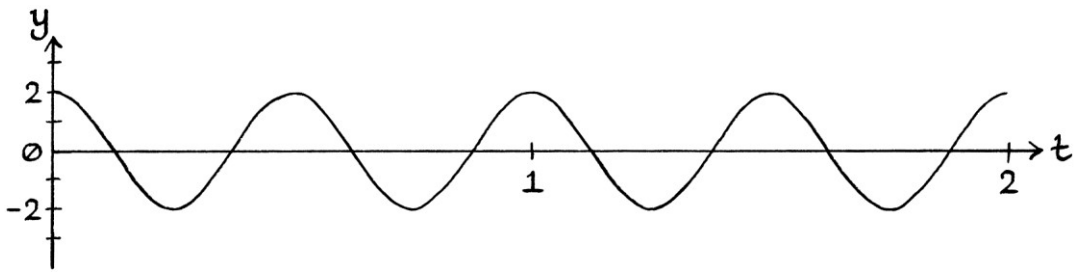
Our first step is to find the corresponding wave, which is:
"y = 2 cos (2π * 2t)"

[We could also have started this example by saying that we had an object rotating at 2 cycles per second around a circle with a radius of 2 units, and we wanted to store its coordinates over time – this would amount to exactly the same thing.]

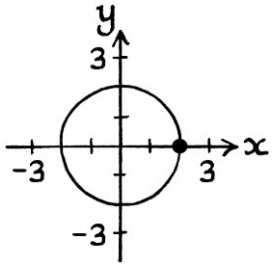
The Sine wave looks like this:



Its corresponding Cosine wave looks like this:



The circle from which these two waves are derived looks like this:



We will take readings from each wave at intervals of 0.05 seconds. This is a sample rate of 20 samples per second. We will record two seconds' worth of samples, and we will turn them into signed bytes. So that we end up with samples in an "x, y" ["Real, Imaginary" or "In-phase, Quadrature"] order, we will put the Cosine wave samples first. This is an arbitrary but generally accepted choice – we could store the values with the Sine wave first, but given that the values can also be thought of as coordinates, it is better to put them in an "x, y" order. The readings from the Cosine wave are also the x-axis coordinates of an object rotating around a circle; the readings from the Sine wave are also the y-axis coordinates of an object rotating around a circle. [As we are thinking in terms of the Complex plane, the x-axis is really the Real axis, and the y-axis is really the Imaginary axis.] The first few samples are as so:

Time (seconds)	**Cosine wave (Real, In-Phase, or x-axis) reading. (Decimal)**	**Sine wave (Imaginary, Quadrature, or y-axis) reading. (Decimal)**
0.00	2	0
0.05	1.6180	1.1756
0.10	0.6180	1.9021
0.15	−0.6180	1.9021
0.20	−1.6180	1.1756
0.25	−2	0
0.30	−1.6180	−1.1756
0.35	−0.6180	−1.9021
0.40	0.6180	−1.9021
0.45	1.6180	−1.1756
0.50	2	0
0.55	1.6180	1.1756
0.60	0.6180	1.9021
0.65	−0.6180	1.9021

0.70	−1.6180	1.1756
0.75	−2	0
0.80	−1.6180	−1.1756
0.85	−0.6180	−1.9021
0.90	0.6180	−1.9021
0.95	1.6180	−1.1756
1.00	2	0
1.05	1.6180	1.1756

... and so on.

We will convert the readings into signed bytes. To do this, we will scale the readings so that they best fit into the available values in a signed byte. The maximum and minimum y-axis values from the curves of the waves are +2 and −2 units. More importantly, the maximum and minimum *readings* from the waves are +2 and −2 units. If we scale all the readings by 63.5, then the maximum reading will become +127, and the minimum reading will become −127. [Note that this scaling assumes that we will not have any larger values to encode if the wave should suddenly change in instantaneous amplitude. It is also worth noting that in this example, the two corresponding waves have the same maximums and minimums, which makes things simpler than they could be.]

After the scaling, we need to turn the values into integers, and then convert them into signed hex bytes.

For practice's sake, we will go through the first few entries by hand. The first Cosine entry is +2. We multiply this by 63.5 to get +127. To convert 127 into hex, we divide it by 16. The result is 7 and a remainder of 15. The number 15 in decimal is 0xf in hex. We put it into the ones number column. The result of 7 is put into the sixteens column, and we end up with 0x7f. The first Sine entry is 0. We multiply it by 63.5, and we still have 0. We convert it to hex, and we have 0x00.

The second Cosine entry is 1.6180. We multiply it by 63.5 to get 102.743. As an integer, this is 103. We divide it by 16 and get 6 and a remainder of 7. The 7 goes in the ones column, and the 6 goes in the sixteens column to produce 0x67. The second Sine wave entry is 1.1756. We multiply it by 63.5 to get 74.6506, which is 75 as an integer. We divide it by 16 to get 4 and a remainder of 11, which is 0xb in hex. We put the remainder in the ones column and the 4 in the sixteens column to produce 0x4b.

The third Cosine entry is 0.6180. We multiply it by 63.5 to get 39.243, which is 39 as an integer. We divide it by 16 to get 2 and a remainder of 7, which ends up as 0x27. The third Sine entry is 1.9021. We multiply it by 63.5 to get 120.78335, which is 121 as a decimal integer. We divide it by 16 to get 7 and a remainder of 9, so the number is 0x79 in hex.

The fourth Cosine entry is −0.6180. We multiply it by 63.5 to get −39.243, which is −39 as an integer. To convert this to a signed byte, we make it positive (it becomes +39), convert it to hex (it becomes 0x27), subtract 1 (it becomes 0x26), and then flip the bits (or subtract it from 0xff). In binary, 0x26 is 00100110, so after the bits are flipped, it becomes 11011001, which is 0xd9 in hex. The fourth Sine number is 1.9021, which as we saw before, becomes 0x79.

The first part of a table showing the converted readings is as so:

Time (seconds)	**Cosine wave reading (signed byte)**	**Sine wave reading (signed byte)**
0.00	0x7f	0x00
0.05	0x67	0x4b
0.10	0x27	0x79
0.15	0xd9	0x79
0.20	0x99	0x4b

... and so on.

We will store the signed bytes for both the Cosine wave and the Sine wave in a file. To do this, we will put down the bytes in pairs, with each pair containing the Cosine byte for one moment in time followed by the Sine byte for that time. Therefore, the file will begin:
0x7f, 0x00, 0x67, 0x4b, 0x27, 0x79, 0xd9, 0x79, 0x99, 0x4b and so on.

If we were to open the completed file in a hex editor, we would see this:

```
0000: 7f 00 67 4b 27 79 d9 79 99 4b 81 00 99 b5 d9 87 ..gK'y.y.K......
0010: 27 87 67 b5 7f 00 67 4b 27 79 d9 79 99 4b 81 00 '.g...gK'y.y.K..
0020: 99 b5 d9 87 27 87 67 b5 7f 00 67 4b 27 79 d9 79 ....'.g...gK'y.y
0030: 99 4b 81 00 99 b5 d9 87 27 87 67 b5 7f 00 67 4b .K......'.g...gK
0040: 27 79 d9 79 99 4b 81 00 99 b5 d9 87 27 87 67 b5 'y.y.K......'.g.
```

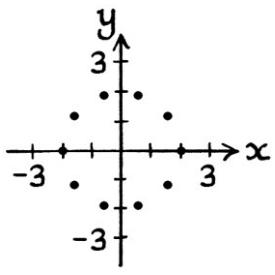
There are 0x50 (in hex) signed bytes in total, which is 80 bytes in decimal. This is 40 pairs of bytes, which, as we are using a sample rate of 20 samples per second, is 2 seconds' worth. Although the ASCII column is just showing coincidences where the signed bytes happen to share the same value as valid ASCII characters, we can use it as a guide to how often the cycles repeat. The ASCII characters "gK'y" repeat 4 times, which suggests [but does not prove] that the cycles of the two waves repeat 4 times in the time that we have recorded them. [It does not prove this because our sample rate might not coincide with the cycles of our wave – there might be peaks that occur between the samples. If we sampled at, say, 33.3333 samples per second, the peaks of the waves would line up with the samples in a different way each cycle. The patterns in the bytes would be harder to recognise.]

We can use the data in the file for various purposes:

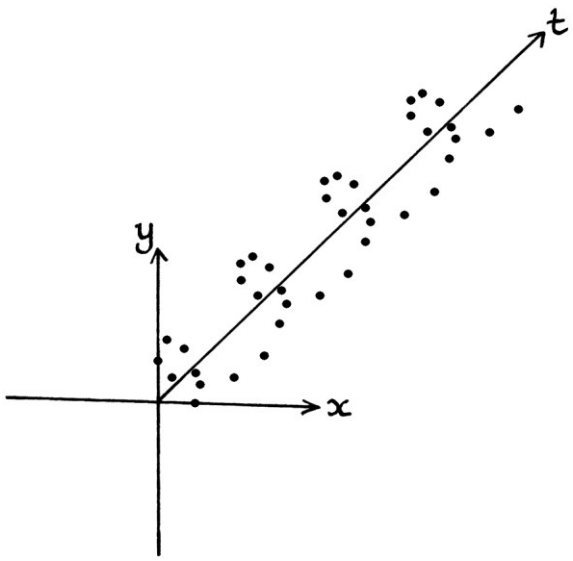
- We can recreate a discrete version of the circle from which our original Sine wave was, or could have been, derived. To do this, we read pairs of bytes, un-scale them by dividing by 63.5, and use the first as the x-axis coordinate, and the second as the y-axis coordinate on the Complex plane or the circle chart.
- We can recreate a discrete version of our original Sine wave. To do this, we read the second byte of each pair, un-scale it by dividing by 63.5, and use it as the y-axis value on a wave graph.
- We can recreate a discrete version of the original Sine wave's corresponding Cosine wave. To do this, we read the first byte of each pair, un-scale it by dividing by 63.5, and use it as the y-axis value on a wave graph.

Having the data as Complex samples also allows some mathematical processes to be performed more easily than if we just had the data for our Sine wave. The compromise being that we need twice as much room to store the data.

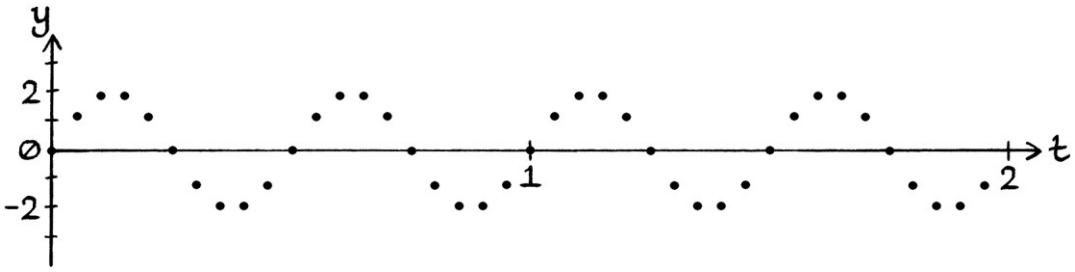
The discrete circle created from the data in the file looks like this:



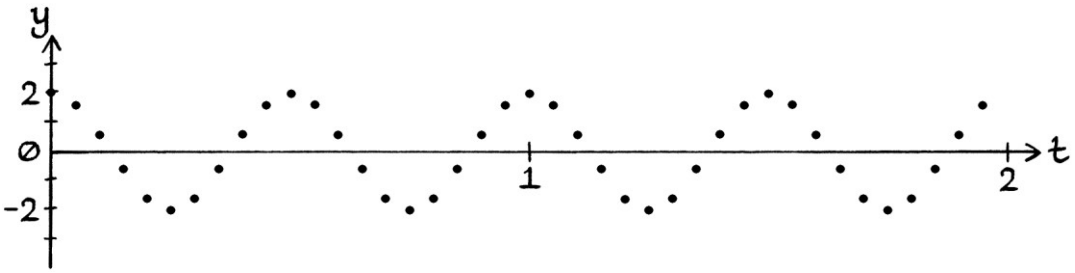
The helix, being the circle viewed from a different perspective, looks like this:



The discrete Sine wave looks like this:



Note how the discrete Sine wave does not contain samples exactly at the peaks of the original continuous Sine wave. The discrete Cosine wave looks like this:



[Note that if we were thinking of the circle chart, then each pair of samples would contain an x-axis value and a y-axis value. If we were thinking of the Complex plane, then each pair of samples would contain a Real value and an Imaginary value. If we were thinking in terms of the way that the Complex plane is treated in signal processing, then each pair of samples would contain an In-phase value and a Quadrature value. If we are thinking in terms of waves, then each pair of samples

would contain a Cosine wave value and a Sine wave value. All of these are really just different ways of expressing the same thing.]

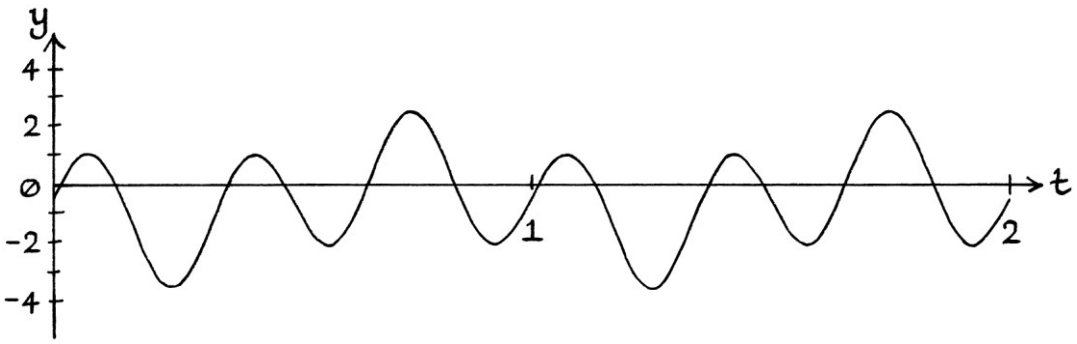
Example 2

We will say that we have been given this signal:
"y = −0.5 + sin ((2π * 1t) + π) + 2 sin (2π * 3t)"
... and that we want to store it as Complex samples in the form of signed words.

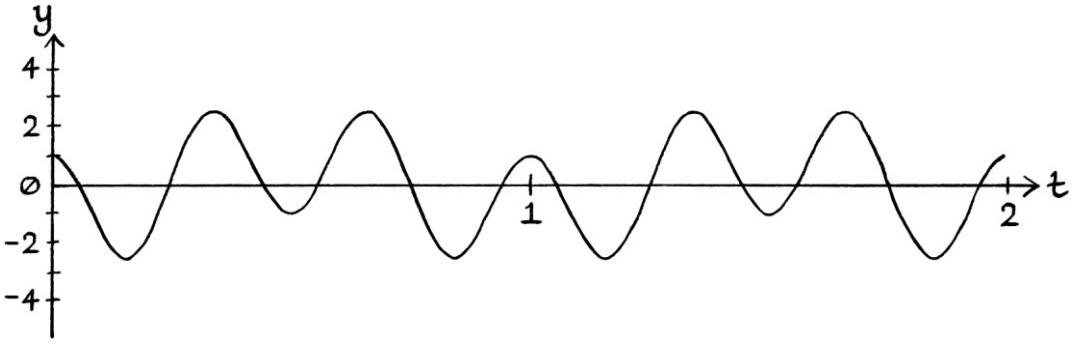
We cannot know the mean level of the corresponding signal, so we will set it to zero. The corresponding signal will therefore be:
"y = cos ((2π * 1t) + π) + 2 cos (2π * 3t)"

[We could also have started this example by saying that we had an object moving around the Complex plane, which had coordinates expressed by these two wave formulas. Everything in this example would be the same.]

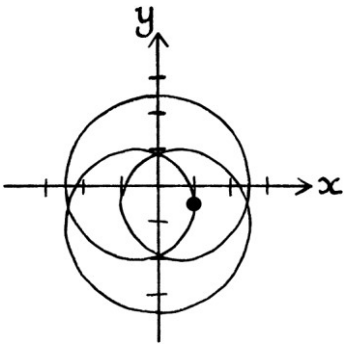
The original signal, which we will call the vertically derived signal, looks like this:



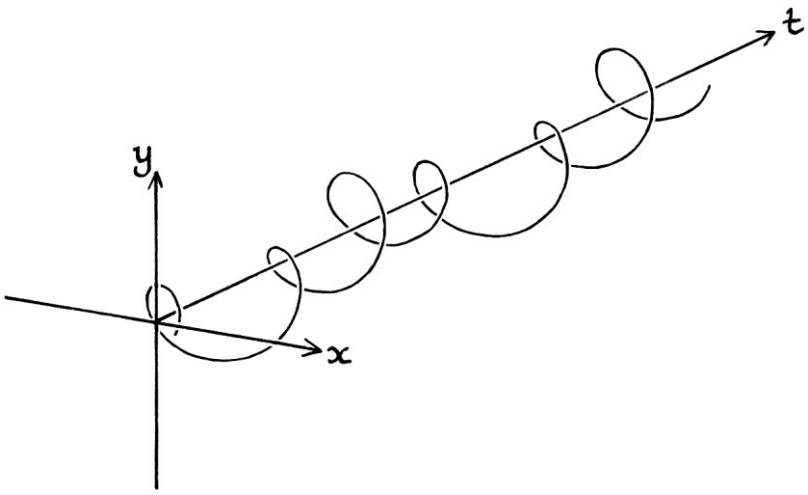
The corresponding signal, which we will call the horizontally derived signal, looks like this:



The shape from which the two signals are derived looks like this:



The helix that represents the shape and the two signals looks like this:



We will take readings from each signal with a sample rate of 20 samples per second. We will record two seconds' worth of samples, and we will turn them into signed words.

The first few readings from each signal are as so:

Time (seconds)	**Horizontally derived signal (decimal)**	**Vertically derived signal (decimal)**
0.00	1	−0.5
0.05	0.2245	0.8090
0.10	−1.4271	0.8143
0.15	−2.4899	−0.6910
0.20	−1.9271	−2.6266
0.25	0	−3.5
0.30	1.9271	−2.6266
0.35	2.4899	−0.6910
0.40	1.4271	0.8143
0.45	−0.2245	0.8090
0.50	−1	−0.5
0.55	−0.2245	−1.8090
0.60	1.4271	−1.8143
0.65	2.4899	−0.3090
0.70	1.9271	1.6266
0.75	0	2.5
0.80	−1.9271	1.6266
0.85	−2.4899	−0.3090
0.90	−1.4271	−1.8143
0.95	0.2245	−1.8090
1	1	−0.5
1.05	0.2245	0.8090

... and so on.

As we are going to be putting the values into signed words, we will need to scale the samples to make best use of the available space. Therefore, we have to find the largest absolute *sample* within both signals. [In other words, we need the largest of the values that, whether positive or negative to start with, is made positive. This is also the value that is furthest from zero in either direction.] The largest negative sample is −3.5 units, which occurs in the vertically derived signal at t = 0.25 seconds. The largest positive sample is +2.5 units, which occurs in the vertically derived signal at t = 0.75 seconds.

[Remember that we are not finding the largest positive or negative values from the original continuous signal's curve, or the corresponding signal's curve – we are finding the largest values from the *readings* that we made from those curves. These may or may not be the same.]

A signed word can contain values from −32,768 up to +32,767 in decimal, which is from −0x8000 to +0x7fff in hex. If our maximum and minimum samples had the same absolute value [the value when a number is made positive], we would need to scale all values to fit between −32,767 and +32,767 inclusive. In this particular example, the minimum (−3.5) is significantly larger in magnitude (3.5) than the maximum (+2.5). Therefore, we can scale every value to fit between −32,768 and +32,768, and we know that our minimum will be exactly −32,768 and our maximum will never reach anywhere near +32,768. This allows us to make the best use of the available range of two's complement numbers without going over the limits.

For −3.5 to be scaled to become −32,768, it must be multiplied by 9,362.2857 in decimal. Therefore, every value in our table must be multiplied by 9,362.2857, turned into an integer, and then turned into a signed 16-bit word.

The first sample from the horizontally derived signal is 1. We multiply this by 9,362.2857, and get 9,362.2857, which is 9,362 as an integer. To convert this to hex, we divide it by 16, which results in 585 and a remainder of 2. We put the 2 in the ones number column. We then divide 585 by 16, which results in 36 and a remainder of 9. We put the 9 in the sixteens column. We then divide 36 by 16, which results in 2 and a remainder of 4. We put the 4 in the 256s column, and put the 2 in the 4096s column. Our final hex number is 0x2492.

The first sample from the vertically derived signal is −0.5. We multiply this by 9,362.2857 and we get −4,681.1429, which is −4,681 as an integer. As this is a negative number, we make it positive, and convert it into a two's complement hex word in the standard way. The number 4,681 divided by 16 produces 292 and a remainder of 9. We put the 9 in the ones number column. We then divide 292 by 16 to produce 18 and a remainder of 4. We put the 4 in the sixteens number column. We then divide 18 by 16 to produce 1 and a remainder of 2. We put the remainder in the 256s column, and put the 1 in the 4096s column. Our hex number is 0x1249. As our original number was negative, we subtract 1 to produce 0x1248. We then flip the bits (or subtract it from 0xffff).

In binary, 0x1248 is:
0001 0010 0100 1000
... and with the bits flipped, it is:
1110 1101 1011 0111
... which is:
0xedb7 in hex.

We continue in this way, or more sensibly get a computer to do it, until we have done our full list of samples. We then put the samples in pairs, where the first of each pair is the sample from the horizontally derived signal, and the second in each pair is the sample from the vertically derived signal. The start of our list of samples looks like this:

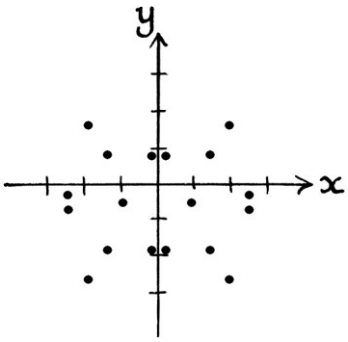
0x2492, 0xedb7, 0x0836, 0x1d96, 0xcbd0, 0x1dc8, 0xa4f2, 0xe6bb, and so on.

If we were to put the samples into a file in the "backwards" little-endian way, and we looked at the file in a hex editor, we would see this:

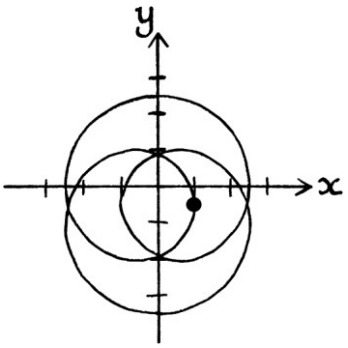
```
0000: 92 24 b7 ed 36 08 96 1d d0 cb c8 1d f2 a4 bb e6 .$..6...........
0010: 87 b9 f2 9f 00 00 01 80 79 46 f2 9f 0e 5b bb e6 ........yF...[..
0020: 30 34 c8 1d ca f7 96 1d 6e db b7 ed ca f7 d8 bd 04......n.......
0030: 30 34 a6 bd 0e 5b b3 f4 79 46 7c 3b 00 00 6d 5b 04...[..yF|;..m[
0040: 87 b9 7c 3b f2 a4 b3 f4 d0 cb a6 bd 36 08 d8 bd ..|;........6...
0050: 92 24 b7 ed 36 08 96 1d d0 cb c8 1d f2 a4 bb e6 .$..6...........
0060: 87 b9 f2 9f 00 00 01 80 79 46 f2 9f 0e 5b bb e6 ........yF...[..
0070: 30 34 c8 1d ca f7 96 1d 6e db b7 ed ca f7 d8 bd 04......n.......
0080: 30 34 a6 bd 0e 5b b3 f4 79 46 7c 3b 00 00 6d 5b 04...[..yF|;..m[
0090: 87 b9 7c 3b f2 a4 b3 f4 d0 cb a6 bd 36 08 d8 bd ..|;........6...
```

The ASCII column shows where the individual bytes of each signed word happen to have the same values as valid ASCII characters. The hex editor has no knowledge of the types of numbers being stored, and treats all data as a sequence of bytes. Therefore, although our data is stored as signed words, which are two bytes long, the hex editor treats them as a sequence of bytes, and so shows the ASCII characters that would apply for those bytes. As usual when dealing with wave-related data in a hex editor, each ASCII character in the right-hand column is just a coincidence. However, the column is still useful for seeing how the data repeats. [Note that it will not always be the case that there will be hex bytes that happen to have valid ASCII characters, so we will not always be able to use the ASCII column to see how a signal repeats. In such situations, we just have to get better at recognising patterns in the hex numbers.]

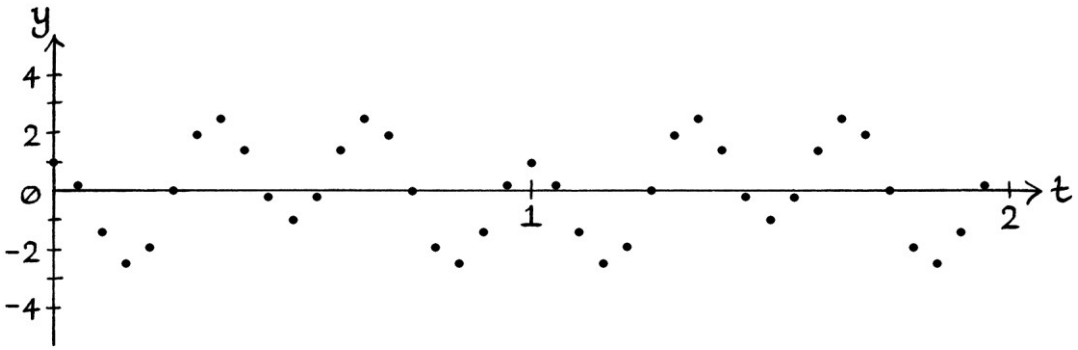
If we were to read the signed words from the file, divide them by 9,362.2857, and use the first of each pair of words as the x-axis coordinate, and the second of each pair as the y-axis coordinate, we would end up drawing this discrete version of the original shape:



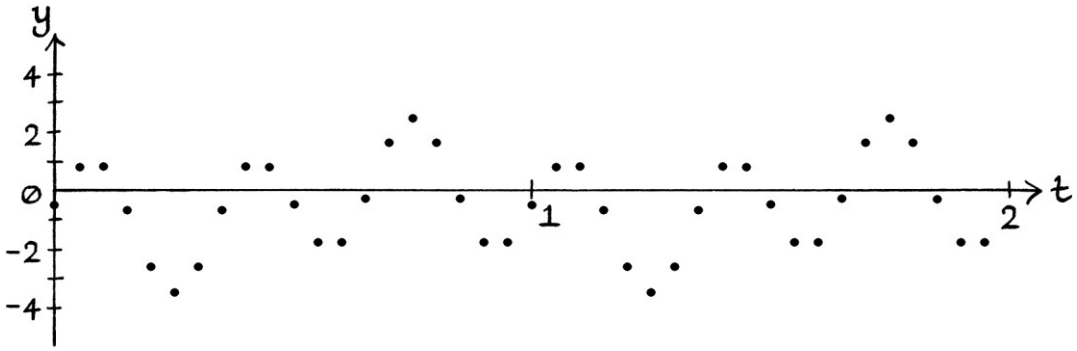
Our sample rate is too low for the actual shape to be clear. For comparison, the original continuous shape looked like this:



If we take the *first* value from each pair of samples, we can create a discrete *horizontally* derived signal that is just sufficient to know the shape of the original continuous horizontally derived signal:



If we take the *second* value from each pair, we can create a discrete version of the continuous *vertically* derived signal. This is the signal that we started with.



Ideally, in this example, we would have chosen a higher sample rate. However, a higher sample rate would have created a larger file and taken more time to do. As it is, the sample rate we chose is sufficient for portraying the general outline of both the original continuous vertically derived signal and its corresponding-by-sum horizontally derived signal. It is not sufficient for making a good picture of the shape made up from both continuous signals.

Creating the other signal for Complex sampling

Complex sampling is useful as it makes certain mathematical procedures easier. The difficult aspect of using Complex samples is that for any given signal, we need the corresponding signal. To put this simply, if we are given a signal that we are treating as the sum of Sine waves, then we need the signal that is the sum of the corresponding Cosine waves. To put this in signal processing terms, if we are given a signal, then we need its quadrature signal. If we can reduce a signal to its constituent waves, we will be able to recreate the corresponding signal. There are also other mathematical ways of calculating the corresponding signal. Many software-defined radios produce data in the form of Complex signals.

File headers

As we saw earlier in this chapter, if we are given a series of hex numbers in a file with no context, it is difficult, if not sometimes impossible, to know whether they represent:

- a series of unsigned bytes, words, dwords, qwords
- a series of signed bytes, words, dwords, or qwords
- a series of 16-bit, 32-bit, 64-bit or 80-bit floating-point numbers
- any one of the above in the form of pairs of Complex samples

If the data consists of a number type larger than a byte, then it is also difficult to know whether the bytes have been stored in the "backwards" little-endian way or the "forwards" big-endian way.

[It is possible to observe patterns in the bytes, and make likely deductions. For example, for a sufficiently high sample rate and a fairly simple signal, we could presume that the values representing a curve would not jump very far between samples. Therefore, two consecutive values could be presumed to be reasonably close. If that is the case, we could try different number types until we stumbled across one that was consistent with this.]

If we have no context to the data in a file, we would not be able to know the sample rate, but depending on what we were doing, this might not be a problem.

A solution to the ambiguities of a file's data is to have a note at the start of the file that describes useful attributes, so that people and programs can know how to work with the data. Such a thing is called a "file header". A header is a section at the beginning of a file that explains everything about the data contained in the rest of the file. A person or program can read the header and then know how to use the data in the intended way.

For a file containing signal data, the header could contain the sample rate, the number type, by how much the contained values were scaled (if at all), the endianness, and whether the data was made up of Complex samples or not. It would be best to put the endianness first so that we knew how to read the rest of the header. This could be a 32-bit unsigned dword that was zero to indicate little-endian and 0xffffffff to indicate big-endian. In this way, the endianness of the number explaining the endianness will not matter when we read it.

We could put the sample rate as an unsigned 32-bit dword.

We could put the number type in the form of an unsigned 32-bit dword set to:

- 0x00000001 to indicate unsigned bytes
- 0x00000002 to indicate signed bytes
- 0x00000003 to indicate unsigned words
- 0x00000004 to indicate signed words
- 0x00000005 to indicate unsigned dwords
- 0x00000006 to indicate signed dwords
- 0x00000007 to indicate unsigned qwords
- 0x00000008 to indicate signed qwords
- 0x00000009 to indicate 16-bit floating-point numbers
- 0x0000000a to indicate 32-bit floating-point numbers
- 0x0000000b to indicate 64-bit floating-point numbers
- 0x0000000c to indicate 80-bit floating-point numbers

We could put by how much the contained values were scaled, if at all, as a 32-bit *floating-point* number.

Whether the data is in the form of Complex numbers could be stored as an unsigned 32-bit dword with 0x00000000 meaning it does not contain Complex numbers, and 0x00000001 meaning it does contain Complex numbers.

Note that this is just one possible type of header that we could have. We could include any information that we wanted in any order, but the more information we include, the larger the file will be.

For our previous Complex sample example, the header would contain the following information:

- It is little-endian, which we would represent with 0x00000000
- The sample rate is 20 samples per second. This is 0x00000014 as a hex dword.
- The number type will be set to 4 to say that the signal is stored as signed words. This is 0x00000004 as a hex dword.
- The scaling amount is 9,362.2857. This is 0x46124925 as a 32-bit *floating-point* number.
- The number that indicates whether we are using Complex samples or not will be 1 to indicate that we are. This is 0x00000001 as a hex dword.

As a list, the header would be:
0x00000000, 0x00000014, 0x00000004, 0x46124925, 0x00000001

When stored in a file using the (backwards) little-endian method of putting bytes down, it would appear as so:

```
0000: 00 00 00 00 14 00 00 00 04 00 00 00 25 49 12 46 ................
0010: 01 00 00 00                                     ....
```

When our header is put at the start of the file containing the data, we would end up with this:

```
0000: 00 00 00 00 14 00 00 00 04 00 00 00 25 49 12 46 ............%I.F
0010: 01 00 00 00 92 24 b7 ed 36 08 96 1d d0 cb c8 1d .....$..6.......
0020: f2 a4 bb e6 87 b9 f2 9f 00 00 01 80 79 46 f2 9f ............yF..
0030: 0e 5b bb e6 30 34 c8 1d ca f7 96 1d 6e db b7 ed .[..04......n...
0040: ca f7 d8 bd 30 34 a6 bd 0e 5b b3 f4 79 46 7c 3b ....04...[..yF|;
0050: 00 00 6d 5b 87 b9 7c 3b f2 a4 b3 f4 d0 cb a6 bd ..m[..|;........
0060: 36 08 d8 bd 92 24 b7 ed 36 08 96 1d d0 cb c8 1d 6....$..6.......
0070: f2 a4 bb e6 87 b9 f2 9f 00 00 01 80 79 46 f2 9f ............yF..
0080: 0e 5b bb e6 30 34 c8 1d ca f7 96 1d 6e db b7 ed .[..04......n...
0090: ca f7 d8 bd 30 34 a6 bd 0e 5b b3 f4 79 46 7c 3b ....04...[..yF|;
00a0: 00 00 6d 5b 87 b9 7c 3b f2 a4 b3 f4 d0 cb a6 bd ..m[..|;........
00b0: 36 08 d8 bd                                     6...
```

As an analysis of our whole file:

At offset 0x0000, the file starts, and the first entry is a 32-bit unsigned dword that indicates the endianness of the file. As this is 0x00000000, it means that the data in the file is stored in the “backwards” little-endian way. A 32-bit dword takes up 4 bytes so the next part of the header starts at offset 0x0004.

At offset 0x0004, there is a 32-bit unsigned dword stored in the little-endian way that contains the sample rate. This is 0x00000014, which is 20 in decimal, which means 20 samples per second. This is 4 bytes long, so the next entry in the header starts at offset 0x0008.

At offset 0x0008, there is a 32-bit unsigned dword stored in the little-endian way that contains the number type. The dword is 0x00000004. In our list of number types, this means that the numbers in the file are stored as 16-bit signed words. This entry is 4 bytes long, so the next entry starts at offset 0x000c [12 in decimal].

At offset 0x000c, there is a 32-bit *floating-point number* stored in the little-endian way that contains the amount by which each original reading was scaled before it was converted to a signed word. This number is 0x46124925 in hex, which is 9,362.2857 in decimal. This is 4 bytes long, so the next entry will start at offset 0x0010.

At offset 0x0010, there is a 32-bit unsigned dword stored in the little-endian way that tells us whether the values in the data part of the file will be non-Complex single values or Complex pairs of values. This number is 0x00000001, which means the data consists of Complex pairs of values. This entry is 4 bytes long and is the final entry in the header. Therefore, the data containing the actual signal starts at offset 0x0014.

At offset 0x0014, the actual data section starts with our pairs of Complex samples.

The header we have here tells us everything that we need to know about the file. The downside is that we have to know how the header works in order to read the header. If someone did not know what the header meant, they would not know how to use the file. If someone did not know that there was a header there at all, they might misinterpret the header as being part of the signal data. Historically, particular file headers have become generally accepted because the files that use them have become extremely popular. They became popular either because an influential private company used the header for their own file format (such as Adobe's PDF files or Microsoft's Word files), or because an influential group of people have decided on that header (such as the Joint Photographic Experts Group's jpeg file). Our header here would be useful for our own purposes, but without general public acceptance, it would require an explanation for anyone who might want to use it. Most people like to use programs to open files, rather than read them with hex editors, so for the header to become popular, there would need to be some popular program that could read it.

As well as having headers, a file can contain different sections for different purposes. For example, a TIFF file can contain a header, the image data, and a smaller version of the image data so that programs can show quick previews without having to load all of the data section.

Nearly all popular file types have specifically defined header layouts. One notable exception is the ASCII text file, which has no header. One simple header is that of Unicode text files that use one 16-bit word for each letter (called UTF-16). These files have a header that is only two bytes long. The two bytes indicate that the file contains UTF-16 Unicode text, and also the endianness of the data representing the

text. If the file starts with 0xff, 0xfe, then the data is stored in the little-endian way. If the file starts 0xfe, 0xff, then the data is stored in the big-endian way. Most file headers are much more complicated and have multiple sections.

Some files representing saved radio signals might not have a header at all. This means that the file just contains the data and nothing else. This might make it easier for some programs to interpret the data because they can treat every number in the file as valid values. However, it also means that the data is likely to be misinterpreted if the program does not know in what form the data is stored.

WAV files

One type of header that is relevant to waves, and reasonably simple, is that of the Microsoft and IBM "WAV" file. This is also called a "wave" file or a ".wav" file on account of its file extension. A WAV file contains audio signal data in the form of consecutive samples. It is the closest popular file type to what we have been doing in this chapter. A WAV file has a header. It also has defined sections, each of which is marked by a piece of text and the size of the section following it. One of these sections will be the "data" section that contains the actual samples.

We will look at a WAV file lasting 0.1 seconds containing a 440 Hz tone. The file is 8,864 bytes in size. In a hex editor, the first part of the file looks like this:

```
0000: 52 49 46 46 98 22 00 00 57 41 56 45 66 6d 74 20 RIFF."..WAVEfmt
0010: 10 00 00 00 01 00 01 00 44 ac 00 00 88 58 01 00 ........D....X..
0020: 02 00 10 00 64 61 74 61 74 22 00 00 00 00 05 08 ....datat"......
0030: 01 10 ef 17 c2 1f 79 27 04 2f 62 36 8a 3d 72 44 ......y'./b6.=rD
0040: 17 4b 6f 51 77 57 26 5d 75 62 66 67 ea 6b 05 70 .KoQwW&]ubfg.k.p
0050: ad 73 e2 76 a0 79 e1 7b aa 7d ee 7e b9 7f ff 7f .s.v.y.{.}.~....
0060: c6 7f 0d 7f d6 7d 1a 7c e8 79 38 77 10 74 75 70 .....}.|.y8w.tup
0070: 68 6c ec 67 0e 63 c1 5d 24 58 1f 52 d6 4b 37 45 hl.g.c.]$X.R.K7E
0080: 55 3e 36 37 dc 2f 57 28 a4 20 d5 18 e7 10 ef 08 U>67./W(. ......
0090: e9 00 e4 f8 e8 f0 f4 e8 22 e1 65 d9 d6 d1 71 ca ........".e...q.
00a0: 44 c3 53 bc a6 b5 47 af 33 a9 7c a3 20 9e 24 99 D.S...G.3.|. .$.
00b0: 95 94 6c 90 b8 8c 74 89 ac 86 57 84 88 82 2c 81 ..l...t...W...,.
00c0: 5b 80 00 80 2f 80 d4 80 06 82 a8 83 d6 85 72 88 [.../.........r.
00d0: 8d 8b 1e 8f 1b 93 8e 97 5f 9c 9f a1 35 a7 2c ad ........_...5.,.
00e0: 71 b3 04 ba e1 c0 f6 c7 4c cf cc d6 7a de 49 e6 q.......L...z.I.
00f0: 2e ee 2a f6 2d fe 33 06 32 0e 23 16 fe 1d ba 25 ..*.-.3.2.#....%
0100: 52 2d ba 34 ee 3b e7 42 9a 49 05 50 20 56 e3 5b R-.4.;.B.I.P V.[
0110: 48 61 51 66 ea 6a 22 6f e2 72 33 76 09 79 6b 7b HaQf.j"o.r3v.yk{
```

... and so on.

We will go through the header entry by entry to see what they mean.

At offset 0x0000, the file starts with the four ASCII characters “RIFF”. The letters “RIFF” stand for “Resource Interchange File Format”. The WAV file layout is based on a more general file layout called the RIFF layout. The letters “RIFF” at the start of the file tell any program opening it that the file is based on this more general RIFF layout. This lets a program know how to deal with it. In practice, most programs expecting to be given a WAV file will treat the “RIFF” characters as a sign that this is a WAV file.

At offset 0x0004, there is the unsigned dword 0x00002298 stored in the “backwards” little-endian way. This is 8856 in decimal. This is the size of the entire file in bytes excluding the “RIFF” text and this dword. By mentioning this in the file, any programs reading the file can tell if the file has been mistakenly truncated, perhaps by a hard drive error or a recording program crashing before it had finished. It also allows a program to prepare memory into which it can load the rest of the file. The fact that this is stored as a dword means that the maximum number that can appear here is 0xffffffff, and therefore the maximum size of the rest of the file can only ever be 0xffffffff bytes, which is 4,294,967,295 bytes in decimal. This limits the maximum possible size of a WAV file.

At offset 0x0008, there are the bytes 0x57, 0x41, 0x56, 0x45, which are the ASCII characters “WAVE”. This tells programs that the file will be a WAV file. This is really the last part of the proper header. Following this are the “sections” of the file, each of which has its own mini header consisting of four bytes of ASCII text to identify it, followed by the size of that section. [Each section in a WAV file is usually called a “chunk”].

At offset 0x000c, there are the bytes 0x66, 0x6d, 0x74, 0x20, which are the ASCII characters “fmt” followed by a space. This is the title of this section of the file. The section title is a marker to indicate that we are at the start of the “fmt ” section of the WAV file header. The abbreviation “fmt ” is short for “format”. The “fmt ” section tells us detailed information about the data section within the file.

At offset 0x0010, there is the unsigned dword 0x00000010 stored in the little-endian way. This tells us the size of the “fmt ” section of the file in bytes excluding the four byte title and this dword. The “fmt ” section in this particular file is 16 bytes long (in decimal). As far as I know, it will always be 16 bytes long for Microsoft WAV files.

At offset 0x0014, there is the unsigned word 0x0001. This tells us that the data in the file is “PCM” data. “PCM” stands for “pulse coded modulation”. It essentially means that the data is stored as sequential y-axis values, in the same way that we have been storing data in this chapter. As far as I can tell, for a Microsoft WAV file, this will always be 0x0001, so the file will always be a PCM file.

At offset 0x0016, there is the unsigned word 0x0001. This tells us that the number of “channels” in the data is 1. There is 1 channel. A channel is the name given to one stream of data. One channel for our file means that it contains mono sound data. If the WAV file contained stereo sound data, then there would be two channels – a left channel and a right channel.

At offset 0x0018, there is the unsigned dword 0x0000ac44 stored in the little-endian way, which is 44,100 in decimal. This tells us the sample rate of the data in the file. There are 44,100 samples per second. This sample rate refers to the sample rate for each separate channel – therefore, for a mono sound signal, we have 44,100 samples per second for that signal. If we had a stereo sound signal, then we would have 44,100 samples per second for the right audio stream and 44,100 samples per second for the left audio stream.

At offset 0x001c, there is the unsigned dword 0x00015888 stored in the little-endian way. This is 88,200 in decimal. This tells us the number of bytes per second that are used to store the data. Coupled with the previous entry, this is a roundabout way of telling us that each sample is stored as a word. If there are 44,100 samples per second, and 88,200 bytes are used per second, then there must be two bytes per sample, and if we are using two bytes per sample, then we must be using one word to store each sample. Note that if the file contained stereo sound, and so used 2 channels, the number of bytes stored per second would be split between the two channels. This means that we would use 88,200 bytes per second *in total*, but 44,100 bytes per second for the left audio channel and 44,100 bytes per second for the right audio channel. This would imply that we would be using *bytes* to store each sample. Although this seems an awkward way of telling us the sample size, it could be useful to account for less common sampling methods.

At offset 0x0020, there is the unsigned word 0x0002 stored in the little-endian way. This shows the number of bytes being used per sample *for all channels together.* For our file, there is only one channel (a mono sound channel). This means that 2 bytes are used to store each sample, which in turn means that 1 word is used to store each sample. This confirms what we calculated from the previous entry. Supposing we had two channels (for stereo sound), then this would be split

between the channels. We would then have one byte for the right audio channel and one byte for the left audio channel.

At offset 0x0022, there is the unsigned word 0x0010, which is 16 in decimal. This indicates how many *bits* there are per sample for each channel. This really just confirms what we have been told twice already. However many channels this file has, there will be 16 bits (a word) for each sample. This is the last entry for the "fmt " section, which we know because we were told earlier that the "fmt " section is 0x10 bytes long.

At offset 0x0024, there are four ASCII characters that spell the word "data". This signifies the start of the data section.

At offset 0x0028, there is the unsigned dword 0x00002274, stored in the little-endian way. This is 8820 in decimal. This tells us the size of the data section within the file, excluding the text "data" and this dword. This is important to know in case there are any sections of the file after the data section. The WAV file format allows there to be many different sections, so the data section might not be the last section. This dword is also useful for programs that need to know how much memory to set aside for loading the whole data section.

From offset 0x002c to the end of the file, we have the samples for our audio signal in the form of signed 16-bit words stored in the "backwards" little-endian way. These are: 0x0000, 0x0805, 0x17ef, 0x1fc2, 0x2779, 0x2f04, 0x3662, 0x3d8a, 0x4472 and so on.

We can see that these numbers are slowly increasing in size. This is a good example of how it is sometimes possible to identify which number system is being used in a data file. If we had mistakenly thought that the samples were signed bytes or signed dwords, the values would not have increased as neatly as this, which would have suggested that we had made a mistake.

This particular file ends at the end of the data section. However, it is possible to have other sections in a WAV file that contain other information. It is also possible to add your own sections into a file that are for your own use. Programs that read WAV files are supposed to ignore sections that they do not understand. Therefore, we could add a section containing literally anything, and it should be ignored by audio software – as long as it had a four-letter section title that was generally not used by audio software, and the title was immediately followed by a dword stating the size of the section (excluding the title and the dword). In practice, WAV playing and editing software is often lazily written and does not follow the proper rules for

WAV files. Therefore, some programs will get confused by unknown sections. Another way that some audio software is badly written is visible in how some programs refuse to open WAV files that have missing parts of the header, even when those parts are irrelevant to decoding the file. For example, some programs will complain that a WAV file is corrupted if the file size dword after the initial "RIFF" signature is set to zero, even though they do not need to know this value to open the file. It would take almost literally no time for such a program to calculate what this value should be and repair the file. [It would not be difficult for a program to recreate all of the sections for most files from just the data in the data section. It would need to look for patterns in the bytes that indicated what type of number the data was stored as, and whether there was more than one audio stream.]

If we are recording audio to a WAV file and the recording software crashes and so does not get a chance to write in the dword after "RIFF" and the dword after "data", it might seem that the file is permanently corrupted. However, having now seen how the file format works, we can see how to repair it. We open the file in a hex editor [after having made a back-up copy]. We then enter the file size minus 8 bytes after the "RIFF" bytes as a "backwards" little-endian dword, and we enter the size of the data section minus 8 bytes after the "data" bytes, again as a little-endian dword. This will restore the WAV file to its proper layout. [When using a hex editor, you need to make sure that you *overwrite* where the values should go, as opposed to entering the values and pushing all the existing values to later in the file. If you do not overwrite, all the values in the header will be pushed along, and end up being interpreted incorrectly.] If we felt so inclined, if we only had the samples from the data section, we could recreate the full WAV file header, the "fmt " section of the header and the "data" entry of the header all by hand.

Thoughts on WAV files

We could use the WAV format to store non-audio data from the signals we saw earlier in this chapter, including Complex samples. The advantage in doing so would be that we would be using a generally accepted file format, so people would be able to understand the nature of the data without any further explanation. A Complex signal could be stored as if it were a stereo signal.

Files such as MP3s use compression to reduce the size of the data section. In its most basic implementation, compression might involve reducing a section of data containing dozens of zero bytes to one or two bytes that indicate how many zero bytes there were originally. As long as a program reading the file understands how

to identify and treat the bytes, doing this will make the file much smaller. In practice, compression usually involves much more complicated algorithms that are beyond the scope of this book. MP3 files store data by analysing the instantaneous constituent frequencies of a signal, removing those whose absence would not be noticed by human hearing, and performing complicated compression routines on the results. MP3s store data in a way that results in the accuracy of the measurements of the signal being reduced. If the actual values of the data were of importance, then MP3s would not be a good way to store them. Because the data in an MP3 file is compressed, if we viewed the data section in a hex editor, it is unlikely that we would be able to discern any obvious patterns.

WAV files are a good introduction to the other types of file layouts that are commonly used, although other file types are not as suited to storing signal data.

Analogue to digital converters

As we know, we can convert the graph of a continuous signal to a discrete signal by carefully measuring the y-axis values of evenly spaced (by time) points along the curve. When it comes to real-world waves, it is unlikely that a signal would fluctuate slowly enough to allow it to be recorded by hand, and it makes sense to automate the process. A device (or a part of an electronic circuit) that converts a continuous signal to a series of evenly spaced readings is called an "analogue to digital" converter or an "ADC" (or sometimes an "A/D converter"). It converts an analogue signal, or in other words, a continuous signal, to a digital signal, or in other words a discrete signal. It performs both sampling (in the sense that it takes readings at evenly spaced times) and quantization (in the sense that it converts the readings into a particular range of values).

Number sizes

Analogue to digital converters are categorised by the number of different values they can produce when representing a signal. For example, an 8-bit ADC produces samples that are each 8 bits in size – they are a byte long. That byte can represent 256 different values. A 16-bit ADC produces samples that are each 16 bits in size. Therefore, the samples it can produce can represent 65,536 different values.

We can calculate how many different values are possible with a certain number of bits by raising two to the power of that number. For example, a 32-bit ADC can produce 2^{32} = 4,294,967,296 different values.

A 10-bit ADC would be able to produce 2^{10} = 1024 different values. What is noteworthy about such an ADC is that the numbers it produces do not take up a full number type. Ten bits is more bits than a byte, but fewer bits than a word. Therefore, to store such numbers, we would have to put them into words (or higher), and the leftmost bits would always be zero. Placed into a word, the bits would range from the binary number:
0000 0000 0000 0000 [with spaces to make it easier to read]
... to:
0000 0011 1111 1111

In hex, these numbers are 0x0000 to 0x03ff. Technically, we could store the bits so that they extended between bytes, so there were no unused bits. This would save space, but would require more effort to store and load them. It would also be harder to identify a particular sample without knowing how many samples preceded it. The extra processing would make it inefficient. It is much better to store the samples as words, dwords or qwords.

The higher the number of bits an ADC can produce, the more detail there is, and the better the approximation of the signal that it is converting. This also means that the discrete signal will have a better dynamic range. However, the more bits an ADC can produce, the more expensive it will be, and the more storage the data will require. Similarly, the higher the *sample rate* that an ADC can manage, the more expensive the ADC will be, and the faster the data will need to be taken to its storage. A hard drive, for example, might not be able to store data fast enough to cope with an extremely fast sample rate. Having a very high number of bits and a very high sample rate is not as important as having a number of bits and a sample rate that can be easily processed.

An example of a situation where there is no need for a high number of bits or a high sample rate is a household digital thermometer. A household digital thermometer really only needs to measure to one decimal place, and it only needs to measure within a small range of temperatures. If it measured from −10.0 degrees Celsius to +40.0 degrees Celsius, it would need to identify 501 different values. It could therefore use a 9-bit ADC. [2^9 = 512.] It does not need to update the temperature particularly often, so it would be perfectly adequate with a sample rate of 2 samples per second. Conversely, the ADC in a digital thermometer in a

scientific laboratory would need to have a much higher bit size and a much higher sample rate.

ADCs appear in many everyday objects. An object that contains several is a modern mobile phone [cell phone]. It will have ADCs for receiving the mobile phone radio signal, for receiving Wi-Fi radio signals, for receiving Bluetooth radio signals, for encoding sound for phone calls, and for recording sound and video for the camera. It might also have other sensors that contain ADCs.

Digital to analogue converters

A device or electronic circuit that converts a discrete signal to a continuous signal is called a "digital to analogue converter" or "DAC". It does the opposite of an ADC. A good DAC does more than just "join up the dots" when converting to a continuous signal, but attempts to recreate the curve that might have existed in the original continuous signal.

A common misconception is that digital audio will always sound inferior to analogue audio because it is consists of discrete values. However, an MP3 player, for example, does not play audio as a series of discrete values, but plays a continuous sound. This is because a DAC converts the discrete signal into a continuous signal before the signal reaches the speakers or the headphones. Any discernable differences between the sounds from, say, an MP3 player and an entirely non-digital record player are probably due to the extra sounds created by the mechanism and structure of the record player.

Signals as lists of samples

A discrete signal can be thought of as a representation of a continuous signal. It is also possible to ignore time completely and treat a discrete signal as just a list of samples. When doing this, it is still possible to use, analyse, and perform maths on the samples.

In everyday life, if we have a list of items, we can identify any item in it by saying such things as "item number 1", "item number 2" and so on. With discrete signals, we can do the same thing. When we have a list of samples, we can identify any sample by saying such things as "sample number 1, "sample number 2" and so on. However, by convention, we identify the first sample as "sample number 0", the

second sample as “sample number 1”, and so on – we count upwards from zero. This convention comes from how if we were counting on a computer, we can count higher using a byte, word, dword, or qword if we start counting at zero. For example, the highest value a byte can hold is 0xff in hexadecimal (255 in decimal). If we start counting from 0x00, we can distinguish between 256 different values. If we start counting from 0x01, we can only distinguish between 255 different values.

In several computer-programming languages, such as C, when we refer to an item in a list, we place the item number in square brackets after the name of the list. For example, “mylist[22]” refers to item number 22 in the list called “mylist”. (In programming, a list is called an “array”.) If we had a list called “mysamples”, and in that list were the numbers:
1, 7, 23, 56, 712
... then we could say that:
mysamples[0] is 1
mysamples[1] is 7
mysamples[2] is 23
mysamples[3] is 56
mysamples[4] is 712

When it comes to discrete signals, maths uses similar notation, but instead of using descriptive names such as “mysamples”, it tends to use single letters such as “f” or “x”. Therefore, if we had this list of samples:
0, 0.3090, 0.5878, 0.8090, 0.9511, 1, 0.9511, 0.8090, 0.5878, 0.3090
... in a signal that we were referring to as “f”, we could say that:
f[0] is 0
f[1] is 0.3090
f[2] is 0.5878
f[3] is 0.8090
f[4] is 0.9511
f[5] is 1
f[6] is 0.9511
f[7] is 0.8090
f[8] is 0.5878
f[9] is 0.3090

If we do not know exactly to which item in the list we are referring, or if the actual item number is unimportant, we can replace the item value number with the letter “n”, such as “f[n]”. The letter “n” acts as a symbol for the item number.

We could say, “Add up every occurrence of f[n], where ‘n’ counts through the integers from 0 to 4.” This would add up the samples identified by
f[0]
f[1]
f[2]
f[3]
f[4]

We could state this more succinctly using a “Σ” sum (sigma sum):

$$\sum_{n=0}^{4} f[n]$$

We can use the letter “n” to mean every value from n = 0 upwards, in which case, “f[n]” becomes shorthand for saying “the signal as a whole”. In such a case, we could say, “Add 1 to f[n],” and it would mean add 1 to every sample in the entire signal. In this way, the use of “f[n]” is analogous to “f(t)”, where “f(t)” can be shorthand for a continuous time-based signal as a whole.

Note that some authors use round brackets instead of square brackets. It does not take long to get used to either method. Much older books use the far less convenient system of subscripts, such as “f_n”.

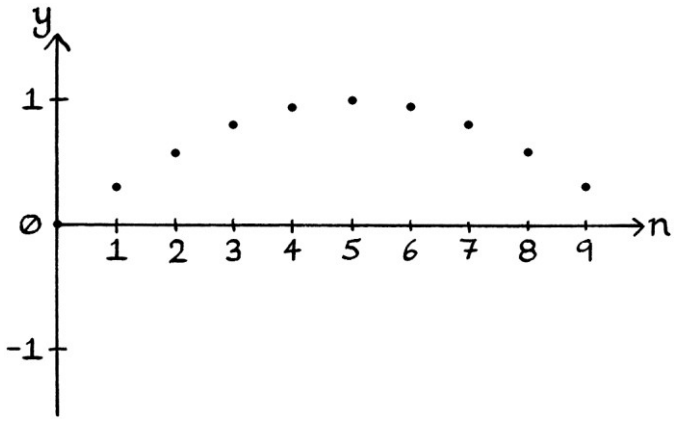
Some authors refer to the first item in a list as the “zeroeth” or “zeroth” item because it is item number 0, and they refer to the second item in a list as the first item because it is item number 1. Personally, I think this is a bad way to describe the items as it conflicts with the standard English language meaning of “first”, “second”, and so on. The “first item” should be the first item – if we are counting from item number 0, then the first item is item number 0. The word “first” cannot be redefined without making its meaning ambiguous elsewhere in a text. That is why in this book, I will use “first” in its standard way – to mean “that which comes before the others”. Note that just because an author misuses ordinal numbers does not imply that what they have to say is not useful. You will get used to the styles of other authors. It is possible to avoid any potential ambiguity by saying “item number 0”, “item number 1” and so on.

Graphs

We will say that we have this list of samples, but we have no idea as to the sample rate:

0, 0.3090, 0.5878, 0.8090, 0.9511, 1, 0.9511, 0.8090, 0.5878, 0.3090

We can create a graph that shows this list of samples with the x-axis being the sample number. The x-axis shows the value of "n". To make things clearer, we will call the x-axis the "n-axis":



Note how the n-axis consists of *integers* from zero upwards. There is nothing between the n-axis numbers – for example, the n-axis value of 1.1 does not exist on the number line because the n-axis consists only of integers. The n-axis is essentially counting through the samples, and we cannot have a non-integer number identifying samples. We cannot refer to, say, the 1.1^{th} sample, mainly because how we count items in general does not work like that. If we have any list of any items, each item would be at an integer place in that list.

There is no mention of time in the graph. We only have a list of samples, and the only information is the value of each sample and the order in which they appear.

Formula

We can make a formula that *describes* what the value of "y" would be in the graph:
"y = f[n]"
... where:

- "f" refers to the list of samples as a whole. It is shorthand for saying "the list of samples".
- "n" is the sample number, which is always an integer. It starts at 0 and increases in steps of 1. The letter "n" is acting as a symbol for the sample number.
- The presence of the square brackets indicates that we are referring to items in a list.

"f[n]" refers to the value of sample "n" in the list of samples being referred to as list "f".

Note how I say that the formula *describes* what "y" would be – we could not use the formula to *calculate* any of the values. The formula "y = f[n]" essentially means that for any sample number in the list of samples, "y" is the value of that sample. In many ways, it is not a particularly useful formula. However, sometimes, it is helpful to be able to make the statement that "y = f[n]".

x[n] and f[n]

In the above examples, I used "f[n]" to identify the samples. Among the other alternatives, you might also see "x[n]" with a lowercase "x". It does not matter which is used as long as the meaning is unambiguous and clear. While "x[n]" is fine, it might be confusing if we are talking about graphs with x-axes – the x-axis of a discrete signal graph is really the n-axis, so the "x" in "x[n]" does not refer to the x-axis. Another problem with "x[n]" is that most of the occasions that "x" is used in maths, it is a variable that changes. With "x[n]", it is the "n" that changes. Despite that, many people prefer "x[n]".

Non-time frequency

In continuous time signals, we measure frequency in cycles per seconds or hertz. In discrete signals *where we know the sample rate*, we can still use cycles per second or hertz because we still know how long a cycle takes. When we deal with time-based signals, either we can talk about normal frequency in terms of cycles per second or hertz, or we can talk about angular frequency in terms of radians per second or degrees per second. With radians, angular frequency is always 2π times faster than normal frequency. With degrees, angular frequency is always 360 times faster than normal frequency. Angular frequency tells us how many angles are passed per second. Angular frequency is an easier concept to understand when we think about the circle from which a wave is, or could have been, derived.

If we do not know the sample rate of a discrete signal, then we have no record of time. However, we can still measure frequency, but instead of using cycles per second, we use cycles per *sample*. In this way, we pay attention to how many cycles occur between each sample. Ideally, this will be a number less than one because if one or more cycles occur between two samples, we will have no record of them. In the following list of samples:
0, 1, 2, 3, 2, 1, 0, −1, −2, −3, −2, −1, 0, 1, 2, 3, 2, 1, 0, −1, −2, −3, −2, −1
... each cycle lasts 12 samples – sample number 0 is the same as sample number 12. Therefore, the time-less frequency is: 1 ÷ 12 = 0.8333 cycles per sample.

The proper name for frequency without time is “normalised frequency”.

We can have a normalised angular frequency, in which case, we measure the frequency in *radians* per sample. This is the cycles-per-sample frequency multiplied by 2π. The above list of samples has a normalised angular frequency of 0.5236 radians per sample, which is $\frac{\pi}{6}$ radians per sample.

It is easier to understand the concept of “radians per sample” if we think about the circle from which a wave is, or could have been derived. With the concept of “radians per sample”, we are counting how many radians an object rotating around a circle completes between each sample.

Discrete-time wave formula

Now we will look at a formula for a time-based discrete wave that allows us to calculate each sample – it is not referring to a pre-calculated list as in the previous section. The formula is:

$$y = sin\ (2\pi * f * T_s * n)$$

... where:

- Sine is working in radians.
- "f" is the normal cycles-per-second frequency.
- "T_s" is *not* the time, but the *sampling period*. In other words, the time between each sample. Time is usually represented by a lowercase "t", while this is an *uppercase* "T", which is usually the symbol for the period of a wave. An uppercase "T" with a subscripted "s" means that it is the period for samples – the sample period. It is important to notice this distinction as misreading it as a lowercase "t" changes the entire sense of the formula. Many people would say that having a capital "T" in a formula where we might expect a lowercase "t" is confusing. However, this is the convention for such a formula, and you will get used to it quickly.
- "n" is the sample number, counting from zero upwards in steps of 1. It is always an integer. It essentially acts as an index or indicator to the sample, but in this formula, it is being multiplied by 2π, the frequency, and the sample period. Its effect in the formula is analogous to that of time in a continuous wave. In a continuous wave formula, we are calculating the effect of all the components on time. In this formula, we are calculating the effect of all the components on "n". Time rises continuously; "n" rises in steps of one. The value of "n" is the only thing that changes in the formula in the same way that the value of "t" is the only thing that changes in a continuous formula.
- To keep the formula simple, for now, we will ignore mean level, amplitude and phase.

We can write the formula slightly more concisely as:

$$y = sin\ (2\pi f T_s * n)$$

[Note that some people do not put the "n" at the end of the formula. I put it at the end to emphasise how it is analogous to "t" for "time" in a continuous wave.]

As an example of the formula in use, we will look at a wave with an amplitude of 1 unit, a frequency of 1 cycle per second and a phase of zero radians. We will have a sample rate of 10 samples per second. If the sample rate is 10 samples per second, then the period of time between each sample is 1 ÷ 10 = 0.1 seconds. Therefore, "T_s" in our formula (the sampling period) will be 0.1 seconds. Our formula becomes:

"y = sin (2π * 1 * 0.1 * n)"
... which is:
"y = sin (2π * 0.1n)"

This formula will calculate every sample in our discrete signal.

We could also phrase the formula using "f[n]" instead of "y" as so:
"f[n] = sin (2π * 0.1n)"

This means the same thing, but instead of saying that we are calculating the y-axis values, we are saying that we are calculating each sample of the discrete signal "f".

For the first sample, which is sample number 0, or f[0], we will have:
sin (2π * 0.1 * 0)
... which is 0 units.

For the second sample – sample number 1 or f[1] – we will have:
sin (2π * 0.1 * 1)
... which is 0.5878 units.

For the third sample – sample number 2 or f[2] – we will have:
sin (2π * 0.1 * 2)
... which is 0.9511 units.

For the fourth sample – sample number 3 or f[3] – we will have:
sin (2π * 0.1 * 3)
... which is 0.9511 units.

For the fifth sample – sample number 4 or f[4] – we will have:
sin (2π * 0.1 * 4)
... which is 0.5878 units.

For the sixth sample – sample number 5 or f[5] – we will have:
sin (2π * 0.1 * 5)
... which is 0 units.

For the seventh sample – sample number 6 or f[6] – we will have:
sin (2π * 0.1 * 6)
... which is −0.5878 units.

For the eighth sample – sample number 7 or f[7] – we will have:
sin (2π * 0.1 * 7)
... which is −0.9511 units.

For the ninth sample – sample number 8 or f[8] – we will have:
sin (2π * 0.1 * 8)
... which is −0.9511 units.

For the tenth sample – sample number 9 or f[9] – we will have:
sin (2π * 0.1 * 9)
... which is −0.5878 units.

The next cycle starts at the eleventh sample – sample number 10 or f[10]. The sample is:
sin (2π * 0.1 * 10)
... which is 0 units.

Our full list of samples so far is:
0, 0.5878, 0.9511, 0.9511, 0.5878, 0, −0.5878, −0.9511, −0.9511, −0.5878, 0

All of the above shows that it is possible to have a formula that produces discrete values.

Our formula:
"$y = \sin (2\pi f * T_s * n)$"
... works in a similar way to a continuous time-based formula:
"$y = \sin (2\pi f * t)$"

However, unlike the time formula, where the part being Sined is increasing in infinitely small divisions, in the discrete formula, the part being Sined is increasing in multiples of an integer (because "n" is always an integer). The value of "n" starts at zero and increases in steps of 1. When it comes to discrete-signal formulas, "n" will always be an integer, and it will always increase in steps of 1. This is important

to remember, and it is the reason why our discrete formula produces a list of individual discrete samples.

We can make our discrete formula have all four characteristics of a wave. The formula becomes:
"$y = h + A \sin ((2\pi f * T_s * n) + \phi)$"
... where:

- "h" is the mean level. It has the same meaning as mean level in a normal continuous wave.
- "A" is the amplitude.
- Sine is working in radians.
- "f" is the frequency in cycles per second.
- "T_s" is the sampling period, which is the time between the samples.
- "n" is the sample number starting from zero and increasing in steps of 1. It is always an integer.
- "ϕ" is the phase in radians.

An example formula for a Sine wave is:
"$y = 1 + 5 \sin ((2\pi * 7 * 0.001 * n) + \pi)$
... which has a mean level of 1 unit, an amplitude of 5 units, a frequency of 7 cycles per second, a sampling period of 0.001 seconds, and a phase of π radians. A sampling period of 0.001 seconds is the same as a sample rate (a sampling frequency) of 1000 samples per second.

Sample rate

If we preferred a discrete formula with the sample rate instead of the sampling period, a simple version with zero mean level and zero phase would look like this:
"$y = A \sin ((2\pi * f * n) / samplerate)$"
... where I am using the word "samplerate" as a symbol for the sample rate to make the formula easier to read.

A version incorporating mean level and phase is harder to read, but is as so:
"$y = h + A \sin (((2\pi * f * n) / samplerate) + \phi)$"

In the above two formulas, the frequency, sample number and 2π frequency correction are all divided by the sample rate. This has the same effect as multiplying them by the sample period because the sample rate is equal to 1 divided by the sample period, and the sample period is equal to 1 divided by the sample rate.

We can make the formula more mathematical by referring to the sample rate as the sampling frequency and representing it with the symbol "f_s". The formula without mean level and phase is:

"$y = A \sin ((2\pi * f * n) / f_s)$"

The formula with mean level and phase is:

"$y = h + A \sin (((2\pi * f * n) / f_s) + \phi)$"

Using the symbol "f_s" makes the formula harder to understand if you have not been made aware of the difference between "f" and "f_s". The symbol "f" is the frequency of the wave – how many times the wave repeats its shape every second. The symbol "f_s" is the frequency of the sampling, as in the sample rate – how many samples we take every second. The symbols "f" and "f_s" are not an ideal choice of symbols to appear within the same academic subject, let alone the same formula. However, it is the mathematical convention, so it pays to get used to it. The more you see such symbols, the easier you will find it to distinguish between them. Sometimes, the cycles-per-second frequency might have a suffix too, such as "f_0" or similar.

In the above formulas where I include mean level, I have used the symbol "h". Previously in this book, I would have used "h_s" to indicate the mean level for a Sine wave and "h_c" to indicate the mean level for a Cosine wave. However, including a subscripted "s" or "c" in these formulas would add yet another source of potential confusion. If we needed to make the distinction between the mean levels, we could use the subscripted letters, but if it is obvious which mean level we are discussing, it makes the formulas clearer not to do so.

Formulas mentioning the sample rate require a division, and therefore, more brackets, which means that formulas are easier to read if we use the sampling period instead of the sample rate.

Duplicate signals

We can make a useful observation from the formula for a discrete wave. As we have seen, the formula for a discrete Sine wave is as so:
"$y = h + A \sin ((2\pi * f * T_s * n) + \phi)$"

The group of multiplications in the middle of the formula show that it is possible for different waves to produce the same samples. There are countless possible values for "f" and "T_s" for which "$f * T_s$" would produce the same result. Therefore, there are countless waves with different frequencies and different sampling periods that will have identical samples.

We can demonstrate this with some wave formulas. If we have this discrete wave:
"$y = \sin (2\pi * 2 * 0.01 * n)$
... then it will have identical samples to this wave:
"$y = \sin (2\pi * 1 * 0.02 * n)$
... and this wave:
"$y = \sin (2\pi * 0.5 * 0.04 * n)$
... and this wave:
"$y = \sin (2\pi * 0.25 * 0.08 * n)$
... and so on.

In each case, the frequency multiplied by the sampling period will be 0.02. Therefore, every wave formula will be the same as:
"$y = \sin (2\pi * 0.02 * n)$

Angular frequency wave formula

In more mathematical descriptions of waves, you are likely to see the formula:
"$y = \sin (2\pi * f * T_s * n)$"
... and its variations given in terms of angular frequency. Therefore, the formula would be given as so:
"$y = \sin (\omega * T_s * n)$"
... or:
"$y = \sin (\omega T_s * n)$"
... or:
"$y = \sin (\omega T_s n)$"

As we know, the letter "ω" is shorthand for "2π * f". In some situations, using "ω" makes things clearer as it uses fewer symbols, and so results in less cluttered formulas. In other situations, it can make the meaning of a formula slightly harder to discern.

The formula including phase and mean level would be given as so:
"$y = h + A \sin((\omega * T_s * n) + \phi)$"
... or:
"$y = h + A \sin((\omega T_s * n) + \phi)$"
... or:
"$y = h + A \sin(\omega T_s n + \phi)$"

Complex Sine formula

As we saw in Chapter 28, we can represent continuous Sine waves and Cosine waves in terms of Complex exponentials. A basic continuous Sine wave can be given with this formula:

$$z = \frac{1}{2i} * (e^{i(2\pi ft)} - e^{-i(2\pi ft)})$$

It is an anticlockwise helix minus a clockwise helix all rotated by 90 degrees clockwise, then halved. If we included amplitude, phase and the Real mean level, we would have this formula:

$$z = h_r + \frac{1}{2i} * (Ae^{i(2\pi ft + \phi)} - Ae^{-i(2\pi ft + \phi)})$$

... where:

- "h_r" is the mean level for the Real axis
- "A" is the amplitude
- "f" is the frequency in cycles per second
- "ϕ" is the phase in radians

We can change the formulas to produce *discrete* Sine waves. Because of the exponents and subscripts, these are easier to read in a larger font. If we ignore phase and mean level, the formula becomes:

$$z = \frac{1}{2i} * (e^{i(2\pi f * T_S * n)} - e^{-i(2\pi f * T_S * n)})$$

... where:

- "f" is the frequency in cycles per second
- "T_s" is the sampling period – it is the amount of time between each sample.
- "n" is the sample number. This will be an integer starting at zero for the first sample, and increasing by 1 each time.

The only difference between the continuous formula and the discrete formula is that the discrete formula replaces "t" (time) with "T_s * n" (the sampling period multiplied by the sample number). This is the same as with a normal non-Complex discrete formula.

If we include amplitude, phase and the Real mean level, the formula is as so:

$$z = h_r + \frac{1}{2i} * (Ae^{i((2\pi f * T_S * n) + \phi)} - Ae^{-i((2\pi f * T_S * n) + \phi)})$$

We can give this formula in terms of angular frequency as so:

$$z = h_r + \frac{1}{2i} * (Ae^{i((\omega * T_S * n) + \phi)} - Ae^{-i((\omega * T_S * n) + \phi)})$$

We can remove the superfluous multiplication signs to end up with the following, which is slightly harder to read:

$$z = h_r + \frac{1}{2i} * (Ae^{i(\omega T_S n + \phi)} - Ae^{-i(\omega T_S n + \phi)})$$

We can test that these formulas work. First, we will find sample number 1 of the discrete wave with the formula:

"y = 0.7 + 2.2 sin ((2π * 11 * 0.02 * n) + 0.25π)"

The discrete wave has a frequency of 11 cycles per second and a sampling period of 0.02 seconds, which means that it has a sample rate of 1 ÷ 0.02 = 50 samples per second.

To calculate sample number 1, we solve the following:
0.7 + 2.2 sin ((2π * 11 * 0.02 * 1) + 0.25π)
... which is:
0.7 + 2.2 sin (2.1677)
... which is:
2.5196

Now, we will use the discrete Complex exponential version of the Sine wave formula. It is:

$$z = 0.7 + \frac{1}{2i} * (2.2e^{i((2\pi * 11 * 0.02 * n) + 0.25\pi)} - 2.2e^{-i((2\pi * 11 * 0.02 * n) + 0.25\pi)})$$

For sample number 1, we need to calculate:

$$z = 0.7 + \frac{1}{2i} * (2.2e^{i((2\pi * 11 * 0.02 * 1) + 0.25\pi)} - 2.2e^{-i((2\pi * 11 * 0.02 * 1) + 0.25\pi)})$$

This is:

$$z = 0.7 + \frac{1}{2i} * (2.2e^{i(2.1677)} - 2.2e^{-i(2.1677)})$$

... which is:

$$z = 0.7 + \frac{1}{2i} * (2.2e^{2.1677i} - 2.2e^{-2.1677i})$$

The first exponential is a point on 2.2-unit-radius circle at an angle of 2.1677 radians. The second exponential is a point on a 2.2-unit-radius circle at an angle of −2.1677 radians. [The circles are flipped versions of each other.]

If we do not have a calculator that can work with Complex numbers, we can calculate the exponentials by drawing and measuring a circle, or more sensibly, by using Sine and Cosine.

The first exponential is: 2.2 cos 2.1677 + 2.2i sin 2.1677 = −1.2366 + 1.8196i
The second exponential is: 2.2 cos −2.1677 + 2.2i sin −2.1677 = −1.2366 – 1.8196i

The calculation as a whole becomes:

$$z = 0.7 + \frac{1}{2i} * (-1.2366 + 1.8196\text{i}) - (-1.2366 - 1.8196\text{i})$$

... which is:

$$z = 0.7 + \frac{1}{2i} * (-1.2366 + 1.8196\text{i} + 1.2366 + 1.8196\text{i})$$

... which is:

$$z = 0.7 + \frac{1}{2i} * (3.6392\text{i})$$

... which is:

$$z = 0.7 + 1.8196$$

... which is:

$$z = 2.5196$$

This is the same result as we had with the non-Complex calculation from earlier, which shows that our Complex exponential formula works.

As in Chapter 28, calculating it mostly by hand helps in understanding how Complex exponentials work. In this example, calculating it entirely with a calculator that can work with Complex numbers would take longer to do because it would require a lot of values to be entered and, depending on your calculator, a lot of brackets. If we had used such a calculator to solve the original calculation, we would have ended up with the same result.

Complex Cosine formula

A basic *continuous* Cosine wave can be given with this formula:

$$z = \frac{1}{2} * (e^{i(2\pi ft)} + e^{-i(2\pi ft)})$$

It is an anticlockwise helix added to a clockwise helix, then halved. If we include amplitude, phase and the Real mean level, the formula is:

$$z = h_r + \frac{1}{2} * (Ae^{i(2\pi ft + \phi)} + Ae^{-i(2\pi ft + \phi)})$$

The Complex exponential formula for a *discrete* Cosine wave is as so, shown in a bigger font to make it clearer:

$$z = h_r + \frac{1}{2} * (Ae^{i((2\pi f * T_S * n) + \phi)} + Ae^{-i((2\pi f * T_S * n) + \phi)})$$

Conclusion

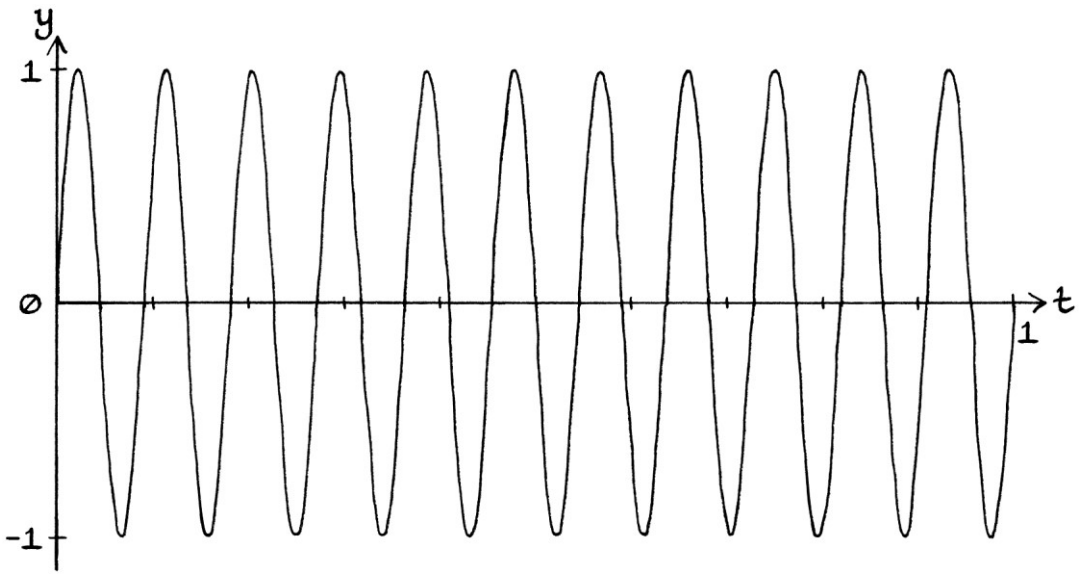
Although discrete signals have some complicated aspects related to sample rates, they can often be easier to comprehend and work with than continuous signals. The most obvious example of this is that computers can only sensibly work with discrete signals. For relatively high sample rates, relatively low frequencies, and a high level of accuracy for the samples, there is not a great deal of difference between a continuous signal and a discrete signal. The differences become more and more significant as sample rates decrease, frequencies increase, and the samples are recorded at a lower level of accuracy.

Chapter 42: Sampling

In this chapter, we will look at the various difficulties that arise when sampling waves and signals. If we sample a continuous signal and we do not have enough samples per second, we will end up portraying a completely different signal altogether. Similarly, the samples from any continuous signal could also have come from countless other faster signals.

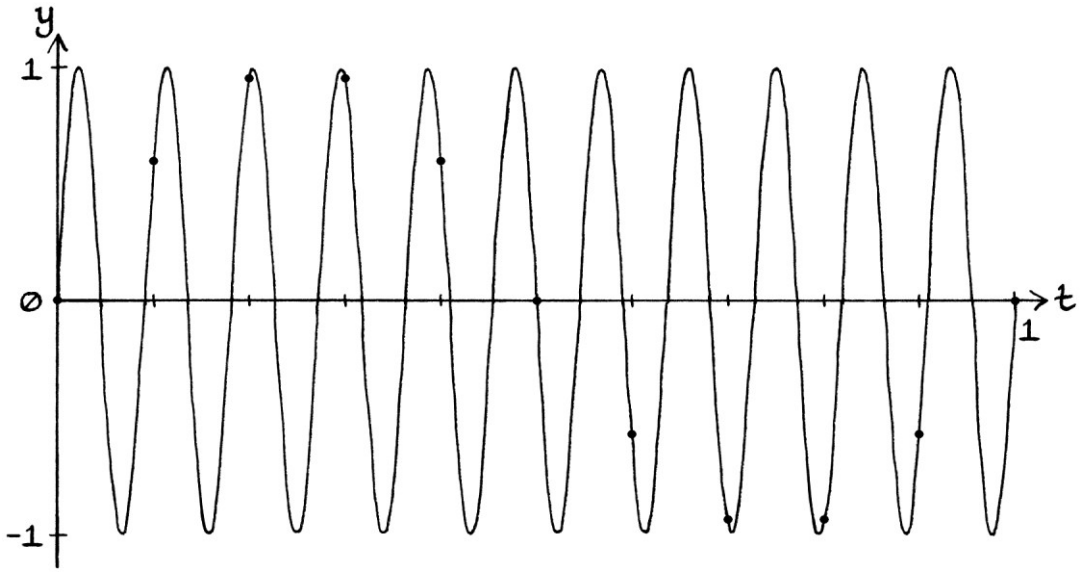
Undersampling

Not using enough samples per second to portray a given signal is called "undersampling". The consequences of undersampling are very significant in the study of discrete signals. As a simple introduction to undersampling, we will look at a continuous 11-cycle-per-second wave with the formula "y = sin (2π * 11t)":



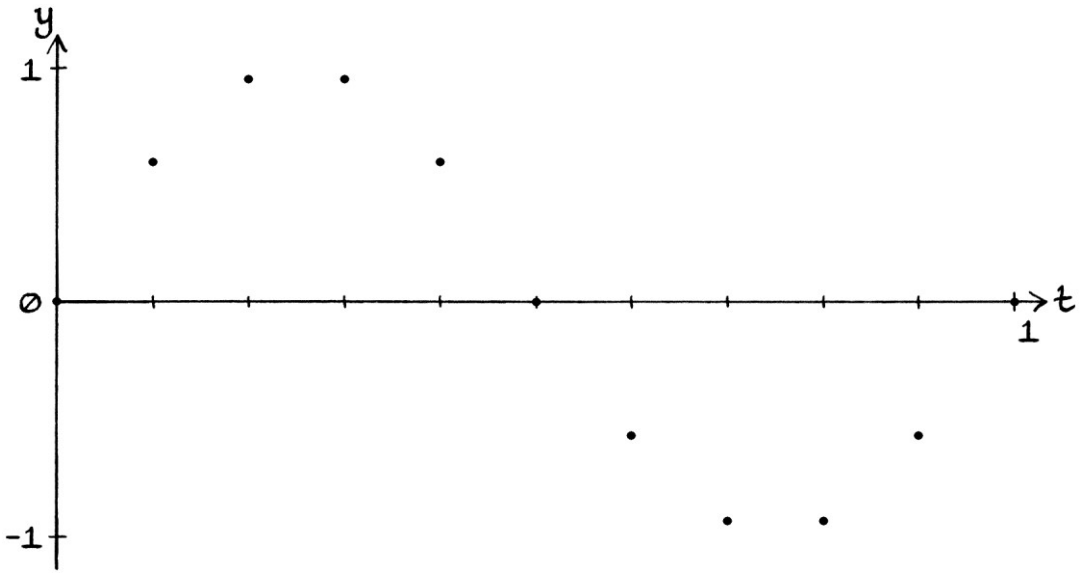
We will use a sample rate of 10 samples per second, and we will take samples from t = 0 up to t = 1 (which means that we will take 11 samples in all).

The samples marked on the graph are as so:



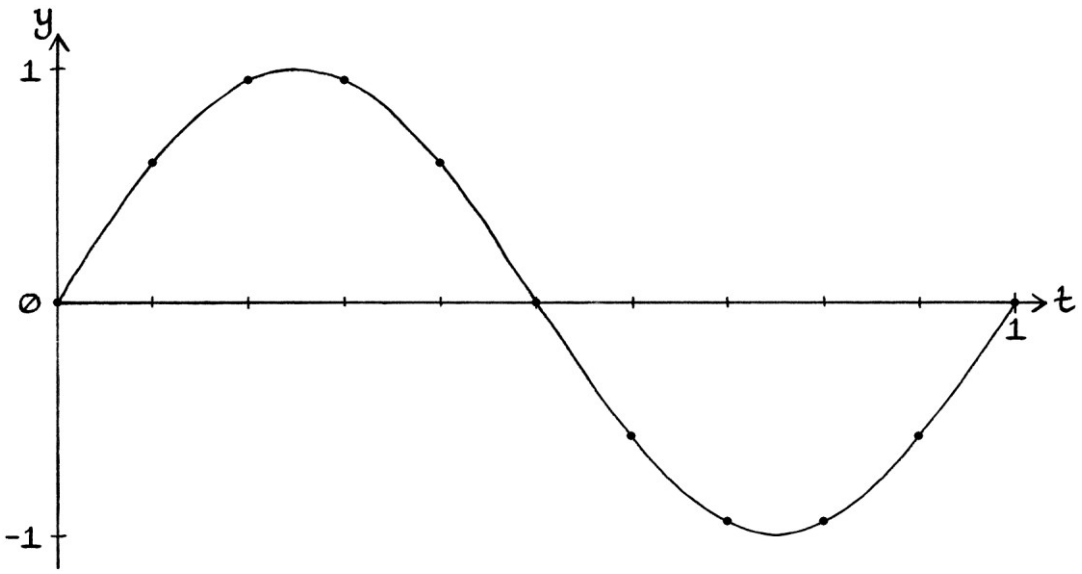
To four decimal places, the samples will be:
0, 0.5878, 0.9511, 0.9511, 0.5878, 0, −0.5878, −0.9511, −0.9511, −0.5878, 0

The samples drawn on their own are as so:



As we can see, there are not enough samples per second to portray our wave accurately. The sampling rate is too low to record the details of an 11-cycle-per-second wave. When looking at the list of samples for the 11-cycle-per-second wave, we would have no way of knowing that they were from a wave of that frequency. In fact, the samples we have are more representative of a 1-cycle-per-second wave.

If we draw the samples over the curve of a 1-cycle-per-second wave, we can see the connection more clearly:



By sampling our 11 cycle-per-second wave with a sample rate of 10 samples per second, we have ended up recording a wave with a frequency of just 1 cycle per second.

Duplicate signals

The way that two signals can have the same samples if they are sampled at a particular sample rate is more obvious if we look at some values. Regardless of the sample rate, a sample taken at t = 0.1 that has a value of 0.5878 could have come from any of the following unit-amplitude, zero-phase, continuous Sine waves:
“y = sin (2π * 1t)”
“y = sin (2π * 4t)”
“y = sin (2π * 11t)”
“y = sin (2π * 14t)”
“y = sin (2π * 21t)”
“y = sin (2π * 24t)”
“y = sin (2π * 31t)”
... and so on. There are an infinite number of unit-amplitude, zero-phase, continuous Sine waves that have a y-axis value of 0.5878 at t = 0.1.

Unit-amplitude, zero-phase, continuous Sine waves that have a y-axis value of 0.9511 at t = 0.2 are:
“y = sin (2π * 1t)”
“y = sin (2π * 6t)”
“y = sin (2π * 11t)”
“y = sin (2π * 16t)”
“y = sin (2π * 21t)”
“y = sin (2π * 26t)”
“y = sin (2π * 31t)”
... and so on.

Unit-amplitude, zero-phase, continuous Sine waves that have both a y-axis value of 0.5878 at t = 0.1, *and* a y-axis value of 0.9511 at t = 0.2, are:
“y = sin (2π * 1t)”
“y = sin (2π * 11t)”
“y = sin (2π * 21t)”
“y = sin (2π * 31t)”
“y = sin (2π * 41t)”
“y = sin (2π * 51t)”
... and so on.

In fact, all of these waves share the same values as each other for all time if they are only sampled at intervals of 0.1 seconds. This means that all of them will produce the same discrete signal for a sample rate of 10 samples per second.

The four types of ambiguity

We have just seen two different problems – first, by not having a fast enough sample rate, our 11-cycle-per-second wave is recorded as a 1-cycle-per-second wave. Second, for a sample rate of 10 samples per second, there are other waves with frequencies faster than 11 cycles per second that would have the same list of samples as an 11-cycle-per-second wave (and a 1-cycle-per-second wave). This leads us to two important aspects of sampling:

- If our sample rate is not fast enough, we will end up with a list of samples that refers to a different signal altogether. We need to have a sufficient sample rate for the wave or signal that is being analysed.
- Any list of samples could have come from countless other faster waves or signals. For any sample rate, there will *always* be faster signals that have the same samples as a given slower signal.

A third aspect of sampling, which we would see if we joined up the points on our sampled signal, is:

- The samples of any sampled *curved* signal are indistinguishable from those taken from a continuous signal with straight lines between those points.

There is a fourth ambiguity that is only relevant if we have a list of samples for which we do not know the sample rate:

- A list of samples that came from a signal with a particular frequency and sample rate could also have come from a signal with a faster frequency and a proportionally slower sample rate, or a slower frequency and a proportionally faster sample rate. This idea can be seen in the discrete wave formula:
  "$y = h + A \sin ((2\pi * f * T_s * n) + \phi)$"
  There are countless possible values for "f" and "T_s" for which "$f * T_s$" would produce the same result. Therefore, if we are presented with a list of samples and we do not know the sample rate, we cannot know the frequency or the sample rate from the list of samples. The frequency and sample rate are ambiguous.

The first two ideas are very important in signal processing. The third idea is important in certain situations, but it is usually overshadowed by the first two ideas. We will ignore the fourth idea in this chapter because we will only be looking at examples where we *do* know the sample rate.

The first three ideas mean that any list of samples might not represent the signal that it is intended to represent. Strictly speaking, any discrete signal is ambiguous. In this chapter, we will give names to the types of ambiguity to make it easier to refer to them in the future:

- The "Undersampling Ambiguity". If the sample rate is too slow, our samples will refer to a different signal. Given that, we cannot be sure that a set of samples was intended to represent the signal that it most closely matches.

- The "Faster Duplicates Ambiguity". Any list of samples could have come from a wave or signal with faster frequencies. Given that, we cannot be sure that our samples did not come from a faster frequency.

- The "Straight Line Ambiguity". The samples taken from a curved signal will be indistinguishable from those taken from a signal that had straight lines connecting those points. Given that, we cannot assume that the signal from which the samples came was a curved one.

All three ambiguities ultimately relate to how we cannot know the details of a discrete signal between its samples.

In this chapter, we will look at these three ideas in more detail.

Undersampling Ambiguity

The "Undersampling Ambiguity" refers to how if we sample a signal with a sample rate that is too low, we will end up recording our signal incorrectly. The samples will refer to a different signal from the one that we intended to record. The problem is called "undersampling" because the sample rate is *under* that needed to record the signal correctly.

The first sample rate rule

There are three useful rules for sampling a wave or a signal. The first rule is:

> "If we have a pure wave of a particular frequency, we must use a sample rate that is greater than twice that frequency. Otherwise, the wave will be recorded incorrectly, and analysis of the samples will find a different wave."

Expressed more concisely, this becomes:

> "The sample rate must be higher than twice the frequency of the wave."

To put this slightly more mathematically, the following must be true:

> "sample rate > 2 * frequency"

The rule is still true for constituent waves in sums of waves. Therefore, we can say:

> "If we have an impure periodic *signal*, we must use a sample rate that is greater than twice the fastest *constituent* frequency. Otherwise, the signal will be recorded incorrectly, and analysis of the samples will find a different set of constituent waves."

Expressed more concisely, this becomes:

> "The sample rate must be higher than twice the fastest *constituent* frequency within the signal."

To put this slightly more mathematically, the following must be true:

> "sample rate > 2 * fastest constituent frequency"

This "constituent wave" version of the rule confirms the idea from the first part of this book that different frequencies do not mix. If we add two or more waves of different frequencies together, it is always possible to separate them again – they do not mix. This is why Fourier series analysis works. With sampling, the "unmixability" of frequencies is relevant again – whether a wave is on its own or whether it is part of a sum of waves, it will not be recorded correctly unless the sample rate is over twice its frequency. If a constituent wave is recorded incorrectly, then the signal as a whole will be recorded incorrectly. [There is probably a way of finding the constituent waves of a signal by sampling it at a range of different sample rates, and observing the results.]

We can think of the rule in reverse. For any given sample rate, there is a maximum frequency that we can record without the samples referring to a different wave or signal. If we are sampling pure waves, we can say:

> "When using a particular sample rate, a wave must have a frequency slower than half the sample rate, or else analysis will not be able to recover it from its samples correctly."

We can state this slightly more mathematically. To record a wave properly, this must be true:

"maximum frequency < sample rate / 2"

For periodic *signals*, we can say:

> "When using a particular sample rate, each *constituent* wave must have a frequency slower than half the sample rate, or else analysis will not be able to recover the signal as a whole from its samples."

If we phrase this slightly more mathematically, we would say that, given a particular sample rate, this must be true:

"maximum *constituent* frequency < sample rate / 2"

We can summarise the "reverse" versions of the first rule as:

- The maximum frequency that we can record correctly will be less than half the sample rate.
- The maximum *constituent* frequency that we can record correctly will be less than half the sample rate.

The second sample rate rule

The second sample rate rule is:

> "Any wave (or constituent wave) with a frequency *equal* to half the sample rate will be recorded incorrectly. If analysis can find any wave at all, it will have the correct frequency, the wrong amplitude, and possibly the wrong phase."

If a wave with a frequency equal to half the sample rate (or a signal containing such a wave) is sampled, then one of three things will happen when analysing the samples:

- There will be no trace of the wave at all.
- Analysis will find a different wave that, after having its amplitude halved, will have matching samples, but it will be not be the correct wave.
- Analysis will find a different wave that, after having its amplitude halved, will be the correct wave.

The phase of the original wave will dictate which of the three happens.

Given all of this, we should never record a wave with a frequency equal to half the sample rate because it might be impossible to recover it. However, if we are given the samples of such a wave or signal, we *might* be able to find the original wave (if we halve the discovered amplitude) or a wave with the same samples (if we halve the discovered amplitude). This will become clearer as this chapter progresses.

The third sample rate rule

The third sample rate rule is:

> "Any wave (or constituent wave) of any frequency will end up being recorded as a wave with a frequency from 0 cycles per second up to and including half the sample rate."

A variation of that is, which is related to the first and second rules, is:

> "Any wave (or constituent wave) with a frequency above, or equal to, half the sample rate will be recorded incorrectly, and analysis will find it as a wave with a positive frequency below, or equal to, half the sample rate."

[Note that if the frequency is equal to half the sample rate, or an integer multiple of the sample rate higher than half the sample rate, the wave might be removed from the signal entirely, and so be unrecoverable.]

If we use a particular sample rate to record a signal, and that signal has both constituent waves with frequencies below half the sample rate, and constituent waves with frequencies above half the sample rate, then the slower waves will be recorded correctly, but the faster waves will be recorded incorrectly. The faster frequency waves become recorded as lower frequencies, thus corrupting the record of the signal. The "fake" lower frequency waves are called "aliases". In the same way that a person might pretend to be someone else by using an alias, so can the false waves portrayed by the samples be thought of as "pretending" to be other waves. Any maths or analysis that is done on the recorded signal will produce incorrect results. We will see examples of this later in this chapter.

In practice, we will often need to use a particular sample rate to sample signals for which we cannot know what the fastest constituent frequency is. If we can only sample at a particular sample rate, we have to make sure that the signals we are sampling do not contain frequencies that are equal to, or above, half our sample rate. For example, if we are sampling at 100 samples per second, the maximum constituent frequency that we can record correctly will be under 50 cycles per second. Anything faster than this will cause erroneous results. When sampling signals in the real world, it makes sense to filter the signal before the sampling is done so that there cannot be any constituent frequencies that are too fast. This removes any constituent frequencies that will be recorded incorrectly, and avoids producing a discrete signal containing aliased frequencies.

The sampling rules can be seen in practice in audio files. For example, in a WAV file with a sample rate of 44,100 samples per second, we would only be able to record frequencies correctly if they were slower than 22,050 cycles per second. We can think of this the other way around. The highest frequency that a typical human can hear is about 20 kHz. To accommodate sounds with constituent frequencies up to 20 kHz in a discrete signal, we would need a sample rate that is over double the highest constituent frequency. Therefore, it must be over 40,000 samples per second. A sample rate of 44,100 samples per second would be sufficient for this.

Nyquist

The name for a sample rate that is *equal* to twice the highest constituent frequency in a signal is called the "Nyquist rate" after the Swedish physicist, Harry Nyquist. In this book, to avoid confusion with a similar term, I will call this the "Nyquist *sampling* rate". We must always sample *above* the Nyquist sampling rate. If we have a continuous wave with a frequency of 123 cycles per second, then the Nyquist sampling rate will be 246 samples per second, and we must sample the wave with a sample rate *above* 246 samples per second. For our 11-cycle-per-second wave, the Nyquist sampling rate is 22 samples per second, and we must sample the wave with a sample rate higher than this.

To make things more confusing, there is a term that indicates the frequency of a wave (or a constituent wave) that is equal to twice the sample rate we happen to be using. This is called the "Nyquist *frequency*". If we are using a sample rate of 10 samples per second, the Nyquist frequency will be 5 cycles per second. We will only be able to record frequencies *under* 5 cycles per second. If we are using a sample rate of 222 samples per second, the Nyquist frequency will be 111 cycles per second, and the highest frequency we will be able to record correctly will be *under* 111 cycles per second.

The term "Nyquist sampling rate" is usually called the "Nyquist rate". Therefore, we have two very similar terms that mean very similar things. We have the "Nyquist *rate*" and the "Nyquist *frequency*". These are often confused with each other for several reasons. First, they have similar names – from a linguistic point of view, "rate" and "frequency" can often mean the same thing. Second, they refer to similar ideas. Third, sometimes, the term "sample rate" is referred to as the "sampling frequency". Fourth, one term refers to a sample rate that is calculated by doubling a frequency; the other term refers to a frequency that is calculated by halving a sample rate. Finally, if you use the wrong term, everyone will know what you mean anyway, so the difference is not always important.

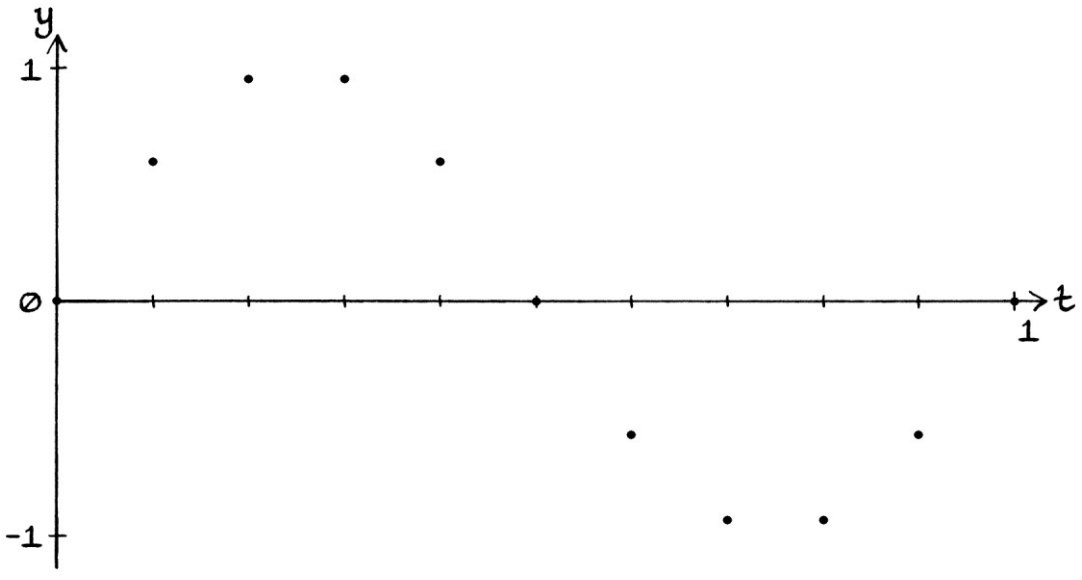
The difference between the two terms relates to the perspective from which we are viewing a situation:

- If we have a signal and we need to choose a sample rate, then we need to know the "Nyquist sampling rate", which will be equal to twice the maximum constituent frequency.
- If we have a sample rate, and we want to know the maximum frequency that we can record, then we need to know the "Nyquist frequency", which will be equal to half the sample rate.

Many people would say that the names for the two ideas are confusing, and furthermore, that there is no need to give names to concepts that just mean "equal to twice the frequency" and "equal to half the sample rate". However, in the study of signal processing, you will often see the terms "Nyquist rate" and "Nyquist frequency". In this book, I will refer to the "Nyquist rate" as the "Nyquist *sampling* rate" to help reduce the confusion.

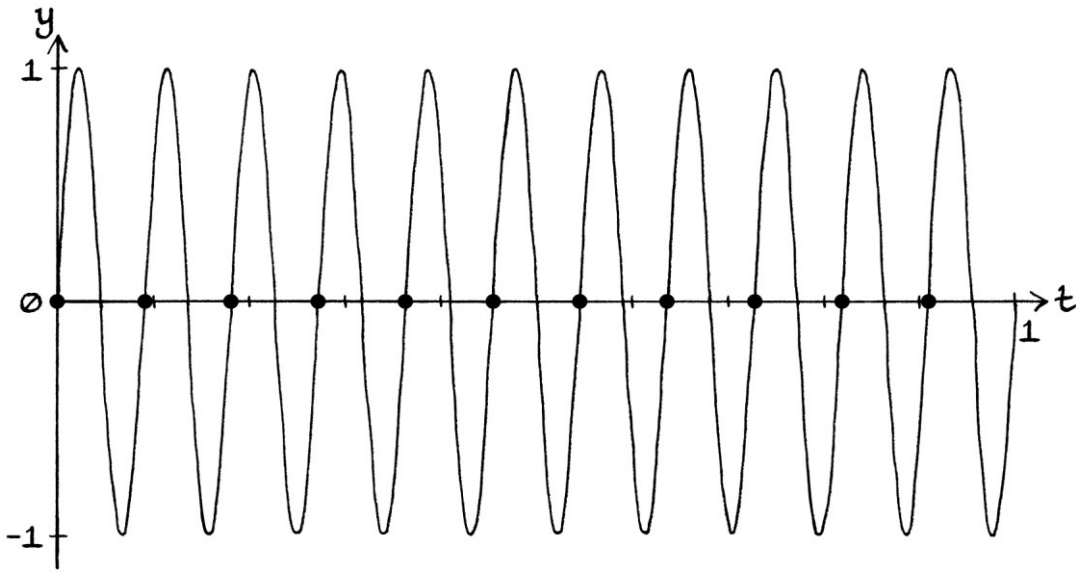
Exploring the rules

In the earlier example of a wave with a frequency of 11 cycles per second ["y = sin (2π * 11t)"], we would have needed to sample it with a sample rate higher than 22 samples per second. The sample rate of 22 samples per second is twice the frequency of the wave. When we sampled the wave at 10 samples per second, we had the following samples for the first second:
0, 0.5878, 0.9511, 0.9511, 0.5878, 0, −0.5878, −0.9511, −0.9511, −0.5878
The samples did not describe the actual wave, but instead a wave of 1 cycle per second. On a graph, the samples looked like this:



If we sample the 11-cycle-per-second wave at 11 samples per second, we will get these samples for the first second: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

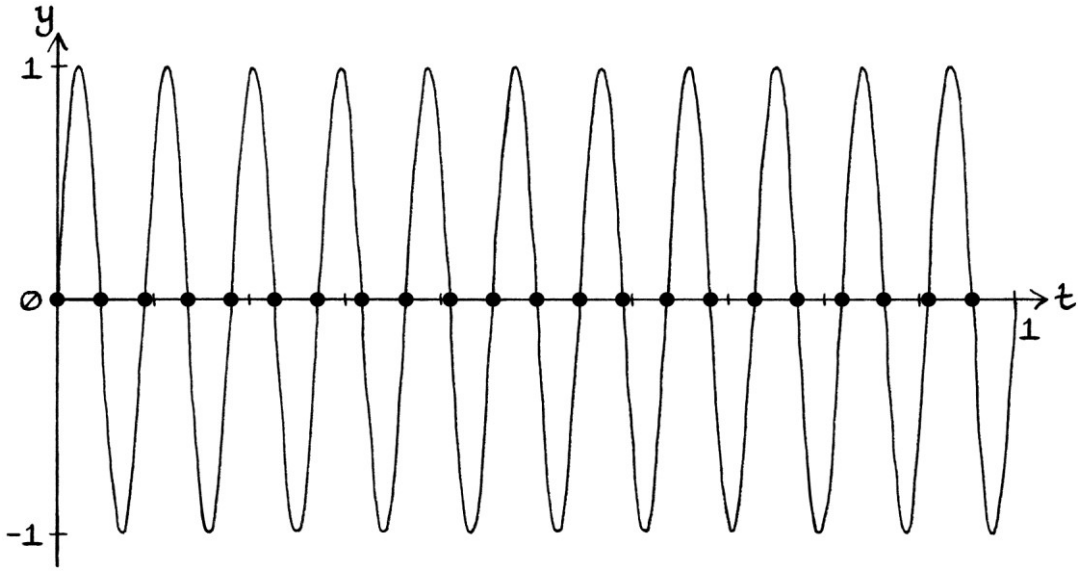
All of the samples are zero because each cycle of the wave occurs entirely between the places where we are taking the samples.



Clearly, a sample rate of 11 samples per second is insufficient to record our wave.

If we sample our wave at exactly 22 samples per second, we will get these samples for the first second: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

All the samples are zero because each peak and each dip occur between the places where we are taking the samples. We are recording the wave at the points when the instantaneous angle is either 0 or 180 degrees (0 or π radians):



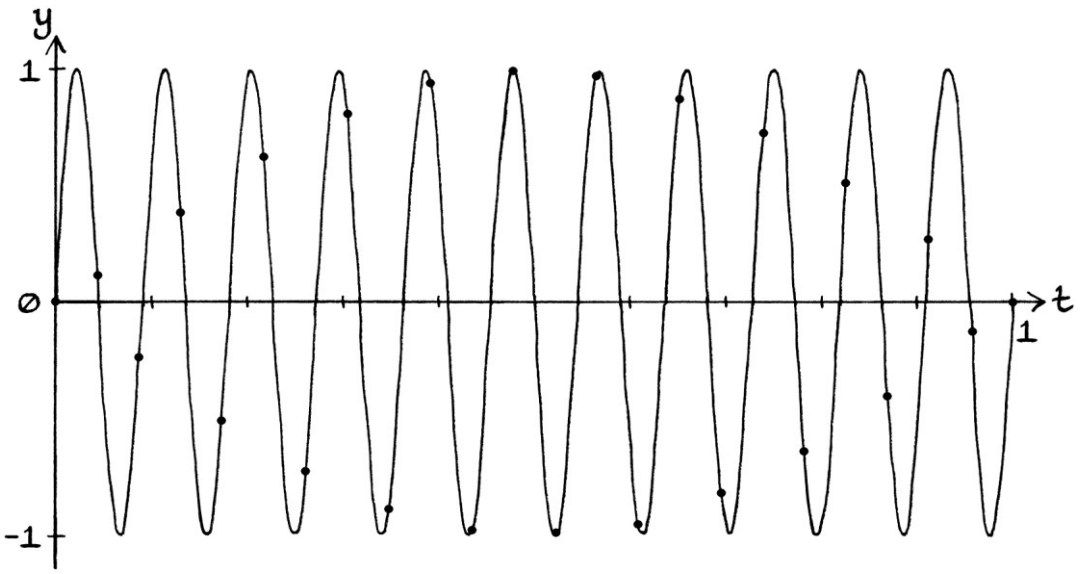
We can see that a sample rate of 22 samples per second is not sufficient to portray our 11-cycle-per-second wave.

If we sample our wave at 23 samples per second, we will have these samples for the first second:

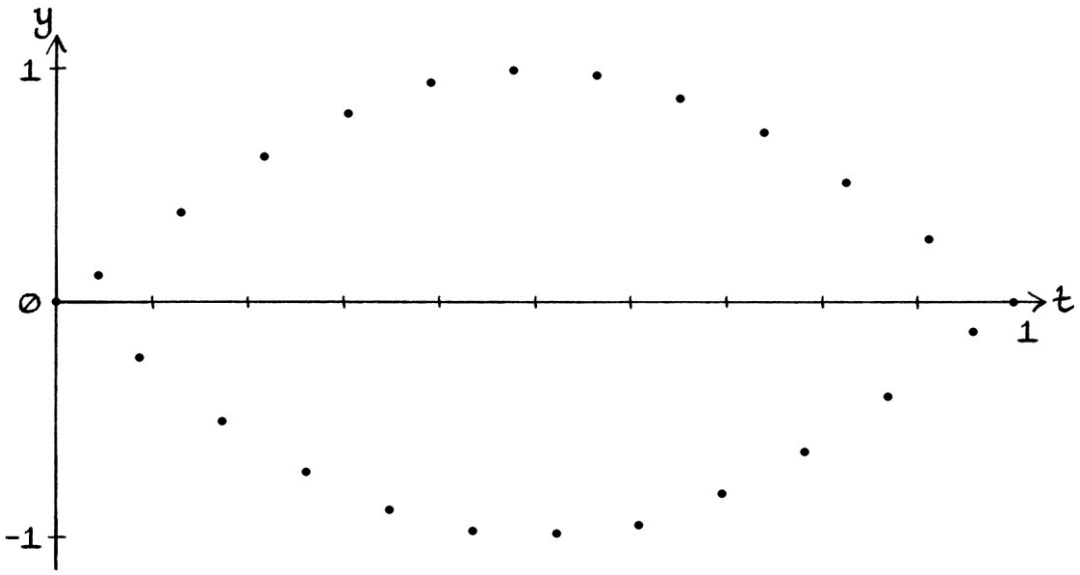
0.0000
0.1362
−0.2698
0.3984
−0.5196
0.6311
−0.7308
0.8170
−0.8879
0.9423
−0.9791
0.9977
−0.9977
0.9791
−0.9423
0.8879
−0.8170
0.7308
−0.6311
0.5196
−0.3984
0.2698
−0.1362
[The next sample would be 0.0000]

Each of these samples is alternately positive then negative. The samples do not record the peaks and dips of the cycles, but various points on the sides of the peaks and dips. Although the maximum and minimum values of our 11-cycle-per-second continuous wave are +1 and −1, the maximum and minimum values of our samples are +0.9977 and −0.9977.

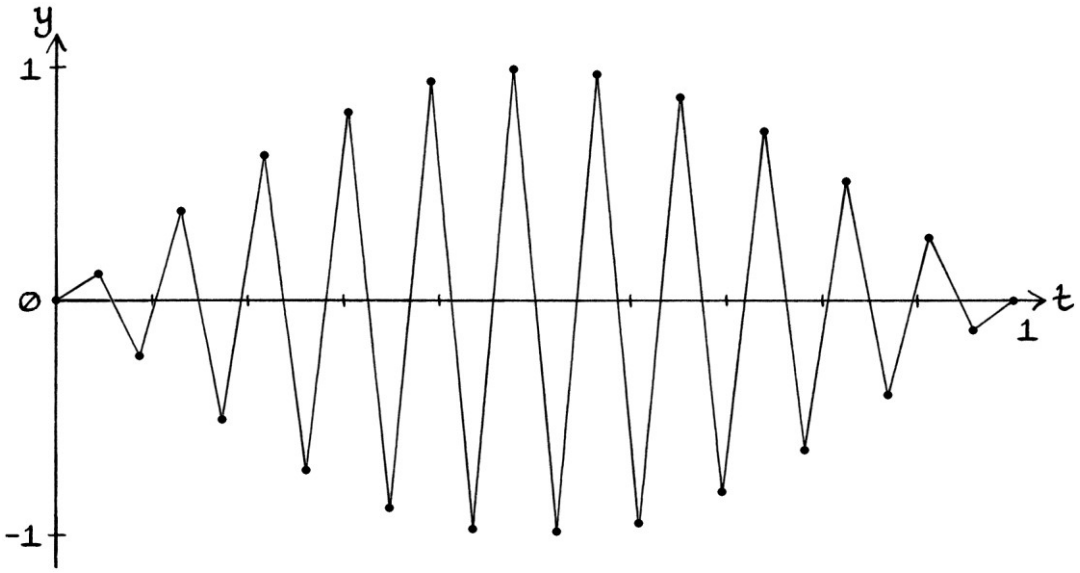
The samples drawn over the 11-cycle-per-second continuous wave are as so:



The samples on their own look like this:



If we join up the samples (solely to make their position on the graph clearer), we end up with this graph:



The graph does not look like our 11-cycle-per-second wave. However, if we analysed the samples using Fourier series analysis (with the same sample rate), we would discover our original 11-cycle-per-second wave.

A note on Fourier series analysis

To analyse a discrete periodic signal, we would typically use "*discrete* Fourier series analysis", which is essentially the same process as normal Fourier series analysis, but taking into account some new ideas. I will explain discrete Fourier series analysis in the next chapter. It helps to understand sampling before trying to understand discrete Fourier series analysis. "Normal" Fourier series analysis is more correctly called "*continuous*" Fourier series analysis to distinguish it from discrete Fourier series analysis.

It is possible to analyse a signal perfectly well using the ideas of continuous Fourier series analysis but with discrete test waves. The test waves need to have the same sample rate as the signal being analysed, and the fundamental frequency needs to be equal to that of the discrete signal and not necessarily that of the continuous signal from which the discrete signal came. The biggest problem with using continuous Fourier series analysis on discrete signals is that it will continue to find duplicate constituent waves as faster frequencies are tested. The duplicates are not actually constituent waves, and just appear as a consequence of sampling. We will see why this happens later in this chapter.

For the rest of this chapter, we will assume that all analysis is done using the ideas of continuous Fourier series analysis, but using discrete test waves with sample rates that match that of the discrete signal being analysed.

To analyse the above 23-sample-per-second discrete version of our 11-cycle-per-second wave, we would need to use test waves with the same sample rate. As our *discrete* signal actually repeats once per second, and not 11 times a second, we would need to have our test frequencies all have frequencies that were integer multiples of 1 cycle per second.

Exploring the rules, continued

We will continue looking at the 23-sample-per-second discrete version of our 11-cycle-per-second wave.

Fourier series analysis of the discrete signal would discover it as a single 11-cycle-per-second wave. Despite this, the discrete signal is identical to a discrete representation of a continuous signal that literally had the same sawtooth shape. Our sample rate is sufficient to *record* our continuous 11-cycles-per-second wave, but it is not sufficient to *distinguish* the 11-cycles-per-second wave from a continuous straight-lined sawtooth signal. Earlier, we referred to this idea as the "Straight Line Ambiguity".

Given that analysis of our discrete signal finds our 11-cycle-per-second wave, we can say that the discrete version of our wave has an adequate sample rate. The discrete signal might not look correct, but it is correct from an analysis point of view.

This sampling example confirms our rule that we need a sample rate higher than twice the frequency. However, it also shows that such a sample rate is the *minimum* sample rate. Anything below this sample rate would produce incorrect samples. [They would be incorrect in the sense that we would not be able to recover the continuous wave by analysing the samples.] The sample rate of 23 samples per second produces a discrete signal that can be analysed and found to be correct. However, a higher sample rate would produce a more accurate representation that would be "visibly" correct as well as correct "analysis-wise". A much higher sample rate would also produce a discrete signal that had the correct fundamental frequency – in this example, our discrete signal repeats once every second, but the original continuous wave repeated 11 times per second.

To obtain a discrete signal that repeated 11 times a second *for this particular wave*, we would need a sample rate of 44 samples per second – each sample would be at y = 0, or at each peak and dip. Each sample would be 0, +1 or −1. We would have four samples for every cycle.

A higher sample rate would also diminish the potential problems of the "Straight Line Ambiguity". No matter what our sample rate, the samples will always be the same as those from a signal with straight lines joining up the points where the samples are taken. However, the faster the sample rate, the less important this ambiguity will become. If we had a low frequency and an extremely high sample rate, the straight-line equivalent signal would be so close to the original continuous wave that any actual differences would be irrelevant.

Non-zero phases

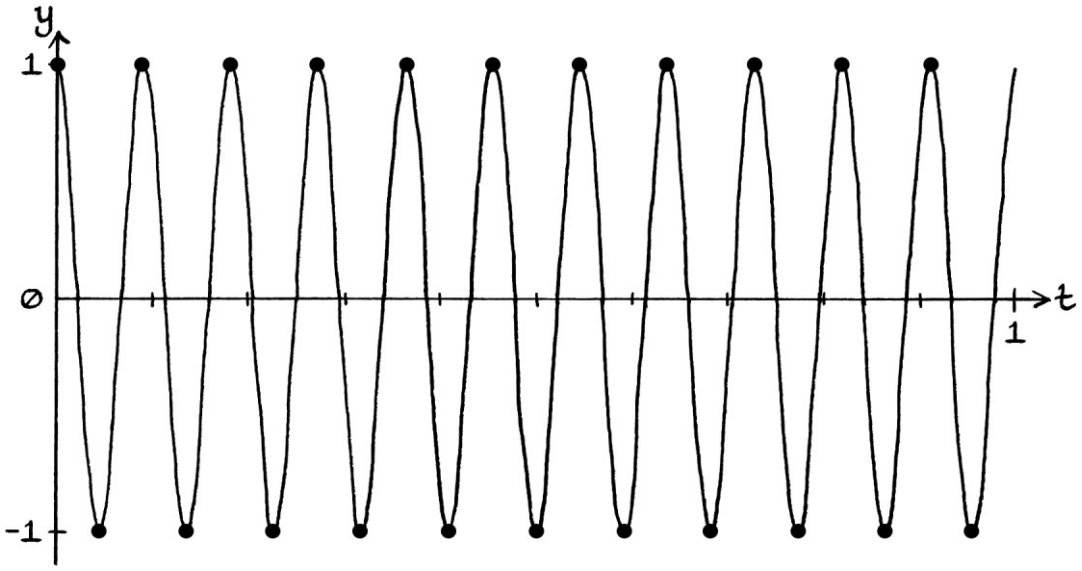
We will give our 11-cycle-per-second Sine wave a phase of 0.5π radians (90 degrees), so it becomes:
"y = sin ((2π * 11t) + 0.5π)"

If we sampled this with a sample rate of 11 samples per second, every sample would be 1. The wave starts at y = 1, and an entire cycle occurs between each sample. There is one sample for each cycle. For any pure wave, no matter what the phase, if the sample rate matches the frequency, all the samples will be the same as each other.

If we sample the wave with a sample rate of 22 samples per second, we will have a record of the wave that *looks* correct – the frequency, amplitude and phase of the discrete signal will match those of the original continuous signal. However, it is important to note that Fourier series analysis (with a matching sample rate) would *not* find the wave by analysing the samples. Instead, it would find a wave with the same characteristics but twice the amplitude. We will learn the reasons for this later in this chapter.

For a sample rate of 22 samples per second, we would have two samples for each cycle, which would be +1 followed by −1.

The samples drawn over the curve are as so:



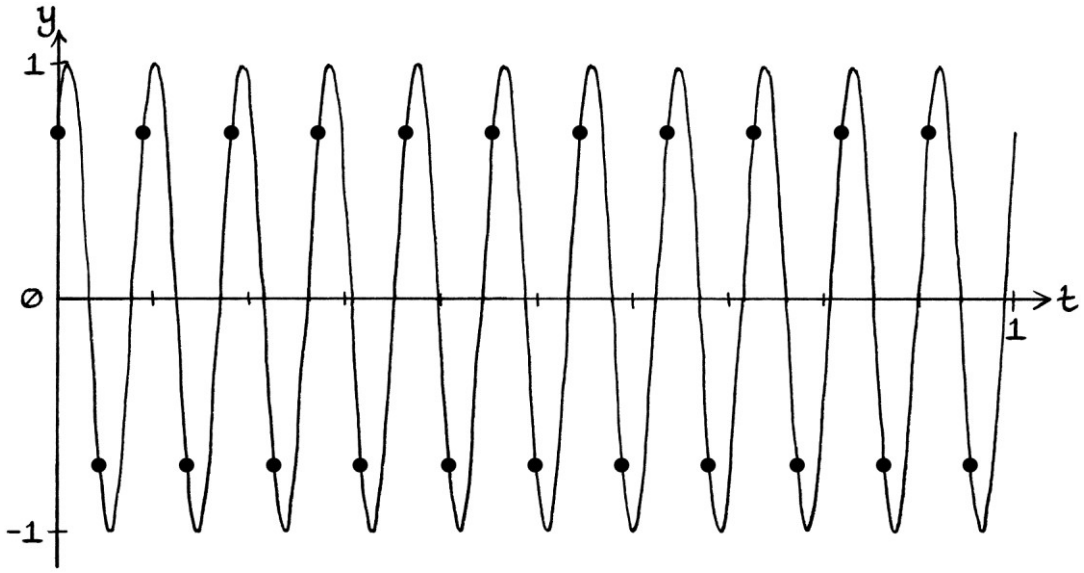
[Note how this situation is different from when our Sine wave had zero phase and a sample rate of 22 samples per second. For a Sine wave with zero phase (and zero mean level) and a sample rate equal to the frequency, every sample will be zero.]

It would be easy to infer from the graph, mistakenly, that a sample rate equal to the frequency of "y = sin ((2π * 11t) + 0.5π)" would be sufficient for this wave. However, analysis would not discover the original wave from the samples. The graph of the samples *looks* correct, but it is not correct from the point of view of analysis. A second mistaken inference is that, because the graph *looks* correct, it must be the case that the sample rate can be equal to twice the frequency for any wave. However, we have seen that this is not true by the way that for a sample rate of 22 samples per second, the samples of "y = sin (2π * 11t)" are all zero. It is always the case that to record a pure *wave* correctly, the sample rate must be over twice the frequency of the wave. To record an impure *signal* correctly, the sample rate must be over twice the frequency of the fastest constituent wave.

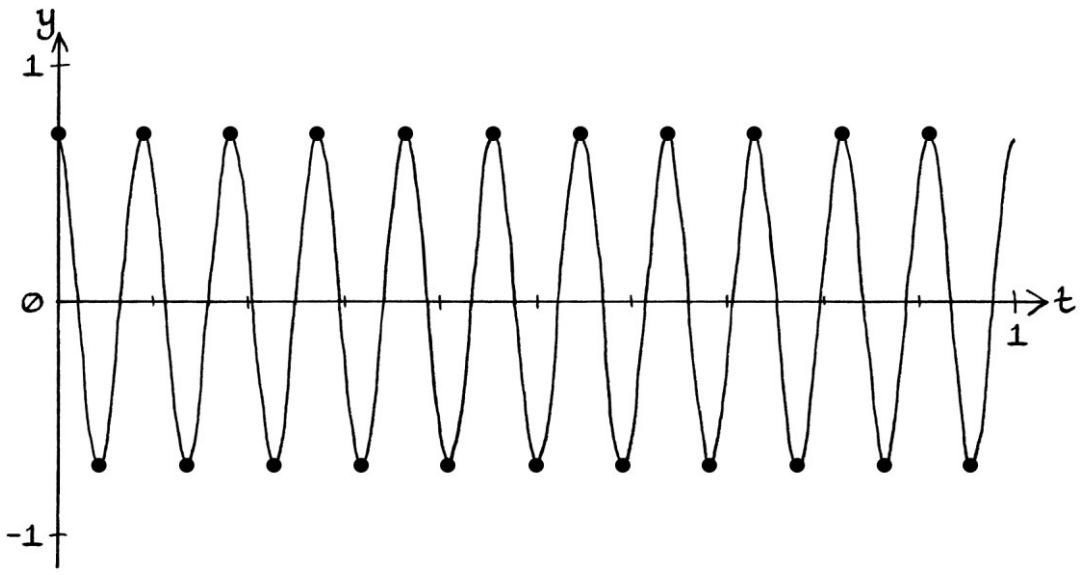
As another example of a non-zero phase, we will give our 11-cycle-per-second wave a phase of 0.25π radians (45 degrees). Its formula will be:
"y = sin ((2π * 11t) + 0.25π)"

We will sample it at 22 samples per second. The samples will be:
0.7071, −0.7071, 0.7071, −0.7071, and so on.

The samples superimposed over the continuous wave's curve are as so:



The first thing to realise is that the discrete samples are identical to those from the wave:
"y = 0.7071 sin ((2π * 11t) + 0.5π)"
... sampled at the same sample rate:



However, Fourier series analysis on the samples would not find either of the above two waves. Instead, it would find:
"y = 1.4142 sin ((2π * 11t) + 0.5π)"
... which has twice the amplitude of the "y = 0.7071 sin ((2π * 11t) + 0.5π)" wave.

Faster Duplicates Ambiguity

In our types of ambiguity, the "Faster Duplicates Ambiguity" refers to how the samples from one wave or signal could also refer to the samples from a faster wave or signal that had been undersampled. The differences between the "Undersampling Ambiguity" and the "Faster Duplicates Ambiguity" are:

- The "Undersampling Ambiguity" refers to how a wave or signal that is not sampled with a fast enough sample rate will end up as a record of a different wave or signal. It will be an incorrect record.

- The "Faster Duplicates Ambiguity" refers to how the samples of a wave or signal, whether they have been recorded correctly or not, are also the samples of waves or signals with faster frequencies that have been undersampled.

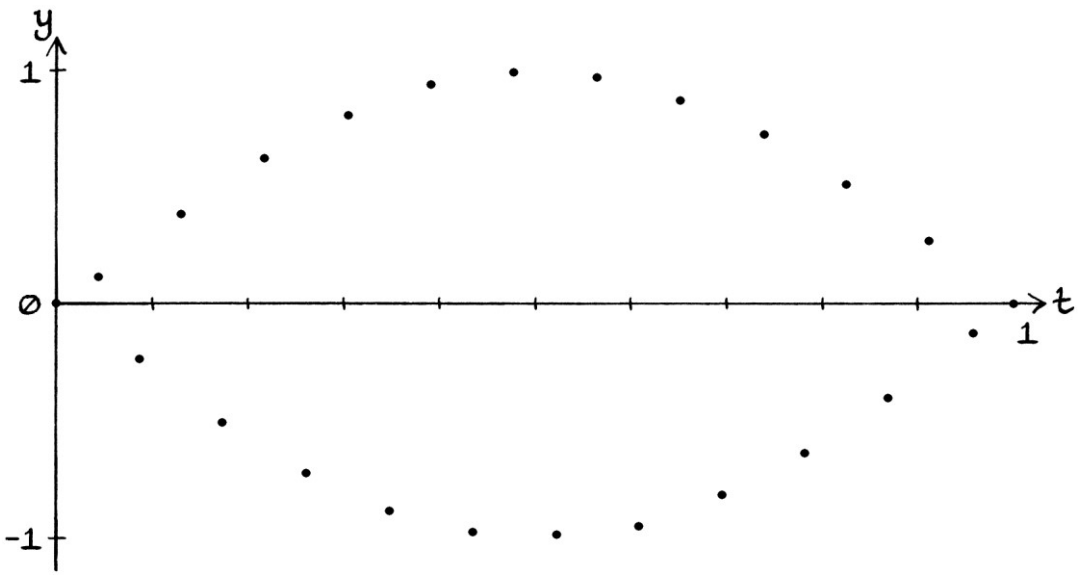
The difference is partly in how we think of a situation. If we have a correctly sampled signal, there will be faster signals with the same samples. If we have an incorrectly sampled signal, it will refer to a different signal, and there will also be faster signals that have the same samples as that different signal.

There is an inherent ambiguity with any list of samples – we cannot know if the samples really came from the continuous wave or signal that is closest to that created by "joining up the dots" or if they came from a different faster wave or signal. Usually, we have to presume the samples represent the signal with the slower frequencies, but we must be aware that there are other signals that would have matching samples (if they had been undersampled).

We will look at various examples to explore the idea of the "Faster Duplicates Ambiguity".

11 cycles per second

We will go back to our "y = sin (2π * 11t)" wave sampled at 23 samples per second.



Although this discrete signal represents our continuous 11-cycle-per-second wave, it also represents countless other possible waves. There are countless continuous waves that when sampled at 23 samples per second will have the same samples as our discrete signal. A wave of 11 cycles per second is the one with the slowest frequency that has these samples. There are other waves with faster frequencies that have the same samples.

If we performed Fourier series analysis on this signal (using the same sample rate, and test frequencies that were integer multiples of 1 cycle per second), and we did not stop once we had found our wave, we would find more waves with matching samples.

The waves we would find would be:

"y = sin (2π * 11t)", which is the correct wave.
"y = sin (2π * 34t)"
"y = sin (2π * 57t)"
"y = sin (2π * 80t)"
"y = sin (2π * 103t)"
"y = sin (2π * 126t)"

... and so on, as well as:

“y = sin ((2π * 12t) + π)”
“y = sin ((2π * 35t) + π)”
“y = sin ((2π * 58t) + π)”
“y = sin ((2π * 81t) + π)”
“y = sin ((2π * 104t) + π)”
“y = sin ((2π * 127t) + π)”
... and so on.

These are all waves that have the same samples as our 11-cycle-per-second wave. Although analysis of the discrete signal would find them as constituent waves, only one of them is actually the true constituent wave. They do not all exist at the same time in our discrete signal as a sum – only one of them exists – but they all have the same samples. Analysis would find all of them, but we should ignore all of them except the slowest frequency, which is the 11-cycle-per-second wave.

The waves in the first group have frequencies that are all equal to 11 cycles per second added to an integer multiple of the sample rate (23 samples per second). The frequencies are:

11 cycles per second, which is: 11 + (0 * 23)
34 cycles per second, which is: 11 + (1 * 23)
57 cycles per second, which is: 11 + (2 * 23)
80 cycles per second, which is: 11 + (3 * 23)
103 cycles per second, which is: 11 + (4 * 23)
126 cycles per second, which is: 11 + (5 * 23)
... and so on.

The second group of waves (the ones with phases of π radians) can be rephrased to be, and are best thought of as, negative-frequency waves with zero phases:

“y = sin (2π * −12t)”
“y = sin (2π * −35t)”
“y = sin (2π * −58t)”
“y = sin (2π * −81t)”
“y = sin (2π * −104t)”
“y = sin (2π * −127t)”
... and so on.

These negative-frequency waves are all equal to 11 cycles per second added to a *negative* integer multiple of the sample rate (23 samples per second).

The frequencies are:

−12 cycles per second, which is: 11 + (−1 * 23)
−35 cycles per second, which is: 11 + (−2 * 23)
−58 cycles per second, which is: 11 + (−3 * 23)
−81 cycles per second, which is: 11 + (−4 * 23)
−104 cycles per second, which is: 11 + (−5 * 23)
−127 cycles per second, which is: 11 + (−6 * 23)
... and so on.

We can see that our wave has duplicates with frequencies higher and lower by positive and negative integer multiples of the sample rate.

When analysing a discrete signal, we can identify (and so, ignore) the positive-frequency and negative-frequency duplicates because their frequencies will all be faster than half the sample rate. Half the sample rate in this case is 23 ÷ 2 = 11.5 samples per second. The negative frequencies are *lower* but not *slower*, but actually, that is irrelevant here because Fourier series analysis would find them as positive-frequency waves anyway. Waves with frequencies above 11.5 cycles per second could not be sampled correctly at 23 samples per second – they would end up being found as (incorrect) waves with frequencies below half the sample rate. Therefore, any waves that are found with frequencies above half the sample rate can only be duplicates of the waves that we have already found.

Another way to identify the duplicates is that once we have found one wave through analysis, we can calculate the formulas of all of its duplicates, so we know to ignore them when we find them.

[Note that we would have no way of knowing if the first-found wave was the wave that was actually sampled because the original wave might have been undersampled. This idea was the “Undersampling Ambiguity”.]

Sums of waves

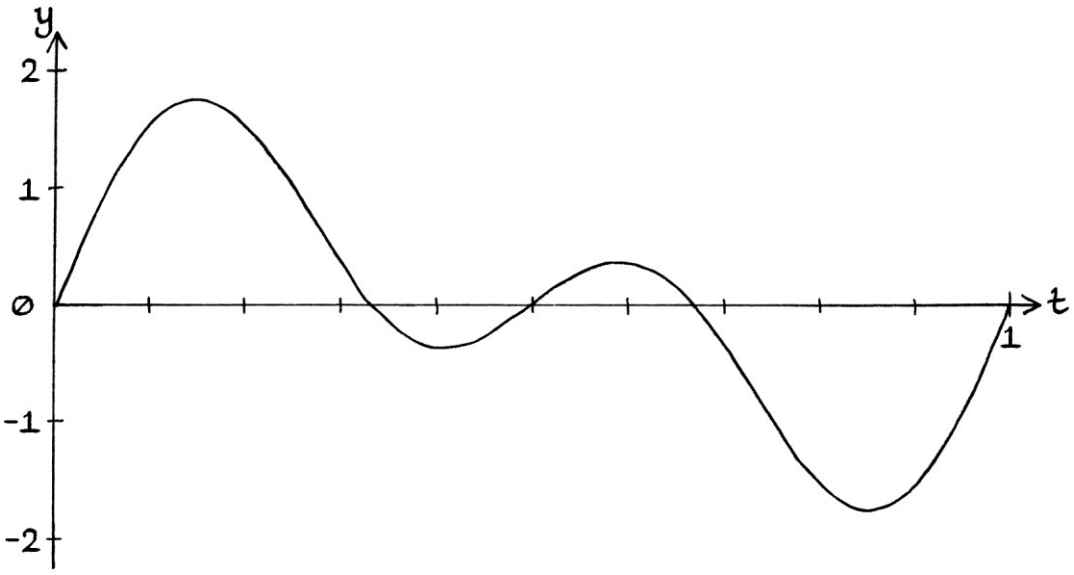
As we have just seen with our 11-cycle-per-second wave, for any sampled wave, there are faster duplicates that have the same samples. The same is true for *sums* of waves – any sampled sum of waves will have faster duplicates. With sums of waves, each constituent wave has duplicates that are higher or lower by positive and negative integer multiples of the sample rate. There are countless sums of faster constituent waves that would share the same set of samples, and Fourier

series analysis would find more and more duplicates as higher and higher test frequencies were used.

We will think about a *continuous* signal made up of the sum of:
"y = sin (2π * 1t)"
... and:
"y = sin (2π * 2t)"

The signal's formula is:
"y = sin (2π * 1t) + sin (2π * 2t)"

The signal looks like this:



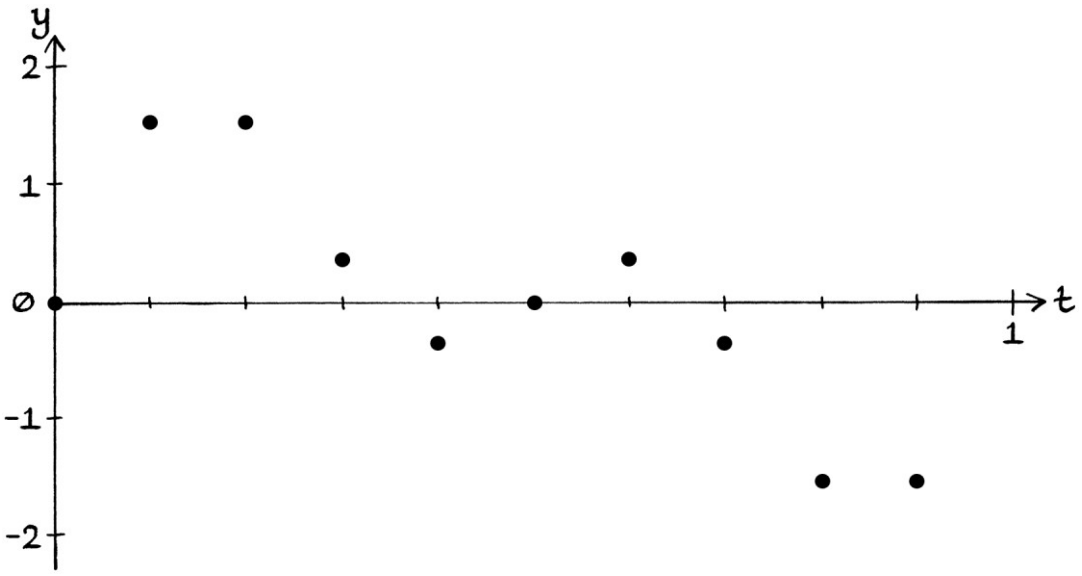
We need to sample the signal with a sample rate that is over twice the fastest constituent frequency. Our fastest constituent frequency is 2 cycles per second. Therefore, we need to sample the signal with a sample rate that is above 4 samples per second. We can say the same thing in terms of the Nyquist sampling rate. The Nyquist sampling rate is twice the fastest constituent frequency – it is 4 samples per second. We need to sample at a higher sample rate than the Nyquist sampling rate. To make things easier, we will sample our signal with a sample rate of 10 samples per second.

The samples for the first cycle are:

t = 0.0 0
t = 0.1 1.5388
t = 0.2 1.5388
t = 0.3 0.3633
t = 0.4 −0.3633
t = 0.5 0
t = 0.6 0.3633
t = 0.7 −0.3633
t = 0.8 −1.5388
t = 0.9 −1.5388

The next sample would be:
t = 1.0 0

These samples do not make a particularly good visual representation of the continuous signal, but that does not matter for the purposes of this example. [If we were to analyse the signal, we would still find the correct constituent waves.] Drawn on a graph, the samples look like this:



We will now find other continuous signals that, if sampled at 10 samples per second, would result in the same samples. To do this, we can use the knowledge that for a sample rate of 10 *samples* per second, each constituent wave will match at frequency steps of 10 *cycles* per second. In other words, the samples from waves with frequencies of 1, 11, 21, 31... cycles per second will match with each other; the samples from waves with frequencies of 2, 12, 22, 32... cycles per second will

match with each other. As our signal consists of frequencies of 1 cycle per second and 2 cycles per second, and it is sampled at 10 samples per second, the following sums of frequencies will match with our sum of 1 and 2 cycles per second:

11 and 12 cycles per second
21 and 22 cycles per second
31 and 32 cycles per second
41 and 42 cycles per second
51 and 52 cycles per second
... and so on.

We will also have the negative-frequency duplicates:

−9 and −8 cycles per second
−19 and −18 cycles per second
−29 and −28 cycles per second
−39 and −38 cycles per second
−49 and −48 cycles per second
−59 and −58 cycles per second
... and so on.

[The negative-frequency duplicates would appear as positive-frequency duplicates when discovered with Fourier series analysis.]

The frequencies given in terms of waves are as so:

“y = sin (2π * 1t) + sin (2π * 2t)” – this is the actual signal.
“y = sin (2π * 11t) + sin (2π * 12t)”
“y = sin (2π * 21t) + sin (2π * 22t)”
“y = sin (2π * 31t) + sin (2π * 32t)”
“y = sin (2π * 41t) + sin (2π * 42t)”
“y = sin (2π * 51t) + sin (2π * 52t)”
... and so on, and:

“y = sin (2π * −9t) + sin (2π * −8t)”
“y = sin (2π * −19t) + sin (2π * −18t)”
“y = sin (2π * −29t) + sin (2π * −28t)”
“y = sin (2π * −39t) + sin (2π * −38t)”
“y = sin (2π * −49t) + sin (2π * −48t)”
“y = sin (2π * −59t) + sin (2π * −58t)”
... and so on.

All of the above signals will have an identical list of samples if they are sampled at 10 samples per second. Similarly, Fourier series analysis will find all of the above signals if we continued to test for waves with frequencies above half the sample rate. Apart from the slowest two waves, the found waves would be duplicates that do not actually exist in the signal. The negative-frequency waves would be found as their positive-frequency equivalents.

The duplicate signals do not have to be pairs of waves that are the *same* integer multiple of the sample rate higher or lower. For example, the following signals all have the same samples as our original signal when sampled at 10 samples per second:

“y = sin (2π * 1t) + sin (2π * 12t)”
“y = sin (2π * 1t) + sin (2π * 22t)”
“y = sin (2π * 1t) + sin (2π * 32t)”

... and so on, and:

“y = sin (2π * 11t) + sin (2π * 2t)”
“y = sin (2π * 21t) + sin (2π * 2t)”
“y = sin (2π * 31t) + sin (2π * 2t)”

... and so on, and:

“y = sin (2π * 21t) + sin (2π * −18t)”
“y = sin (2π * −49t) + sin (2π * 22t)”
“y = sin (2π * 41t) + sin (2π * 92t)”
“y = sin (2π * 51t) + sin (2π * −38t)”

... and all the other possible combinations.

If we were given the samples from any of the above signals, and we analysed them to find the constituent waves, we would first find the two slowest waves:
“y = sin (2π * 1t)” and “y = sin (2π * 2t)”.

If we continued the analysis, we would find the faster duplicate frequencies too, with the negative frequencies made positive. These waves would not be part of the sum, but just duplicates of the two constituent waves that we had already found.

Formulas

We can make a formula that describes how, for a sample rate of 10 samples per second, waves with frequencies that differ by 10 cycles per second will have the same samples as each other. The formula includes negative frequencies too:

$f_0 = f_0 + (x * 10)$
... where:

- "f_0" is the frequency of a wave. [It is just the letter "f" with a subscript of "0" to distinguish it from any other "f" symbols we might use. We could have used any subscript, but it is easier to start with "0".]
- "x" is any positive or negative integer.
- 10 is the sample rate in samples per second.

The formula means that for a sample rate of 10 samples per second, a wave with any frequency will be recorded *in exactly the same way* as a wave with a frequency any integer multiple of 10 cycles per second higher or lower. [Note that waves with frequencies above half the sample rate will not be recorded *correctly*, but they will be recorded in the same way.]

We can make the formula apply to all sample rates:
$f_0 = f_0 + (x * samplerate)$
... where:

- "samplerate" is the sample rate.

We can make the formula more consistent with signal processing conventions:
$f_0 = f_0 + (x * f_s)$
... where:

- "f_0" is the frequency of a wave.
- "x" is a positive or negative integer.
- "f_s" is the sample rate in samples per second, but given as if we referred to it as the sampling *frequency*. Having "f_0" to mean a cycles-per-second frequency and "f_s" to mean the sampling frequency (sample rate) might seem slightly confusing at first.

[Note that despite how "f_s" seems a difficult symbol to distinguish from "f_0" or "f", it and other symbols become much easier to use as you become more used to them. If you find it confusing having "f_0" and "f_s" in the same formula, you should know that it will become straightforward in very little time.]

We can adapt the formula so that it works with wave formulas:

If we have this continuous wave sampled at "f_s" samples per second:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... then it will have identical samples to this continuous wave sampled at the same sample rate:
"$y = h + A \sin ((2\pi * (f_0 + (x * f_s)) * t) + \phi)$"
... where:

- "h" is the mean level.
- "A" is the amplitude.
- "f_0" is the frequency of the original wave.
- "x" is any positive or negative integer.
- "f_s" is the sample rate.
- "t" is the time in seconds.
- "ϕ" is the phase in radians.

Every attribute apart from the overall frequency will remain the same. For example, for this wave sampled at 12 samples per second:
"$y = 1 + 2 \sin ((2\pi * 3t) + 0.125\pi)$"
... the following positive-frequency waves sampled at 12 samples per second would have the same samples:
"$y = 1 + 2 \sin ((2\pi * 15t) + 0.125\pi)$"
"$y = 1 + 2 \sin ((2\pi * 27) + 0.125\pi)$"
"$y = 1 + 2 \sin ((2\pi * 39t) + 0.125\pi)$"
"$y = 1 + 2 \sin ((2\pi * 51t) + 0.125\pi)$"

Similarly, the following *negative*-frequency waves sampled at 12 samples per second would have the same samples:
"$y = 1 + 2 \sin ((2\pi * -9t) + 0.125\pi)$"
"$y = 1 + 2 \sin ((2\pi * -21t) + 0.125\pi)$"
"$y = 1 + 2 \sin ((2\pi * -33t) + 0.125\pi)$"
"$y = 1 + 2 \sin ((2\pi * -45t) + 0.125\pi)$"
... and so on.

[Fourier series analysis would find the negative-frequency waves as their *positive*-frequency equivalents, which would all have phases of 0.875π radians.]

We will refer to the duplicate rule as the "$f_0 = f_0 + (x * f_s)$" rule. The rule means that waves of any frequency will be recorded in the same way as waves with frequencies any integer multiple of the sample rate higher or lower. Again, note

how I say that the waves will be recorded *in the same way*. A wave might not be recorded correctly, but it will be recorded in the same way.

We can summarise the consequences of the rule in the following statement. Note how this refers to the sample rate and not *half* the sample rate.

> "A wave with a frequency *outside* the range of 0 cycles per second up to just under the sample rate will, when sampled, produce the same samples as a wave with a frequency from 0 cycles per second up to just under the sample rate."

When analysing a discrete signal using Fourier series analysis, a wave that originally had any frequency above the sample rate will end up being found as a wave with a frequency from 0 cycles per second up to, and including, *half* the sample rate. A negative-frequency wave will also end up being found as a wave with a frequency from 0 cycles per second up to, and including, half the sample rate. [This should be expected as a negative-frequency wave formula can be rephrased to have a positive frequency.] If we continued to analyse a signal past the point of half the sample rate, we would then find all the countless duplicates of the waves we had already found.

For a sample rate of 10 *samples* per second, any frequency that is equal to, or faster than, 10 *cycles* per second will be recorded *in the same way* as a frequency between 0 and just under 10 cycles per second. Again, note how I emphasise that it will be recorded *in the same way* as a frequency between 0 and just under 10 cycles per second.

For a sample rate of 10 samples per second, any waves with frequencies from 5 cycles per second upwards will be recorded incorrectly – this is because the sample rate is equal to, or below, twice those frequencies. [We could also say that the sample rate is equal to, or below, the Nyquist sampling rate for those frequencies.] At 10 samples per second, a wave with a frequency of 6 cycles per second will be recorded incorrectly – the sample rate is too low to record the wave correctly. A wave with a frequency of 16 cycles per second will be recorded with the same samples as a wave with a frequency of 6 cycles per second – it will be recorded incorrectly too, but in the same incorrect way as that of the 6-cycle-per-second wave. The frequencies of 16 cycles per second and 6 cycles per second are above half the sample rate, so waves with those frequencies end up being recorded as waves with frequencies below half the sample rate.

Some examples of how faster frequencies are mapped to lower frequencies for the sample rate of 10 samples per second are as follows:

- A pure wave with a frequency of 15 cycles per second will end up being recorded *in the same way* as a pure wave with a frequency of 5 cycles per second. [Therefore, it will definitely be recorded incorrectly, and it might end up being removed from the signal altogether.]

- A pure wave with a frequency of 19 cycles per second will end up being recorded *in the same way* as a pure wave with a frequency of 9 cycles per second.

- A pure wave with a frequency of 25 cycles per second will end up being recorded *in the same way* as a pure wave with a frequency of 5 cycles per second.

- A pure wave with a frequency of 142 cycles per second will end up being recorded *in the same way* as a wave with a frequency of 2 cycles per second.

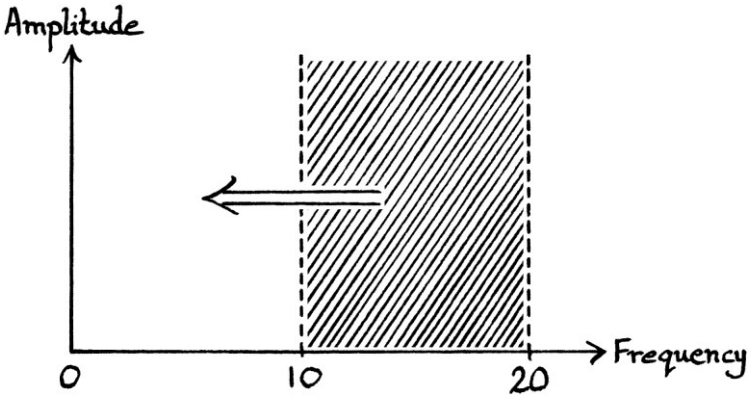
Whether a wave is on its own or part of a sum, the same recording rules apply.

Frequency domain graphs

The "$f_0 = f_0 + (x * f_s)$" rule means that any recorded discrete signal will have countless possible duplicates along the frequency axis of the frequency domain graph.

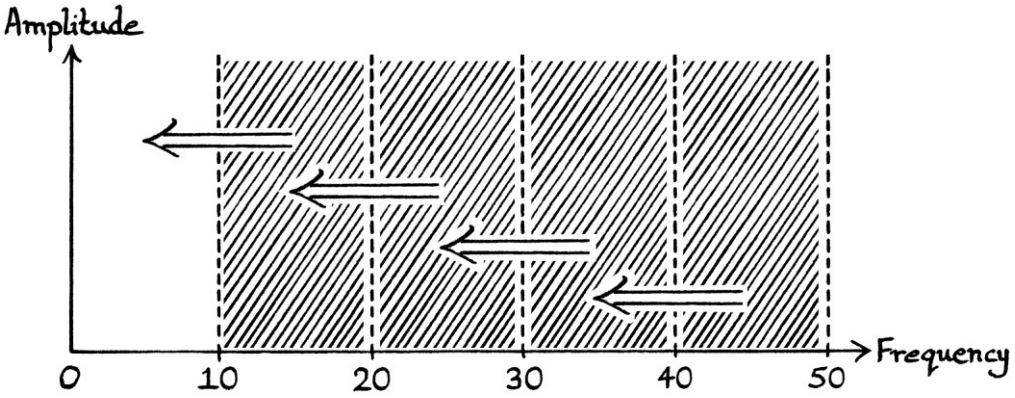
For a sample rate of 10 samples per second, if we sample any continuous waves with frequencies from 10 cycles per second up to just below 20 cycles per second, the samples will be the same as those waves with frequencies from 0 cycles per second up to just under 10 cycles per second. (We could also say that the frequencies will become "mapped" to frequencies from 0 cycles per second up to just below 10 cycles per second).

We can see this in the following graph:

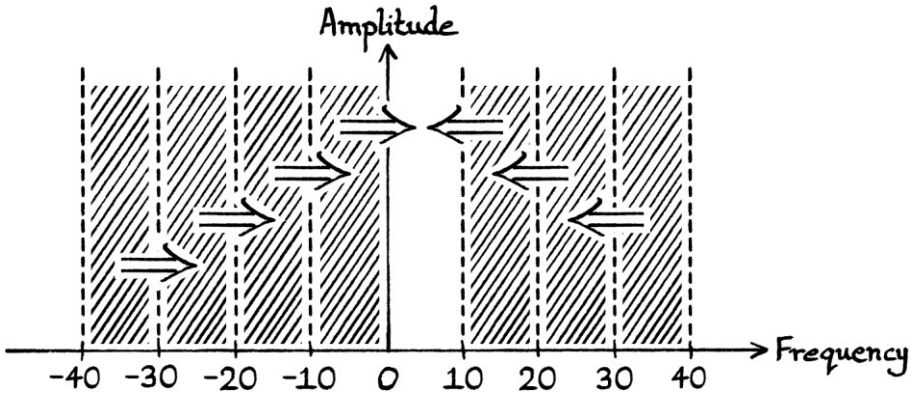


[Remember that the graph is not suggesting that any of the frequencies mentioned will be recorded *correctly* – it is saying that the higher frequencies will be recorded *in the same way* as the lower frequencies.]

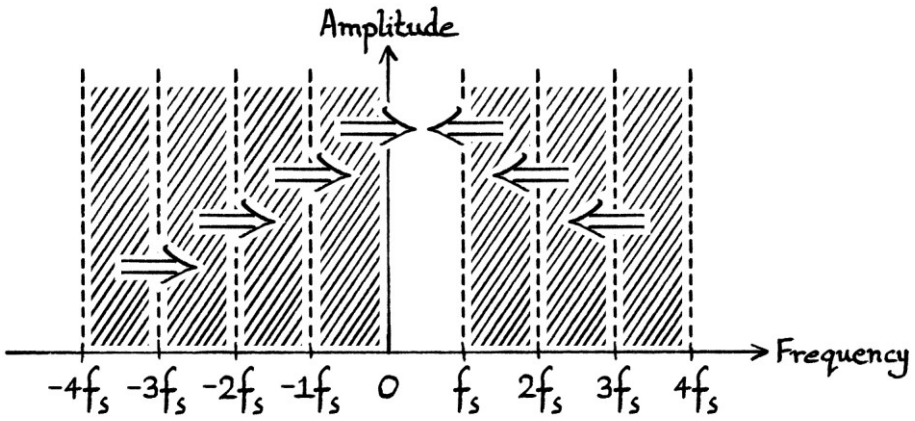
We can make the graph extend further to the right to show all the ranges of frequencies that will be recorded as frequencies below the sample rate:



We can draw the graph to take into account negative frequencies:



We can make the graph more generic by referring to the sample rate as "f_s" [in other words, the frequency of sampling or the sampling frequency]. We can then mark the frequency borders (beyond which frequencies will end up being recorded as slower frequencies) with multiples of "f_s". In this way, the graph can apply to any sample rate.



Negative frequencies

We will look at negative frequencies in more depth. We will look at some examples that use a sample rate of 10 samples per second:

- A wave with a frequency of −2 cycles per second will end up being recorded *in the same way* as a wave with a frequency of +8 cycles per second. [We just add 10 to the negative frequency.]

- A wave with a frequency of −7 cycles per second will end up being recorded *in the same way* as a wave with a frequency of +3 cycles per second.

- A wave with a frequency of −10 cycles per second will end up being recorded *in the same way* as a wave with a frequency of 0 cycles per second.

- A wave with a frequency of −11 cycles per second will end up being recorded *in the same way* as a wave with a frequency of −1 cycles per second, and also in the same way as a wave with a frequency of +9 cycles per second. [We add a multiple of 10 to the negative frequency.]

- A wave with a frequency of −124 cycles per second will end up being recorded *in the same way* as a wave with a frequency of +6 cycles per second. [We add a multiple of 10 to the negative frequency.]

Again, note how I say "*in the same way*". Positive and negative frequencies will be mapped to frequencies between 0 and just under the sample rate, but whether or not they are mapped to frequencies that can be recorded correctly is irrelevant to how they are mapped. If they are mapped to frequencies from half the sample rate to just under the sample rate, they will be recorded incorrectly. [Frequencies equal to half the sample rate might be removed from the signal completely.] Waves with frequencies from just over half the sample rate up to just under the sample rate will end up being recorded as having frequencies from just over 0 cycles per second up to just under half the sample rate.

As we know, any wave formula with a negative frequency can be rephrased to be one with a positive frequency. This means that the negative frequencies in our mapping rule could also be said to be positive frequencies with possibly different phases. This in turn means that positive frequencies from just over 0 cycles per second up to just under half the sample rate will have duplicates from just over half the sample rate to just under the sample rate.

[Generally, we would not need to sample a negative-frequency wave. We would be more likely to experience the negative frequencies when analysing a discrete signal, and then the negative frequencies would be duplicate waves appearing as positive frequencies. However, it is good to be able to think about negative frequencies with the same ease with which we think about positive frequencies.]

Rules for negative frequencies

The general wave formula for the "Faster Duplicates Ambiguity" was as so:

If we have this continuous wave sampled at "f_s" samples per second:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... then it will have identical samples to this continuous wave sampled at the same sample rate:
"$y = h + A \sin ((2\pi * (f_0 + (x * f_s)) * t) + \phi)$"
... where:

- "h" is the mean level.
- "A" is the amplitude.
- "f_0" is the frequency of the original wave.
- "x" is any positive or negative integer.
- "f_s" is the sample rate.
- "t" is the time in seconds.
- "ϕ" is the phase in radians.

The above formula includes negative frequencies as negative frequencies. We can adjust the formula so that the duplicate negative frequencies are given as positive frequencies. To do this, we have to adjust the phase and the frequency for the negative-frequency waves.

For a sample rate of 10 samples per second, these two continuous waves have the same samples:
"y = sin (2π * −8t)"
"y = sin (2π * 2t)"

A *continuous* version of:
"y = sin (2π * −8t)"
... has the exact same curve as a *continuous* version of:
"y = sin ((2π * 8t) + π)"

Therefore, we can say that for a sample rate of 10 samples per second, these two continuous waves have the same samples:
"y = sin ((2π * 8t) + π)"
"y = sin (2π * 2t)"
[Note, however, that because "y = sin ((2π * 8t) + π)" has a frequency over half the sample rate, its samples would not be a correct representation.]

For continuous waves in degrees, the rule for converting negative-frequency Sine waves to positive-frequency Sine waves is that we see how many degrees the phase is above or below 90 degrees, and we then change the phase to be that number of degrees below or above 90 degrees. We could also measure from 270 degrees instead of 90 degrees. We then make the frequency of the wave positive, and the new wave formula will have exactly the same curve as the old one. For Cosine waves, we see how many degrees the phase is above or below 0 degrees, and then we set it to that number of degrees below or above 0 degrees. We could also measure from 180 degrees instead of 0 degrees. In radians, for Sine waves, we measure to 0.5π radians or 1.5π radians, and for Cosine waves, we measure to 0 radians or π radians. This was all explained in Chapter 11.

In degrees, we can express this as:

If the original continuous negative-frequency wave has a phase of:
90 + z
... then the continuous positive-frequency wave with the same curve will have a phase of:
90 – z

"90 + z" is the phase of the first wave, but given in terms of how far away that phase is from 90 degrees. The second phase is "90 – z". This phase is the same distance from 90 degrees but in the other direction. As an example of this idea working, if the first wave had a phase of 91 degrees, we would express it as "90 + 1". Therefore, the second wave would have a phase of "90 – 1" which is 89 degrees.

In radians, if the negative-frequency Sine wave has a phase of:
$0.5\pi + z$
... then the positive-frequency Sine wave will have a phase of:
$0.5\pi - z$

We can also calculate the phase that is the other side of 90 degrees (0.5π radians) by subtracting that phase from 180 degrees (π radians). Doing this makes each formula easier to read, but at the expense of making them slightly less intuitive. If the negative-frequency Sine wave has the phase:
"ϕ" degrees
... then the positive-frequency Sine wave will have the phase:
"$180 - \phi$" degrees.

In radians, if the negative-frequency Sine wave has the phase:
"ϕ" radians
... then the positive-frequency Sine wave will have the phase:
"$\pi - \phi$" radians.

We will incorporate the above idea into our final duplicate wave formula shortly.

For waves with frequencies between zero and just under the sample rate, the first negative-frequency duplicate will be from the negative of the sample rate up to just under zero. To calculate its frequency, we just subtract the sample rate from the wave that we have. We can express this as follows. If the frequency of a sampled wave is:
f_0
... then the first negative-frequency duplicate wave will have the frequency:
$f_0 - f_s$

When that frequency is made positive, it will become:
$-(f_0 - f_s)$
... which can be written as:
$-f_0 + f_s$
... which is:
$f_s - f_0$

We now know how to express mathematically the first negative-frequency duplicate and how it appears as a positive frequency. We also know how to adjust the phase. We can combine these methods into one formula that gives the negative-frequency-made-positive Sine wave formula duplicate for a positive-frequency Sine wave:

If we have a discrete wave based on this continuous wave:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... then there will be a negative-frequency-made-positive discrete wave with a frequency under the sample rate that has the same samples, which will be based on this wave:
"$y = h + A \sin ((2\pi * (f_s - f_0) * t) + \pi - \phi)$"
... where:

- "h" is the mean level
- "A" is the amplitude
- "f_0" is the frequency of the first wave
- "f_s" is the sample rate
- "ϕ" is the phase of the first wave

As an example, if we have a sample rate of 23 samples per second, then this wave:
"$y = 3 \sin ((2\pi * 6t) + 0.2\pi)$"
... will have exactly the same samples as the negative-frequency-made-positive wave:
"$y = 3 \sin ((2\pi * (23 - 6) * t) + (\pi - 0.2\pi)$"
... which is:
"$y = 3 \sin ((2\pi * 17t) + 0.8\pi)$"

[If we had not used the formula, we would have had to calculate the negative-frequency wave first. It is "$y = 3 \sin ((2\pi * -17t) + 0.2\pi)$". We would then have to convert it to a positive-frequency wave.]

We now know that, for a sample rate of 23 samples per second, the samples from:
"$y = 3 \sin ((2\pi * 6t) + 0.2\pi)$"
... and:
"$y = 3 \sin ((2\pi * 17t) + 0.8\pi)$"
... are identical to each other.

Just as the original wave will have duplicates that are higher by integer multiples of the sample rate, so will the negative-frequency-made-positive duplicate have duplicates that are higher by integer multiples of the sample rate.

We can express them with this formula:

"$y = h + A \sin ((2\pi * (f_s - f_0 + (x * f_s)) * t) + \pi - \phi)$"
... where:

- "x" is a *positive* integer from zero upwards.

The formula is the same as the previous negative-frequency-made-positive formula, but now it contains its own higher frequency duplicates. This formula is general enough to refer to every possible negative-made-positive duplicate. Although we could have stayed with the original "$f_0 = f_0 + (x * f_s)$" idea and had "x" as a negative number to refer to negative frequencies, when we actually analyse a discrete wave, we will only find positive frequencies. Therefore, the negative-frequency-made-positive formula is more useful for describing what we will actually find.

Rule for the Faster Duplicates Ambiguity

We can now say that any discrete record of this continuous wave:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... will have identical samples to the waves described by these two wave formulas:
"$y = h + A \sin ((2\pi * (f_0 + (x * f_s)) * t) + \phi)$"
"$y = h + A \sin ((2\pi * (f_s - f_0 + (x * f_s)) * t) + \pi - \phi)$"
... as long as "x" is any positive integer.

We will look at an example that uses these formulas.

We will use a sample rate of 13 samples per second, and we will look at the samples of the wave:
"$y = 1.1 \sin ((2\pi * 3t) + 0.9\pi)$"

The samples of this discrete wave for the first second are as so:

0.3399
−0.9976
−0.5804
0.8576
0.7872
−0.6679
−0.9482
0.4393

1.0541
−0.1852
−1.0987
−0.0797
1.0795

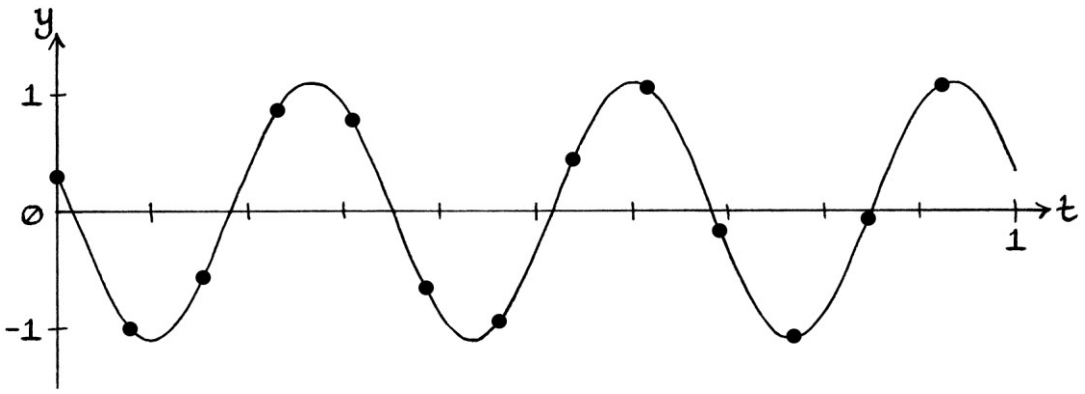
If we analysed these samples using Fourier series analysis and:

- We used test waves with frequencies that were sequential integer multiples of *one* cycle per second. [In other words, the test frequencies would be 1, 2, 3, 4, 5, 6 ... cycles per second and so on.]
- We used the same sample rate of 13 samples per second.
- We continued testing frequencies after we had found our wave, and after we had passed the frequency equal to half the sample rate, and after we had passed the frequency equal to the sample rate.

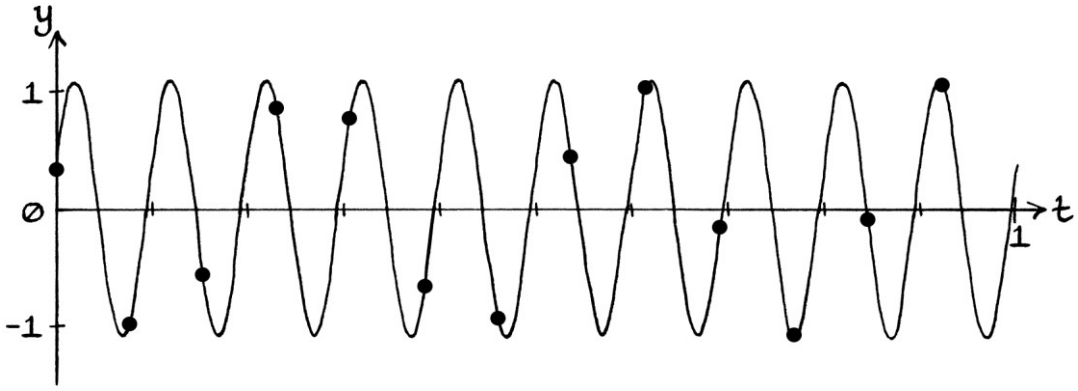
... then we would find an infinite number of waves that had exactly the same samples, and which were all based on these two formulas:
"y = 1.1 sin ((2π * (3 + (x * 13)) * t) + 0.9π)"
"y = 1.1 sin ((2π * (13 – 3 + (x * 13)) * t) + π – 0.9π)"
... where "x" is every positive integer from zero upwards.

We can make the formulas tidier as so:
"y = 1.1 sin ((2π * (3 + 13x) * t) + 0.9π)"
"y = 1.1 sin ((2π * (10 + 13x) * t) + 0.1π)"

The first two waves that we would find would have frequencies under the sample rate. The first would be the wave that we started with (where "x" is zero):
"y = 1.1 sin ((2π * 3t) + 0.9π)"

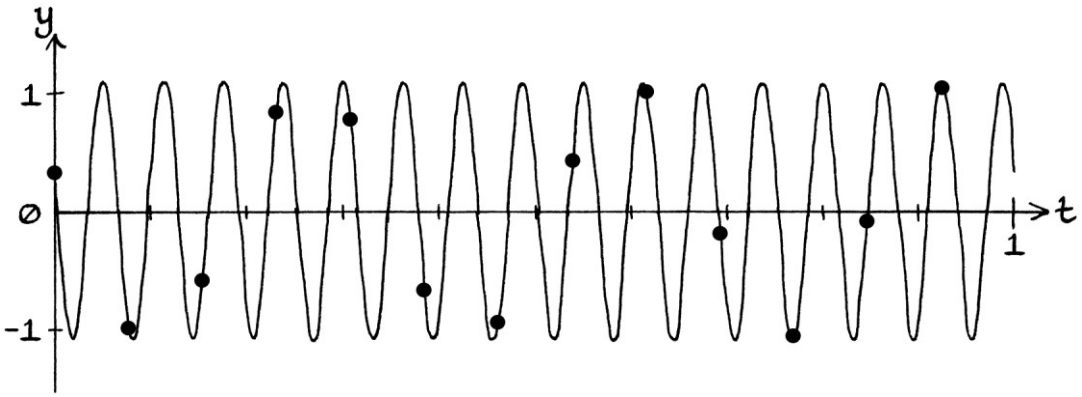


The second would be the negative-frequency-made-positive wave:
"y = 1.1 sin ((2π * 10t) + 0.1π)"

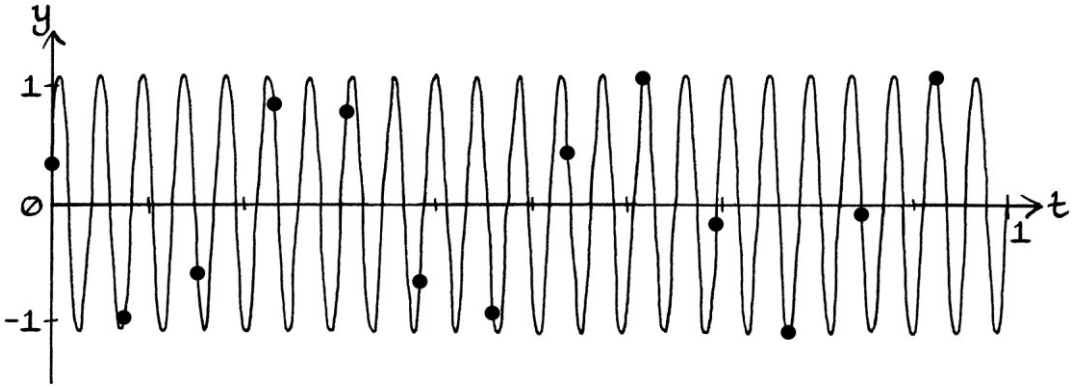


As we can see, the samples for the above wave are not a correct representation of the wave, but, nevertheless, they are the samples that we would find.

The next two waves we would find would have frequencies over the sample rate, but under twice the sample rate. The first would be:
"y = 1.1 sin ((2π * 16t) + 0.9π)"



The second wave would be:
"y = 1.1 sin ((2π * 23t) + 0.1π)"



The next two waves we would find would have frequencies over twice the sample rate, but under three times the sample rate. They would be:
"y = 1.1 sin ((2π * 29t) + 0.9π)"
"y = 1.1 sin ((2π * 36t) + 0.1π)"

The next two waves we would find would have frequencies over three times the sample rate, but under four times the sample rate. They would be:
"y = 1.1 sin ((2π * 42t) + 0.9π)"
"y = 1.1 sin ((2π * 49t) + 0.1π)"

As we continued, we would find:

"y = 1.1 sin ((2π * 55t) + 0.9π)"
"y = 1.1 sin ((2π * 62t) + 0.1π)"

"y = 1.1 sin ((2π * 68t) + 0.9π)"
"y = 1.1 sin ((2π * 75t) + 0.1π)"

"y = 1.1 sin ((2π * 81t) + 0.9π)"
"y = 1.1 sin ((2π * 88t) + 0.1π)"

"y = 1.1 sin ((2π * 94t) + 0.9π)"
"y = 1.1 sin ((2π * 101t) + 0.1π)"

... and so on.

Summary

A summary of this section is that, for a given sample rate "f_s", the wave:
"$y = h + A \sin((2\pi * f_0 * t) + \phi)$"
... will have identical samples to both of these two wave formulas:
"$y = h + A \sin((2\pi * (f_0 + (x * f_s)) * t) + \phi)$"
"$y = h + A \sin((2\pi * (f_s - f_0 + (x * f_s)) * t) + \pi - \phi)$"
... where "x" is any positive integer.

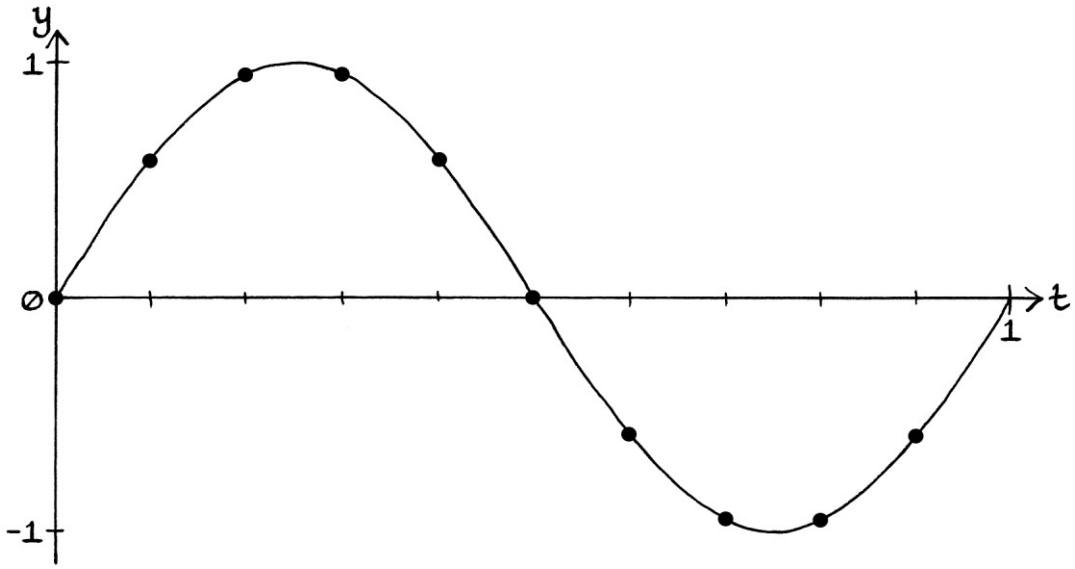
If we were analysing a discrete signal, we would find the possible duplicates if we continued testing frequencies past the frequency equal to half the sample rate.

Straight Line Ambiguity

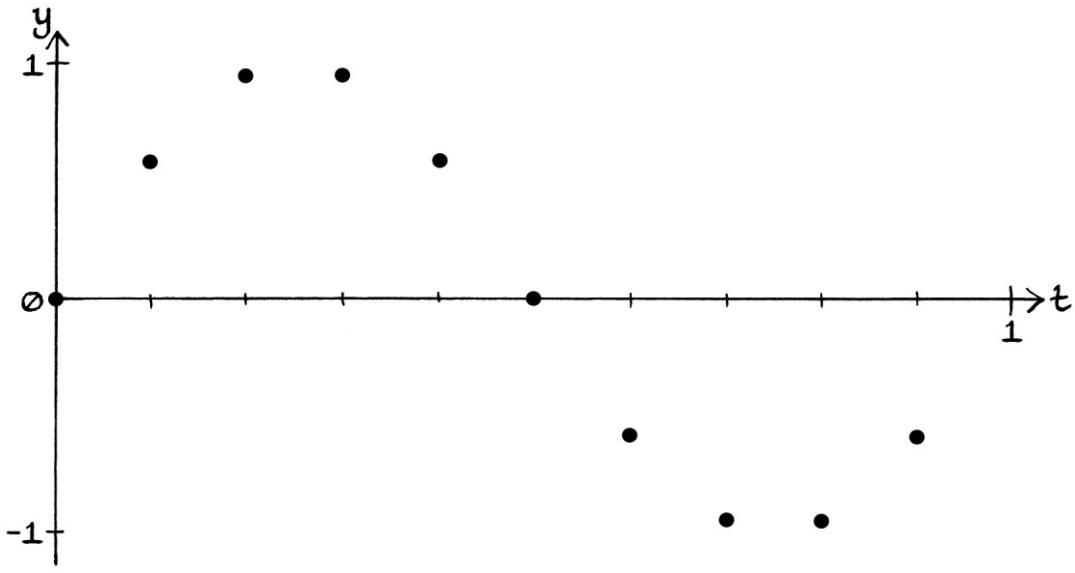
The “Straight Line Ambiguity” refers to the idea that the samples taken from a curved signal will be indistinguishable from those taken from a signal that had straight lines connecting the places from where the samples were taken.

As an example, we will sample the continuous wave:
“y = sin (2π * 1t)”
... with a sample rate of 10 samples per second.

The continuous wave with the positions of the samples looks like this:



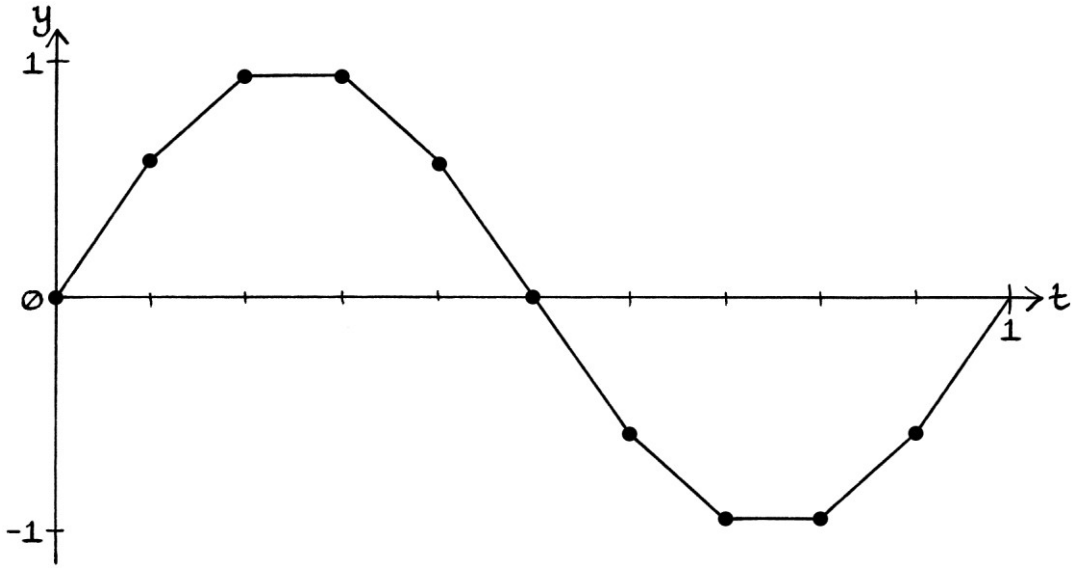
The discrete wave looks like this:



The samples for one second are as follows:

Time (seconds)	**Sample**
0	0
0.1	0.5878
0.2	0.9511
0.3	0.9511
0.4	0.5878
0.5	0
0.6	−0.5878
0.7	−0.9511
0.8	−0.9511
0.9	−0.5878

The samples are identical to the samples that we would obtain from a continuous signal where those points were connected with straight lines. In other words, this continuous signal has exactly the same samples:



This straight-line continuous signal is actually the sum of the following continuous waves:

“y = 0.9675 sin (2π * 1t)”
“y = 0.0119 sin ((2π * 9t) + π)”
“y = 0.0080 sin (2π * 11t)”
“y = 0.0027 sin ((2π * 19t) + π)”
“y = 0.0022 sin (2π * 21t)”

"y = 0.0012 sin ((2π * 29t) + π)"
"y = 0.0010 sin (2π * 31t)"
... and other waves with lower amplitudes.

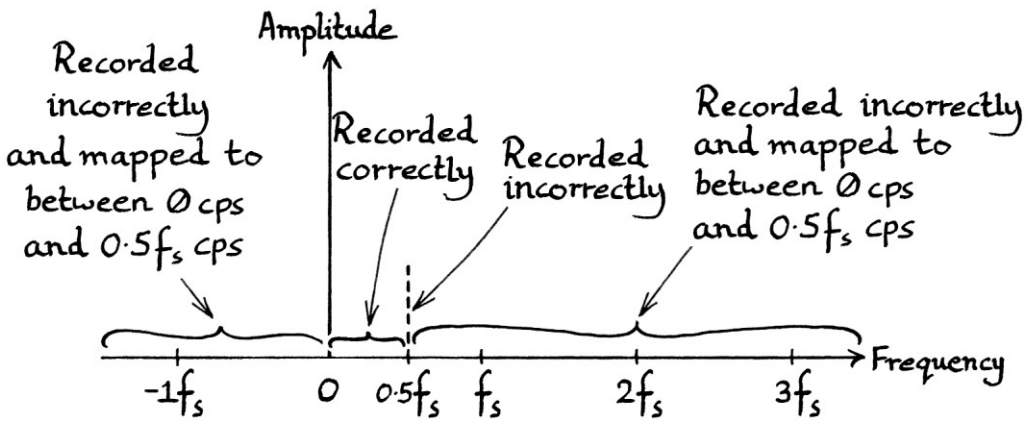
It is a completely different signal from our "y = sin (2π * 1t)" pure wave, but it has the same samples. For any list of samples, we cannot know if the samples were intended to represent a curved wave or signal, or a signal with straight lines connecting the same points. This is the "Straight Line Ambiguity".

The frequency domain

We know that waves with frequencies that are outside the range of 0 cycles per second up to just under the sample rate will be recorded in the same way as waves with frequencies from 0 cycles per second up to just under the sample rate.

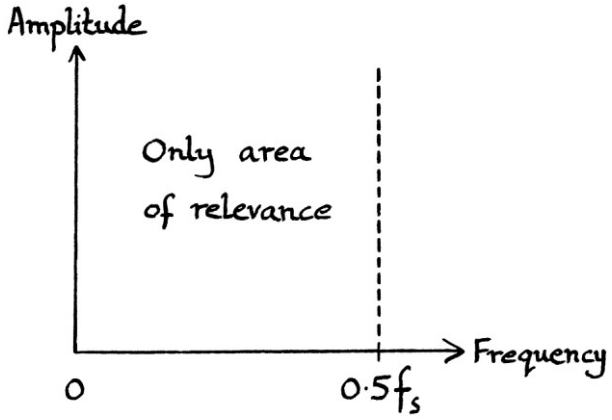
We also know that any wave with a frequency equal to, or above, *half* the sample rate will be recorded as a wave with a frequency from 0 cycles per second up to and including half the sample rate. [If a frequency that is equal to half the sample rate is not recorded as all zeroes, it would be possible to analyse the signal and find a wave with the correct frequency that, after halving the amplitude, had the correct samples, but it might not be the actual original wave. Any wave with a frequency higher than half the sample rate by a multiple of the sample rate will end up being recorded in the same way as a wave with a frequency equal to half the sample rate.]

A descriptive frequency domain graph that shows how waves are recorded is as so:



If we analysed a discrete signal, we would first find the constituent waves with frequencies from zero cycles per second up to, and including, half the sample rate. If we continued searching, we would find the first set of negative-frequency-made-positive wave duplicates, and then all the countless duplicate waves with faster frequencies.

Given all of these facts, there is never any point in searching for constituent frequencies above half the sample rate. There cannot be anything worth searching for above the frequency equal to half the sample rate because no continuous wave above that frequency would be recorded as being above that frequency. Anything found there must be a duplicate. If we were analysing a signal sensibly, we would stop just after we had tested for the frequency equal to half the sample rate. Similarly, a discrete-wave frequency domain graph has no need to extend past the frequency equal to half the sample rate. Anything past that point can only ever be a duplicate of the part of the graph we have. A discrete-wave frequency domain graph can be truncated as so:



Note that we can only truncate it because we are dealing with *discrete* signals, and the frequencies of waves in discrete signals, first, always end up being recorded as being equal to or below half the sample rate, and second, have duplicates that repeat along the frequency axis. There is no need to have the later part of the frequency axis because it will just consist of copies of the part of the frequency axis that we do have. We would not be able to truncate a *continuous*-wave frequency domain graph because it would not repeat in that way. [We can truncate the frequency domain graph of a continuous signal so that we only include the waves that exist in the signal or those in which we are interested. However, we cannot truncate it for repetition reasons.]

Periodic and discrete frequencies

Given that analysis of a signal with ever-faster test waves would continue to find duplicates of the waves we had already found, it is common for people to say that the frequency domain for discrete periodic signals is, itself, *periodic*. In other words, the frequencies repeat forever in both directions along the frequency axis. The term "periodic" is usually used with waves and signals, but it is just as apt to use it to describe the frequency domain for discrete signals.

Although it is not relevant to this chapter, it is also common for people to say that the discovered frequencies of either a continuous periodic signal or a discrete periodic signal are *discrete*, in the sense that they are separate individual frequencies. The word "discrete" is usually used with waves, but it is just as apt to use it with the frequency domain too. The usefulness of calling the frequencies "discrete" will only become apparent when we see frequencies that are "continuous" in the frequency domain. When frequencies are continuous in the frequency domain, it means that we could zoom into any part of the frequency domain, and we would still see a frequency – there would be no gaps between the frequencies. Such an idea occurs with the Fourier transform, which is used to analyse aperiodic signals.

10-sample-per-second examples

We will go through every integer frequency for a sample rate of 10 samples per second to see how the different Ambiguities become relevant, and what other matters of interest arise. We will use the continuous wave formula:
"$y = \sin(2\pi * f * t)$"
... and we will change the value of "f" so that we can examine every wave.

0 cycles per second:

If the frequency of our wave is zero cycles per second, the continuous wave will be a straight line, and all the samples of the discrete version will be the same as each other. In this way, a sample rate of 10 samples per second will record the wave correctly. What is more, *any* sample rate will record a frequency of 0 cycles per second correctly. As every value of the continuous signal is the same, any number of readings taken per second will all have that value. However, we would not be able to distinguish our list of identical samples from an undersampled wave that,

by chance, had that value at the places where the samples were taken. For the wave "y = sin (2π * 0t)" sampled at 10 samples per second, its samples would be the same as those from:

- "y = sin (2π * 5t)" because each peak and each dip of this 5-cycle-per-second wave would occur between the places where the readings were taken.
- "y = sin (2π * 10t)" because each entire *cycle* of this 10-cycle-per-second wave would occur between the places where the readings were taken.
- "y = sin (2π * 15t)" because two peaks and one dip would occur between the places where the readings were taken. To put this another way, 1.5 cycles would occur between each sample reading.
- "y = sin (2π * 20t)" because two entire cycles would occur between each of the places where the readings were taken.
- "y = sin (2π * 25t)" because 2.5 cycles would occur between each sample reading.

... and so on.

The samples would also be the same as the samples from these negative-frequency waves:

- "y = sin (2π * −5t)", which is the same as "y = sin ((2π * 5t) + π)". Each peak and dip occurs between the places where the samples are read. As the phase of the wave given a positive frequency is π radians (180 degrees), the values are still all zero, as they were with the wave "y = sin (2π * 5t)".
- "y = sin (2π * −10t)", which is the same as "y = sin ((2π * 10t) + π)". The wave given a positive frequency starts at π radians (180 degrees) when it is zero. Each cycle occurs entirely between where the samples are read.
- "y = sin (2π * −15t)", which is the same as "y = sin ((2π * 15t) + π)".
- "y = sin (2π * −20t)", which is the same as "y = sin ((2π * 20t) + π)".
- "y = sin (2π * −25t)", which is the same as "y = sin ((2π * 25t) + π)".
- "y = sin (2π * −30t)", which is the same as "y = sin ((2π * 30t) + π)".

... and so on.

We will look at what happens if we have a wave with a non-zero phase. We will use "y = sin ((2π * 0t) + 0.25π)" with a sample rate of 10 samples per second. Whatever the sample rate, all the samples will be sin (0.25π) = 0.7071 units. In this case, our discrete wave will have the same samples as:

- "y = sin ((2π * 10t) + 0.25π)"
- "y = sin ((2π * 20t) + 0.25π)"
- "y = sin ((2π * 30t) + 0.25π)"
- "y = sin ((2π * 40t) + 0.25π)"

... and so on.

The 5-cycle-per-second wave is not a duplicate in this case because the non-zero phase means that the samples of "y = sin ((2π * 5t) + 0.25π)" are:
0.7071
−0.7071
0.7071
−0.7071
... and so on. Therefore, the non-zero phase stops the 5-cycle-per-second wave having the same samples as the zero-frequency wave. If the phase had been π radians (180 degrees), the samples *would* have all been the same as each other. Therefore, we can see that the value of the phase affects the possible waves that have the same samples. For a sample rate of 10 samples per second, the frequencies higher by multiples of 10 cycles per second will definitely have the same samples, but the frequencies at 5-cycle-per-second intervals between those 10-cycle-per-second waves will only have the same samples if the phase is 0 or π radians.

For the non-zero-phase wave "y = sin ((2π * 0t) + 0.25π)", there are also the negative-frequency waves that have the same samples for a sample rate of 10 samples per second. These are:

- "y = sin ((2π * −10t) + 0.25π)", which is "y = sin ((2π * 10t) + 0.75π)", which, given the sample rate, is also "y = sin ((2π * 0t) + 0.75π)"
- "y = sin ((2π * −20t) + 0.25π)", which is "y = sin ((2π * 20t) + 0.75π)"
- "y = sin ((2π * −30t) + 0.25π)", which is "y = sin ((2π * 30t) + 0.75π)"
- "y = sin ((2π * −40t) + 0.25π)", which is "y = sin ((2π * 40t) + 0.75π)"

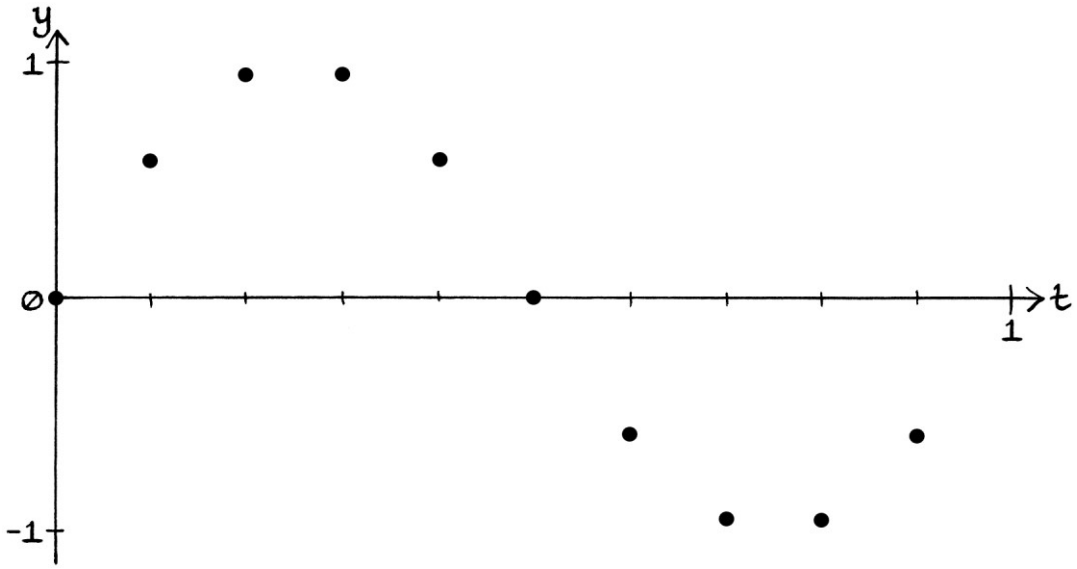
... and so on.

For a sample rate of 10 samples per second and a frequency of 0 cycles per second, we only really see the "Faster Duplicates Ambiguity" in effect – there are duplicates with higher frequencies that have the same samples as our zero-frequency wave, but our zero-frequency wave will be recorded correctly. The "Straight Line Ambiguity" does not apply here because a zero-frequency continuous wave is a straight line, so it would be identical to a continuous wave that had straight lines between where the samples were read.

1 cycle per second

A pure wave with a frequency of 1 cycle per second will be recorded correctly, in the sense that we would be able to recover the wave's characteristics exactly by analysing the discrete signal.

For example, the wave “y = sin (2π * 1t)” will be recorded correctly:



Its samples over one cycle are:
0
0.5878
0.9511
0.9511
0.5878
0
−0.5878
−0.9511
−0.9511
−0.5878

As with all the non-zero frequencies, the “Straight Line Ambiguity” will be in effect – these samples describe our continuous 1-cycle-per-second Sine wave, but would also describe a continuous signal that had straight lines between those points.

For a sample rate of 10 samples per second, other waves would produce the same samples. The following waves all have the same samples:

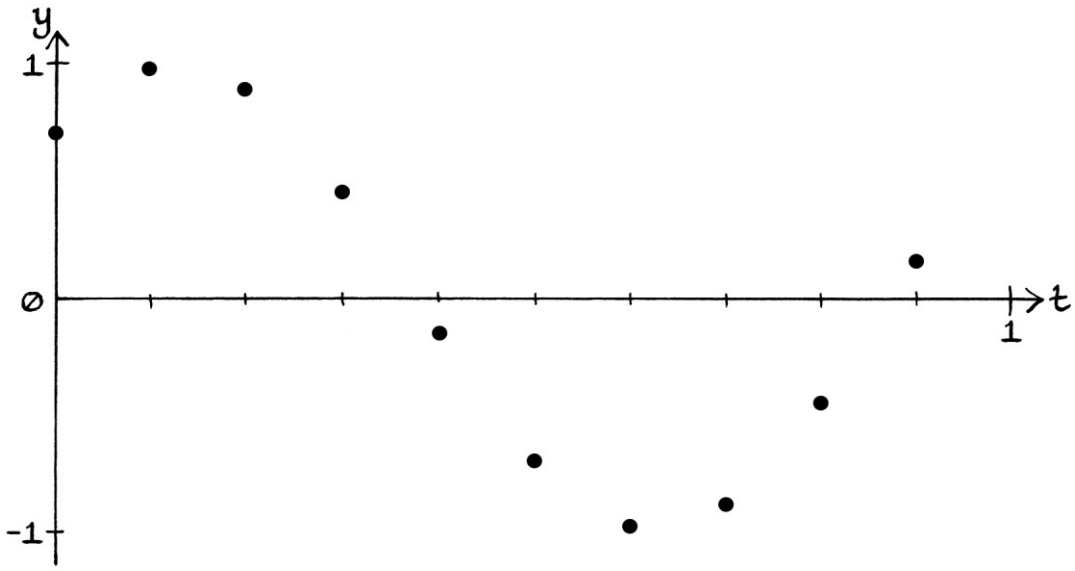
- “y = sin (2π * 1t)”
- “y = sin (2π * 11t)”
- “y = sin (2π * 21t)”
- “y = sin (2π * 41t)”
- “y = sin (2π * 51t)”
- “y = sin (2π * 61t)”

... and so on.

The samples are also those of these negative-frequency waves:
"y = sin (2π * −9t)", which is "y = sin ((2π * 9t) + 0.5π)"
"y = sin (2π * −19t)", which is "y = sin ((2π * 19t) + 0.5π)"
"y = sin (2π * −29t)", which is "y = sin ((2π * 29t) + 0.5π)"
"y = sin (2π * −39t)", which is "y = sin ((2π * 39t) + 0.5π)"
"y = sin (2π * −49t)", which is "y = sin ((2π * 49t) + 0.5π)"
"y = sin (2π * −59t)", which is "y = sin ((2π * 59t) + 0.5π)"
"y = sin (2π * −69t)", which is "y = sin ((2π * 69t) + 0.5π)"
... and so on.

As we have seen before in this chapter, waves that are different in frequency by an integer multiple of the sample rate have the same samples. This is the "Faster Duplicates Ambiguity".

We will look at a wave with a non-zero phase: "y = sin ((2π * 1t) + 0.25π)"



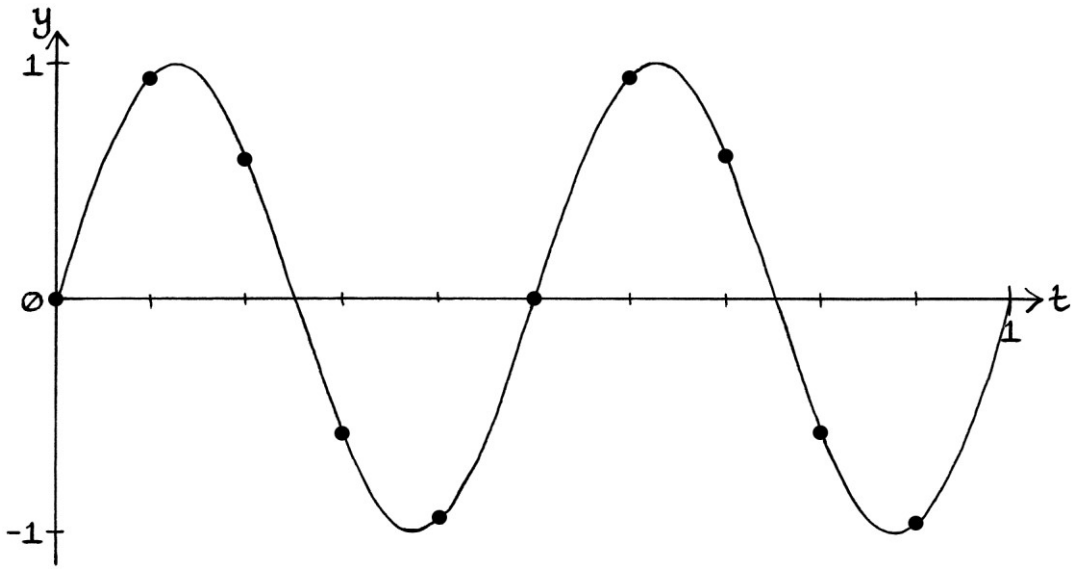
Its samples over one cycle are:
0.7071
0.9877
0.8910
0.4540
−0.1564
−0.7071
−0.9877
−0.8910
−0.4540
0.1564

For a sample rate of 10 samples per second, this list of samples belongs to the following waves:
“y = sin ((2π * 1t) + 0.25π)”
“y = sin ((2π * 11t) + 0.25π)”
“y = sin ((2π * 21t) + 0.25π)”
“y = sin ((2π * 31t) + 0.25π)”
“y = sin ((2π * 41t) + 0.25π)”
“y = sin ((2π * 51t) + 0.25π)”
... and so on.

The list of samples also belongs to these waves:
“y = sin ((2π * −9t) + 0.25π)”, which is “y = sin ((2π * 9t) + 0.75π)”
“y = sin ((2π * −19t) + 0.25π)”, which is “y = sin ((2π * 19t) + 0.75π)”
“y = sin ((2π * −29t) + 0.25π)”, which is “y = sin ((2π * 29t) + 0.75π)”
“y = sin ((2π * −39t) + 0.25π)”, which is “y = sin ((2π * 39t) + 0.75π)”
“y = sin ((2π * −49t) + 0.25π)”, which is “y = sin ((2π * 49t) + 0.75π)”
“y = sin ((2π * −59t) + 0.25π)”, which is “y = sin ((2π * 59t) + 0.75π)”
... and so on.

2 cycles per second

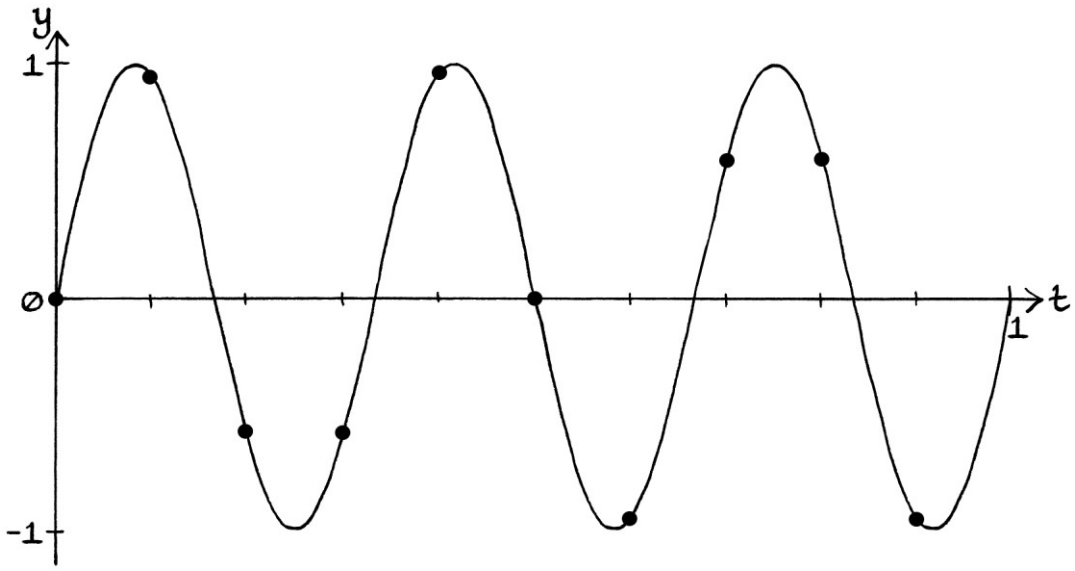
A pure wave with a frequency of 2 cycles per second will be recorded correctly, in the sense that it will be possible to analyse the discrete signal and discover its characteristics. There will be faster frequency waves that have the same samples, and straight-line continuous signals will share the same samples too. The samples of “y = sin (2π * 2t)” drawn over its curve look like this:



Note how the frequency of the discrete wave is the same as that of the continuous wave – the values of the discrete samples repeat twice per second. This will not always be the case, as we shall shortly see.

3 cycles per second

A pure wave with a frequency of 3 cycles per second will be recorded correctly. The samples of "y = sin (2π * 3t)"drawn over its curve look like this:



The samples from the discrete version of "y = sin (2π * 3t)" over 1 second are:

Time (seconds)	**Sample value**	
0.0	0	
0.1	0.9511	
0.2	−0.5878	
0.3	−0.5878	
0.4	0.9511	
0.5	0	
0.6	−0.9511	
0.7	0.5878	
0.8	0.5878	
0.9	−0.9511	
1.0	0	[This is where the samples repeat]
1.1	0.9511	

1.2	−0.5878
1.3	−0.5878
1.4	0.9511
1.5	0
1.6	−0.9511
1.7	0.5878
1.8	0.5878
1.9	−0.9511

Note that although the continuous wave has a frequency of 3 cycles per second, the samples do not repeat until the sample at t = 1.0 seconds. In other words, the frequency of the discrete wave is actually 1 cycle per second, and not 3 cycles per second. Despite this, Fourier series analysis would still discover the correct continuous wave from these samples if we used test waves that were integer multiples of the frequency of the *discrete* signal. Whenever we analyse a discrete signal, we have to use test waves that are integer multiples of the frequency of the *discrete signal*, and not integer multiples of the continuous signal that was sampled.

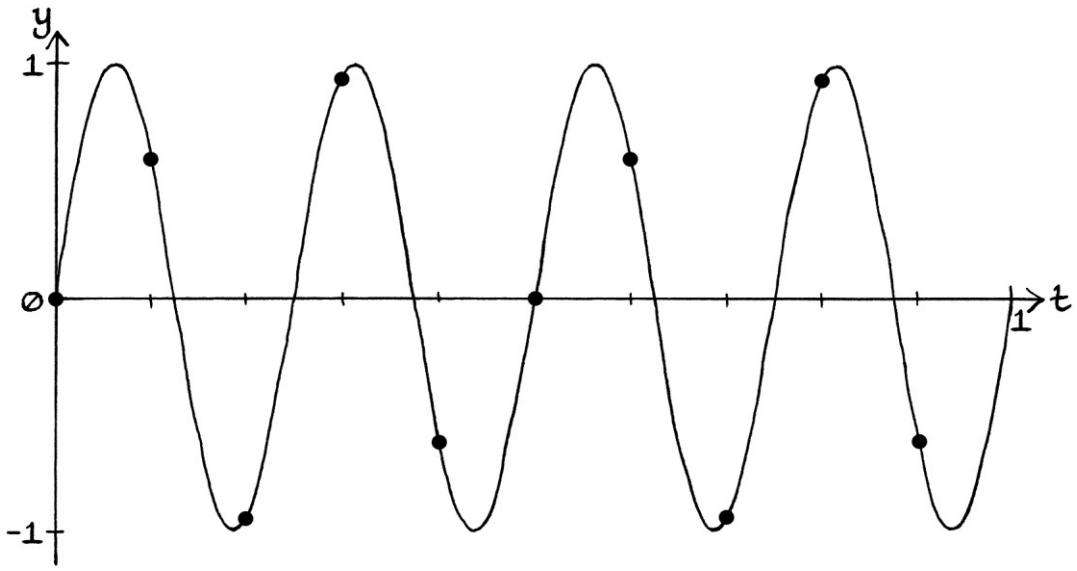
4 cycles per second

A pure wave with a frequency of 4 cycles per second will be recorded correctly. It will not have samples that make it visually obvious that it is a 4-cycles-per-second wave, but analysis of the samples would reveal the characteristics of the wave. For example, over one second, the wave "y = sin (2π * 4t)" will have the samples:

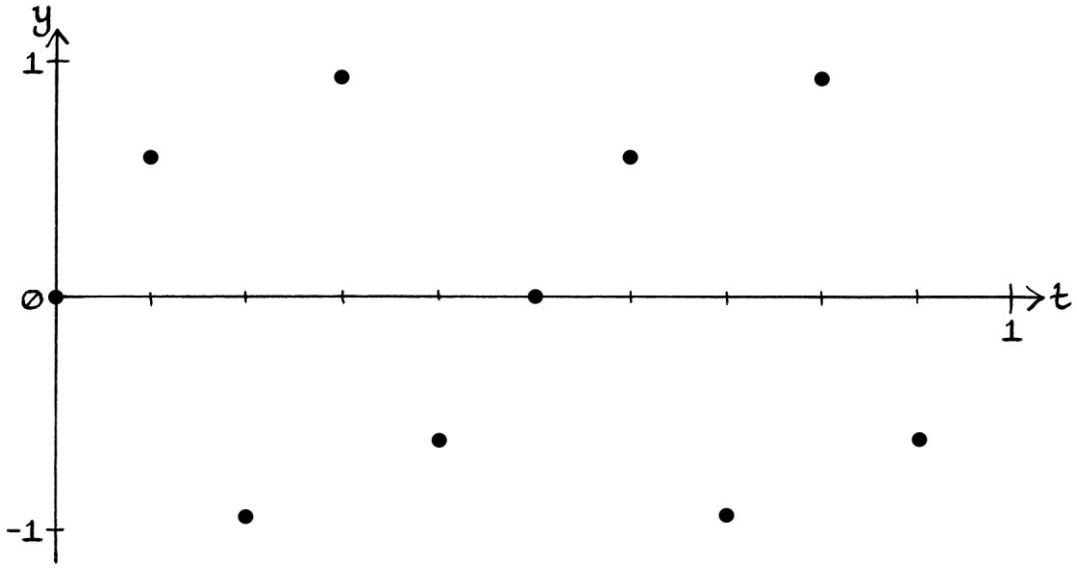
Time (seconds)	**Sample value**
0.0	0
0.1	0.5878
0.2	−0.9511
0.3	0.9511
0.4	−0.5878
0.5	0
0.6	0.5878
0.7	−0.9511
0.8	0.9511
0.9	−0.5878

Notice how the continuous wave has a frequency of 4 cycles per second, but the discrete wave has a frequency of 2 cycles per second. Fourier series analysis would need test frequencies that were integer multiples of 2 cycles per second.

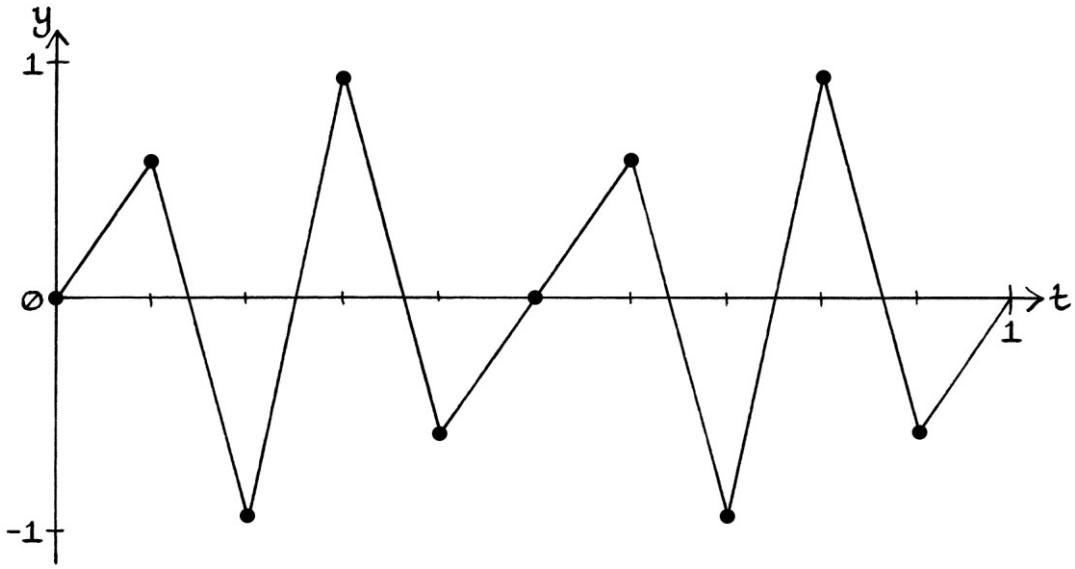
The samples drawn on the 4-cycle-per-second continuous wave look like so:



When we see the samples on their own, we can see how the samples do not particularly look like those from a 4-cycle-per-second wave:



This is clearer still if we join up the samples (for the sole purpose of making their position clearer):



5 cycles per second

For a sample rate of 10 samples per second, a pure wave with a frequency of 5 cycles per second will end up being recorded *incorrectly.* If we analyse the samples using Fourier series analysis, we will not find the original wave, but we might be able to alter our results afterwards to find the correct wave or a wave that has the same samples. For a *Sine* wave with a zero mean level, there are three different situations that we can have for a wave that has a frequency equal to half the sample rate:

- For phases of zero or π radians (0 or 180 degrees), every sample will be zero, so the wave may as well not exist. The original wave will be unrecoverable.

- For phases of 0.5π radians or 1.5π radians (90 or 270 degrees), Fourier series analysis will recover the correct frequency and phase, but the amplitude will be twice what it should be.

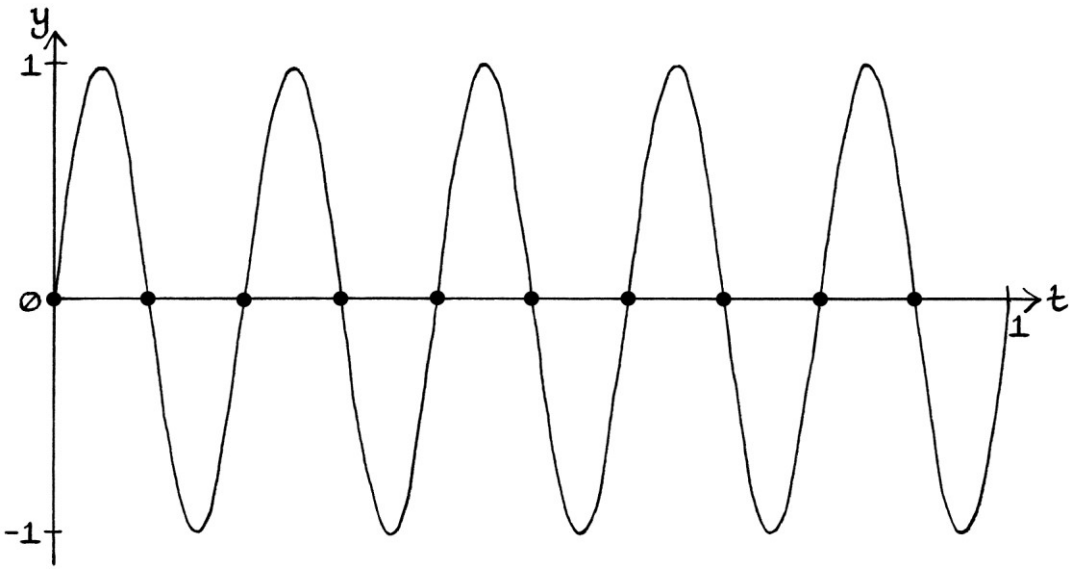
- For phases other than 0 radians, 0.5π radians, π radians and 1.5π radians, Fourier series analysis will be able to discover the correct frequency, but the phase will be miscalculated as 0.5π or 1.5π radians, and the amplitude will be twice that of a different wave that has the same samples.

For the first situation, it would be impossible to find the wave because such a wave is erased when it is sampled. For the second situation, we can find the correct wave by analysing the signal and then halving the amplitude. For the third situation, analysing the signal and then halving the amplitude will find a wave that has the correct samples, but it will not be the original wave. It is worth noting that usually when analysing a discrete signal, we would not be able to tell which of the second or third situations had occurred – we would have no way of knowing if we had the correct wave with twice the amplitude or a different wave with twice the amplitude that produced the same samples.

One of the main sampling rules is that to be sure of recording a wave so that Fourier series analysis can recover it, the frequency must be less than half the sample rate. In this situation, the frequency is 5 cycles per second and the sample rate is 10 samples per second, so we are trying to record a frequency equal to half the sample rate.

The most obvious example of how we cannot record a 5-cycle-per-second wave correctly with a 10 sample-per-second sample rate is in the sampling of a wave with a phase of 0 or π radians (0 and 180 degrees). For the waves:
"$y = \sin (2\pi * 5t)$"
... and:
"$y = \sin ((2\pi * 5t) + \pi)$"
... all the samples will be zero because each peak and each dip of the wave will occur between the points where the samples are taken.

The samples of "$y = \sin (2\pi * 5t)$" drawn over its curve are as so:

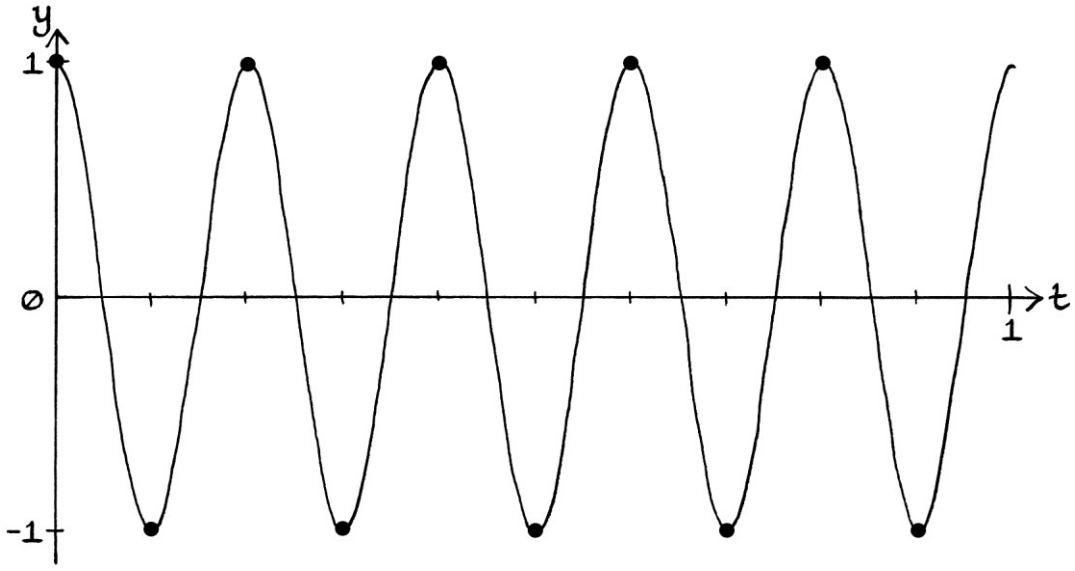


For *any* Sine wave with a phase of zero radians or π radians (0 degrees or 180 degrees), and zero mean level, if its frequency is equal to half the sample rate, every sample will be zero. For any Cosine wave with a phase of 0.5π radians or 1.5π radians (90 degrees or 270 degrees), and zero mean level, if its frequency is equal to half the sample rate, every sample will be zero. The first 10 samples of the wave will be:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0

Obviously, there is no way to recover the original wave from these samples. If the wave is part of a sum of waves, it will have no effect on the sum, so it may as well not exist at all.

We will now look at the wave "y = sin ((2π * 5t) + 0.5π)". This wave has a phase of 0.5π radians (90 degrees). Every sample is alternately +1 and −1. The first 10 samples are:
1, −1, 1, −1, 1, −1, 1, −1, 1, −1

The samples drawn over the continuous wave are as so:



The frequency of the discrete wave is 5 cycles per second, which is the same as that of the continuous wave. Although the samples for this wave look correct, if we analysed them using Fourier series analysis with the same sample rate of 10 samples per second, we would, instead, end up discovering the wave:
"y = 2 sin ((2π * 5t) + 0.5π)"

In other words, the frequency, phase and mean level would be discovered correctly, but the amplitude would be *twice* what it should be. The samples for:
"y = 2 sin ((2π * 5t) + 0.5π)"
... are: 2, −2, 2, −2, 2, −2, 2, −2, 2, −2

For any Sine wave with a phase of 0.5π radians and a frequency equal to half the sample rate, Fourier series analysis would discover the amplitude as being twice what it should be.

[Note that we can know all the characteristics of the continuous wave that created the samples "1, −1, 1, −1, 1, −1, 1, −1, 1, −1" by just looking at the samples. We know that a Sine wave with a 0.5π radian (90 degree) phase [or a Cosine wave with zero phase] and an amplitude of 1 unit, would create those samples. However, we cannot use Fourier series analysis to calculate those characteristics. If the wave were part of a sum of waves, there would be no way to know *visually* that the wave was in the sum, and Fourier series analysis would find the wave but with the wrong amplitude. [We would know that if we found a wave with a frequency equal to half the sample rate that we should halve the amplitude.]

Fourier series analysis (as we have been performing it) will always fail for waves with frequencies equal to half the sample rate. This is because the test waves we use for analysis will also be sampled at that sample rate, and therefore, they will be limited in their values.

We will look at why analysis fails. For a 5-cycle-per-second discrete wave, the only two test waves that will play a part in the analysis will be the 10-sample-per-second discrete versions of:
"y = 2 sin (2π * 5t)"
... and:
"y = 2 sin (2π * 5t) + 0.5π)"

[Remember that test waves always have amplitudes of *2 units*. If they did not have amplitudes of 2 units, the amplitudes of analysed waves that had been recorded correctly would be half of what they should be. This is a consequence of multiplying two waves together, and was explained in Chapters 16 and 18.]

If we were dealing with continuous signals, the range of values for each test wave would go through every value from +2 to −2. However, because we are dealing with discrete signals, the problems with sampling affect our test waves too.

For a sample rate of 10 samples per second, the first 10 samples for our two test waves will be as so:

Samples from "y = 2 sin (2π * 5t)"	**Samples from "y = 2 sin ((2π * 5t) + 0.5π)"**
0	2
0	−2
0	2
0	−2
0	2
0	−2
0	2
0	−2
0	2
0	−2

Every value of the first test wave is zero. Every value of the second test wave is alternately +2, then −2. In fact, these values are not just the values for the 5-cycle-per-second test waves with a sample rate of 10 samples per second, they are the values for *any* test waves that have frequencies equal to half the sample rate. For example, for a sample rate of 124 samples per second, the test waves for the frequency of 62 cycles per second would have zero values for the zero-phase Sine test wave, and values of +2, −2, +2, −2 and so on for the 0.5π-phase Sine test wave. For a sample rate of 21 samples per second, the test waves for the frequency of 10.5 cycles per second would have the same values.

Every value of the first test wave is zero. Therefore, when we multiply the test wave's samples by the corresponding samples of the discrete wave that we are analysing, the resulting signal will consist only of samples that are zero. Its mean level will be zero. From a Fourier analysis point of view, the "first mean level" will be zero. For *any* wave that has a frequency equal to half the sample rate, no matter what its amplitude, phase or mean level, when we use Fourier series analysis, the first mean level will be zero. This instantly means that the phase of every discovered wave that has a frequency equal to half the sample rate can only be either 0.5π radians or 1.5π radians (90 degrees or 270 degrees). [The phase point of the circle from which the discovered wave could be said to be derived, will have an x-axis coordinate of zero, so whatever the y-axis value, the point will be at an angle of 0.5π or 1.5π radians.]

For the second test wave, every value is alternately +2 and −2. This will be the case for any second test wave if the frequency is equal to half the sample rate. When it comes to analysing waves, this produces particular results.

We will look at how we would analyse our "y = sin ((2π * 5t) + 0.5π)" wave. The first ten samples of our wave, next to the first ten samples of our *first* test wave, and the result of the multiplication are as so:

Sample from our 5 cps wave	**Sample from "y = 2 sin (2π * 5t)"**	**Multiplied together**
1	0	0
−1	0	0
1	0	0
−1	0	0
1	0	0
−1	0	0
1	0	0
−1	0	0
1	0	0
−1	0	0

Every value of our test wave is zero. Therefore, the result of multiplying each sample from our discrete wave by our test wave is zero. Therefore, the "first mean level" will be zero. The first ten samples of our "y = sin ((2π * 5t) + 0.5π)" wave, next to the first ten samples of our *second* test wave, and the result of the multiplication are as so:

Sample from our 5 cps wave	**Sample from "y = 2 sin ((2π * 5t) + 0.5π)"**	**Multiplied together**
1	2	2
−1	−2	2
1	2	2
−1	−2	2
1	2	2
−1	−2	2
1	2	2
−1	−2	2
1	2	2
−1	−2	2

The values from our second test wave are all either +2 or −2. Therefore, when we multiply them by the corresponding values of the wave being analysed, every result is +2. The average is 2, so the "second mean level" is 2.

To calculate the amplitude of the analysed wave, we use Pythagoras's theorem:
$\sqrt{first\ mean\ level^2 + second\ mean\ level^2}$
... which, because the first mean level is zero, will be:
$\sqrt{second\ mean\ level^2}$
... which is just the value of the second mean level made positive. In this case, it is 2 units.

To calculate the phase of the analysed wave, we would normally use arctan, but as the first mean level is always zero, we can just imagine the coordinates of the phase point on a circle – it can only be at 90 degrees or 270 degrees. In this case, it will have the coordinates (0, 2), so it will be 2 units away from the origin of the axes at an angle of 90 degrees. Therefore, the recovered wave in our example will be "y = 2 sin ((2π * 5t) + 0.5π)". The difference between the actual wave and the recovered wave is that the amplitude is twice what it should be.

We will look at the samples of various 5-cycle-per-second waves sampled at 10 samples per second:

"y = sin (2π * 5t)" will have the samples:
0, 0, 0, 0, and so on.

"y = sin ((2π * 5t) + 0.1π)" will have the samples:
0.3090, −0.3090, 0.3090, −0.3090, and so on.

"y = sin ((2π * 5t) + 0.2π)"will have the samples:
0.5878, −0.5878, 0.5878, −0.5878, and so on.

"y = sin ((2π * 5t) + 0.3π)"will have the samples:
0.8090, −0.8090, 0.8090, −0.8090, and so on.

"y = sin ((2π * 5t) + 0.5π)"will have the samples:
1, −1, 1, −1, and so on.

"y = sin ((2π * 5t) + 0.9π)"will have the samples:
0.3090, −0.3090, 0.3090, −0.3090, and so on.

"y = sin ((2π * 5t) + π)"will have the samples:
0, 0, 0, 0, and so on.

"y = sin ((2π * 5t) + 1.1π)"will have the samples:
−0.3090, 0.3090, −0.3090, 0.3090, and so on.

"y = sin ((2π * 5t) + 1.4π)"will have the samples:
−0.9511, 0.9511, −0.9511, 0.9511, and so on.

"y = sin ((2π * 5t) + 1.5π)"will have the samples:
−1, 1, −1, 1, and so on.

"y = sin ((2π * 5t) + 1.6π)"will have the samples:
−0.9511, 0.9511, −0.9511, 0.9511, and so on.

"y = 12.9 sin ((2π * 5t) + 1.7127π)"will have the samples:
−10.1256, 10.1256, −10.1256, 10.1256, and so on.

As we can see, there are patterns for the samples of every wave. These are:

- Each wave's samples will be the same absolute value – in other words, each value made positive will be the same.

- If the wave has a phase between 0 and π radians exclusive (0 and 180 degrees exclusive), each sample will be alternately positive, then negative. The samples follow the pattern:
  +A, −A, +A, −A, +A, −A, and so on.

- If the wave has a phase between π and 2π radians exclusive (180 and 360 degrees exclusive), each sample will be alternately negative, then positive. The samples follow the pattern:
  −A, +A, −A, +A, −A, +A and so on.

- If the wave has a phase of 0 or π radians, each sample will be zero.

Given these facts, and given that the second test wave consists of the values +2, −2, +2, −2, and so on, it means that multiplying the second test wave by the samples of any of the waves will result in:

- A series of identical positive samples if the phase is between 0 and π radians exclusive.
- A series of identical negative samples if the phase is between π and 2π radians exclusive.

Therefore, the "second mean level" will be any one of the resulting samples made positive if the phase is between 0 and π radians exclusive, and it will be any one of the samples made negative if the phase is between π and 2π radians exclusive. [The "first mean level" will always be zero.]

To phrase this in another way: when we multiply the samples from our second test wave with the corresponding samples from the wave we are testing, we will achieve one of two things: either we will double them and make them all positive, or we will double them and make them all negative. Whichever happens, all the samples of the resulting signal will have the same value as each other, so the mean level will be the same as any one sample. This means that the "second mean level" will either be twice the absolute value of any sample of the wave being analysed or the negative of twice the absolute value.

As the "first mean level" is always zero, if the "second mean level" consists of just positive values, the phase of the discovered wave will *always* be 0.5π radians (90 degrees). If the "second mean level" consists of just negative values, the phase of the discovered wave will *always* be 1.5π radians (270 degrees). This means that, as long as the phase of our *original* 5-cycle-per-second Sine wave is not 0 or π radians, the phase of the *discovered* Sine wave will always be calculated as being 0.5π radians or 1.5π radians. It cannot be anything else. Therefore, Fourier series analysis cannot recover the *phase* of a wave correctly unless by chance it was actually 0.5π or 1.5π radians to start with. Note that despite not finding the correct wave, if we halve the amplitude, the wave we *do* find will have the correct samples.

As the "first mean level" will always consist of zero samples, the amplitude of the discovered wave will always be equal to any sample of the second mean level made positive. To be more specific, it will be twice the absolute value of any sample from the original wave being analysed. For example, if the samples of the original wave are −A, +A, −A, +A and so on, or +A, −A, +A, −A and so on, then the amplitude of the discovered wave will be +2A.

It is easiest to see what is happening in a table, and it is easiest to see if we work in degrees. We will look at a series of results for the wave:
"y = sin ((360 * 5t) + ϕ)"
... where:

- The sample rate is 10 samples per second.
- The frequency of the wave is 5 cycles per second.
- Sine is working in degrees.
- The phase is in degrees.
- The amplitude of the wave is 1 unit.

Phase of wave	**1st sample**	**2nd sample**	**Amplitude calculated with Fourier series analysis**	**Phase calculated with Fourier series analysis**
0	0	0	0	[Not relevant]
1	0.0175	−0.0175	0.0349	90
45	0.7071	−0.7071	1.4142	90
60	0.8660	−0.8660	1.7321	90
89	0.9998	−0.9998	1.9997	90
90	1	−1	2	90
91	0.9998	−0.9998	1.9997	90
95	0.9962	−0.9962	1.9924	90
135	0.7071	−0.7071	1.4142	90
180	0	0	0	[Not relevant]
185	−0.0872	0.0872	0.1743	270
225	−0.7071	0.7071	1.4142	270
265	−0.9962	0.9962	1.9924	270
270	−1	1	2	270
275	−0.9962	0.9962	1.9924	270
315	−0.7071	0.7071	1.4142	270
359	−0.0175	0.0175	0.0349	270

[In each signal, all the samples after the first and second samples will be the same as the first and second samples.]

From the table, we can draw the following conclusions about Sine waves with frequencies equal to half the sample rate:

- If the phase of the original Sine wave is zero or 180 degrees, every sample will be zero, and Fourier series analysis will not be able to find the wave.

- Whatever the phase of the original Sine wave, the first sample will be the Sine of the phase, the second sample will be the negative of the Sine of the phase, the third sample will be the Sine of the phase, the fourth sample will be the negative of the Sine of the phase, and so on.

- Given that, if the phase is between 0 and 180 degrees exclusive, every sample will be alternately positive then negative. If the phase is between 180 and 360 degrees exclusive, every sample will be alternately negative then positive. If the phase is 90 degrees, every sample will be alternately +1, then −1. If the phase is 270 degrees, every sample will be alternately −1, then +1.

- When we analyse the samples, if the original phase was between 0 and 180 degrees exclusive, we will find a phase of 90 degrees. If the original phase was between 180 and 360 degrees exclusive, we will find a phase of 270 degrees.

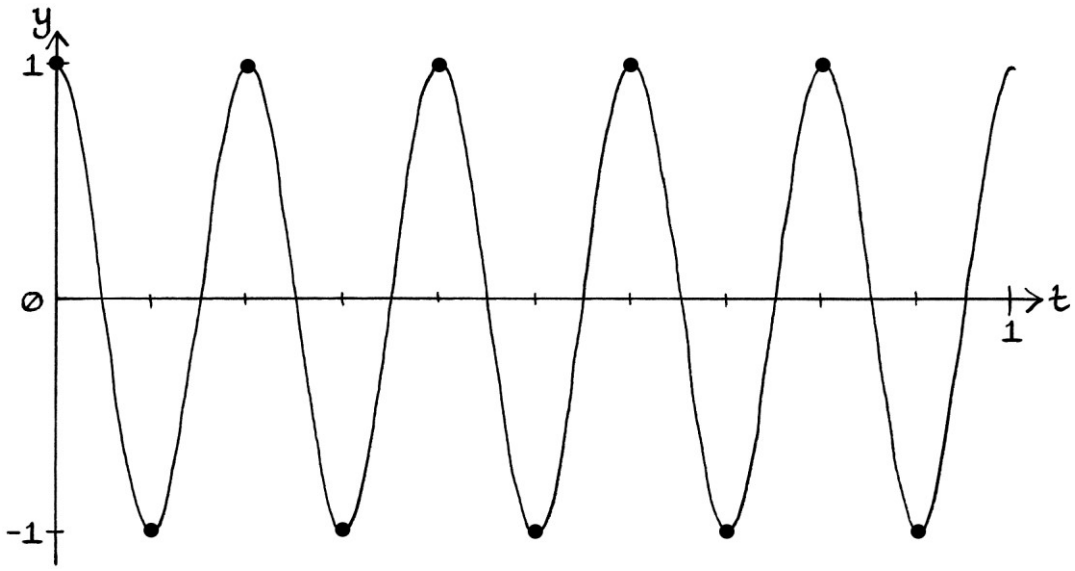
- When we analyse the samples, if the original phase was exactly 90 degrees or exactly 270 degrees, we will find the correct phase, but the discovered amplitude will be twice what it should be.

- There are waves with different phases that will be recovered as the same wave. For example, a Sine wave with a phase between 0 and 180 degrees exclusive that is a particular number of degrees higher than 90 degrees will be indistinguishable from a wave with a phase that number of degrees lower than 90 degrees. A wave with a phase between 180 and 360 degrees exclusive that is so many degrees higher than 270 degrees will be indistinguishable from a wave with a phase that number of degrees lower than 270 degrees.

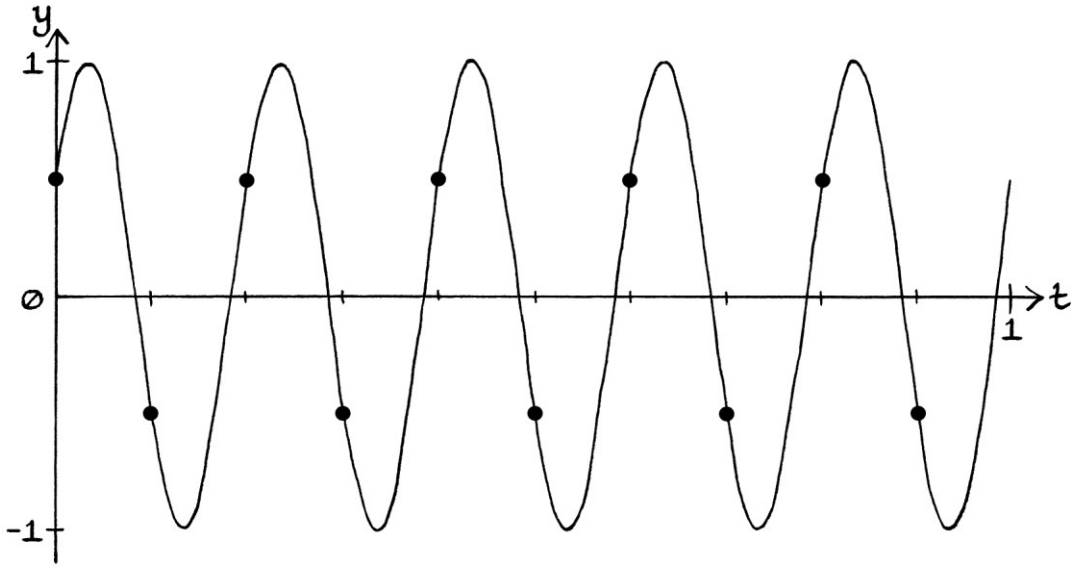
Analysis discovers each amplitude as being twice the absolute value of the Sine of the original phase – in other words, the Sine of the original phase made positive, even if it is negative. If the Sine of the original phase is negative, the discovered phase will be 270 degrees; if the Sine of the original phase is positive, the discovered phase will be 90 degrees. We could rephrase this and say that every

discovered amplitude is equal to twice the value of the Sine of the original phase, and the discovered phase will be 90 degrees. If this leads to a negative-amplitude wave, we can rearrange it to be a positive-amplitude wave with a phase of 270 degrees, and it will have the same curve. [To turn a negative-amplitude wave formula into a positive-amplitude wave formula and maintain the same curve, we just add 180 degrees to the phase.]

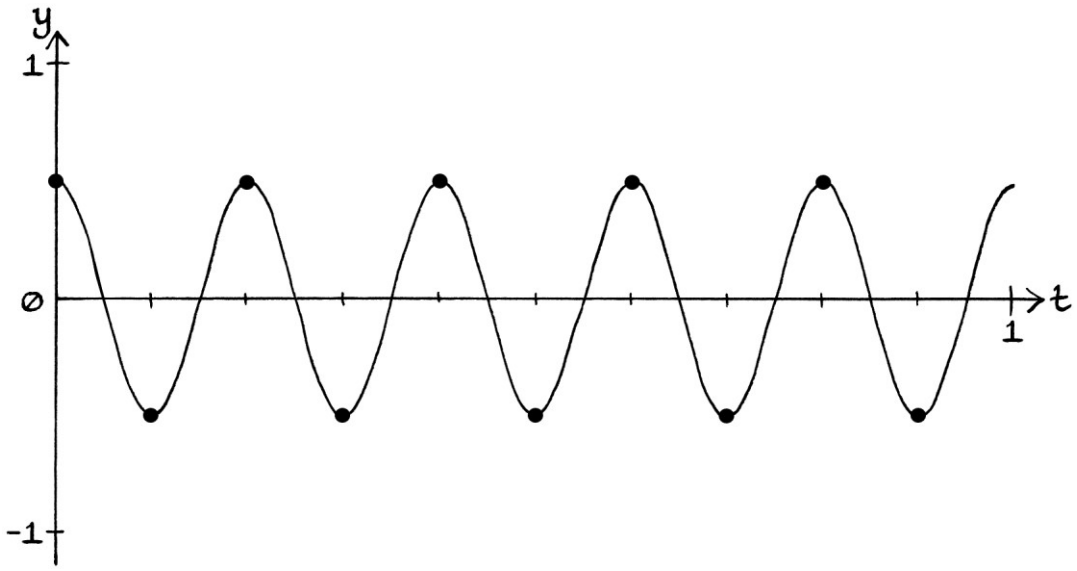
Another way to understand that the discovered phases will always be either 90 degrees or 270 degrees is through looking at the graphs of sampled waves. If a sampled wave has a phase of 90 degrees or 270 degrees, each sample will be at a peak or dip:



If a sampled wave has a phase other than 0 degrees, 90 degrees, 180 degrees, or 270 degrees, then no samples will be at a peak or dip:



If this is the case, the actual peaks and dips are not recorded, and the samples will be identical to those from a wave that had those samples at its peaks and dips. In other words, the samples will be the same as those from a wave with the same frequency, a lower amplitude and a phase that is 90 degrees or 270 degrees:



The equal-to-half-the-sample-rate rule

From all of the above, we can formulate a rule for a 5-cycle-per-second wave with a sample rate of 10 samples per second. If we have the samples from this wave in degrees:
"y = A sin ((360 * 5t) + ϕ)"
... then Fourier series analysis of the samples would find this wave:
"y = 2A sin (ϕ) * sin ((360 * 5t) + 90)"
... where:

- the amplitude of the calculated wave might be negative, in which case, rearranging the formula would result in a positive amplitude and a phase of 270 degrees.
- Sine is working in degrees, and "ϕ" is an angle in degrees.

The above rule takes into account phases of 0 or 180 degrees. In such cases, the phase of the calculated wave will be 90 degrees, but the amplitude will be 2 * 0 = 0 units.

As an example of the rule in use, if we had the wave:
"y = 2.11 sin ((360 * 5t) + 277.3)"
... and we sampled it at 10 samples per second, then Fourier series analysis would misinterpret the samples as being from:
"y = 2 * 2.11 * sin (277.3) * sin ((360 * 5t) + 90)"
... which is:
"y = 4.22 * sin (277.3) * sin ((360 * 5t) + 90)"
... which is:
"y = −4.1858 * sin ((360 * 5t) + 90)"
... which is:
"y = 4.1858 * sin ((360 * 5t) + 270)"

The wave "y = 2.11 sin ((360 * 5t) + 277.3)" sampled at 10 samples per second has the samples: −2.0929, +2.0929, −2.0929, +2.0929 and so on.

The wave "y = 4.1858 * sin ((360 * 5t) + 270)" sampled at 10 samples per second has the samples: −4.1858, +4.1858, −4.1858, +4.1858 and so on. These samples are double those of "y = 2.11 sin ((360 * 5t) + 277.3)". The wave we have found is incorrect, but if we halve the amplitude, we end up with this wave:
"y = 2.0929 * sin ((360 * 5t) + 270)"
... which produces *the correct samples*.

The formula in radians is as follows. If we have the samples from this wave:
"y = A sin ((2π * 5t) + ϕ)"
... then Fourier series analysis of the samples will find this wave:
"y = 2A sin (ϕ) * sin ((2π * 5t) + 0.5π)"
... where:

- the amplitude of the calculated wave might be negative, in which case, rearranging the formula would result in a positive amplitude and a phase of 1.5π radians.
- Sine is working in radians, and "ϕ" is an angle in radians.

We will look at an example that uses a different sample rate. We will sample the following degrees-based wave at 50 samples per second:
"y = 5.5 sin ((360 * 25t) + 128)"

The first two samples, and all the samples afterwards, are:
4.3341 and −4.3341

If we analysed the sampled wave using Fourier series analysis, we would find the wave:
"y = 8.6681 sin ((360 * 25t) + 90)"

The value of 8.6681 is twice the amplitude multiplied by the Sine of 128 degrees. Therefore, the rule still works for any wave that has a frequency that is equal to half the sample rate.

For waves in degrees, the rule for *any* sample rate is as follows:

If we have this wave being sampled at "f_s" samples per second:
"$y = A \sin ((360 * 0.5f_s * t) + \phi)$"
... then Fourier series analysis of the samples would find this wave:
"$y = 2A \sin (\phi) * \sin ((360 * 0.5f_s * t) + 90)$"
... where:

- the amplitude of the calculated wave might be negative, in which case, rearranging the formula would result in a positive amplitude and a phase of 270 degrees.
- Sine is working in degrees, and "ϕ" is an angle in degrees.

For waves in radians, the rule for *any* sample rate is as follows:

If we have this wave being sampled at "f_s" samples per second:
"$y = A \sin ((2\pi * 0.5f_s * t) + \phi)$"
... then Fourier series analysis of the samples would find this wave:
"$y = 2A \sin (\phi) * \sin ((2\pi * 0.5f_s * t) + 0.5\pi)$"
... where:

- the amplitude of the calculated wave might be negative, in which case, rearranging the formula would result in a positive amplitude and a phase of 1.5π radians.
- Sine is working in radians, and "ϕ" is an angle in radians.

We will call the above rule, "the equal-to-half-the-sample-rate" rule.

The most important things to know about what we have learnt in this and the previous section are:

If we analyse a wave with Fourier series analysis, and the wave has a frequency equal to half the sample rate:

- If the phase is 0 or π radians, the wave will end up with samples that are all zero, so it may as well not exist.
- If the phase is 0.5π or 1.5π radians, we will find the correct wave, but its recovered amplitude will be twice what it should be.
- If the phase is any other angle, the wave will be recovered as one with a phase of 0.5π or 1.5π radians. If we halve the discovered amplitude, we will have a wave with the correct samples, but it will not be the correct wave.
- Different original waves can end up being recovered as the same wave.

Given all of the above, there are two useful facts:

- It makes sense never to sample a wave with a frequency equal to half the sample rate. We cannot be sure that we will be able to recover it correctly or at all.

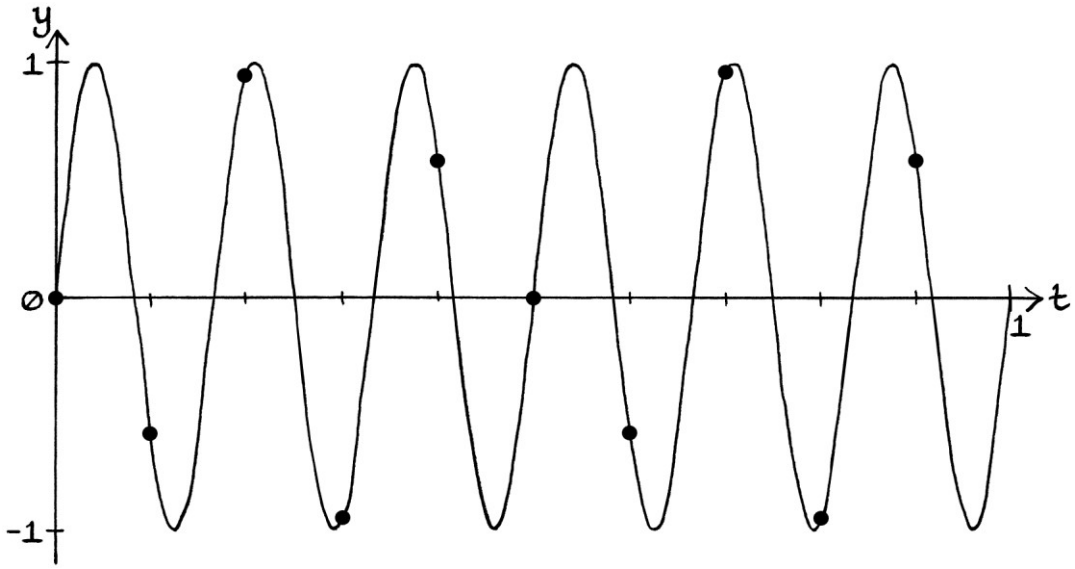
- When analysing a signal, we cannot be sure if it was sampled correctly, so we *do* need to test for the frequency equal to half the sample rate. In which case, we must halve the discovered amplitude. We will end up with a wave with the correct samples, but it might not be the original constituent wave that was in the signal.

6 cycles per second

A pure wave with a frequency of 6 cycles per second will end up being recorded incorrectly because the sample rate is too low. The frequency is greater than half the sample rate. We could also say that the sample rate is less than twice the frequency. For “y = sin (2π * 6t)” over one second, the samples will be:

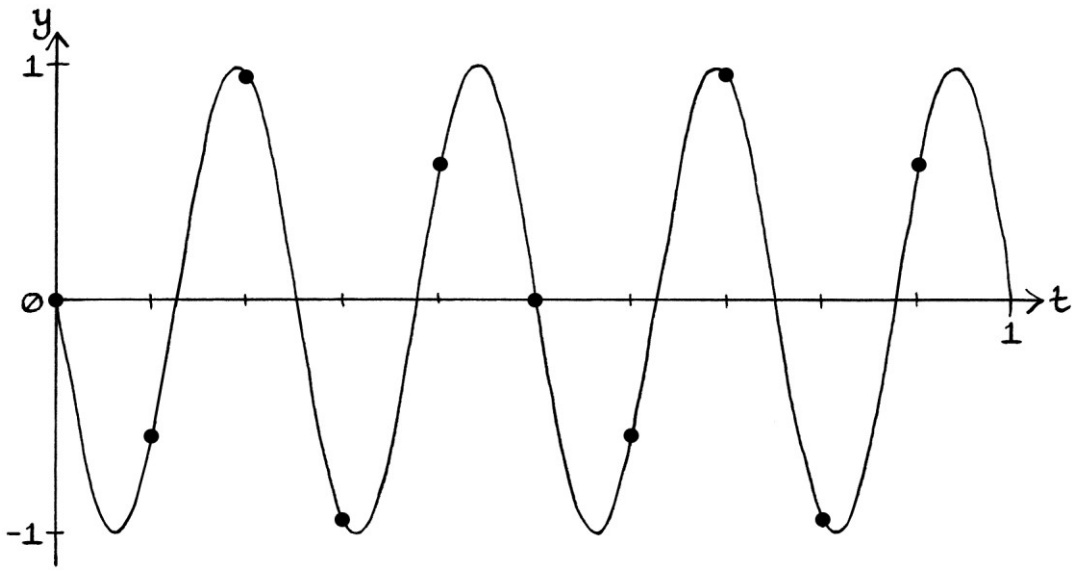
0
−0.5878
0.9511
−0.9511
0.5878
0
−0.5878
0.9511
−0.9511
0.5878

These samples drawn over the continuous wave are as so:



The samples are actually those from a 4 cycle-per-second wave (or a −4 cycles per second wave). The samples are the same as those from:
“y = −sin (2π * 4t)”
... or:
“y = sin (2π * 4t) + π)”
... or:
“y = sin (2π * −4t)”

The above three continuous waves have exactly the same curve as each other. The samples drawn over that curve are as so:



Exactly in which way the wave is recorded incorrectly is clearer if we have a wave with a non-zero phase. The wave "y = sin ((2π * 6t) + 0.25π)" would have the same samples as:
"y = sin ((2π * −4t) + 0.25π)" [A phase equal to 45 degrees.]
... or:
"y = sin ((2π * 4t) + 0.75π)" [A phase equal to 135 degrees.]
... or:
"y = −sin ((2π * 4t) + 1.75π)" [A phase equal to 315 degrees.]
[These three continuous waves also have the same curve as each other.]

It is most straightforward to say that the samples for "y = sin ((2π * 6t) + 0.25π)" are the same as those from a wave with the same phase, but a frequency of −4 cycles per second. The rule for what is happening is as so:

If we have this wave:
"y = A sin ((2π * 6t) + ϕ)"
... then its samples will be the same as those from:
"y = A sin ((2π * −4t) + ϕ)"
... and Fourier series analysis will discover it as being:
"y = A sin ((2π * 4t) + (π − ϕ)"

Note how the frequency −4 cycles per second is 10 cycles per second less than 6 cycles per second. It is less by the sample rate. This is the "Faster Duplicates Ambiguity" relating to negative-frequency duplicates. The rule was explained towards the end of the "Faster Duplicates Ambiguity" section. It stated that, for the samples of the discrete wave that record:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... where "f_0" is any frequency from zero cycles per second up to just under the sample rate, there will be a negative-frequency-turned-positive-frequency wave with a frequency under the sample rate that has identical samples. It will have the formula:
"$y = h + A \sin ((2\pi * (f_s - f_0) * t) + \pi - \phi)$"

All of this means that our record of the 6-cycle-per-second wave will be recorded in exactly the same way as if it were a negative-frequency (−4-cycle-per-second) wave.

The over-half-the-sample-rate rule

We can create a formula that shows how a continuous wave with a frequency above half the sample rate and below the sample rate will be incorrectly recorded as a discrete version of a continuous wave with a frequency below half the sample rate. We will call this, "the over-half-the-sample-rate rule".

If we have this continuous wave:
"$y = h + A \sin ((2\pi * f_1 * t) + \phi)$"
... where "f_1" is a frequency over half the sample rate and under the sample rate, then its samples will be identical to the samples from (and Fourier series analysis will discover it as a wave that has the samples from):
"$y = h + A \sin ((2\pi * (f_s - f_1) * t) + \pi - \phi)$"

In other words, to calculate the positive frequency that it will become, we subtract the original frequency from the sample rate. Therefore, 6 cycles per second becomes 10 – 6 = 4 cycles per second. To calculate the new phase, we subtract the old phase from π radians. Therefore, 0.25π becomes π – 0.25π = 0.75π radians.

As an example of this formula in use, we will calculate in what way the continuous wave:
"$y = 3 \sin ((2\pi * 22t) + 0.1\pi)$"
... will end up being recorded for a sample rate of 30 samples per second. This will also be the formula that Fourier series analysis would discover when analysing the signal. We will fill in the formula:
"$y = h + A \sin ((2\pi * (f_s - f_1) * t) + \pi - \phi)$"
... as so:
"$y = 3 \sin ((2\pi * (30 - 22) * t) + \pi - 0.1\pi)$"
... which is:
"$y = 3 \sin ((2\pi * 8t) + 0.9\pi)$"

We can check this is correct by looking at the samples of:
"$y = 3 \sin ((2\pi * 22t) + 0.1\pi)$"
... and:
"$y = 3 \sin ((2\pi * 8t) + 0.9\pi)$"
... when sampled at 30 samples per second. They are both:
0.9271
−2.9344
−0.3136
3
−0.3136

−2.9344
0.9271
2.7406
−1.5
−2.4271
2.0074
... and so on.

The way that a 6-cycle-per-second wave is recorded as a 4-cycle-per-second wave is an example of "aliasing". If we had a signal made up of adding waves that were otherwise under half the sample rate, the presence of the 6-cycle-per-second wave would alter the signal – it would become a 4-cycle-per-second wave that is not supposed to be there. If we were given nothing but the samples and the sample rate, we would not be able to tell if that 4-cycle-per-second wave was supposed to be there or not. We can avoid aliasing by filtering a signal before it is sampled to remove any frequencies that are equal to, or above, half the sample rate.

5 cycles per second again

Before we look at 7 cycles per second, we will use what we have just learnt to re-examine 5-cycle-per-second waves. The "equal-to-half-the-sample-rate rule" from earlier was formulated by observing the behaviour of different waves. We can actually use the "over-half-the-sample-rate" rule to calculate exactly the same thing. The rule can be used for waves with frequencies equal to half the sample rate.

When we test for waves in Fourier series analysis, we test for increasing integer multiples of the fundamental frequency of the signal. For discrete signals, we find the positive-frequency waves under half the sample rate, and then, if we continue testing, we find the negative-frequency-made-positive waves over half the sample rate but under the sample rate. Then, as we continue to test, we find countless faster duplicates. Any frequency slower than half the sample rate will have a negative-frequency-made-positive duplicate faster than half the sample rate.

When it comes to the frequency equal to half the sample rate, we are actually finding the positive-frequency wave *and* the negative-frequency-made-positive duplicate, combined into one. They are inseparable.

We can demonstrate this by looking at the "over-half-the-sample-rate" rule that we used in the 6-cycles-per-second section:

> If we have this continuous wave:
> "$y = h + A \sin ((2\pi * f_1 * t) + \phi)$"
> ... where "f_1" is a frequency over half the sample rate and under the sample rate, then its samples will be identical to the samples from:
> "$y = h + A \sin ((2\pi * (f_s - f_1) * t) + \pi - \phi)$"

The rule also means that:

> If we have this continuous wave:
> "$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
> ... where "f_0" is a frequency *under* half the sample rate, then Fourier series analysis will find the negative-frequency-made-positive duplicate with the formula:
> "$y = h + A \sin ((2\pi * (f_s - f_0) * t) + \pi - \phi)$"

We can ignore the idea that the frequency of the first wave needs to be under half the sample rate, and say:

> If we have this continuous wave:
> "$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
> ... where "f_0" is any frequency, then Fourier series analysis will find a duplicate with the formula:
> "$y = h + A \sin ((2\pi * (f_s - f_0) * t) + \pi - \phi)$"

Given the new rule, for a sample rate of 10 samples per second, if we have this wave (or a signal containing this wave):
"$y = 1 \sin (2\pi * 5t)$"
... then we **should* also find the duplicate with this formula:
"$y = 1 \sin ((2\pi * (10 - 5) * t) + \pi - 0)$"
... which is:
"$y = 1 \sin ((2\pi * 5t) + \pi)$"

However, those two waves have the same frequency. Therefore, when we analyse a wave or signal, they will always be found added together. We can never find either wave on its own. We are really finding the 5-cycle-per-second wave and its negative-frequency-made-positive duplicate combined.

We would actually find:
“y = 1 sin (2π * 5t) + 1 sin ((2π * 5t) + π)”
... which is the same as:
“y = 0”

[The second wave is an upside down version of the first wave, so they add up to nothing.]

We will look at another example. If we have this wave:
“y = 1 sin ((2π * 5t) + 0.5π)”
... then we *should* find a duplicate with this formula:
“y = 1 sin ((2π * (10 – 5) * t) + π – 0.5π)”
... which is:
“y = 1 sin ((2π * 5t) + 0.5π)”

However, as before, analysis would find both waves added together as:
“y = 1 sin ((2π * 5t) + 0.5π) + 1 sin ((2π * 5t) + 0.5π)”
... which is:
“y = 2 sin ((2π * 5t) + 0.5π)”

Therefore, we find the correct wave, but with double the amplitude.

As another example, if we have this wave:
“y = 1 sin ((2π * 5t) + 0.25π)”
... then we *should* find the duplicate with this formula:
“y = 1 sin ((2π * (10 – 5) * t) + π – 0.25π)”
... which is:
“y = 1 sin ((2π * 5t) + 0.75π)”

However, analysis would find both waves added together as:
“y = 1 sin ((2π * 5t) + 0.25π) + 1 sin ((2π * 5t) + 0.75π)”
... which is:
“y = 1.4142 sin ((2π * 5t) + 0.5π)”

[We have to use some maths to calculate the result in this case.]

The three examples are consistent with the "equal-to-half-the-sample-rate" rule from earlier, which was:

> If we have this wave being sampled at "f_s" samples per second:
> "$y = A \sin ((2\pi * 0.5f_s * t) + \phi)$"
> ... then Fourier series analysis of the samples would find this wave:
> "$y = 2A \sin (\phi) * \sin ((2\pi * 0.5f_s * t) + 0.5\pi)$"
> ... where:
> - the amplitude of the calculated wave might be negative, in which case, rearranging the formula would result in a positive amplitude and a phase of 1.5π radians.

Generally, it is simpler to use the "equal-to-half-the-sample-rate" rule to calculate what would be found for a frequency equal to half the sample rate. However, the "over-half-the-sample-rate" rule explains why the "equal-to-half-the-sample-rate" rule works.

7 cycles per second

A pure wave with a frequency of 7 cycles per second will end up being recorded incorrectly because the frequency is higher than half the sample rate. The "over-half-the-sample-rate" rule applies:

> If we have this continuous wave:
> "$y = h + A \sin ((2\pi * f_1 * t) + \phi)$"
> ... where "f_1" is a frequency over half the sample rate and under the sample rate, then its samples will be identical to the samples from (and Fourier analysis will discover it as wave that has the samples from):
> "$y = h + A \sin ((2\pi * (f_s - f_1) * t) + \pi - \phi)$"

As an example, the wave "$y = 1 + 0.3 \sin ((2\pi * 7t) + 1.2\pi)$" will have the same samples as:
"$y = 1 + 0.3 \sin ((2\pi * (10 - 7) * t) + \pi - 1.2\pi)$"
... which is:
"$y = 1 + 0.3 \sin ((2\pi * 3t) - 0.2\pi)$"
... which is:
"$y = 1 + 0.3 \sin ((2\pi * 3t) + 1.8\pi)$"

The samples from both:
“y = 1 + 0.3 sin ((2π * 7t) + 1.2π)”
... and:
“y = 1 + 0.3 sin ((2π * 3t) + 1.8π)”
... for one second are:
0.8237
1.2853
1
0.7147
1.1763
1.1763
0.7147
1
1.2853
0.8237

Fourier series analysis on the samples would find the wave:
“y = 1 + 0.3 sin ((2π * 3t) + 1.8π)”
... which shows that the frequency of 7 cycles per second cannot be recorded correctly with a sample rate of 10 samples per second.

8 cycles per second

A pure wave with a frequency of 8 cycles per second will end up being recorded incorrectly. It will end up as a wave with a frequency of 2 cycles per second.

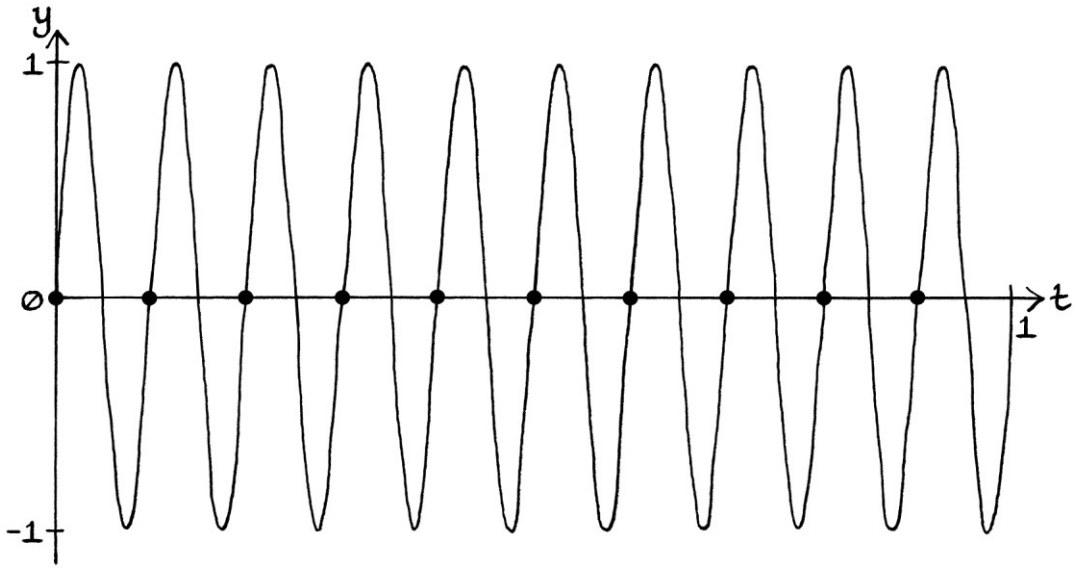
9 cycles per second

A pure wave with a frequency of 9 cycles per second will end up being recorded as a wave with a frequency of 1 cycle per second.

10 cycles per second

A pure wave with a frequency of 10 cycles per second will end up being recorded incorrectly. Every sample will be identical because exactly one complete cycle will occur between the points where the samples are taken. For a Sine wave with zero mean level and a phase of zero radians, every sample will be zero.

For example, the samples of "y = sin (2π * 10t)" drawn over the curve appear as so:



For a Sine wave with a zero mean level and a non-zero phase, every sample will be the amplitude multiplied by the Sine of that phase. In this way, a wave of 10 cycles per second will be identical to a wave with the same characteristics but a frequency of zero cycles per second. Given that, we can say that a wave with a frequency that is equal to the sample rate ends up as a wave with a zero frequency. [It is mapped to a frequency of zero cycles per second.] Therefore, the wave "y = sin (2π * 10t)" has the same samples as "y = sin (2π * 0t)".

This fits in with our frequency-mapping rule:
$f_0 = f_0 + (x * f_s)$
... where:

- "f_0" is a frequency from 0 up to just under the sample rate.
- "x" is a positive or negative integer.
- "f_s" is the sample rate in samples per second.

For a frequency of 10 samples per second, any wave will have samples that are identical to the same wave with a frequency any integer multiple of 10 higher or lower.

A wave with a frequency of 10 cycles per second will be recorded incorrectly because its frequency is over half the sample rate. In addition, it will be recorded as a wave with a frequency 10 cycles per second lower – it will be recorded as a wave with a frequency of 0 cycles per second. Therefore, it will become a mean level.

Other frequencies

A pure wave with a frequency of 11 cycles per second will end up being recorded incorrectly. For example, the wave:
“y = sin (2π * 11t)”
... will end up being recorded as:
“y = sin (2π * 1t)”

A pure wave with a frequency of 19 cycles per second will end up being recorded incorrectly in two different ways. First, it will end up with the same samples as a wave with a frequency of 9 cycles per second. Then, that 9 cycles-per-second wave will end up being recorded as a wave with a frequency of −1 cycles per second, and that will be found as a wave with a positive frequency of +1 cycles per second and a possibly different phase.

Summary

There are several ideas in play in the above examples:

- For a given sample rate, we can only correctly record waves that have frequencies that are under half the sample rate. We can phrase this the other way around: for a wave with a particular frequency, we must sample it with a sample rate over twice that frequency.

- For a given sample rate, any frequency that is less than *half* that sample rate will end up being analysed correctly.

- For a given sample rate, any frequency that is *equal* to *half* that sample rate will end up being sampled incorrectly – if its samples are not all zero, they will have the same absolute value, but will be alternately positive and negative, or negative and positive. Knowing this is useful for recognising patterns in samples.

- When analysing a signal that contains a wave with a frequency equal to half the sample rate, if that wave can be detected at all, it will be found as having an amplitude equal to twice its actual amplitude, and the phase will probably be incorrect because it will always be either 0.5π radians or 1.5π radians. However, if we halve the amplitude, the found wave will have identical samples to the actual wave that was part of the original signal.

- For a given sample rate, any wave with a frequency that is under the sample rate but above *half* the sample rate, will be sampled incorrectly. [The "over-half-the-sample-rate rule".] It will be end up as a wave with a frequency below half the sample rate. We can think of it as being recorded with a frequency lower by the sample rate (which means that it will have a negative frequency), or we can think of it as being recorded with a positive frequency below half the sample rate with a different phase.

- For a given sample rate, any frequency that is equal to, or above, that sample rate, will become mapped to a frequency from 0 to just under that sample rate. [The same is true for negative frequencies, but generally, we would not be trying to sample negative frequencies, but instead their positive-frequency equivalents.]

- For a given sample rate, any frequency that is equal to that sample rate will end up with samples of the same value for all time. It will end up as a zero-frequency wave, and so will act as a mean level.

Other books usually express "half the sample rate" as the "Nyquist sampling rate" or the "Nyquist rate", and they usually express the frequency equal to half the sample rate as the "Nyquist frequency".

A sum of waves example

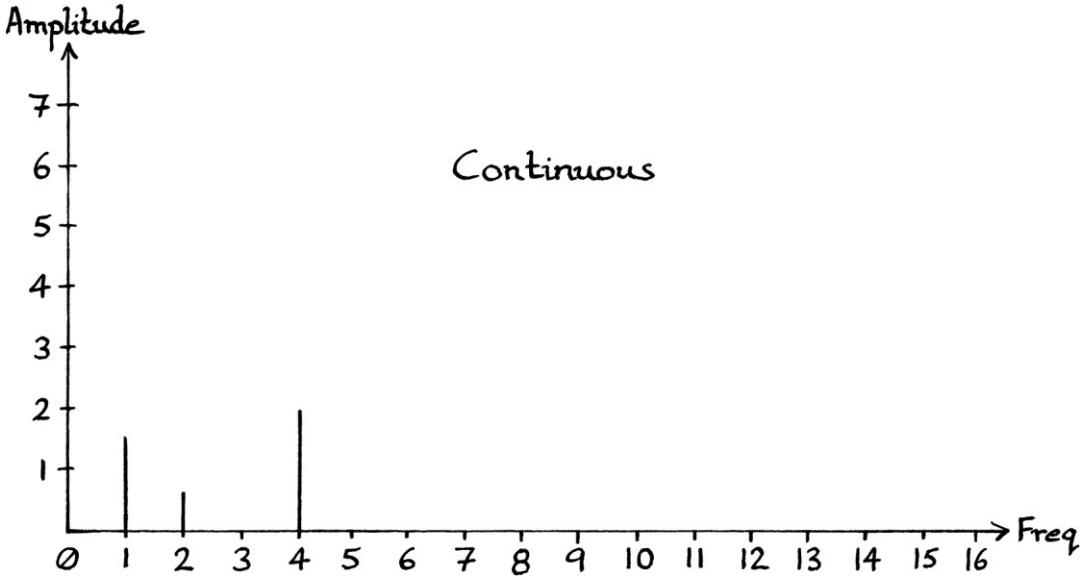
We will now go through an example signal to which we will keep adding waves to see what happens.

Initial signal

We will imagine that we have a continuous signal made up of the following waves:

"$y = 1.5 \sin (2\pi * 1t)$"
"$y = 0.6 \sin (2\pi * 2t)$"
"$y = 2 \sin (2\pi * 4t)$"

In a frequency domain graph, these *continuous* waves appear as so:



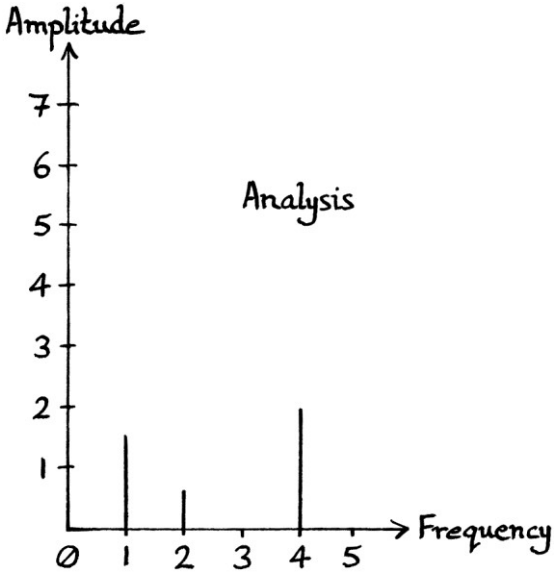
We will sample our signal at 10 samples per second. The samples for one second will be:

0
2.6279
−0.1229
2.9760
−0.8645
0
0.8645
−2.9760
0.1229
−2.6279

From what we know about sampling, each wave will be recorded correctly because each frequency is less than half the sample rate. If we analysed the discrete signal made up of these samples using Fourier series analysis (and stopping after we had tested for the frequency equal to half the sample rate), we would find it to be made up of these waves:

“y = 1.5 sin (2π * 1t)”
“y = 0.6 sin (2π * 2t)”
“y = 2 sin (2π * 4t)”
... which shows that they were all recorded correctly.

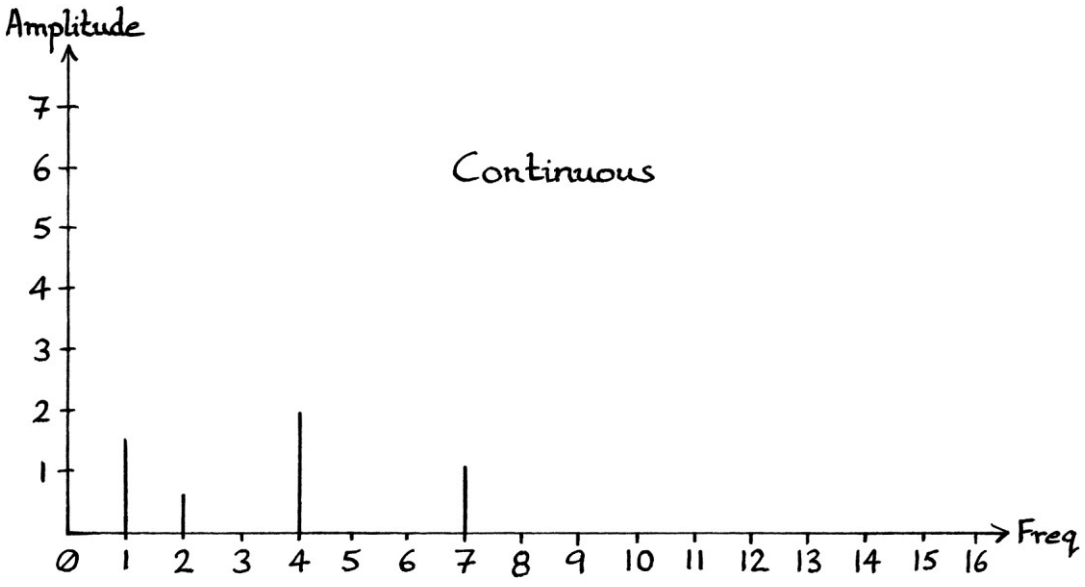
The found waves appear as so in a frequency domain graph:



7-cycle-per-second wave

We will now add a 7-cycle-per-second wave to our *continuous* signal. Our continuous signal will be made up of the following waves:
"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"
"y = 1.1 sin (2π * 7t)"

In a frequency domain graph, these continuous waves appear as so:



If we sample our signal at 10 samples per second, the samples for our new signal over one second will be:

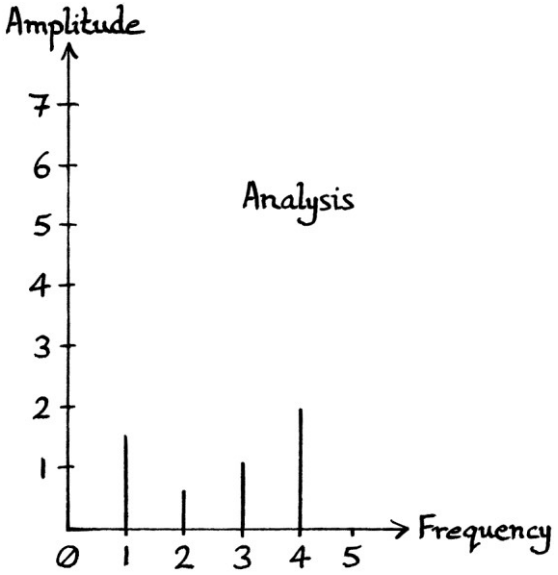
0
1.5817
0.5237
3.6226
−1.9107
0
1.9107
−3.6226
−0.5237
−1.5817

As the frequency of "y = 1.1 sin (2π * 7t)" is above half the sample rate, when it is sampled, it will end up as the positive-frequency equivalent of:
"y = 1.1 sin (2π * −3t)"
... which is the wave:
"y = 1.1 sin ((2π * 3t) + π)"

This means that our sampled signal will really be the discrete equivalent of this sum of continuous waves:
"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"
"y = 1.1 sin ((2π * 3t) + π)"
... which we can put in order of frequency as:
"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 1.1 sin ((2π * 3t) + π)"
"y = 2 sin (2π * 4t)"

These four waves are what we would find if we analysed our discrete signal using Fourier series analysis, and stopped after testing for the frequency equal to half the sample rate. As we can see, our discrete signal does not represent the four continuous waves that were added together to make up our signal.

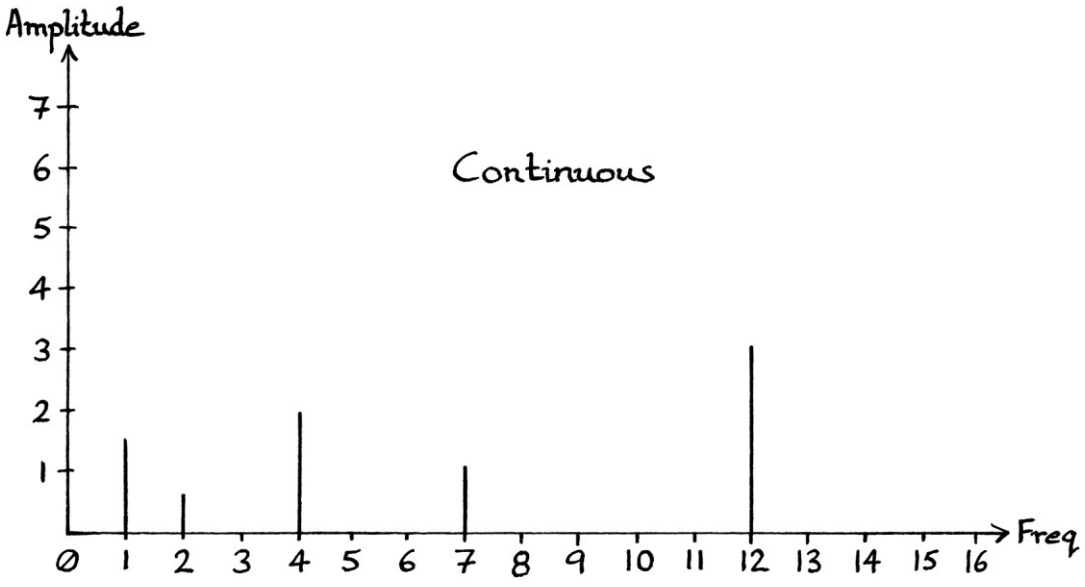
In the frequency domain, these discovered constituent waves would look like this:



12-cycle-per-second wave

We will now add "y = 3.1 sin (2π * 12t)" to our *continuous* signal. The sum is:
"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"
"y = 1.1 sin (2π * 7t)"
"y = 3.1 sin (2π * 12t)"

In a frequency domain graph, these continuous waves appear as so:



If we sample our signal at 10 samples per second, the samples for our new signal over one second will be:

0
4.5300
2.3458
1.8005
−4.8590
0
4.8590
−1.8005
−2.3458
−4.5300

We already know that:
“y = 1.1 sin (2π * 7t)”
... will be recorded as:
“y = 1.1 sin ((2π * 3t) + π)”

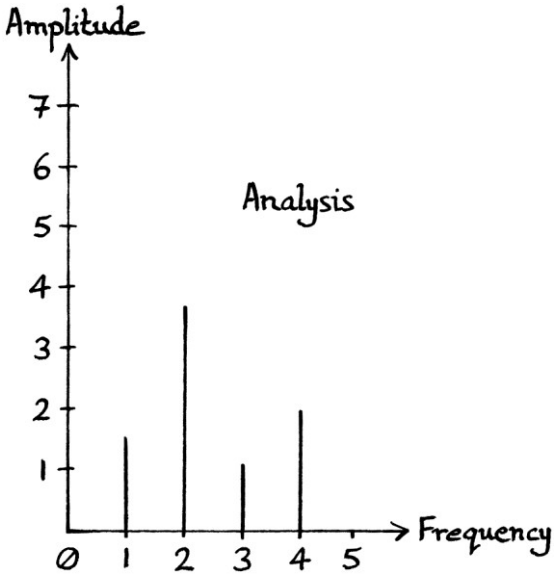
We now have the wave “y = 3.1 sin (2π * 12t)”, which has a frequency over the sample rate. This means that it will be recorded with the same samples as the wave with a frequency 10 cycles per second lower (because the sample rate is 10 samples per second). It will become:
“y = 3.1 sin (2π * 2t)”

Taking into account how we already corrected “y = 1.1 sin (2π * 7t)”, our discrete signal will actually be a sampled version of these five continuous waves:
“y = 1.5 sin (2π * 1t)”
“y = 0.6 sin (2π * 2t)”
“y = 1.1 sin ((2π * 3t) + π)”
“y = 2 sin (2π * 4t)”
“y = 3.1 sin (2π * 2t)”

As this sum of waves has two waves with frequencies of 2 cycles per second, those waves become added together in the sum. We end up with just four waves:
“y = 1.5 sin (2π * 1t)”
“y = 3.7 sin (2π * 2t)”
“y = 1.1 sin ((2π * 3t) + π)”
“y = 2 sin (2π * 4t)”

These four waves are the ones that we would discover if we analysed the samples with Fourier series analysis and stopped just after testing for the frequency equal to half the sample rate.

The four waves appear as so in a frequency domain graph:

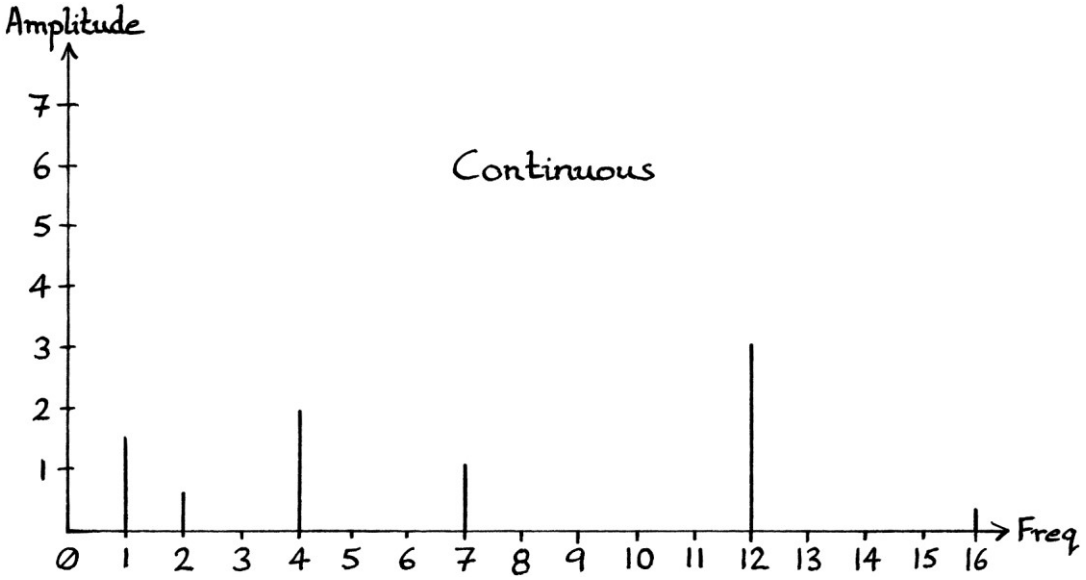


16-cycle-per-second wave

We will now add another wave to our continuous signal – one with a frequency of 16 cycles per second. Our continuous signal will be made up of the following waves:

“y = 1.5 sin (2π * 1t)”
“y = 0.6 sin (2π * 2t)”
“y = 2 sin (2π * 4t)”
“y = 1.1 sin (2π * 7t)”
“y = 3.1 sin (2π * 12t)”
“y = 0.3 sin (2π * 16t)”

These waves appear as so in the frequency domain graph:



If we sample our signal at 10 samples per second, the samples for our new signal over one second will be:

0
4.3537
2.6312
1.5151
−4.6826
0
4.6826
−1.5151
−2.6312
−4.3537

We already know that:
“y = 1.1 sin (2π * 7t)”
... will be recorded as:
“y = 1.1 sin ((2π * 3t) + π)”

We also already know that:
“y = 3.1 sin (2π * 12t)”
... will be recorded as:
“y = 3.1 sin (2π * 2t)”
... and will be added to the existing 2-cycle-per-second wave in the sum.

The last wave in our list is:
“y = 0.3 sin (2π * 16t)”
This has a frequency above the sample rate. Therefore, it will be recorded as the same wave with a frequency lower by the sample rate. It will become:
“y = 0.3 sin (2π * 6t)”
However, *this* wave is above *half* the sample rate. Therefore, it will be recorded as the same wave with a frequency lower by the sample rate again. It will become:
“y = 0.3 sin (2π * −4t)”
... which can be rephrased to have a positive frequency as:
“y = 0.3 sin ((2π * 4t) + π)”

If we take into account all of our changed waves, our discrete signal will be the sampled version of these continuous waves:
“y = 1.5 sin (2π * 1t)”
“y = 3.7 sin (2π * 2t)”
“y = 1.1 sin ((2π * 3t) + π)”
“y = 2 sin (2π * 4t)”
“y = 0.3 sin ((2π * 4t) + π)”

As we have two waves with frequencies of 4 cycles per second, those waves become added together within the sum of waves. Adding:
“y = 2 sin (2π * 4t)”
... to:
“y = 0.3 sin ((2π * 4t) + π)”
... is easiest if we rephrase:
“y = 0.3 sin ((2π * 4t) + π)”
... to be a negative amplitude wave with zero phase:
“y = −0.3 sin (2π * 4t)”

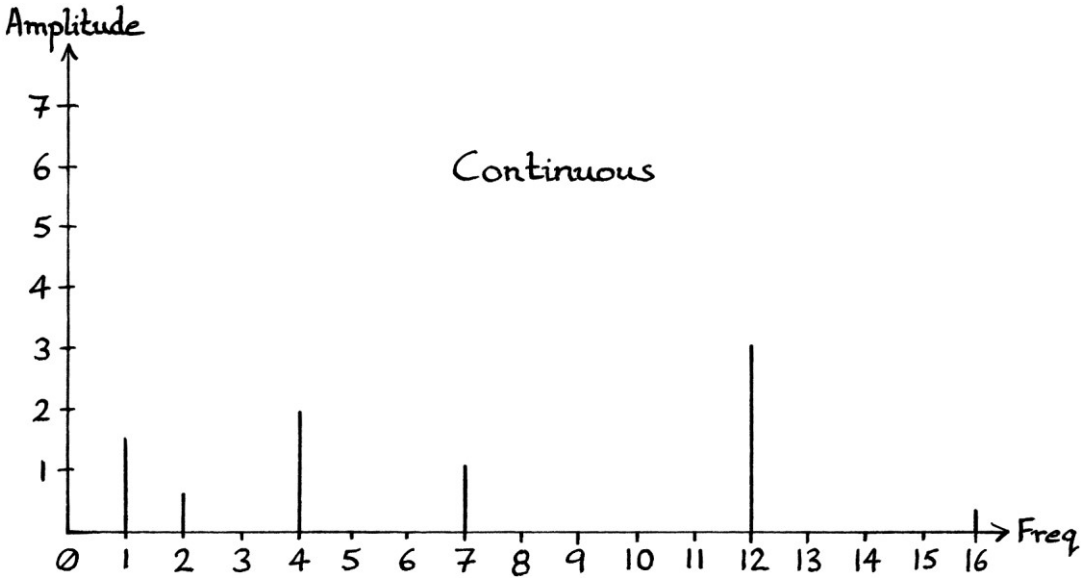
We then just add “y = 2 sin (2π * 4t)” to “y = −0.3 sin (2π * 4t)” to obtain:
“y = 1.7 sin (2π * 4t)”

[If we could not simplify the addition to two waves with the same phase, we would have had to imagine the phase points on the two circles from which the waves could have been derived, and arrange the circles for adding. This was explained in Chapter 14.]

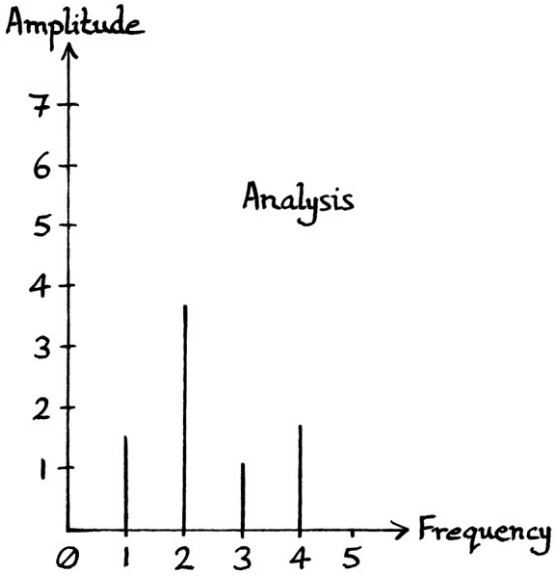
We now have four waves in our list of waves that we would discover by analysing the discrete signal:
"y = 1.5 sin (2π * 1t)"
"y = 3.7 sin (2π * 2t)"
"y = 1.1 sin ((2π * 3t) + π)"
"y = 1.7 sin (2π * 4t)"

If we analysed the samples with Fourier series analysis and stopped just after the test frequency equal to half the sample rate, we would find the above four waves.

The *continuous* signal consists of six different waves:



However, the discrete signal consists of only these four waves:



Note how they all have frequencies that are below half the sample rate.

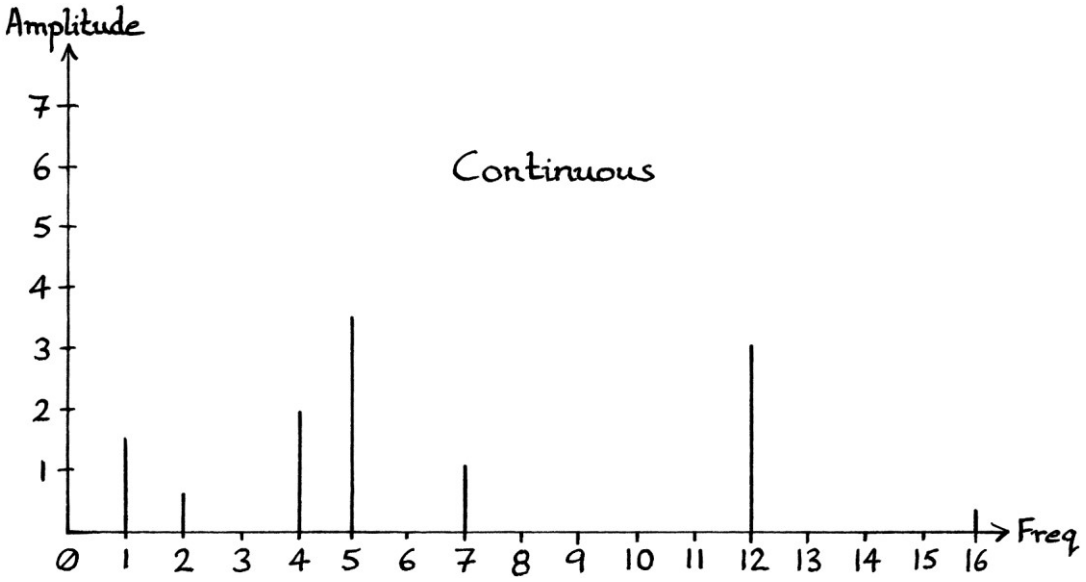
Our discrete signal is not a representation of the continuous signal, in the sense that we would not be able to recover the original continuous signal from analysing the discrete signal. Two of the constituent waves of the continuous signal were too fast to be recorded correctly, and they ended up as aliased waves in our discrete signal. The four-wave signal that we found has the same samples as a discrete version of the original continuous signal, but the continuous signal cannot be recovered from those samples.

5-cycle-per-second wave

We will add the wave "y = 3.5 sin ((2π * 5t) + π)" to our signal.

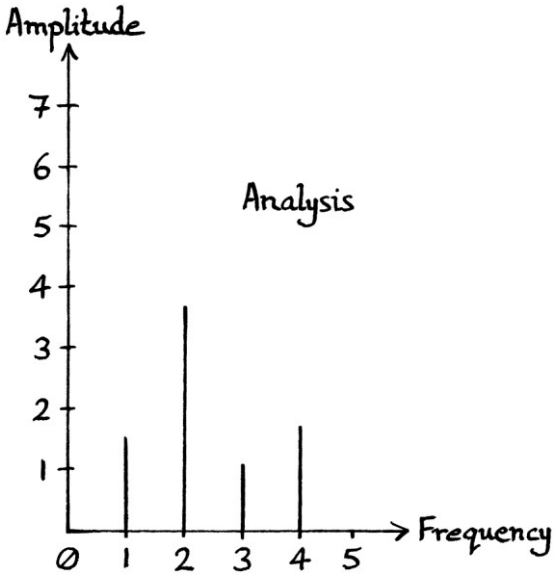
We already know that a continuous Sine wave with a frequency equal to half the sample rate and a phase of 0 or π radians will produce samples consisting entirely of zeroes. Therefore, our new wave will not exist in the discrete signal.

Our *continuous* signal is now the sum of these waves:
"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"
"y = 3.5 sin ((2π * 5t) + π)" [This is the new wave]
"y = 1.1 sin (2π * 7t)"
"y = 3.1 sin (2π * 12t)"
"y = 0.3 sin (2π * 16t)"



If we analysed the samples from the discrete version of the signal (sampled at 10 samples per second), and we stopped after testing for the frequency equal to half the sample rate, we would only find these 4 waves:

"y = 1.5 sin (2π * 1t)"
"y = 3.7 sin (2π * 2t)"
"y = 1.1 sin ((2π * 3t) + π)"
"y = 1.7 sin (2π * 4t)"



Another 5-cycle-per-second wave

We will add another 5-cycle-per-second wave to our continuous signal, but this time, one with a non-zero phase. We will add:
"y = 7.7 sin ((2π * 5t) + 0.34π)"

Our continuous signal is now the sum of these waves:

"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"
"y = 3.5 sin ((2π * 5t) + π)"
"y = 7.7 sin ((2π * 5t) + 0.34π)" [This is the new wave]
"y = 1.1 sin (2π * 7t)"
"y = 3.1 sin (2π * 12t)"
"y = 0.3 sin (2π * 16t)"

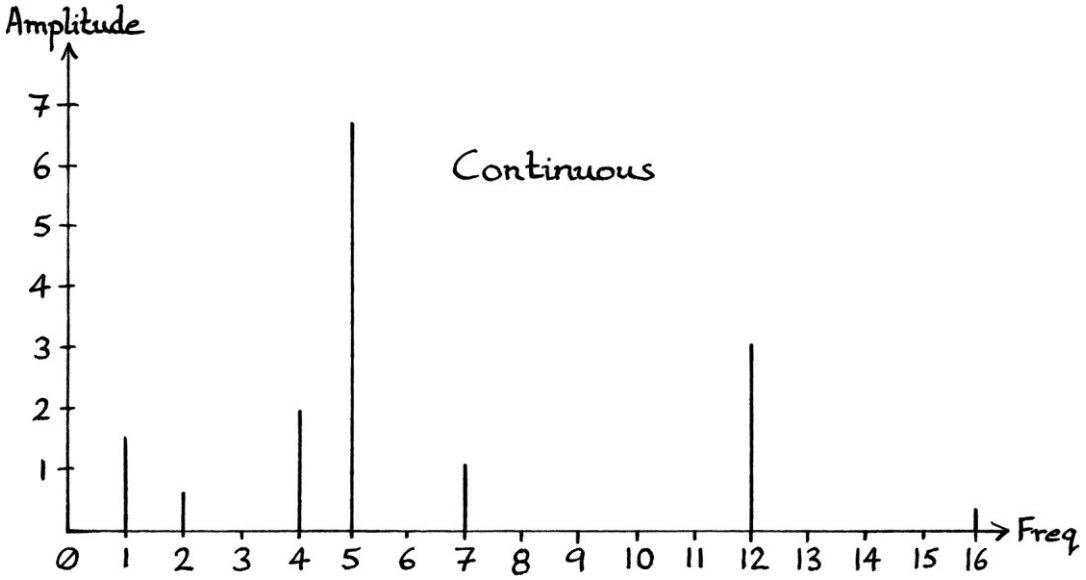
As we have two continuous waves with the same frequency, they become added together in the continuous signal. As a reminder of how we do this, we imagine the circles from which the two waves were, or could have been, derived at t = 0, and arrange the circles for adding. The angle of the phase point of the outer circle with respect to the centre of the axes will be the phase of the resulting wave. Its distance from the centre of the axes will be the amplitude. [This was explained in Chapter 14.] The phase point will have an x-axis position of: 3.5 cos (π) + 7.7 cos (0.34π) = 0.2095; its y-axis position will be: 3.5 sin (π) + 7.7 sin (0.34π) = 6.7476. We use Pythagoras's theorem to calculate its distance from the origin of the axes. The distance will be 6.7508 units. We use arctan to calculate the angle. It will be 1.5398 radians. The two waves when added together become:
"y = 6.7508 sin ((2π * 5t) + 1.5398)"

[Note that it is only the discrete version of the 5-cycle-per-second Sine wave with a phase of π radians that, when sampled at 10 samples per second, ends up as zero samples, and so ceases to exist. The continuous version of the wave has a range of values and always exists. Therefore, the result of adding the two 5-cycle-per-second *continuous* waves is different from the result of adding the two *discrete* versions of those waves, when sampled at 10 samples per second. Depending on whether we add the waves as continuous waves or as discrete waves, the result of the addition will be different.]

Our continuous signal is now the sum of these waves:

"y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"
"y = 6.7508 sin ((2π * 5t) + 1.5398)"
"y = 1.1 sin (2π * 7t)"
"y = 3.1 sin (2π * 12t)"
"y = 0.3 sin (2π * 16t)"

The frequency domain graph is as so:

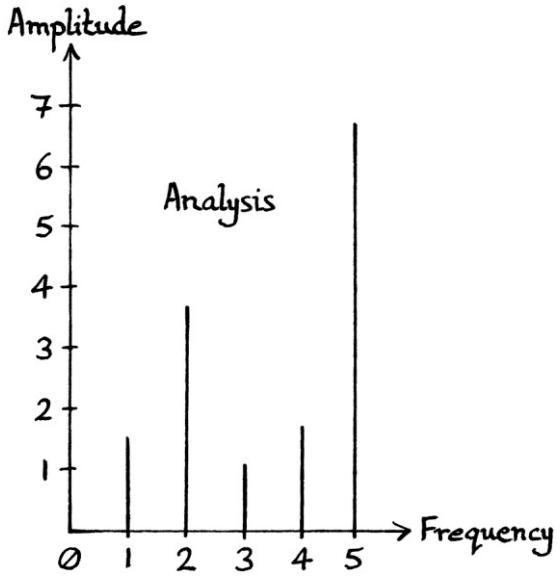


Our 5-cycle-per-second wave has a phase other than 0 or π radians, and its phase is between 0 and π radians. Therefore, its samples will have the same absolute value but will be alternately positive then negative. The samples for the wave are actually: 6.7476, −6.7476, 6.7476, −6.7476, 6.7476, −6.7476 and so on. We also know that if we analysed this wave with Fourier series analysis, we would find a wave with twice the amplitude that it should have, and a phase of 0.5π radians. We would actually find this wave:
"y = 13.4951 sin ((2π * 5t) + 0.5π)"
If we halved the amplitude [13.4951 ÷ 2 = 6.74755], we would have a wave with exactly the same samples as the original wave:
"y = 6.7476 sin ((2π * 5t) + 0.5π)"
The halved-amplitude wave's samples are: 6.7476, −6.7476, 6.7476, −6.7476 and so on.

If we analysed the samples from the discrete version of the entire signal (sampled at 10 samples per second), we would need to make sure that when we tested for the frequency equal to half the sample rate, we halved the amplitude of the discovered wave. The resulting wave might not be the exact wave that was in the signal, but it would produce the same samples (for a sample rate of 10 samples per second).

If we remembered to halve the amplitude of the discovered wave that had a frequency equal to half the sample rate, then the list of discovered waves would be:

"y = 1.5 sin (2π * 1t)"
"y = 3.7 sin (2π * 2t)"
"y = 1.1 sin ((2π * 3t) + π)"
"y = 1.7 sin (2π * 4t)"
"y = 6.7476 sin ((2π * 5t) + 0.5π)"

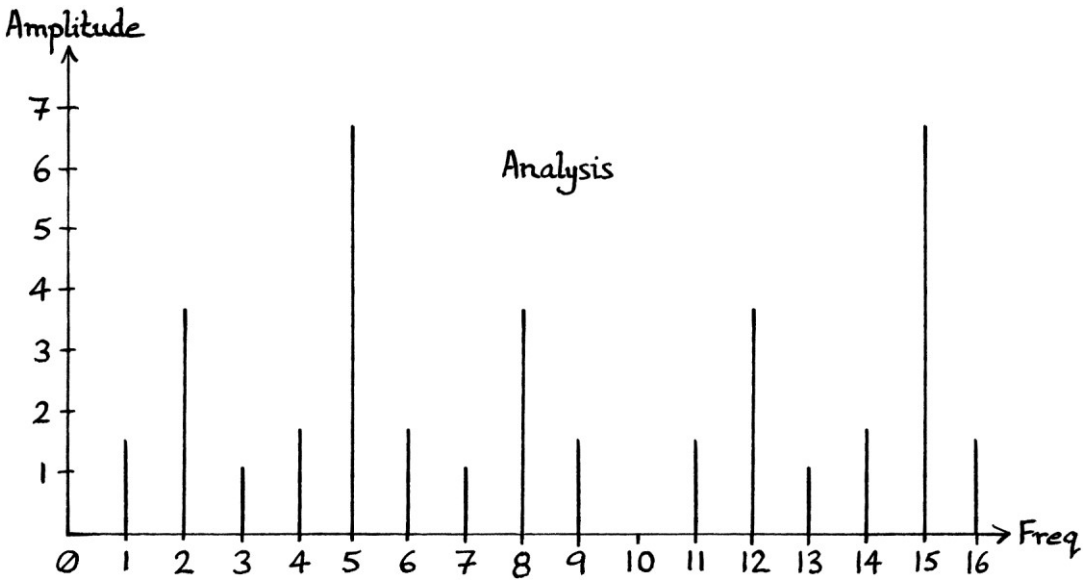


The signal created by adding these continuous waves would have exactly the same samples as our sum of 7 different continuous waves (for a sample rate of 10 samples per second). They would not be the *same* waves, but the sum would still have the same samples.

Unnecessary analysis

If we were to analyse the current discrete signal, and we continued to test for waves with frequencies higher than the frequency equal to half the sample rate, we would find duplicates of the waves that we had already found. First, we would find the negative-frequency-made-positive duplicates, and then we would find the positive-frequency duplicates. [We would have to make sure we halved the discovered amplitude for the frequency equal to half the sample rate, and also for frequencies that were integer multiples of the sample rate higher than that.]

The frequency domain graph showing the analysis would look like this for frequencies up to and including 16 cycles per second:



We can know which waves are duplicates by how they have frequencies higher than half the sample rate. Therefore, in this case, we can ignore all the waves with frequencies higher than 5 cycles per second.

[As always, remember that we cannot see the phases on a frequency domain graph such as this, so the graph is missing useful information.]

Summary

Our final signal consists of a sum of seven different waves. How those waves (sampled at 10 samples per second) would appear in the results of Fourier series analysis are as follows:

Original continuous wave	**What Fourier series analysis would find**
"y = 1.5 sin (2π * 1t)"	Found correctly as: "y = 1.5 sin (2π * 1t)"
"y = 0.6 sin (2π * 2t)"	Combined with the 12 cps wave to become: "y = 3.7 sin (2π * 2t)"
"y = 2 sin (2π * 4t)"	Combined with the 16 cps wave to become: "y = 1.7 sin (2π * 4t)"
"y = 6.7508 sin ((2π * 5t) + 1.5398)"	Found as a different wave with the same samples: "y = 6.7476 sin ((2π * 5t) + 0.5π)" [as long as we halved the discovered amplitude]
"y = 1.1 sin (2π * 7t)"	Becomes a 3 cps wave with the formula: "y = 1.1 sin ((2π * 3t) + π)"
"y = 3.1 sin (2π * 12t)"	Becomes a 2 cps wave with the formula: "y = 3.1 sin (2π * 2t)", which becomes added to the other 2 cps wave.
"y = 0.3 sin (2π * 16t)"	Becomes a 4 cps wave with the formula: "y = 0.3 sin ((2π * 4t) + π)", which is added to the other 4 cps wave.

Formulas so far

As a reminder, we will look again at some of the formulas we have created so far:

Duplicate frequencies

The rule for which waves have the same samples is as so:

If we have this continuous wave sampled at "f_s" samples per second:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... then it will have identical samples to this continuous wave sampled at the same sample rate:
"$y = h + A \sin ((2\pi * (f_0 + (x * f_s)) * t) + \phi)$"
... where:

- "h" is the mean level.
- "A" is the amplitude.
- "f_0" is the frequency of the original wave.
- "x" is any integer, whether positive or negative.
- "f_s" is the sample rate.
- "t" is the time in seconds.
- "ϕ" is the phase in radians.

In other words, for the sample rate of "f_s" and a frequency of "f_0":
$f_0 = f_0 + (x * f_s)$
... where:

- "x" is any positive or negative integer.

Positive duplicate frequencies

The "$f_0 = f_0 + (x * f_s)$" rule can be rewritten so that it applies only to positive-frequency discrete waves, or in other words, positive frequencies and negative frequencies made positive. The rule is as so:

Any discrete record of this continuous wave:
"$y = h + A \sin ((2\pi * f_0 * t) + \phi)$"
... will have identical samples to both of these two continuous wave formulas:
"$y = h + A \sin ((2\pi * (f_0 + (x * f_s)) * t) + \phi)$"
"$y = h + A \sin ((2\pi * (f_s - f_0 + (x * f_s)) * t) + \pi - \phi)$"
... as long as "x" is any positive integer.

Frequencies above half the sample rate

The rule for how waves with frequencies above half the sample rate, but below the sample rate, will be recorded as waves with frequencies between 0 and half the sample rate is as so:

If we have this continuous wave:
"$y = h + A \sin ((2\pi * f_1 * t) + \phi)$"
... where "f_1" is a frequency over half the sample rate and under the sample rate, then its samples will be identical to the samples from (and Fourier analysis will discover it as a wave that has the samples from):
"$y = h + A \sin ((2\pi * (f_s - f_1) * t) + \pi - \phi)$"

[This is the "over-half-the-sample-rate rule".]

Frequencies equal to half the sample rate

The rule for how Fourier series analysis will misinterpret the samples from a wave that has a frequency equal to half the sample rate is as so:

If we have this wave being sampled at "f_s" samples per second:
"$y = A \sin ((2\pi * 0.5f_s * t) + \phi)$"
... then Fourier series analysis of the samples would find this wave:
"$y = 2A \sin (\phi) * \sin ((2\pi * 0.5f_s * t) + 0.5\pi)$"
... where:

- the amplitude of the calculated wave might be negative, in which case, rearranging the formula would result in a positive amplitude and a phase of 1.5π radians.

When testing for a frequency equal to half the sample rate, we would find a wave that had the correct samples if we halved the amplitude of the wave that we found. This would not necessarily be the actual original wave, but it would have the same samples as the original wave. The wave that had the correct samples would have the formula:
"$y = A \sin (\phi) * \sin ((2\pi * 0.5f_s * t) + 0.5\pi)$"
["A" has been divided by 2.]

Non-integer frequencies

So far, the frequencies in all of our examples have been integers per second, and all of our sample rates have been integer samples per second. This does not have to be the case, but it can complicate things slightly to deal with non-integers.

We have seen how waves with integer frequencies per second have been sampled for a sample rate of 10 samples per second. We will look at some non-integer examples for a sample rate of 10 samples per second.

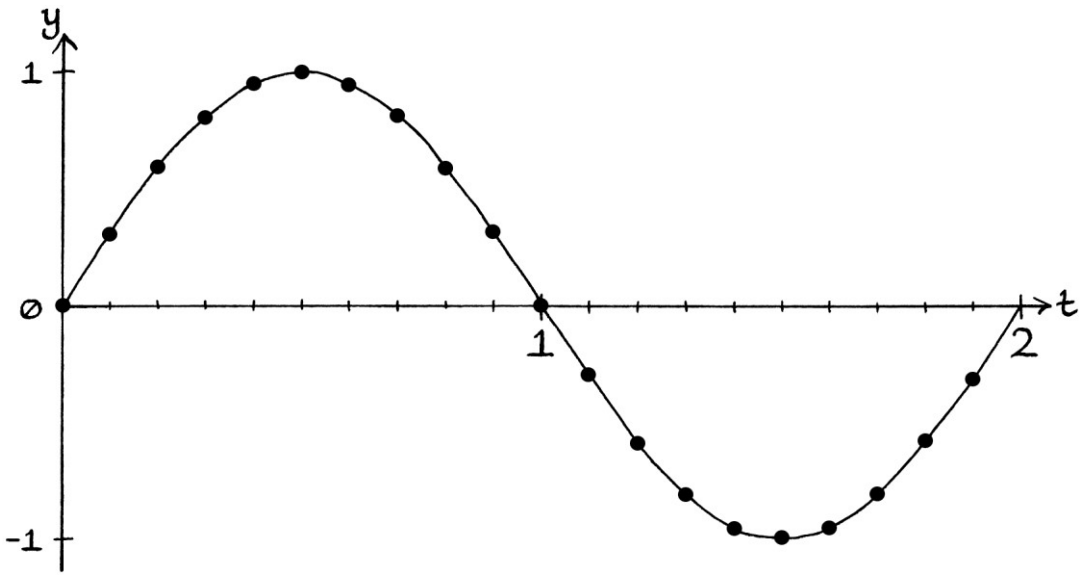
0.5 cycles per second

If we have the wave "y = sin (2π * 0.5t)", it will take 1 ÷ 0.5 = 2 seconds to complete one cycle. At 10 samples per second, the samples for one cycle will be:

Time (seconds)	**Sample value**
0.0	0
0.1	0.3090
0.2	0.5878
0.3	0.8090
0.4	0.9511
0.5	1
0.6	0.9511
0.7	0.8090
0.8	0.5878
0.9	0.3090
1.0	0
1.1	−0.309017
1.2	−0.587785
1.3	−0.809017
1.4	−0.951057
1.5	−1.000000
1.6	−0.951057
1.7	−0.809017
1.8	−0.587785
1.9	−0.309017

The next sample would be:
2.0 0

The graph of the samples drawn over the wave curve is as so:



The sample rate is more than enough to record this frequency. If we analysed the discrete signal, we would recover the original wave.

The "$f_0 = f_0 + (x * f_s)$" rule still works for frequencies that are not integers. Therefore, for a sample rate of 10 samples per second, the samples of the continuous wave:
"y = sin (2π * 0.5t)"
... are also the samples of the (undersampled) continuous waves:
"y = sin (2π * 10.5t)"
"y = sin (2π * 20.5t)"
"y = sin (2π * 30.5t)"
... and so on, and also:
"y = sin (2π * −9.5t)"
"y = sin (2π * −19.5t)"
"y = sin (2π * −29.5t)"
... and so on.

0.1 cycles per second

If we have the wave "y = sin (2π * 0.1t)", it will take 1 ÷ 0.5 = 10 seconds to complete one cycle. As the sample rate is 10 samples per second and it takes 10 seconds to complete 1 cycle, we will have 100 samples for one cycle of the wave. Therefore, the sample rate is more than adequate to record our wave.

1.25 cycles per second

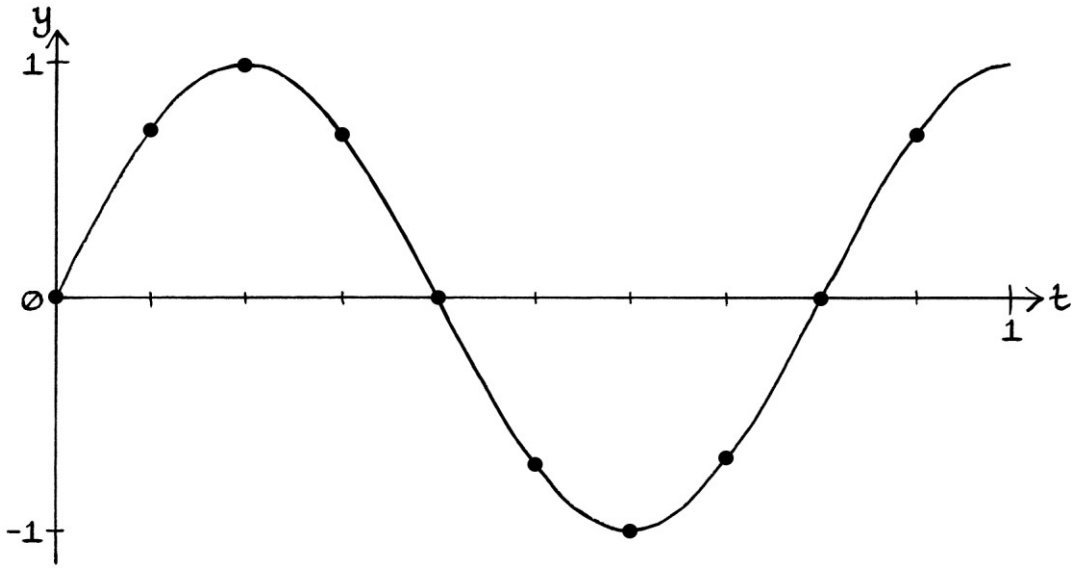
If we have the continuous wave "y = sin (2π * 1.25t)", it will take 1 ÷ 1.25 = 0.8 seconds to complete one cycle. At 10 samples per second, the samples for one cycle will be as in this table:

Time (seconds)	**Sample value**
0	0
0.1	0.7071
0.2	1
0.3	0.7071
0.4	0
0.5	−0.7071
0.6	−1
0.7	−0.7071

The next sample will be the start of the next cycle:

0.8	0

The samples drawn over the curve of the continuous wave for one second are as so:



If we analyse the signal using Fourier series analysis (with test frequencies that are integer multiples of 1.25 cycles per second), we would find the wave correctly. Therefore, we can say that the sample rate is sufficient for this frequency.

It is still the case that the "$f_0 = f_0 + (x * f_s)$" rule applies. For a sample rate of 10 samples per second, the following waves have the same samples:
"$y = \sin (2\pi * 1.25t)$"
"$y = \sin (2\pi * 11.25t)$"
"$y = \sin (2\pi * 21.25t)$"
"$y = \sin (2\pi * 31.25t)$"
"$y = \sin (2\pi * 41.25t)$"
... and so on, as do these waves:
"$y = \sin (2\pi * -8.75t)$", which is also: "$y = \sin ((2\pi * 8.75t) + \pi)$"
"$y = \sin (2\pi * -18.75t)$", which is also: "$y = \sin ((2\pi * 18.75t) + \pi)$"
"$y = \sin (2\pi * -28.75t)$", which is also: "$y = \sin ((2\pi * 28.75t) + \pi)$"
"$y = \sin (2\pi * -38.75t)$", which is also: "$y = \sin ((2\pi * 38.75t) + \pi)$"
"$y = \sin (2\pi * -48.75t)$", which is also: "$y = \sin ((2\pi * 48.75t) + \pi)$"
... and so on.

1.2 cycles per second

We will now learn that not all non-integer frequencies are as straightforward to record. We will look at a wave with a frequency of 1.2 cycles per second:
"y = sin (2π * 1.2t)"

This wave takes 1 ÷ 1.2 = 0.8333 seconds [to 4 decimal places] to complete one cycle. Therefore, its period is 0.8333 seconds [to 4 decimal places]. The significant aspect of this period is that it cannot be expressed with a finite number of decimal digits. This becomes relevant when we are using a sample rate of 10 samples per second.

At 10 samples per second, the samples taken from one cycle of the continuous wave will be:

Time (seconds)	**Sample value**
0.0	0
0.1	0.6845
0.2	0.9980
0.3	0.7705
0.4	0.1253
0.5	−0.5878
0.6	−0.9823
0.7	−0.8443
0.8	−0.2487
0.9	0.4818

The point where a full cycle is completed is between the samples taken from 0.8 seconds and 0.9 seconds. Therefore, although the continuous wave repeats once every 0.8333 seconds, the discrete samples *do not* repeat once every 0.8333 seconds.

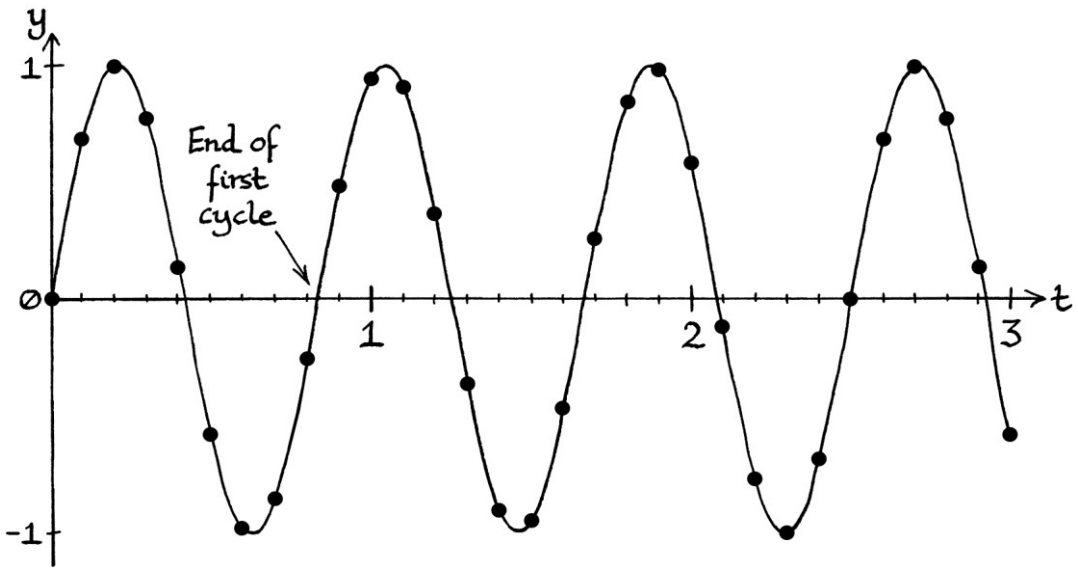
We will look at more samples to see where they *do* repeat:

1.0	0.9511
1.1	0.9048
1.2	0.3681
1.3	−0.3681
1.4	−0.9048

1.5	−0.9511	
1.6	−0.4818	
1.7	0.2487	
1.8	0.8443	
1.9	0.9823	
2.0	0.5878	
2.1	−0.1253	
2.2	−0.7705	
2.3	−0.9980	
2.4	−0.6845	[This is the last sample of the first discrete cycle.]
2.5	0	[The samples repeat here.]
2.6	0.6845	
2.7	0.9980	
2.8	0.7705	
2.9	0.1253	
3.0	−0.5878	

It turns out that the samples repeat at 2.5 seconds. Therefore, our continuous wave repeats after 0.8333 seconds, but our discrete wave repeats after 2.5 seconds. Our discrete wave has a frequency of 0.4 cycles per second.

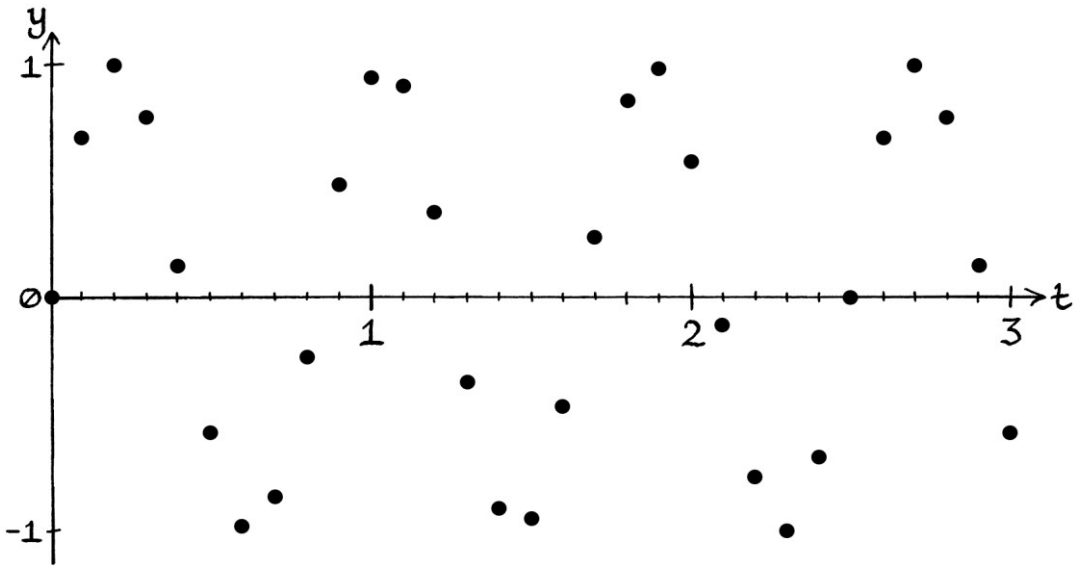
We can see what is happening when we superimpose the samples over the curve of the continuous wave:



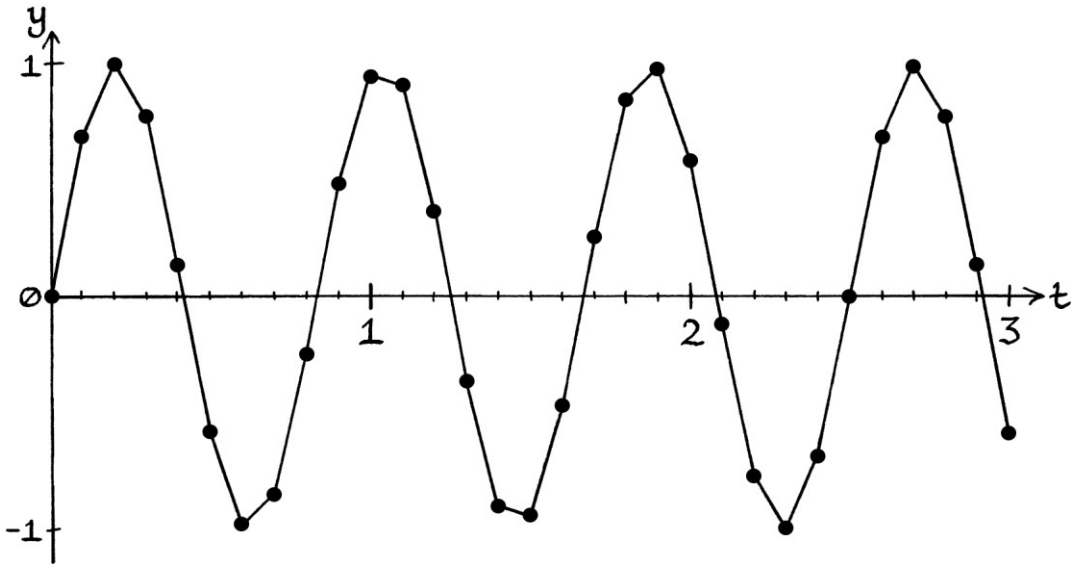
No cycles of the continuous wave end exactly at a place where a sample is taken until 3 cycles have occurred (at 2.5 seconds). In the previous example, we had a wave with a frequency of 1.25 cycles per second. In that case, each cycle started and finished exactly at one of the sample reading points. Therefore, the samples

recorded the start and finish point of each cycle exactly. In this example, with a frequency of 1.2 cycles per second, the samples do not record the start and finish points of every cycle.

If we look at the samples on a graph without the continuous curve, we can still make out the path of the discrete wave, but we cannot be sure where the cycles actually repeat:



If we join up the points with straight lines (solely to make their positions clearer), we can see how the vertical positions of the samples are different for the first three cycles of the continuous wave:



For the "straight-line" discrete signal, the first cycle has a period of roughly 0.83 seconds, so the frequency is roughly 1.2 cycles per second. The second cycle starts at about 0.83 seconds and continues to about 1.65 seconds – therefore, its period is roughly: 1.65 – 0.83 = 0.82 seconds. The third cycle starts at about 1.65 seconds and continues to 2.5 seconds. Therefore, its period is 2.5 – 1.65 = 0.85 seconds. In each of these calculations, we are making the assumption that the curve of the discrete wave continues in roughly the same way between samples. This is true in this case, but with discrete signals, we can never be sure that it is true. One of the characteristics of a discrete signal is that the values between the samples are unknown. An obvious example of when our assumption would be wrong is if the samples had come from an undersampled "y = sin (2π * 11.2t)", which would have the same samples.

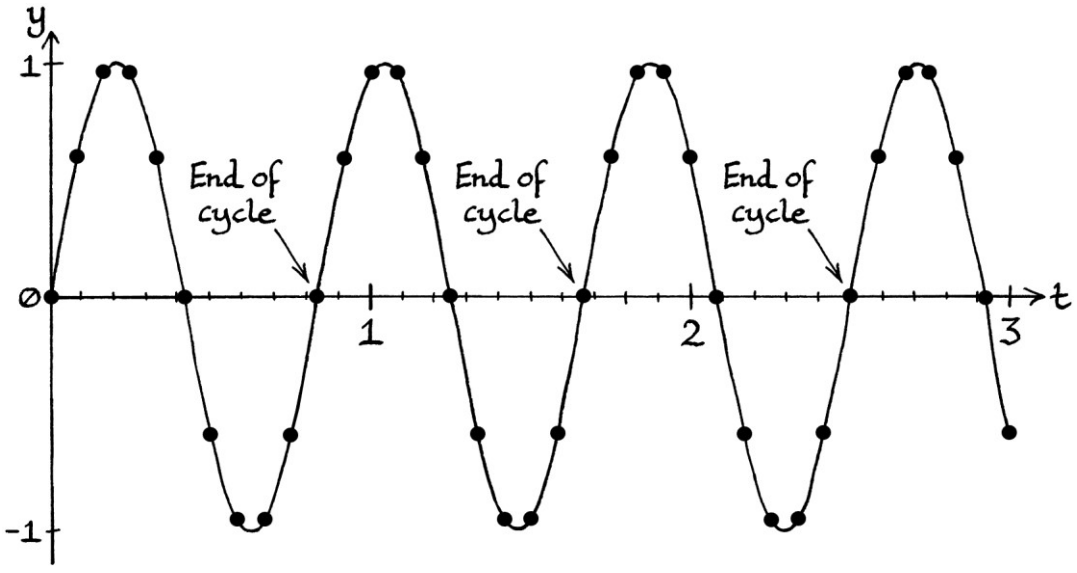
It is a common problem to have each cycle of the continuous wave producing different samples because of a mismatch between the frequency and the sample rate. Generally, we would use one sample rate for all our signals, as opposed to altering the sample rate to deal with each signal that we needed to record. Therefore, we will often have a continuous wave or signal that has cycles that do not finish and start exactly at the point where a sample reading is taken. This means that the discrete wave or signal will not repeat where the continuous wave or signal repeats.

If we are given a discrete signal as a list of samples, a simple way to calculate the frequency is to look through the samples until we find a group that matches those at the very start of the signal. For example, if a discrete signal starts with these samples:
0, 0.3090, 0.5878, 0.8090
... then we look through the other samples until we find a place where we see those samples again in that order. The place where the samples repeat will *probably* be the start of the second cycle – we would then need to continue checking samples to make sure that it really was the start of a new cycle, and not just a place where a few samples repeated. This method works for the situation we have here, *but it finds only the discrete signal's frequency* – it does not find the continuous wave's frequency. If the places where the continuous cycles start and finish are not always exactly where a sample is read, then the method would count a much longer cycle for the signal, and the discrete and continuous waves would have different frequencies. It might be that the method will never find an end of a cycle at all. This can happen for a reason we will see shortly, or because the accuracy of the samples varies throughout the sampling. [For example, a y-axis value of 3 units on a continuous signal might be sampled as 3.0000001 units at one moment in time and

2.9999999 units at another time. This would stop us finding the frequency of the discrete signal as easily.]

The way that the discrete wave has a different frequency from the continuous wave it is supposed to represent affects how we perform Fourier series analysis. Instead of basing our test waves on the frequency of the continuous wave (1.2 cycles per second), we must base the test waves on the frequency of the discrete wave (0.4 cycles per second). If we use test waves that are sequential integer multiples of 1.2 cycles per second, we will not discover our wave. If we use test waves that are sequential integer multiples of 0.4 cycles per second, we will discover our original continuous wave exactly. [If we are given a list of samples and a sample rate, with no information as to which continuous wave they are supposed to represent, performing Fourier series analysis in this way is the only option we have anyway.] From the point of view of Fourier series analysis, it does not matter that our discrete wave has a different frequency from the continuous wave – we can recover the continuous wave's details using the discrete wave's frequency.

If we had a sample rate of 12 samples per second for our wave of 1.2 cycles per second, the frequency of the continuous wave and the discrete wave would be the same. The starts and ends of the cycles would coincide with the times when the samples were being read:



For a sample rate of 12 samples per second, the first few samples are as so:

Time (seconds)	**Sample value**	
0	0	
0.0833	0.5878	
0.1667	0.9511	
0.2500	0.9511	
0.3333	0.5878	
0.4167	0	
0.5000	−0.5878	
0.5833	−0.9511	
0.6667	−0.9511	
0.7500	−0.5878	[This is where the first cycle ends]
0.8333	0	[This is where the second cycle starts]
0.9167	0.5878	

The next sample would be:

1	0.9511	

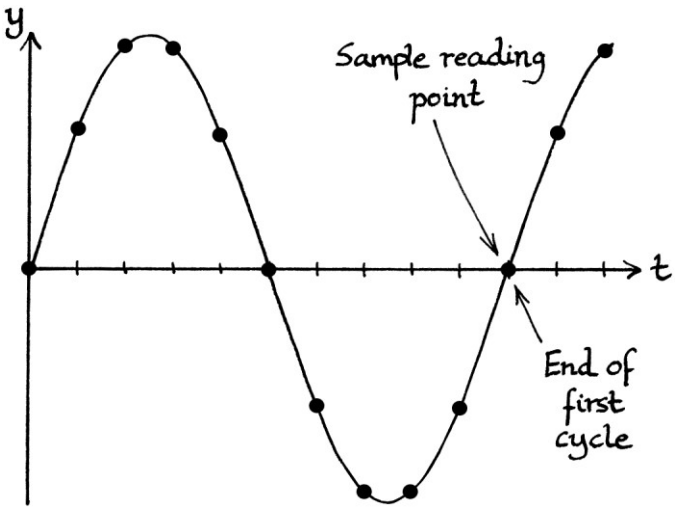
If we had a sample rate of 12 cycles per second, we would still be likely to have the frequency mismatch problem occur with other frequencies instead.

Mismatched frequencies and sample rates

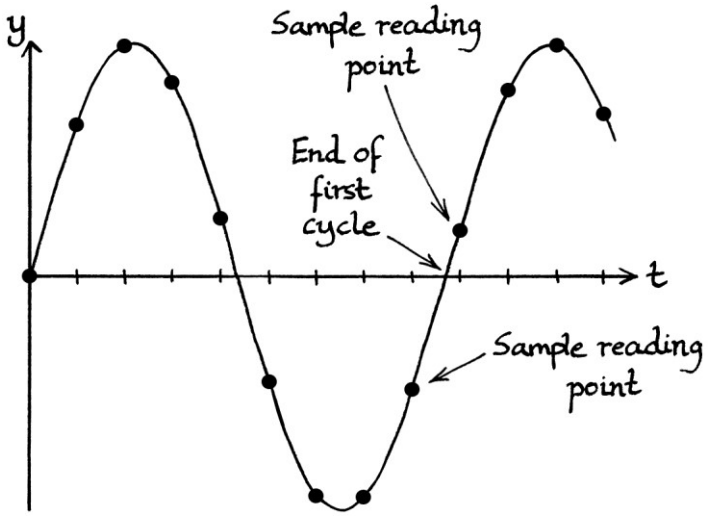
The problem we saw with a frequency of 1.2 cycles per second and a sample rate of 10 samples per second is a result of there being a mismatch between the places where the samples are being taken and the places where the cycles start and end.

A continuous wave with a particular frequency will end up as a discrete wave with a different frequency if the end of the first cycle of the continuous wave does not coincide exactly with a point where a sample is being taken. This problem does not just occur with non-integer frequencies – we saw it earlier in this chapter, when a 3-cycle-per-second continuous wave turned into a 1 cycle-per-second discrete wave, and when a 4-cycle-per-second continuous wave turned into a 2 cycle-per-second discrete wave.

We can portray the idea with drawings. The following continuous Sine wave will be recorded as a discrete wave with the correct frequency:



On the other hand, the following continuous Sine wave will *not* be recorded as a discrete wave with the correct frequency:



For a sample rate of 10 samples per second, we take samples every 0.1 seconds. For the discrete wave to have the same frequency as the continous wave, the cycles of the continuous wave must all end exactly at a 0.1 second point. If they do not, the discrete wave will have the wrong frequency.

In most cases, if the end of the first cycle does not occur exactly when a sample is being read, there will be a point later on where a cycle *does* end where a sample is being read. In such cases, the discrete wave's frequency will be slower than that of the continuous wave by an integer multiple. In such cases, we can still use Fourier

series analysis to recover the original wave (if the test waves are integer multiples of the discrete's wave's frequency).

Rule

For a given sample rate and frequency, we can test if there will be a mismatch and, if so, what the frequency of the discrete wave will be from that mismatch. We first divide the sample rate by the frequency. We then multiply the result by consecutive integers until we end up with an integer result. That integer result will be the *sample number* at which the first end of a cycle is at the exact place where a sample is read. [Remember that for sample numbers, we count from zero.] The integer by which we multiply the result of the division to obtain an integer result will be the number of times the discrete frequency is slower than the continuous frequency.

We can express the rule for calculating the details of mismatches slightly more mathematically as:

"sample number match place = (sample rate ÷ frequency) * x"
... where:

- "sample number match place" is an integer, and refers to the sample at which the first end of a cycle of the continuous wave coincides with a sample reading. The sample numbers start at sample number 0.
- "x" is the lowest integer that is necessary to turn "sample rate ÷ frequency" into an integer. When we have found "x", we will know that the frequency of the discrete wave will be "x" times slower than the frequency of the continuous wave.

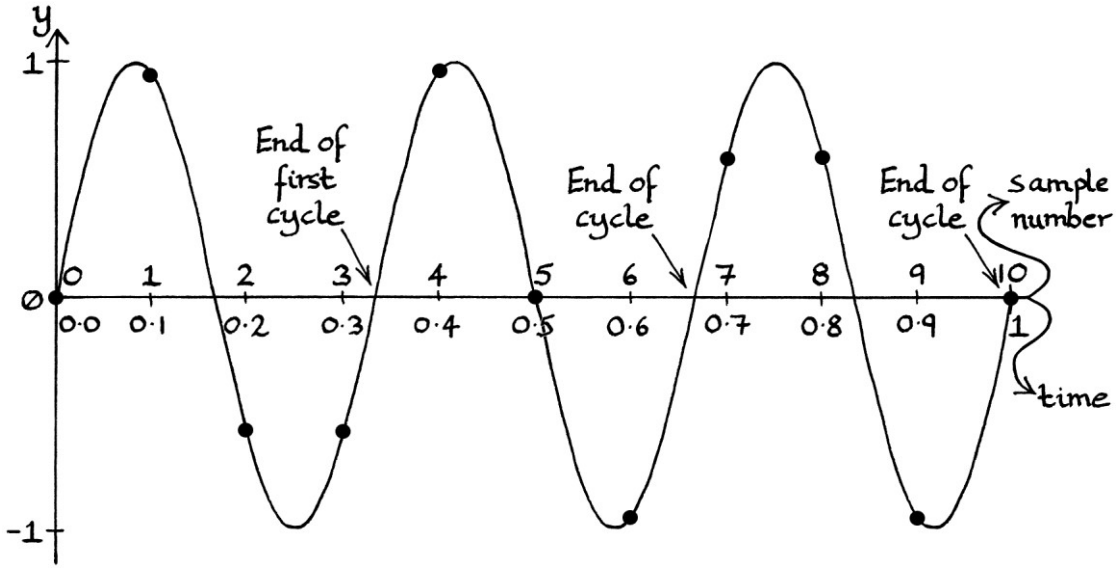
[Note that there are probably better ways to calculate or express these details, but this method emphasises how we need the samples and the cycles to coincide.]

3 cycles per second

As an example, we will use a sample rate of 10 samples per second and a frequency of 3 cycles per second. We divide the sample rate by the frequency:
10 ÷ 3 = 3.33333333 [to 8 decimal places]

We then multiply 3.33333333 by sequential integers until the result is an integer:
3.33333333 * 1 = 3.33333333
3.33333333 * 2 = 3.66666667
3.33333333 * 3 = 10, which is an integer.

Therefore, the first place where a cycle of our 3-cycle-per-second wave will end exactly at the point where a sample is being read will be at sample number 10. [Remember that we start counting samples at sample number 0, so sample number 10 is the eleventh sample.] As we have 10 samples per second, sample number 10 is at t = 1. We can confirm this on a graph with the y-axis marked as both the time axis and the sample number axis. The first place where the end of a cycle coincides with where a sample is read is at t = 1 or sample number 10:

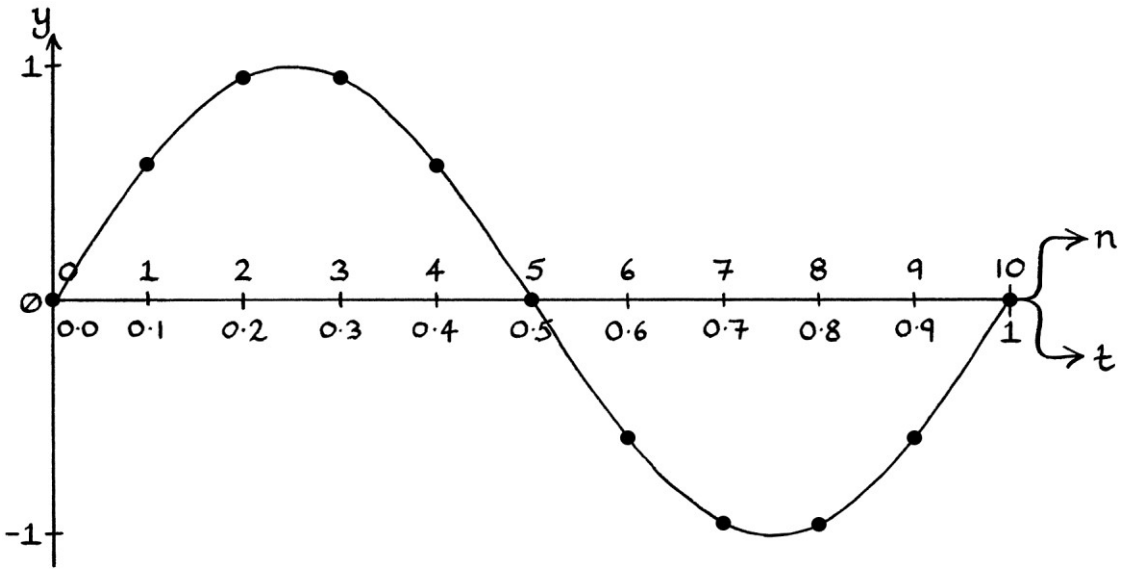


We multiplied the result of the division by 3 to turn it into an integer. The frequency of the continuous wave is 3 times the frequency of the discrete wave. The continuous wave had a frequency of 3 cycles per second; the discrete wave has a frequency of 1 cycle per second. [We saw this occur earlier in this chapter, which confirms that the method works.]

1 cycle per second

As another example, we will use a sample rate of 10 samples per second and a frequency of 1 cycle per second. We divide the sample rate by the frequency:
10 ÷ 1 = 10

This is already an integer, so we know that the first place where the end of a cycle will coincide exactly with where a sample is read will be at sample number 10. We can confirm this on the following graph. The graph has the y-axis as both the time axis (t-axis) and the sample axis (n-axis) labelled with an "n".



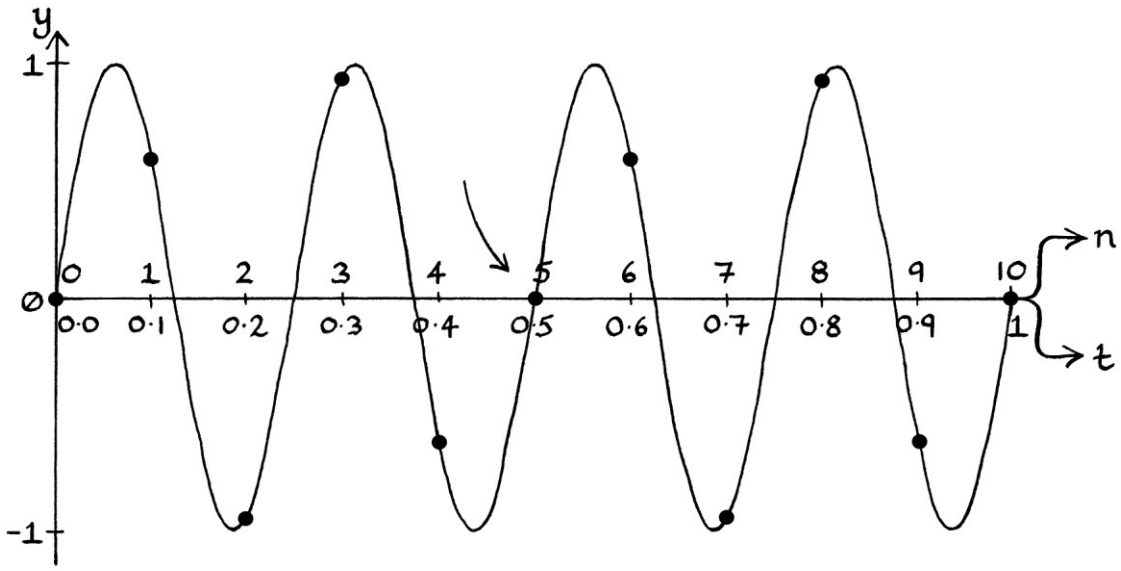
The discrete wave will have a frequency 1 time slower than that of the continuous wave, so, in other words, the frequencies will be the same.

4 cycles per second

For a frequency of 4 cycles per second and a sample rate of 10 samples per second, we divide the sample rate by the frequency:
10 ÷ 4 = 2.5

We then multiply 2.5 by sequential integers until the result is an integer:
2.5 * 1 = 2.5
2.5 * 2 = 5

This means that the first place where the end of a cycle will coincide with where a sample is read will be at sample number 5: [Again, the graph has a t-axis showing time in seconds and an n-axis showing the sample number, starting at sample number zero.]



We multiplied the division by 2 to turn it into an integer. Therefore, the frequency of the discrete wave will be 2 times slower than the continuous wave – it will be 2 cycles per second instead of 4 cycles per second. We saw earlier in this chapter that this is true, and we can also tell this from the graph.

5 cycles per second

For a frequency of 5 cycles per second and a sample rate of 10 samples per second, we divide the sample rate by the frequency as before:
10 ÷ 5 = 2

As the result is already an integer, we do not need to do any multiplications, but we can still say that:
2 * 1 = 2

The first place where the end of a cycle will coincide with where a sample is read will be at sample number 2. The frequency of the discrete wave will be 5 ÷ 1 = 5 cycles per second. [Note that the discrete signal will not be recorded correctly.]

1.2 cycles per second

For a frequency of 1.2 cycles per second and a sample rate of 10 samples per second, we divide the sample rate by the frequency:
10 ÷ 1.2 = 8.33333333 [to 8 decimal places]

We then multiply 8.33333333 by consecutive integers:
8.33333333 * 1 = 8.33333333
8.33333333 * 2 = 16.66666667
8.33333333 * 3 = 25, which is an integer.

Therefore, the first place where an end of a cycle of our 1.2 cycle-per-second wave will coincide with where a sample is being read will be at sample number 25. As we are using a sample rate of 10 samples per second, this means it will happen at t = 2.5 seconds. At 2.5 seconds, our continuous wave will have completed 1.2 * 2.5 = 3 cycles, but our discrete wave will have completed only 1 cycle. We multiplied 8.33333333 by 3 to make it an integer, which means that the discrete wave will have a frequency 3 times slower than that of the continuous wave – it will be 1.2 ÷ 3 = 0.4 cycles per second. We saw earlier that this is true.

3.4 cycles per second

For a frequency of 3.4 cycles per second and a sample rate of 10 samples per second, we divide the sample rate by the frequency:
10 ÷ 3.4 = 2.94117647 [to 8 decimal places]

We then multiply 2.94117647 by consecutive integers:
2.94117647 * 1 = 2.94117647
2.94117647 * 2 = 5.88235294
2.94117647 * 3 = 8.82352941
2.94117647 * 4 = 11.76470588
2.94117647 * 5 = 14.70588235
... and so on until:
2.94117647 * 17 = 50, which is an integer.

Therefore, the first place where the end of a cycle from our 3.4 cycle-per-second continuous wave will coincide with where a sample is being read will be at sample number 50. As we are using a sample rate of 10 samples per second, this will be at t = 5 seconds. At 5 seconds, our continuous wave will have completed 3.4 * 5 = 17 cycles, but our discrete wave will have completed only 1 cycle. Therefore, our

discrete wave will have a frequency of 1 ÷ 5 = 0.2 cycles per second. We can also calculate our discrete frequency by knowing it will be 17 times slower than that of the continuous frequency: 3.4 ÷ 17 = 0.2 cycles per second.

More non-integer frequencies

Now that we know more about mismatched frequencies and sample rates, we will continue to look at various waves with non-integer frequencies.

π cycles per second

If we had a frequency of π cycles per second, and a sample rate of 10 samples per second, the discrete wave would never repeat. We can show this to be true by calculating when an end of a cycle would coincide with the time of a sample reading. First, we divide the sample rate by the frequency:
10 ÷ π = 3.18309886 [to 8 decimal places]

Next, we multiply 3.18309886 by consecutive integers until the result is an integer. Given that π is an irrational number, 10 ÷ π will be an irrational number too. Therefore, there is no integer that we can multiply by 3.18309886 that will result in an integer.

For the formula:
"sample number match place = (sample rate ÷ frequency) * x"
... we would continue forever, increasing "x" by 1, but never producing an integer. The value of "x" would become infinitely high. This means that the frequency of the discrete signal would be infinitely slower than the frequency of the continuous wave. The frequency would be zero. There would never be a time when the pattern of samples repeated. The discrete signal would have an infinitely long period. [Note that a zero-frequency impure *signal* is not the same as a zero-frequency pure *wave*. A zero-frequency pure wave, whether continuous or discrete, has the same values for all time, while a zero-frequency impure signal, whether continuous or discrete, has many different values and never repeats its pattern.] In practice, given that we would probably be using a computer to calculate the samples, and given that a computer has a limited level of accuracy, rounding errors would eventually make it seem as if the discrete signal *did* repeat.

[For more information about irrational numbers, see Chapter 13 on the addition of waves in the section "Larger fractional ratios".]

Despite the problems with irrational frequencies, for a sample rate of 10 samples per second, the continuous wave “y = sin (2π * πt)” would have the same samples as the following continuous waves:
“y = sin (2π * (10 + π) * t)”
“y = sin (2π * (20 + π) * t)”
“y = sin (2π * (30 + π) * t)”
“y = sin (2π * (40 + π) * t)”
... and so on, and also:
“y = sin (2π * (−10 + π) * t)”
“y = sin (2π * (−20 + π) * t)”
“y = sin (2π * (−30 + π) * t)”
“y = sin (2π * (−40 + π) * t)”
... and so on.

For a frequency of “π” cycles per second, and a sample rate of 10 samples per second, Fourier series analysis would be incapable of finding the correct wave. The main reason for this is that there would be no discrete frequency around which to base our test frequencies. Fourier series analysis uses test waves that are integer multiples of the frequency of the signal we are testing – not the frequency of the wave we think it should be. When analysing discrete signals, we use the discrete signal’s frequency, and not that of the continuous signal from which the discrete signal came. Therefore, using test waves that were integer multiples of π would not find the original wave.

The problems with a frequency of π cycles per second also occur with any irrational frequency, as long as the sample rate is not a multiple of that frequency.

If we had a frequency of “π” cycles per second and a sample rate of, say, “10π” samples per second, then our discrete wave would end up with the same frequency as the continuous wave, and the sample rate would be fast enough to record the wave correctly:
10π ÷ π = 10
... and:
10 * 1 = 10, which is an integer. Therefore, the frequency of the discrete wave is the same as that of the continuous wave. However, there would be little point in using a sample rate of 10π samples per second, as it would mean that any frequency that was not a multiple of π would end up as a discrete signal with zero frequency and an infinitely long period.

For interest's sake, we will round "π" cycles per second to exactly 3.14 cycles per second, and use a sample rate of 10 samples per second. Our wave will be:
"y = sin (2π * 3.14t)".

We will calculate the discrete frequency:
10 ÷ 3.14 = 3.18471338 [to 8 decimal places]

3.18471338 * 1 = 3.18471338
3.18471338 * 2 = 6.36942675
3.18471338 * 3 = 9.55414013
... and so on until:
3.18471338 * 157 = 500, which is an integer.

The first cycle that ends at an exact point where we take a sample reading is at sample number 500, which is at t = 50.0 seconds. The discrete wave has a frequency of 1 ÷ 50.0 = 0.02 cycles per second. We could also say that the discrete wave's frequency would be 157 times slower than that of the continuous wave. It would be 3.14 ÷ 157 = 0.02 cycles per second.

Fourier series analysis using test waves that are integer multiples of 0.02 cycles per second would find the details of the original continuous wave correctly – in other words, despite the frequency of the discrete wave being different from that of the continuous wave, analysis of the samples would find "y = sin (2π * 3.14t)".

4.99 cycles per second

When sampling a wave, the highest frequency that we can record that can be recovered correctly with Fourier series analysis must be less than half the sample rate. Therefore, for a sample rate of 10 samples per second, the fastest frequency we can record correctly would be *under* 5 cycles per second [where by "correctly", I mean that the original continuous wave can be recovered from the samples with Fourier series analysis.]

To demonstrate how we can record close to 5 cycles per second, we will sample this wave:
"y = sin (2π * 4.99t)"
... at 10 samples per second.

Experience has taught us that the discrete wave will have a different frequency from the continuous wave. We will work out its frequency by seeing when the first cycle of the continuous wave ends exactly at a point where a sample is being read:
10 ÷ 4.99 = 2.00400802 [to 8 decimal places]

2.00400802 * 1 = 2.00400802
2.00400802 * 2 = 4.00801603
2.00400802 * 3 = 6.01202404
2.00400802 * 4 = 8.01603206
... and so on until:
2.00400802 * 498 = 997.99599200
2.00400802 * 499 = 1,000, which is an integer.

Therefore, the first time a cycle ends at a place where a sample is being taken is at sample number 1,000, which is when t = 100 seconds. The frequency of the discrete wave will be 499 times less than that of the continuous wave. It will be: 4.99 ÷ 499 = 0.01 cycles per second.

We will be able to recover the original wave correctly if we use test waves that are integer multiples of 0.01 cycles per second.

Rational numbers

[As a reminder, a rational number is the opposite of an irrational number – in other words, a rational number is one that can be expressed as one integer divided by another integer. We looked at irrational numbers in Chapter 13.]

For any rational sample rate, a wave with a rational frequency will eventually have a point where the end of a cycle coincides exactly with where a sample is being read. It might take a long time, but there will always be a match. If there were not a match, then either the frequency or the sample rate must be an irrational number. This must be true because it would mean that the sample rate divided by the frequency would produce an irrational number, and for that to happen, one of them must be an irrational number – it is impossible to create an irrational number by dividing one integer by another. [Any division of two rational non-integers can be scaled to be one integer divided by another integer.]

If there is a point where the end of a cycle coincides with where a sample is being read, then the discrete wave will have a non-zero frequency, and we will be able to analyse the signal to recover the original wave.

Odd sample rates

Sample rates that are odd numbers can sometimes cause problems if you forget that half the sample rate will not be an integer.

A signal consisting of waves with integer constituent frequencies will have an integer fundamental frequency. For an even sample rate, we can add to the signal a constituent wave with a frequency equal to half the sample rate, and the fundamental frequency of the sum will still have an integer frequency. This is because the frequency equal to half the sample rate will still be an integer. For odd sample rates, a wave with a frequency equal to half the sample rate will be a non-integer. Adding such a wave to a signal with an integer fundamental frequency will cause the fundamental frequency to change.

To explore this idea, we will look at a sum of waves sampled at 11 samples per second. We will add each continuous wave in turn and see what happens for each wave.

"y = sin (2π * 1t)":
This continuous wave will be recorded as a discrete wave with a frequency of 1 cycle per second.

Addition of "y = sin (2π * 2t)":
After adding "y = sin (2π * 2t)", the continuous signal will still have a frequency of 1 cycle per second. The discrete signal will still have a frequency of 1 cycle per second.

Addition of "y = sin (2π * 5t)":
After adding "y = sin (2π * 5t), the continuous signal and the discrete signal will still both have frequencies of 1 cycle per second. If we were analysing our discrete signal, we would need 1 second's worth of samples.

Addition of "y = sin (2π * 5.5t)":
We will now add "y = sin (2π * 5.5t)" to our continuous signal. Because this frequency is not an integer, the frequency of our continuous and discrete signals will change. The addition of this wave will cause it to change to 0.5 cycles per second.

To analyse our signal now, we would need to have 2 seconds' worth of samples. One cycle of our discrete signal now requires 22 samples, whereas before, it needed 11 samples.

Apart from the half-the-sample-rate wave changing the fundamental frequency of a signal, odd sample rates can be treated in the same way as even sample rates. The "over half the sample rate" rule still holds true for odd sample rates. For example, for a sample rate of 11 cycles per second (in which case, half the sample rate is 5.5 cycles per second), the following are true:

- A frequency of 6 cycles per second is 0.5 cycles per second *above* half the sample rate, so it becomes recorded as 0.5 cycles per second *below* half the sample rate, which is 5 cycles per second.

- 7 cycles per second is 1.5 cycles per second above half the sample rate, so it becomes 1.5 seconds below half the sample rate, which is 4 cycles per second.

- 8 cycles per second becomes 3 cycles per second.

- 9 cycles per second becomes 2 cycles per second.

- 10 cycles per second becomes 1 cycle per second.

- 11 cycles per second becomes 0 cycles per second

Non-integer sample rates

We do not have to have a sample rate that is an integer. We can have non-integer sample rates such as 10.5 samples per second or 0.1 samples per second. Generally, it is easier to visualise everything and to perform calculations if we have integer sample rates. If we are going to have a non-integer sample rate, ideally, it would be a rational number. We *could* have an irrational number as the sample rate, but it would make everything more complicated, and many continuous waves would become discrete signals with zero frequencies.

Negative sample rates

Theoretically, we could have negative sample rates, although they would probably not be useful in sampling waves and signals. Despite the idea seeming strange, it pays to consider ideas that would be impossible or useless in real life.

Other sampling thoughts

Duplicate frequencies and the discrete formula

Earlier in this chapter, we saw the "$f_0 = f_0 + (x * f_s)$" rule. For a given sample rate, the samples from a wave will be identical to those from a wave with a frequency an integer multiple of the sample rate higher or lower. At the end of Chapter 41, we saw a formula for a discrete wave: "$y = h + A \sin((2\pi * f * T_s * n) + \phi)$". There is no way of knowing the "$f_0 = f_0 + (x * f_s)$" rule from our discrete wave formula because our discrete wave formula only refers to the resulting discrete signal – there is no mention of the original continuous signal from which it is (or might have been) derived. Therefore, despite having a discrete formula, it cannot help us in knowing about the faster duplicate frequencies from the sampling process.

Negative frequencies made positive

When we analyse a discrete signal, we use test frequencies that are integer multiples of the frequency of the signal (the fundamental frequency). You might wonder if it is always the case that the frequency of a negative-frequency-made-positive-frequency duplicate will be an integer multiple of the fundamental frequency. If it were not, then that duplicate could never be found. [Generally, whether it is found or not does not matter because the duplicates are not really part of the signal and are best ignored.] As far as I can tell, the frequency of a negative-frequency-made-positive-frequency duplicate will always be an integer multiple of the fundamental frequency of the discrete signal. Therefore, such duplicates will always be found (if analysis continues past the frequency equal to half the sample rate).

We can see examples in the following table of various frequencies for a sample rate of 10 samples per second. In the first column is the frequency of the continuous wave that we want to record. The second column shows the frequency of the discrete wave consisting of samples of the continuous wave. As explained earlier in this chapter, this frequency might be different from that of the continuous wave. In the third column is the frequency of the first negative-frequency-made-positive duplicate of the discrete wave. Note that this is based on the frequency of the continuous wave, and not on the frequency of the discrete wave. The fourth column says whether the duplicate is an integer multiple of the discrete frequency. If it is an integer multiple then it will be possible for analysis to find it.

The letters "nfmpfd" are an abbreviation for "negative-frequency-made-positive-frequency duplicate."

Continuous Frequency	**Discrete Frequency**	**Frequency of first nfmpfd**	**Is nfmpfd an integer multiple of discrete frequency?**
0 cps	0 cps	10 cps = 0 cps	Yes
1 cps	1 cps	9 cps	Yes
2 cps	2 cps	8 cps	Yes
3 cps	1 cps	7 cps	Yes
4 cps	2 cps	6 cps	Yes
5 cps	5 cps	5 cps	Yes
6 cps (= 4 cps)	2 cps	6 cps	Yes
7 cps (= 3 cps)	1 cps	7 cps	Yes
8 cps (= 2 cps)	2 cps	8 cps	Yes
9 cps (= 1 cps)	1 cps	9 cps	Yes
10 cps (= 0 cps)	0 cps	10 cps = 0 cps	Yes
1.2 cps	0.4 cps	8.8 cps	Yes
3.4 cps	0.2 cps	6.6 cps	Yes
4.99 cps	0.01 cps	5.01 cps	Yes

In all the examples, the negative-frequency-made-positive duplicate is an integer multiple of the discrete wave's frequency. Therefore, it will always be found for the examples in the table. From what I can tell, it will *always* be the case for any frequency and any sample rate.

If we ignore irrational continuous frequencies and continuous frequencies over half the sample rate, we can make three interesting observations from the table:

- The continuous frequency will always be equal to, or an integer multiple of, the discrete wave's frequency. We know that this will always be true from the "Mismatched frequencies" section earlier in this chapter.
- The frequency of the first negative-frequency-made-positive wave will always be equal to, or a multiple of, the discrete wave's frequency. [I have not proved this, but I have not yet found any exceptions.]
- The discrete frequency is always the highest number for which the continuous frequency and the negative-frequency-made-positive frequency are integer multiples. In other words, the discrete frequency is the highest common divisor. [Again, I have not proved this, but I have not yet found any exceptions.]

These three observations give us another way to calculate the discrete frequency of a sampled continuous wave with a rational frequency below or equal to half the sample rate. We need the continuous frequency and the negative-frequency-made-positive frequency. We then calculate the highest number that goes into both an integer number of times. In other words, we find the highest number for which the two frequencies are both integer multiples. We are finding the highest common divisor.

We will call the continuous frequency "f_0". The negative-frequency-made-positive will always be the continuous frequency minus the sample rate, and then made positive. It will be "$f_0 - f_s$" made positive. As "$f_0 - f_s$" will always be negative, we can give its positive form as:
$-(f_0 - f_s)$
... which is:
$-f_0 + f_s$
... which is:
$f_s - f_0$

We then find the highest number which divides into both an integer number of times. This will be the highest possible "z" in the following two formulas:
$f_0 \div x = z$
$(f_s - f_0) \div y = z$
... where "x" and "y" are both integers.

We can phrase this the other way around:
$z * x = f_0$
$z * y = (f_s - f_0)$
... where "x" and "y" are both integers.

The quickest way to calculate "z" is to use a "highest common divisor" ("greatest common factor") calculator program or website. [We will see another way shortly.] As these usually work with integers, we need to multiply our frequencies by the same power of 10 so that they both become integers, then find the highest common divisor, and then divide the result by the amount by which we multiplied the frequencies. If we were doing it by hand, we could scale the frequencies by a power of 10 so that they were both integers, then write out all the integer factors of each frequency until we found the highest factor shared by both. We would then divide that factor by the number by which we multiplied the frequencies to start with. [Note that this is not the fastest way, but it is the easiest to understand.]

We will go through an example with a sample rate of 12 samples per second and a continuous frequency of 4.75 cycles per second. The negative-frequency-made-positive frequency will be 12 – 4.75 = 7.25 cycles per second. We will multiply each frequency by 10 until they are both integers. We will need to multiply them by 100 to produce 475 and 725.

- The factors of 425 are: 1, 5, 17, 25, 85, 425
- The factors of 725 are: 1, 5, 25, 29, 145, 725
- The highest shared factor is 25.
- We divide 25 by 100 to get 0.25.

Therefore, 0.25 cycles per second will be the frequency of the discrete version of our continuous wave.

We can confirm it using the method from the "Mismatched frequencies and sample rates" section from earlier in this chapter. We divide the sample rate by the frequency:
12 ÷ 4.75 = 2.52631579 (to 8 decimal places).

We then multiply 2.52631579 by consecutive integers until we obtain an integer result:
2.52631579 * 1 = 2.52631579
2.52631579 * 2 = 5.05263158
2.52631579 * 3 = 7.57894737
2.52631579 * 4 = 10.10526316
2.52631579 * 5 = 12.63157895
2.52631579 * 6 = 15.15789474
2.52631579 * 7 = 17.68421053
... and so on until:
2.52631579 * 19 = 48.

We had to multiply the number by 19 to turn it into an integer. Therefore, the frequency of the discrete wave will be 19 times slower than the frequency of the continuous wave. It will be:
4.75 ÷ 19 = 0.25 cycles per second.

This matches the result we had earlier, which shows that both methods work.

An irrelevant section on highest common divisors

In the previous section, both methods worked to find the discrete frequency of a continuous wave sampled at a particular frequency. An interesting consequence of this (which is completely unrelated to waves) is that the "Mismatched frequencies and sample rates" method is actually a simple way to calculate the highest common divisor of any two numbers. If we are given two numbers, we have to rephrase them so that they can pretend to be two continuous frequencies based around a sample rate. The bigger number will be "f_0" and the smaller number will be "$f_0 - f_s$".

For example, if we have the numbers 144 and 24, we will say that:
"f_0" = 144
"$f_0 - f_s$" = 24

This allows us to calculate the number that, if were dealing with waves, would be the sample rate "f_s":
"$f_0 - f_s$" = 24
$144 - f_s = 24$
$-f_s = 24 - 144$
$-f_s = -120$
$f_s = 120$

Then, we can use the method from the "Mismatched frequencies and sample rates" section to calculate what, if we were dealing with waves, would be the discrete frequency. We divide the "sample rate" by the "frequency", and then multiply the result by consecutive integers until the result of that is an integer.

The "sample rate" divided by the "frequency" is:
120 / 144 = 0.83333333

0.83333333 * 1 = 0.83333333
0.83333333 * 2 = 1.66666667
0.83333333 * 3 = 2.5
0.83333333 * 4 = 3.33333333
0.83333333 * 5 = 4.16666667
0.83333333 * 6 = 5, which is an integer.

Therefore, the "discrete frequency" (which is actually the highest common divisor) is 6 times lower than the original "continuous frequency". Therefore, it is 144 ÷ 6 = 24. The highest common divisor is 24. We have found it using a reasonably simple-to-perform method.

We can rephrase the method to make it simpler and less related to waves. We will say that we have the two numbers "A" and "B", where "A" is the larger of the two, and that we want to find the highest common divisor. In this way, "A" is standing in for "f_0" and "B" is standing in for "$f_0 - f_s$", which becomes "$A - f_s$". We calculate "f_s" as so:

$B = A - f_s$

$f_s + B = A$

$f_s = A - B$

Our fraction will be the sample rate divided by the frequency. In terms of "A" and "B", it becomes

$$\frac{A - B}{A}$$

Therefore, we need to find the number that when multiplied by:

$$\frac{A - B}{A}$$

... results in an integer. We then divide "A" by the multiple that created the integer, and we will have the highest common divisor.

We can express this more concisely as:

- "A" and "B" are either both positive or both negative.
- "A" is greater than B.
- $x * \left(\frac{A - B}{A}\right)$ is an integer, where "x" is the lowest possible integer that makes it so.
- (A / x) is the highest common divisor of "A" and "B".

The method works for integers and non-integers.

Although the method can involve a lot of multiplications, it is very easy to program. [If you write a computer program to do this, remember that rounding errors might cause an integer not to be recognised as an integer.]

As an example of the method, if we have the numbers 160 and 56, we calculate:

$$\frac{160 - 56}{160} = 0.65$$

Then:
0.65 * 1 = 0.65
0.65 * 2 = 1.3
0.65 * 3 = 1.95
0.65 * 4 = 2.6
0.65 * 5 = 3.25
... and so on until:
0.56 * 20 = 13.

Then we divide "A" by the amount by which we multiplied our fraction (20):
160 ÷ 20 = 8
This means that the highest common divisor is 8.

We will find the highest common divisor of the numbers 148.4 and 6.4. [The divisor will have one decimal place of accuracy.] We will say that "A" is 148.4 and that "B" is 6.4. We need to find the number that when multiplied by:

$$\frac{148.4 - 6.4}{148.4}$$

... results in an integer. The fraction is: 0.95687332 to 8 decimal places. We then start the multiplications:

0.95687332 * 1 = 0.95687332
0.95687332 * 2 = 1.91374663
0.95687332 * 3 = 2.87061995
0.95687332 * 4 = 3.82749326
... and so on until:
0.95687332 * 371 = 355, which is an integer.

Therefore, the highest common divisor of 148.4 and 6.4 (that has one decimal place) is:
148.4 ÷ 371 = 0.4

Meanings of "correctly"

When it comes to sampling, there are really three different meanings of "correctly" when we say that a continuous wave has been recorded correctly.

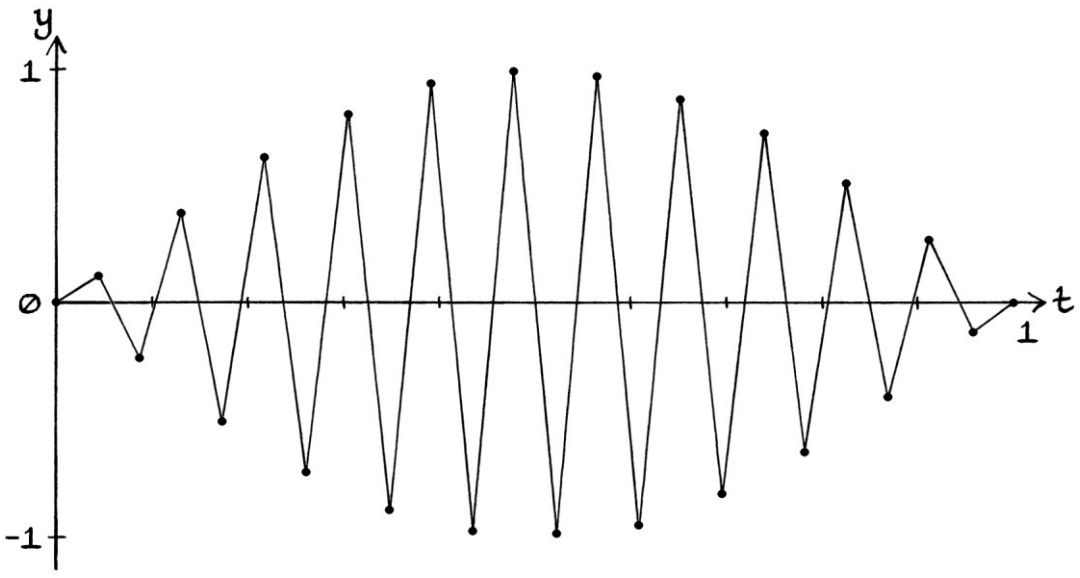
- We can have a wave recorded "correctly" in the sense that we will be able to recover its attributes with Fourier series analysis. We could say that the samples are "*analytically* correct". From the point of view of waves and signals, this is the definition of "correct" that I think is most sensible.

- We can have a wave recorded "correctly" in the sense that a graph of the samples appears to follow the outline of the wave. We could say that the samples are "*visually* correct".

- We can have a wave recorded "correctly" in the sense that the discrete samples have the same frequency, amplitude, and phase as the continuous wave. We could say that the samples are "*characteristically* correct" because the characteristics of the continuous wave are all seen in the discrete wave.

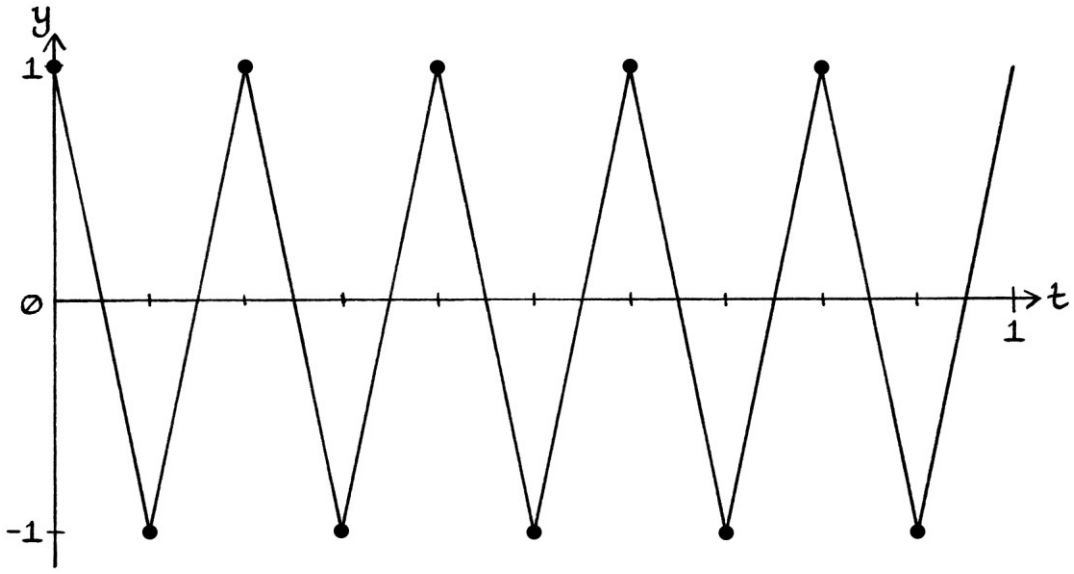
For a sample rate that is sufficiently fast in relation to the frequencies of the waves being sampled, we can have discrete waves that obey all three definitions of "correct". For slower sample rates, we might have discrete waves that are only correct according to just one or two of these definitions.

An "analytically correct" discrete wave that is not "characteristically correct" or "visually correct" is "y = sin (2π * 11t)" sampled at 23 samples per second. The samples are shown here (joined together to make their position clearer):



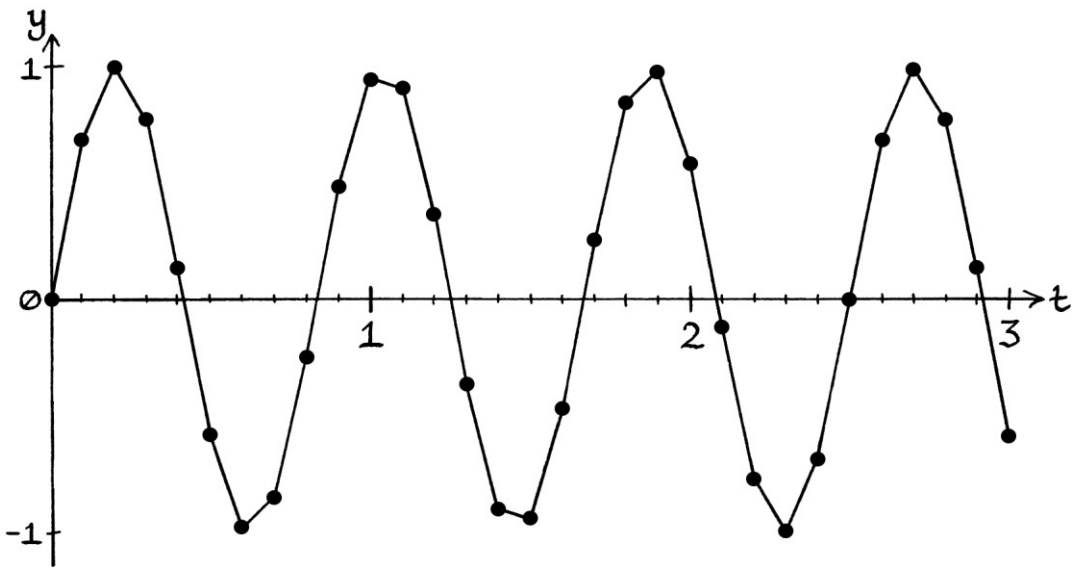
Despite how the signal does not look like an 11-cycle-per-second wave, and despite how it has a frequency of 1 cycle per second, if we analysed it using Fourier series analysis (and test frequencies based around a frequency of 1 cycle per second), we would find the original continuous wave formula.

An example of a "visually correct" and "characteristically correct" discrete wave that is not "analytically correct" is "y = sin ((2π * 5t) + 0.5π)" with a sample rate of 10 samples per second. The graph is shown here with the samples joined together with straight lines to make their position clearer:



The samples record each peak and dip correctly – we can tell the actual amplitude and frequency. We also know that the samples probably came from a Sine wave with a phase of 0.5π radians. However, if we analysed this discrete signal using Fourier series analysis, we would not find the original continous wave, but instead we would find "y = 2 sin ((2π * 5t) + 0.5π)". The amplitude would be twice what it should be.

Earlier, we saw the example with "y = sin (2π * 1.2t)" sampled at 10 samples per second. The graph is shown here with the samples joined together with straight lines to make their position clearer:



We can roughly tell where the cycles are "supposed" to repeat, although the actual discrete signal has a frequency of 0.4 cycles per second and not 1.2 cycles per second. The discrete wave is "analytically correct" (if we use the discrete wave's frequency, and not that of the continuous wave), but whether it is "visually correct" and "characteristically correct" is a matter of opinion.

As we have seen in this chapter, we can have a discrete version of a continuous wave that has a different frequency, or it might not reach up and down to the points reached by the continuous wave, or its phase might be unclear. However, it can still be analysed to find the original continuous signal that created it. Therefore, requiring that the discrete wave look correct or that it have the same frequency, amplitude and phase as the continuous wave are unnecessary for the purposes of analysis.

Sampling summary

Here are some of the main facts about sampling that we have learnt in this chapter:

- For a continuous wave (or continuous constituent wave), to be recorded so that Fourier series analysis can recover its characteristics exactly, the wave must have a rational positive frequency that is less than half the sample rate. [The sample rate must also be a rational number.]

- For a given sample rate, the samples taken from any continuous *wave* could also have come from any continuous wave with a frequency any integer multiple of the sample rate higher or lower.

- For a given sample rate, the samples taken from any continuous *signal* could also have come from any signal made up of constituent waves with frequencies any integer multiple of the sample rate higher or lower than the constituent waves in the original signal.

- The samples taken from any continuous wave or signal could also have come from a continuous wave or signal that consisted of straight lines joining up those samples.

- If a continuous wave (or continuous constituent wave) has a frequency above half the sample rate and under the sample rate, it will be recorded as a different wave with a positive frequency below half the sample rate.

- If a continuous wave (or continuous constituent wave) has a frequency equal to half the sample rate, how it will be recorded depends on its phase. Depending on its phase, either Fourier series analysis will not detect the wave at all, or it will find a wave that, if its amplitude is halved, would have the correct samples, but it might not be the correct wave.

- If we are analysing a discrete signal using Fourier series analysis, there is no point in testing waves that have frequencies above half the sample rate. If there had been any continuous waves in the original signal that had frequencies above half the sample rate, they would have ended up being recorded incorrectly with frequencies below half the sample rate. Therefore, we would have already found their aliases by the time we had tested for the frequency equal to half the sample rate.

- Given that, if we portray the analysis on a frequency domain graph, there is no point in the frequency axis extending past the frequency equal to half the sample rate.

Complex exponentials

We will look at some of the sampling rules from this chapter as they apply to helices portrayed by Complex exponentials. This section just introduces some of the rules, and there is a lot more that could be said.

Earlier in this book, we saw that we can portray a circle or helix as a Complex exponential. In Chapter 28, we saw that a pure wave can be portrayed as one Complex exponential added to, or subtracted from, another with an opposing frequency (and rotated if it is a Sine wave). As we know, any periodic signal is equal to, or approximately equal to, a sum of pure waves. We can also say that any periodic signal is equal to, or approximately equal to, a sum of pairs of Complex exponentials. Therefore, it pays to understand some aspects of sampling Complex exponentials.

Duplicate frequencies

If we have a *helix* portrayed by a single Complex exponential, then it will still be the case that the "$f_0 = f_0 + (x * f_s)$" rule applies. For example, if we have a continuous Complex exponential sampled at 10 samples per second, and with a frequency of 4 cycles per second, then it will have the same samples as that exponential given a frequency of 14, 24, 34, 44, 54 cycles per second and so on. It will also have the same samples as that exponential given a negative frequency of −6, −16, −26, −36 cycles per second and so on. The rule is as so:

The following Complex exponential, sampled at "f_s" samples per second:

$$z = Ae^{i((2\pi * f_0 * t) + \phi)}$$

... will have identical samples to this Complex exponential sampled at the same sample rate:

$$z = Ae^{i((2\pi * (f_0 + (x * f_s)) * t) + \phi)}$$

... where:

- "A" is the amplitude or radius.
- "f_0" is the original frequency.
- "x" is any integer, whether positive or negative.
- "f_s" is the sample rate.
- "t" is the time in seconds.
- "ϕ" is the phase in radians.

This is the "$f_0 = f_0 + (x * f_s)$" rule that applied for waves. The rule also works for the sums of Complex exponentials too, regardless of whether the frequencies in the sum are all positive, all negative or both positive and negative.

We will look at an example discrete Complex exponential, sampled at a sample rate of 10 samples per second:

$$z = e^{i(2\pi * 4t)}$$

As we are dealing with a Complex exponential, we can think of the samples in three different ways:

- as the samples of a derived Cosine wave and Sine wave
- as Real samples and Imaginary samples
- as Complex number samples

These are all essentially the same as each other. The first of the three should help you understand why the "$f_0 = f_0 + (x * f_s)$" rule is the same for Complex exponentials as it is for waves. The third of these is the way that the samples would normally be written outside of a computer program.

The samples for one second for the derived Cosine and Sine waves are:

Cosine or Real values: 1.0000, −0.8090, 0.3090, 0.3090, −0.8090, 1.0000, −0.8090, 0.3090, 0.3090, −0.809017

Sine or Imaginary values: 0.0000, 0.5878, −0.9511, 0.9511, −0.5878, 0.0000, 0.5878, −0.9511, 0.9511, −0.5878

If we give the samples as Complex numbers, we have:
1.0000 + 0.0000i
−0.8090 + 0.5878i
0.3090 − 0.9511i
0.3090 + 0.9511i
−0.8090 − 0.5878i
1.0000 − 0.0000i
−0.8090 + 0.5878i
0.3090 − 0.9511i
0.3090 + 0.9511i
−0.8090 − 0.5878i

If we give the exponential a frequency that is higher by a multiple of the sample rate, it will still have the same samples. For example, the following exponential has a frequency of 14 cycles per second, and it has the same samples as our 4-cycle-per-second exponential when sampled at 10 samples per second.

$$z = e^{i(2\pi * 14t)}$$

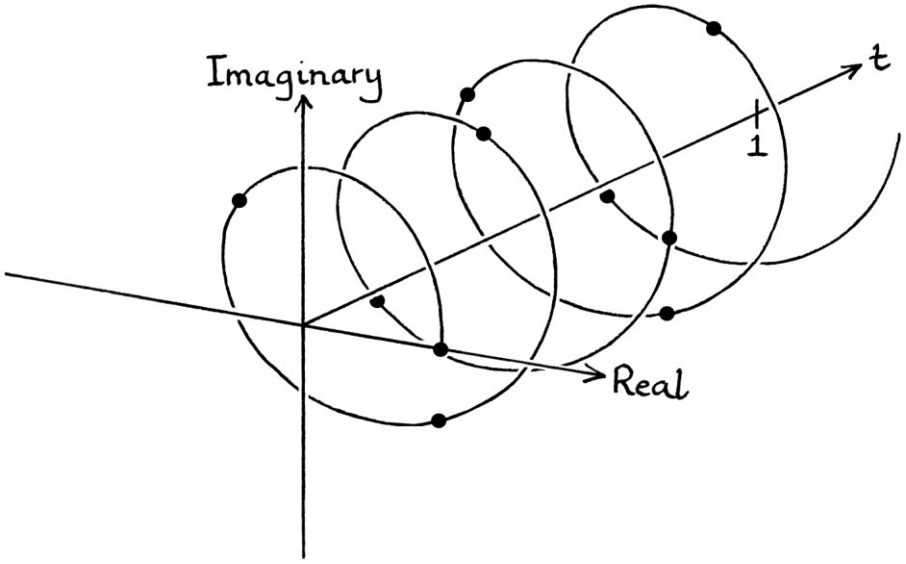
We can also change the frequency to be lower by a multiple of the sample rate. As an example, we will give it a frequency of −6 cycles per second:

$$z = e^{i(2\pi * -6t)}$$

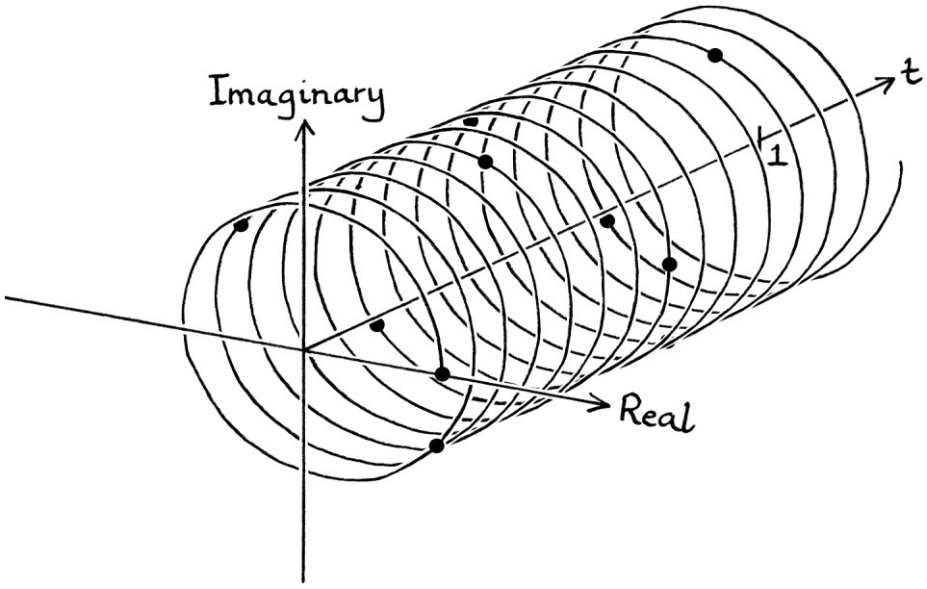
Note that this is now a clockwise helix (instead of an anticlockwise helix), yet, when it is sampled at 10 samples per second, it still has the same samples as our original 4-cycle-per-second Complex exponential. The fact that it is clockwise instead of anticlockwise is lost in the sampling. A *continuous* anticlockwise helix can never be thought of as a *continuous* clockwise helix (and vice versa), but when we sample them, we can lose the attributes of which way around they rotate. Therefore, it is possible for a discrete anticlockwise helix to have the same samples as a discrete clockwise helix. [With any discrete signal, we do not know what happens between the samples. Therefore, strictly speaking, it is impossible to be sure that any discrete anticlockwise helix is not supposed to be a discrete clockwise helix, and vice versa.]

We can get a better understanding of why the samples of our different Complex exponentials are the same by looking at pictures of the helices with the sampling points marked.

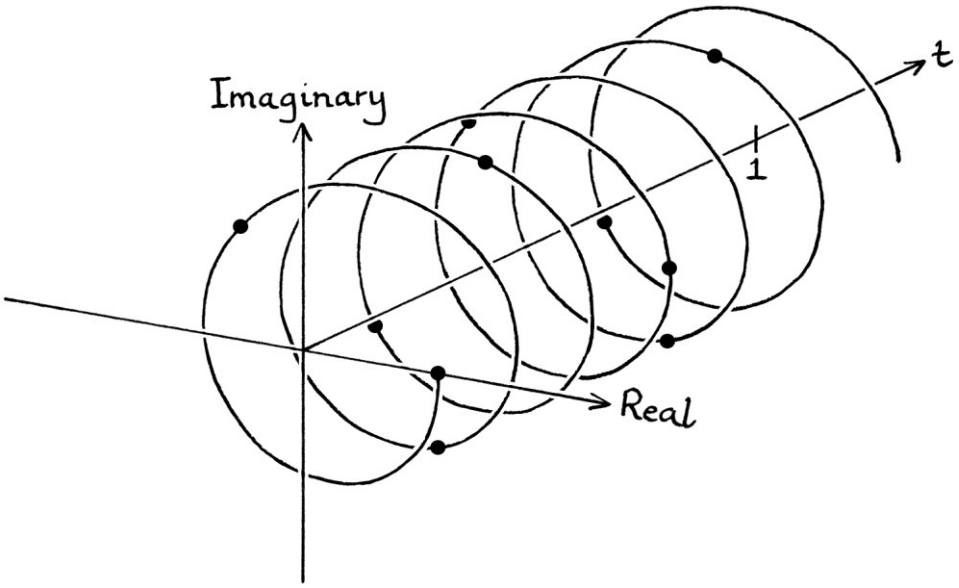
For the Complex exponential with a frequency of 4 cycles per second, the helix looks like this:



For the Complex exponential with a frequency of 14 cycles per second, the helix appears as in the following picture. The points where the helix is sampled are in the same places as before, but there are cycles that occur between the samples, and these are missed by the sampling process.



For the Complex exponential with a frequency of −6 cycles per second, the helix looks like this:



Despite how the helix rotates in a different way, when the samples are taken, there are points on the curve that are in the same place as in the previous two pictures. Hence, the samples from the helix are identical to before.

Sampled Cosine waves and sampled Sine waves follow the "$f_0 = f_0 + (x * f_s)$" rule. A circle or helix can be thought of as consisting of points described by a Cosine wave and a Sine wave. Therefore, a sampled helix follows the "$f_0 = f_0 + (x * f_s)$" rule. A Complex exponential describes a circle or helix, and therefore, it too follows the "$f_0 = f_0 + (x * f_s)$" rule.

The over-half-the-sample-rate rule

When we sample a continuous **wave** with a frequency between half the sample rate and the sample rate (exclusive), the samples will be the same as those from a wave with a frequency between 0 cycles per second and half the sample rate (exclusive). To be more specific, a wave with a frequency so many cycles per second *above* half the sample rate will have the same samples as a wave with a frequency that number of cycles per second below half the sample rate, and possibly a different phase. [We saw the rule in the "6 cycles per second" part of the "10-sample-per-second examples" section earlier.]

An example is that for a sample rate of 10 samples per second, this wave:
"$y = \sin((2\pi * 7t) + 0.9\pi)$"
... will have the same samples as this wave:
"$y = \sin((2\pi * 3t) + 0.1\pi)$"

The rule is true because a wave with a frequency from just over half the sample rate up to the sample rate is really a negative-frequency wave obeying the "$f_0 = f_0 + (x * f_s)$" rule, and that has been rephrased to have a positive frequency. We can see this is the case by going through more steps to get to our equivalent wave. If we start with this continuous wave:
"$y = \sin((2\pi * 7t) + 0.9\pi)$"
... then the continuous wave with a frequency 10 cycles per second lower (and which has the same samples) is:
"$y = \sin((2\pi * -3t) + 0.9\pi)$"
... and if we convert that to a positive-frequency continuous wave, we have:
"$y = \sin((2\pi * 3t) + 0.1\pi)$"

A consequence of this idea is that there is no point in searching for constituent waves above the frequency equal to half the sample rate – if a signal contained such waves, we would have already found a wave that had the same samples by the time we had reached the frequency equal to half the sample rate. Continuing past the frequency equal to half the sample rate would just find duplicates of the waves we had already found.

The rule works because of how a *continuous* negative-frequency wave formula can be rephrased to be a *continuous* positive-frequency wave formula with a possibly different phase while keeping the same curve. A continuous positive-frequency wave will always have the same curve as a particular continuous negative-frequency wave. We cannot tell by looking at a continuous wave curve whether it is supposed to be a positive-frequency wave or a negative-frequency wave, because, ultimately, there is no difference.

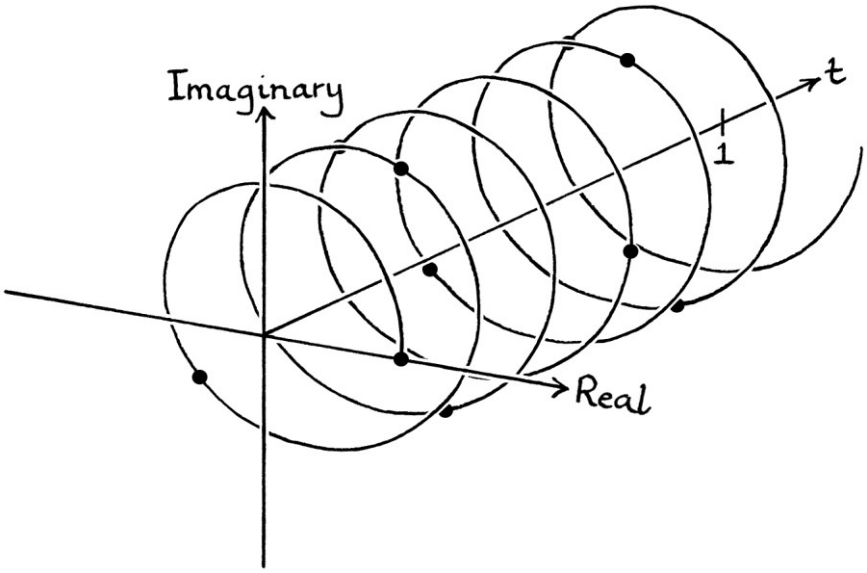
When we have a continuous negative-frequency *helix*, it has a different "twist" to a continuous positive-frequency helix. The helices are different shapes. An object moving around a circle and progressing down the time axis is moving anticlockwise for a positive frequency, and clockwise for a negative frequency. There are no *continuous* positive-frequency formulas for a helix that can be rephrased to be a *continuous* negative-frequency formula. This all means that the over-half-the-sample-rate rule that applies to waves does not apply to individual helices. An individual sampled helix with a frequency between half the sample rate and the sample rate will not have a duplicate that is between 0 cycles per second

and half the sample rate. It will not be aliased to become a helix between 0 cycles per second and half the sample rate.

From all of the above, we can say that individual helices will be sampled correctly if they have a frequency from 0 cycles per second up to just under the sample rate. Note that *individual* helices will be sampled correctly. When it comes to sums of helices, matters are more complicated as we will see later.

As an example of how we can record a helix with a frequency over half the sample rate, we will look at the Complex exponential:
$z = e^{i(2\pi * 6t)}$

When it is sampled at 10 samples per second, the samples drawn over the curve would appear as so:



The Complex samples are:

1.000 + 0.0000i
−0.8090 – 0.5878i
0.3090 + 0.9511i
0.3090 – 0.9511i
−0.8090 + 0.5878i
1.000 – 0.0000i
−0.8090 – 0.5878i
0.3090 + 0.9511i
0.3090 – 0.9511i
−0.8090 + 0.5878i

There are no other continuous unit-amplitude, zero-mean-level Complex exponentials with frequencies from 0 cycles per second up to the sample rate that share the same samples.

If we were to look at just the Real (Cosine) samples, we could find a second continuous Cosine wave with a frequency below half the sample rate that had the same samples. Similarly, if we looked at just the Imaginary (Sine) samples, we could find a duplicate Sine wave with a frequency below half the sample rate. However, there is no combination of Cosine and Sine waves that can be expressed as a helix (or Complex exponential) with a frequency below half the sample rate that has the same samples.

For our 6-cycle-per-second helix, a matching helix that obeys the "$f_0 = f_0 + (x * f_s)$" rule is:

$$z = e^{i(2\pi * -4t)}$$

[Its frequency is lower by the sample rate.] However, that helix cannot be rephrased to be one with a frequency between 0 cycles per second and the sample rate.

When it comes to *sums* of helices or Complex exponentials, the rules are more complicated, as we will see.

Discrete negative frequencies

The "$f_0 = f_0 + (x * f_s)$" rule means that for any sample rate, if we sample a continuous negative-frequency helix, the samples will be the same as those from a continuous positive-frequency helix (and vice versa). To put this another way, a *discrete* negative-frequency helix can be rephrased to be a *discrete* positive-frequency helix (and vice versa) while maintaining the same samples. As we have just seen, for a sample rate of 10 samples per second, the samples from:

$$z = e^{i(2\pi * -4t)}$$

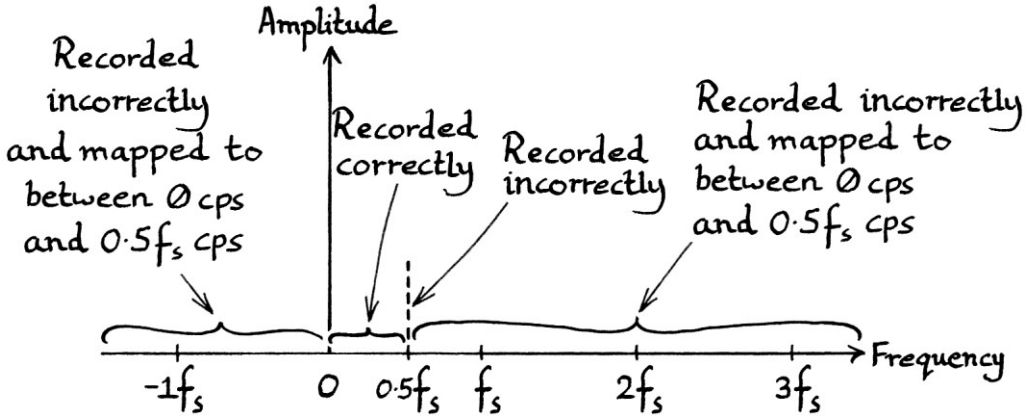
... are the same as those from:

$$z = e^{i(2\pi * 6t)}$$

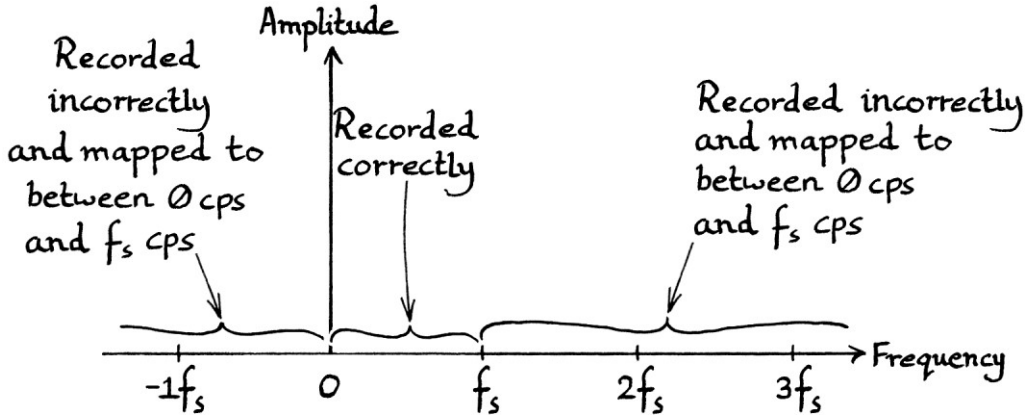
Whereas a *continuous* positive-frequency helix can never be described as a *continuous* negative-frequency helix (and vice versa), a *discrete* positive-frequency helix *can* be described as a *discrete* negative-frequency helix (and vice versa).

Frequency domain graphs

A frequency domain graph that shows how discrete **waves** are mapped to other frequencies is as so:



A frequency domain graph that shows how individual discrete **helices** are mapped to other frequencies is as so:



If we were sampling continuous **waves** at 10 samples per second, then the following continuous waves would produce the same samples as each other:

$y = \sin(2\pi * 4t)$
$y = \sin(2\pi * 14t)$
$y = \sin(2\pi * 24t)$
$y = \sin(2\pi * 34t)$
... and so on, and:
$y = \sin(2\pi * -6t)$
$y = \sin(2\pi * -16t)$
$y = \sin(2\pi * -26t)$
$y = \sin(2\pi * -36t)$
... and so on, which would be found with positive frequencies as so:

y = sin ((2π * 6t) + π)
y = sin ((2π * 16t) + π)
y = sin ((2π * 26t) + π)
y = sin ((2π * 36t) + π)
... and so on.

If we were sampling continuous *_helices_* at 10 samples per second, given here as Complex exponentials, then *only* the following continuous helices would produce the same samples as each other:

$z = e^{i(2\pi * 4t)}$

$z = e^{i(2\pi * 14t)}$

$z = e^{i(2\pi * 24t)}$

$z = e^{i(2\pi * 24t)}$

... and so on, and:

$z = e^{i(2\pi * -6t)}$

$z = e^{i(2\pi * -16t)}$

$z = e^{i(2\pi * -26t)}$

$z = e^{i(2\pi * -36t)}$

The negative frequencies would not be found as positive frequencies.

Sums of opposing frequencies

The rule that helices will be recorded correctly if they have a frequency from 0 cycles per second up to just under the sample rate is more complicated if we add or subtract two opposing-frequency helices to create a two-dimensional Sine wave or Cosine wave. In such cases, the characteristics of the created wave become more important than the helices that created it. If the created two-dimensional wave has a frequency between half the sample rate and the sample rate (exclusive), it will have the same samples as a wave between 0 cycles per second and half the sample rate (exclusive). We could say that it will be aliased to a frequency below half the sample rate. In that case, the Complex exponentials that created the original wave will be recorded as the pair of Complex exponentials that would create the aliased wave.

As an example, we will create a two-dimensional Sine wave in the three-dimensional Complex helix chart by subtracting one Complex exponential from another:

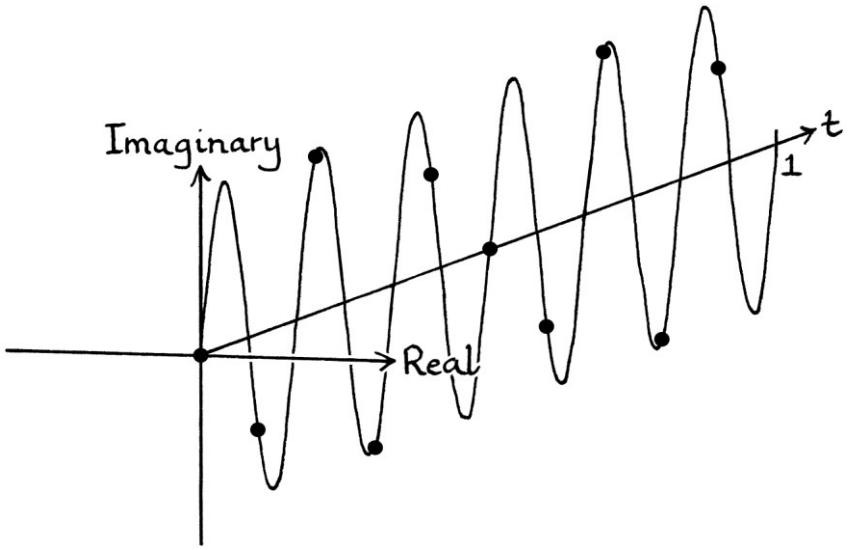
$$z = 0.5e^{i(2\pi * 6t)} - 0.5e^{i(2\pi * -6t)}$$

These Complex exponentials create an Imaginary Sine wave with an amplitude of 1i unit, a frequency of 6 cycles per second, and a phase of zero units. The wave has the formula:

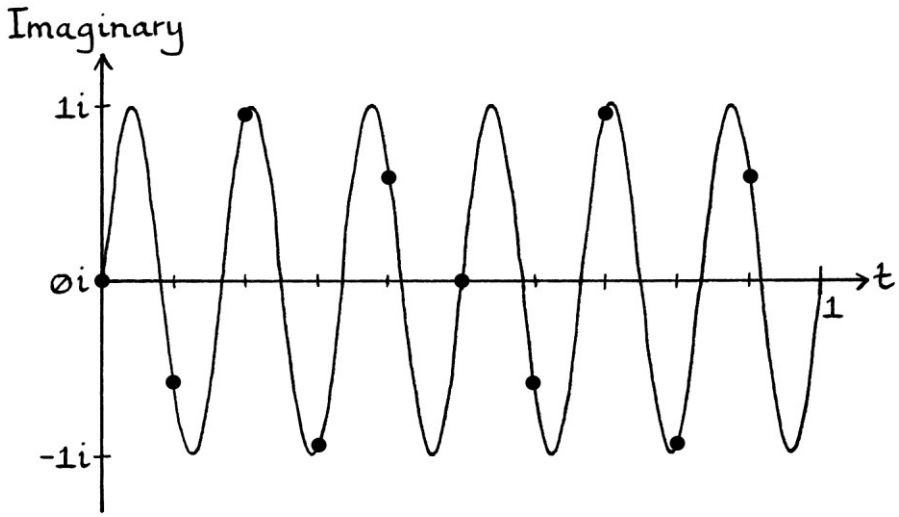
"y = 1i sin (2π * 6t)"

[If it were rotated by −90 degrees, it would be entirely Real, and it would have the formula: "y = 1 sin (2π * 6t)" but it keeps the pictures for this example simpler to keep it Imaginary.]

We will use a sample rate of 10 samples per second. In the Complex helix chart, the resulting Sine wave looks like this, with the samples marked on the curve:



Viewed from the side, the Sine wave looks like this:



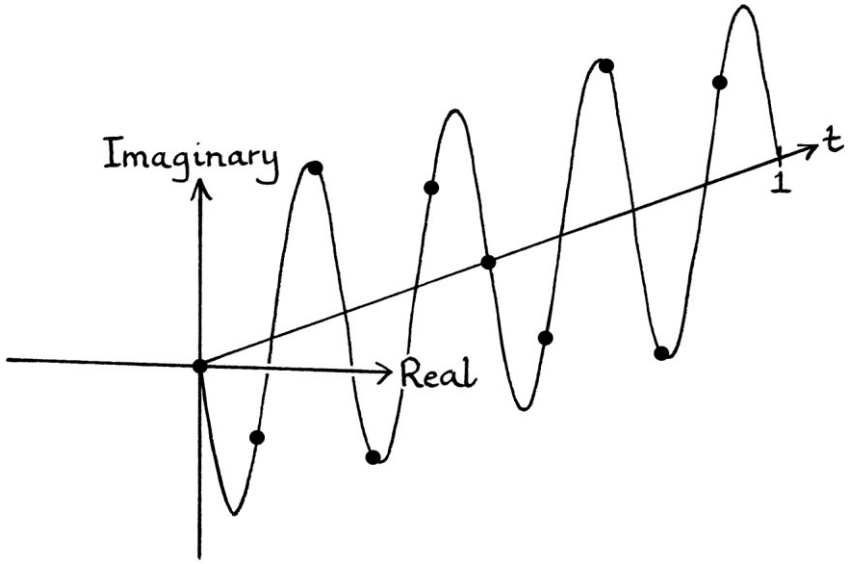
The samples from the signal are as so:
0 + 0i
0 − 0.5878i
0 + 0.9511i
0 − 0.9511i
0 + 0.5878i
0 + 0i
0 − 0.5878i
0 + 0.9511i
0 − 0.9511i
0 + 0.5878i

From what we know about waves and sampling, we can know that our two-dimensional wave will have identical samples to those of:
“y = 1i sin ((2π * 4t) + π)”

We can create that 4-cycle-per-second wave in the three-dimensional Complex helix chart with the following Complex exponential formula:

$$z = 0.5e^{i((2\pi * 4t)+\pi)} - 0.5e^{i((2\pi * -4t)+\pi)}$$

The wave looks like this:



This wave has the same samples as our 6-cycle-per-second wave. However, the samples are more representative of the 4-cycle-per-second wave than they are of the 6-cycle-per-second wave. The 6-cycle-per-second wave was undersampled.

The rule we have seen happening is as follows: when it comes to two-dimensional waves created by the addition or subtraction of a pair of opposing-frequency Complex exponentials, if the wave has a frequency above half the sample rate, then *the wave* will be sampled as if it had a frequency under half the sample rate. In which case, the underlying Complex exponentials will be sampled as if they represented that slower wave.

As we are using discrete helices, we can find duplicates of the Complex exponentials that have the same samples. For our example, we ended up with:

$$z = 0.5e^{i((2\pi * 4t) + \pi)} - 0.5e^{i((2\pi * -4t) + \pi)}$$

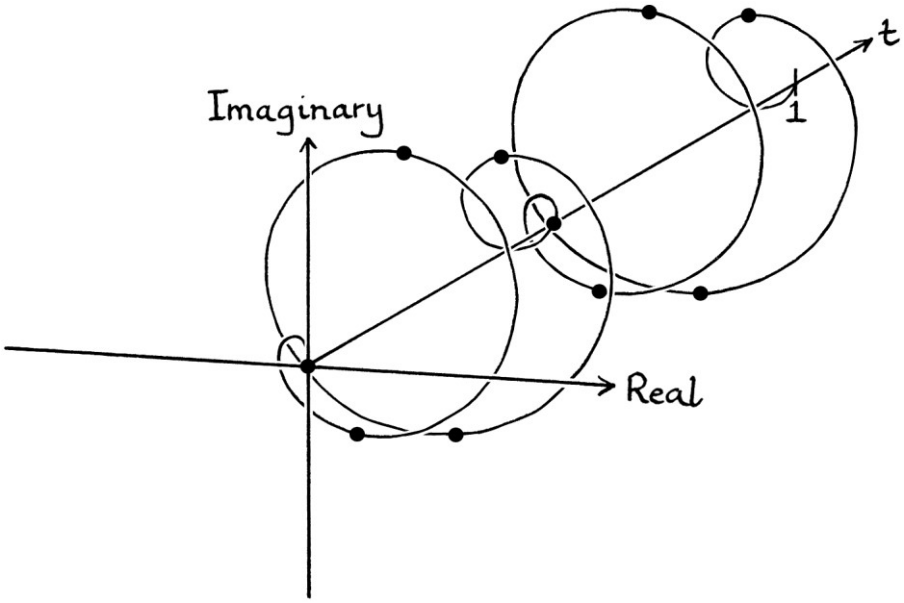
We can make the negative-frequency Complex exponential have a positive frequency by adding the sample rate to its frequency. [This is the "$f_0 = f_0 + (x * f_s)$" rule.] We will end up with the following Complex exponentials:

$$z = 0.5e^{i((2\pi * 4t) + \pi)} - 0.5e^{i((2\pi * 6t) + \pi)}$$

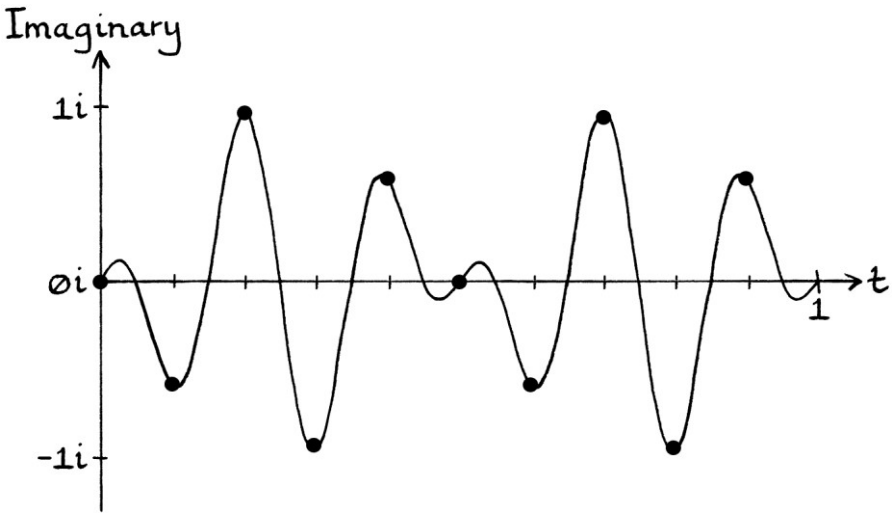
This is no longer a positive-frequency Complex exponential minus a negative-frequency Complex exponential. The frequencies of each Complex exponential are both positive and are different from each other. Therefore, the *continuous curve*

will not be a two-dimensional wave or signal. However, it will still contain the same samples for a sample rate of 10 samples per second. [Note how the second exponential has a frequency above half the sample rate – this is still fine because it is the two-dimensional wave that has been created that is important.]

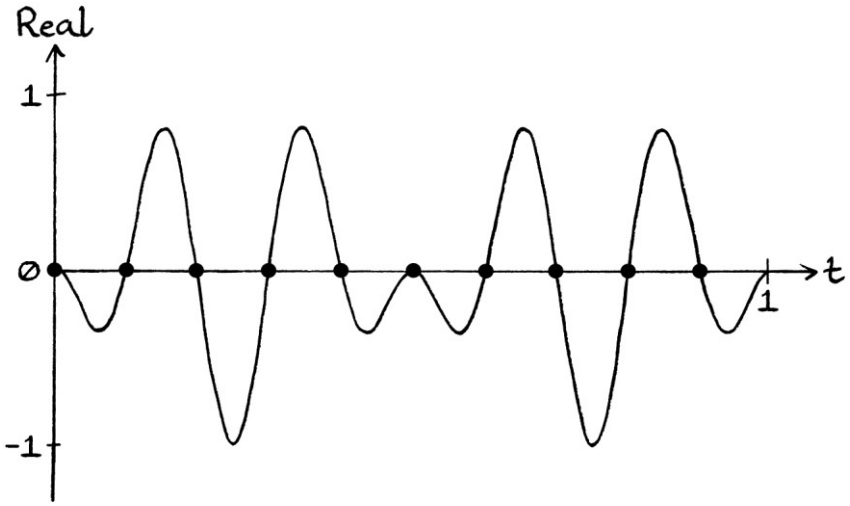
The curve looks like this (with dotted lines to emphasise the position of the samples):



Although this is a more complicated helix-like shape, when it is sampled at 10 samples per second, it has the same samples as our +4 and −4 cycle-per-second helix, and our +6 and −6 cycle-per-second helix. All the loops are missed out in the sampling process. The helix-like shape viewed from the side (the vertically derived signal or the Imaginary part of the signal) looks like this:



The helix-like shape viewed from underneath (the horizontally derived signal or the Real part of the signal) looks like this:

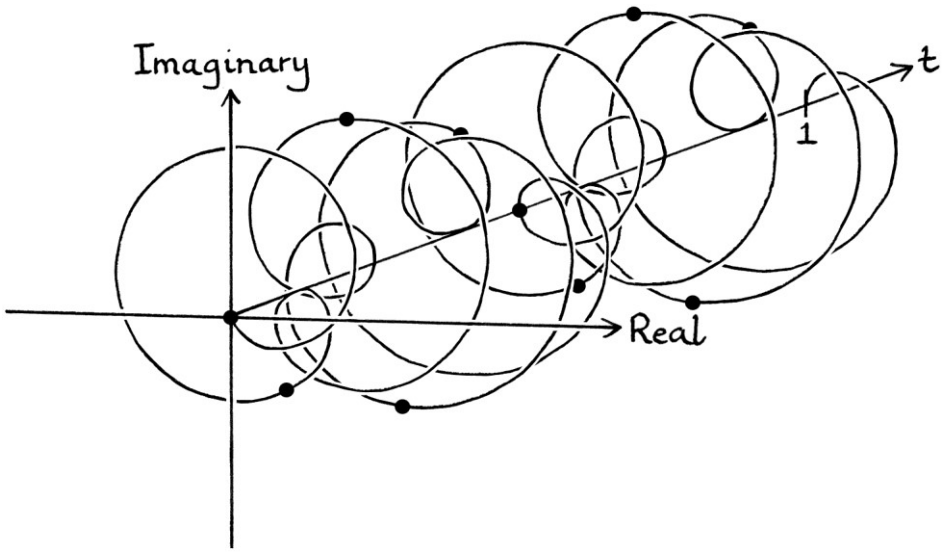


Note how the curve is at zero at all the places where samples are taken.

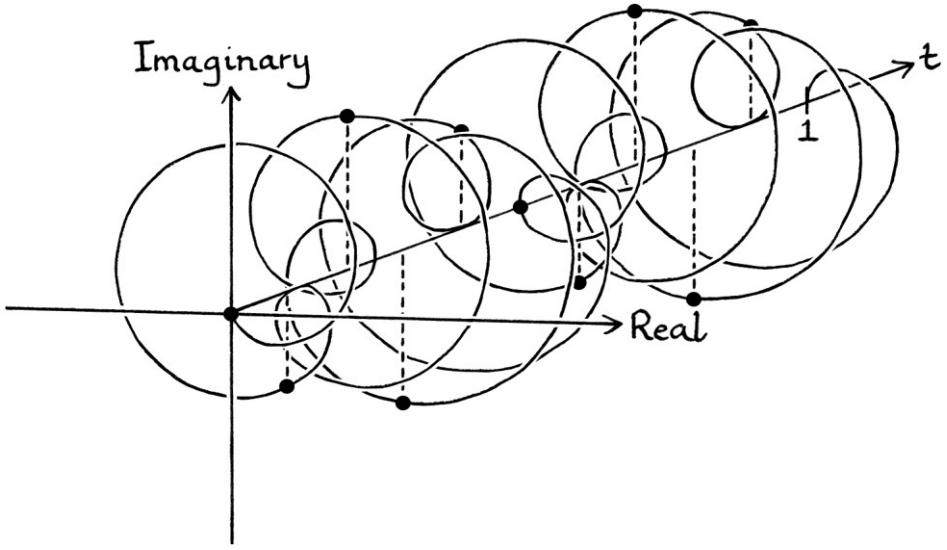
For interest's sake, we will add the sample rate to the frequency of the first Complex exponential of our pair. Because of the "$f_0 = f_0 + (x * f_s)$" rule, this will make no difference to the samples. We end up with this sum of Complex exponentials:

$$z = 0.5e^{i((2\pi * 14t) + \pi)} - 0.5e^{i((2\pi * 6t) + \pi)}$$

The resulting curve looks like this:



Although this shape looks very complicated, a perfect discrete Sine wave cuts across it lengthwise, and all the Real samples are zero. It is slightly easier to see how a Sine wave cuts across the shape if vertical lines are drawn from the t-axis to the curve:



Note that the:

$z = 0.5e^{i((2\pi * 14t) + \pi)} - 0.5e^{i((2\pi * 6t) + \pi)}$

... shape has a Complex exponential with a frequency above the sample rate. Therefore, on sampling the sum would end up as our previous pair of Complex exponentials:

$z = 0.5e^{i((2\pi * 4t)+\pi)} - 0.5e^{i((2\pi * 6t)+\pi)}$

Our original continuous Complex exponentials were:

$z = 0.5e^{i(2\pi * 6t)} - 0.5e^{i(2\pi * -6t)}$

... which, for a sample rate of 10 samples per second, produce the same samples as:

$z = 0.5e^{i((2\pi * 4t) + \pi)} - 0.5e^{i((2\pi * -4t) + \pi)}$

... and:

$z = 0.5e^{i((2\pi * 4t) + \pi)} - 0.5e^{i((2\pi * 6t) + \pi)}$

... and:

$z = 0.5e^{i((2\pi * 14t) + \pi)} - 0.5e^{i((2\pi * 6t) + \pi)}$

We can draw some inferences from this example:

- If we have a sum of two identical helices with opposing frequencies creating a two-dimensional wave in the Complex helix plane, and that wave has a frequency between half the sample rate and the sample rate (exclusive), then the samples will be the same as those from the sum of two helices creating a wave with a frequency between 0 cycles per second and half the sample rate (exclusive). [The helices themselves will have frequencies between the negative of the sample rate and the sample rate, and using the "$f_0 = f_0 + (x * f_s)$" rule, it will be possible to rephrase them both to have frequencies between 0 cycles per second and the sample rate (exclusive).]

- The sum of two identical helices with opposing frequencies can be rephrased to be the sum of two positive-frequency helices with frequencies between 0 cycles per second and the sample rate (exclusive), and they will still produce the same samples.

- The sum of two continuous helices, both with positive frequencies or both with negative frequencies, might end up producing a two-dimensional discrete wave in the Complex helix chart when sampled.

Summary for Complex exponentials

- A Complex exponential will be recorded correctly if its frequency is from zero cycles per second up to just below the sample rate.

- The discrete version will have duplicates that are higher or lower by an integer multiple of the sample rate.

- For a pair of Complex exponentials with opposing frequencies, if the wave they create has a frequency above half the sample rate, then the samples will be the same as those from a wave with a frequency below half the sample rate. We can then use a slower pair of Complex exponentials to express that wave. The same situation can arise for some pairs of Complex exponentials that do not have opposing frequencies.

Conclusion

In this chapter, we have seen a great deal about the potential problems that occur when sampling signals. There is still more to know about sampling, but this chapter will have provided a reasonable foundation for further learning.

Chapter 43: Discrete Fourier series

In Chapter 18, we learnt how to perform Fourier series analysis and we learnt why the process worked. In Chapter 30 on calculus, we saw formulas that described the process of Fourier series analysis. In those chapters, we were performing *continuous* Fourier series analysis because our waves and signals were continuous waves and signals. Discrete Fourier series analysis is very similar to continuous Fourier series analysis, but the nature of discrete signals makes some aspects slightly more complicated. [We performed discrete Fourier series analysis, without acknowledging the fact, in Chapter 18 when we analysed a picture of wave. In Chapter 42, we were essentially using discrete Fourier series analysis all the time.]

Continuous Fourier series analysis

As a reminder, here are the steps needed to perform *continuous* Fourier series analysis. In other words, these are the steps for finding the constituent waves in a continuous periodic signal. These steps were first explained in Chapter 18, and the calculus aspects were explained in Chapter 30.

- Calculate the mean level of the signal. One way of doing this is to use integration to calculate the area between the signal's curve and the t-axis for one cycle of the signal, and then to divide that by the length of one cycle (the period). Subtract the mean level from the signal, and make a note of the mean level for later.

- Find the frequency of the signal, and make a list of test frequencies that are sequential integer multiples of this number.

- For every test frequency in the list, multiply the signal by each of the following test waves in turn, swapping "f" for the frequency being tested:
  "$y = 2 \sin (2\pi * f * t)$"
  ... and:
  "$y = 2 \cos (2\pi * f * t)$"

For each frequency being tested:

- Find the mean level of the signal created from the first multiplication. This can be done with integration to calculate the area between the signal's curve and the t-axis for one cycle, and then dividing that by the length of one cycle. We will call this the "first mean level".

- Find the mean level of the signal created from the second multiplication. This can be done in the same way. We will call this the "second mean level".

- If both mean levels are zero, that frequency does not exist in the signal. Otherwise, the amplitude of the wave for that frequency will be the square root of the sum of the square of these mean levels. The phase of the wave for that frequency will be the arctan of "the second mean level divided by the first mean level" (and we need to check that the arctan result is the one that we want). If either mean level is zero, we can skip using arctan and just imagine the mean levels as the coordinates of a phase point on a circle – the first mean level will be the x-axis coordinate, and the second mean level will be the y-axis coordinate.

Do not forget the mean level, if any, that was removed from the signal at the start.

When performing continuous Fourier series analysis, we continue until we have the constituent waves that when added together make the signal that we are analysing. If our signal were not equal to a sum of waves, then, in theory, we would continue forever. In practice, we would choose an arbitrary frequency at which to stop.

Discrete Fourier series analysis

The process of discrete Fourier series analysis is, in essence, identical to continuous Fourier series analysis, except it uses discrete signals instead of continuous signals. For sample rates that are significantly faster than the highest constituent frequency, the differences are irrelevant if we only test for waves with frequencies less than half the sample rate.

Useful facts

Some useful facts about discrete Fourier series analysis are:

- The test waves must have the same sample rate as the discrete signal that we are analysing.

- We have to base our test waves on the frequency of the *discrete* signal. They should not be based on the frequency of the continuous signal from which the discrete signal came. Nor should the test waves be based on the frequency that we think the discrete wave *should* have. [For example, just because we might see the outline of cycles by joining up the samples on a graph does not mean that we should use those perceived cycles as the basis for our test waves.]

- We do not need to use integration to calculate mean levels – we can just average the samples in one period (one cycle) of the signal for which we want the mean level.

- As we saw in Chapter 42 on sampling, we do not need to test for frequencies that are *above* half the sample rate. Original frequencies that were above half the sample rate would have become recorded as frequencies equal to or below half the sample rate, so there is no point looking for them after the halfway point. We will find their aliases below (or at) half the sample rate. If we continued to test for frequencies after the frequency equal to half the sample rate, we would just find waves with faster frequencies that had the same samples as the waves that we had already found. These would not be part of the sum that makes up the signal being analysed, and if we counted them as such, the sum of the discovered waves would be incorrect.

- We still need to test for frequencies *equal* to half the sample rate, but only if a frequency equal to half the sample rate will be one of our test frequencies. Not all combinations of fundamental frequencies and sample rates will end up with the frequency equal to half the sample rate being in the list of test frequencies. If our sample rate is 32 samples per second and the frequency of the signal that we are analysing is 1 cycle per second, then we will test for the frequencies of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16 cycles per second. The frequency of 16 cycles per second is equal to half the sample rate. On the other hand, if our sample rate is 33 samples per second, and the frequency of the signal we are analysing is 1 cycle per second, then we will test for the same frequencies, but none of them will be equal to half the sample rate (which is 16.5 cycles per second).

- If we find a constituent wave for the frequency equal to half the sample rate, we must halve its amplitude. As explained in Chapter 42, if the wave was not removed from the signal in the sampling process, we might find the correct wave or we might find a wave that contained the same samples. Whether this difference matters or not depends on what we are doing. A well-sampled signal should not contain frequencies equal to half the sample rate. However, we cannot know if a signal was well sampled or not until we have analysed it.

- Perhaps most importantly, discrete Fourier series analysis will find a set of constituent waves that will recreate the analysed signal's samples *exactly*.

If you understand normal (continuous) Fourier series analysis, then knowing the above ideas is enough to perform discrete Fourier series analysis. The hardest part of understanding discrete Fourier series analysis concerns the formulas that describe the process.

The process

The process of discrete Fourier analysis, expressed in words, is as follows:

- Calculate the frequency of the discrete signal and see how many samples are in one period (one cycle) of the signal.

- Calculate the mean level of the signal by finding the average of the samples over one period of the signal. Remove the mean level from the signal, and make a note of it for later.

- Make a list of test frequencies that are sequential integer multiples of the frequency of the discrete signal. The list should start at the frequency of the discrete signal and end at the frequency equal to half the sample rate, or just before that if there is no integer multiple of the frequency of the signal that is equal to half the sample rate.

- For every test frequency in the list, multiply the signal by each of the following test waves in turn, swapping "f" for the frequency being tested and "T_s" for the sampling period:
  "$y = 2 \sin (2\pi * f * T_s * n)$"
  ... and:
  "$y = 2 \sin (2\pi * f * T_s * n)$"

For each frequency being tested:

- Find the mean level of the signal created from the first multiplication. This will be the average of the samples over one period. We will call this the "first mean level".

- Find the mean level of the signal created from the second multiplication. This will also be the average of the samples over one period. We will call this the "second mean level".

- If both mean levels are zero, that frequency does not exist in the signal. Otherwise, the amplitude of the wave for that frequency will be the square root of the sum of the square of the two mean levels. If the test frequency is equal to half the sample rate, we halve the calculated amplitude. The phase of the wave for that frequency will be the arctan of the second mean level divided by the first mean level (and we need to check that the arctan result

is the one that we want). If either mean level is zero, we can skip using arctan and just imagine the mean levels as the coordinates of a phase point on a circle – the first mean level will be the x-axis coordinate, and the second mean level will be the y-axis coordinate.

Do not forget the mean level, if any, that was removed from the signal at the start.

[Multiplication of a wave and a signal involves multiplying corresponding samples by each other. For example, we multiply the first sample from the original signal by the first sample from the wave, and that becomes the first sample of our new signal. We then multiply the second sample from the original signal by the second sample from the wave, and that becomes the second sample of our new signal. We continue until we have gone through all the samples from the wave and the signal. When doing Fourier series analysis, all our test waves will have frequencies that are integer multiples of that of our original discrete signal. Therefore, in practice, we can stop the multiplication after completing one cycle of the original signal – we know that all the later cycles of the original signal and the waves will be identical, so our new signal will repeat after this point.]

Formulas

As a reminder of the formulas for *continuous* Fourier series analysis, we will look again at the longer group of formulas that I gave in Chapter 30. These formulas have two parts. The first part states that a periodic signal is made up of the sum of waves of various amplitudes and phases, and of frequencies that are integer multiples of the frequency of the signal being analysed. The second part consists of formulas that describe how to calculate those waves. The formulas are as so:

$$f(t) = \sum_{k=0}^{\infty} c_k \, sin \, ((2\pi * f * k * t) + \phi_k)$$

... where:

$$c_0 = \frac{1}{T}\int_0^T f(t) \; dt$$

$$\phi_0 = 0.5\pi$$

$$g(t) = f(t) - c_0 \, sin\,((2\pi * f * 0 * t) + \phi_0)$$

$$a_k = \frac{2}{T}\int_0^T g(t) * (sin\,(2\pi f k t))\,dt$$

$$b_k = \frac{2}{T}\int_0^T g(t) * (cos\,(2\pi f k t))\,dt$$

$$c_k = \sqrt{(a_k)^2 + (b_k)^2}$$

$$\phi_k = arctan\,(b_k \,/\, a_k)$$

The details of the first formula are:

- "f(t)" refers to the continuous periodic signal that we are discussing.
- "Σ" (the Greek letter sigma) is shorthand for a series of additions involving the part to the right, where each addition involves filling in a different value for "k".
- "k" counts through the different constituent waves from wave number 0 (which will represent the mean level) upwards. It is always an integer, and it increases in steps of 1.
- "c_k" is the amplitude of each wave, where the letter "k" distinguishes between the amplitudes for the different waves, but does not specify their actual value. The different values of "c" might be the same as each other, different from each other, or even zero.
- "f" is the frequency of the signal we are analysing. It is the "fundamental frequency".
- The multiplication "f * k" means that each wave in the sum will have a frequency that is an integer multiple of that of the fundamental frequency. When "k" is zero, this will produce a frequency of zero cycles per second, and the wave will act as a mean level.
- "ϕ_k" is the phase of each wave, where the letter "k" distinguishes between the phases for the different waves, but does not specify their values. The phases of each wave might be the same as each other, or different from each other.

The details for the other formulas are:

- "g(t)" refers to the signal we are discussing after we have removed the mean level.
- "a_k" is the mean level of the result of multiplying "g(t)" by a Sine wave with a frequency dictated by "f" and the value of "k" for the particular wave in the sum. The mean level is calculated with integration. Although the process can be calculated with a Sine wave with an amplitude of 2 units, here we have changed the amplitude to 1 unit and put a multiplication by 2 before the integral. This produces the same result, but it is slightly easier to read.
- "b_k" is the mean level of the result of multiplying "g(t)" by a Cosine wave with a frequency dictated by "f" and the value of "k" for the particular wave in the sum. The mean level is also calculated with integration. Again, although the process can be calculated with a Cosine wave with an amplitude of 2 units, here we have changed the amplitude to 1 unit and put a multiplication by 2 before the integral.

Refer to Chapter 30 again if these formulas are not clear.

The synthesis formula

To convert the formula to work with discrete signals, we will first look again at the "synthesis" formula for continuous signals. As we have seen in part one of this book, the synthesis formula is the standard name for what could also be called the "descriptive" formula. Either we could say that it *describes* the nature of a periodic signal, or we could say that it shows how to *synthesise* a periodic signal by adding waves.

To convert the formula to work with discrete signals, we will first look again at the synthesis formula for continuous signals. This is the formula that is intended to state that any continuous periodic signal is equal to, or approximately equal to, the sum of pure waves of various amplitudes and phases and different frequencies. It does not actually tell us how to calculate the amplitudes and phases – it just describes the situation.

$$f(t) = \sum_{k=0}^{\infty} c_k \, sin\left((2\pi * f * k * t) + \phi_k\right)$$

To make this "Σ" sum ("Sigma sum") work with discrete signals, we have to alter the part to the right of the "Σ" so that it describes a general discrete wave. The general formula for a discrete wave is as so:

$$y = h + A\ sin\ ((2\pi * f *\ T_s * n) +\ \phi)$$

[In Chapter 18 on continuous Fourier series analysis, we counted the different waves in the Fourier "Σ" sum using "n". In Chapter 30 on calculus, we counted the different waves using the letter "k". The switch to using "k" was so that when we use the Fourier sum with discrete waves, we are not using "n" to mean two different things.]

Now we can alter the formula so that it applies to discrete signals. The first thing to realise is that instead of referring to our signal with "f(t)", which implies a continuous signal, we need to refer to it with "f[n]". Note how this has square brackets instead of rounded brackets. A word or letter followed by square brackets indicates a discrete signal (such as "mysignal[n]"), or a particular sample from a discrete signal (such as "mysignal[3]"). The same idea is used in programming arrays, where an array in programming is just a list of values in the same way that a discrete signal is just a list of values. Note that not everyone in signal processing uses this convention. The term "f[n]" essentially means "the discrete function that we are discussing that involves 'n'".

Our *discrete* synthesis formula becomes:

$$f[n] = \sum_{k=0}^{\infty} c_k\ sin\ ((2\pi * f * k * T_s * n) + \phi_k)$$

... where:

- "f[n]" means the periodic *discrete* signal that we are discussing.
- "c_k" is the amplitude of each wave.
- "f" is the frequency of the original discrete signal, which we could also call the "fundamental frequency".
- "k" is the counter that identifies each amplitude and phase, and which sets each frequency to be an integer multiple of that of the discrete signal. It identifies and describes each individual Sine wave in the sum of waves that will make up the signal. It will always be an integer and it increases in steps of 1.
- "T_s" is the sampling period, which is the time between the samples.

- "n" is analogous to the time in a continuous wave formula. It is the sample number starting from zero and increasing in steps of 1. It is always an integer. It essentially means every integer from 0 upwards, in the same way that "t" for time essentially meant every moment in time from t = 0 upwards.
- "ϕ" is the phase in radians.
- The mean level is the wave that has a zero frequency, which is the wave when k = 0.

Note that this formula is not in its finished form yet. It has a huge flaw that we will examine in a moment.

In the context of Fourier series analysis, the ultimate meaning of the sum written out in full is:

The *discrete* periodic function "f[n]" is *exactly* equal to the sum of:
$c_0 \sin((2\pi * f * 0 * T_s * n) + \phi_0)$ [This acts as the mean level, if any.]
+
$c_1 \sin((2\pi * f * 1 * T_s * n) + \phi_1)$
+
$c_2 \sin((2\pi * f * 2 * T_s * n) + \phi_2)$
+
$c_3 \sin((2\pi * f * 3 * T_s * n) + \phi_3)$
+
$c_4 \sin((2\pi * f * 4 * T_s * n) + \phi_4)$
... and so on.

An example

To make the meaning of the above synthesis formula slightly clearer, we can create an actual discrete signal with proper values. We will say that the discrete signal has a frequency of 2 cycles per second, a sampling period of 0.01 seconds (so a sample rate of 100 samples per second), and a mean level of 7 units. We will say that the discrete signal consists of a mean level and three different waves. The mean level is portrayed as a wave with zero frequency. [Note that these are the waves that we are adding together, and not the waves that we found through analysis.]

f[n] =
7 sin ((2π * 2 * 0 * 0.01n) + 0.5π) [This ends up as: 7 sin (0.5π) = 7 units]
1 sin (2π * 2 * 1 * 0.01n)
3 sin (2π * 2 * 2 * 0.01n)
0 sin (2π * 2 * 3 * 0.01n) [This ends up as nothing.]
0 sin (2π * 2 * 4 * 0.01n) [This ends up as nothing.]
2 sin ((2π * 2 * 5 * 0.01n) + 1.2π)
0 sin (2π * 2 * 6 * 0.01n) [This ends up as nothing.]
... and all the later waves will have amplitudes of zero units, so will not contribute to the sum.

We can write the discrete waves more concisely, while ignoring those with zero amplitudes:

f[n] =
7 sin (0.5π)
1 sin (2π * 2 * 0.01n)
3 sin (2π * 4 * 0.01n)
2 sin ((2π * 10 * 0.01n) + 1.2π)

An improvement

The discrete synthesis formula that we have at the moment is:

$$f[n] = \sum_{k=0}^{\infty} c_k \, sin \, ((2\pi * f * k * T_s * n) + \phi_k)$$

This adds up waves of frequencies that are all integer multiples of the frequency of the signal (the fundamental frequency). It multiplies the fundamental frequency by "k", and "k" rises from zero up to infinity. If we were dealing with continuous signals and waves, having "k" rise to infinity would not be a problem (although it would make things difficult in practice). However, we are now dealing with *discrete* waves and signals, which means that we would have found every constituent wave once we had tested for the frequency equal to half the sample rate. If we continued testing, we would still find more waves, but they would be duplicates of the ones we had already found. If we continued testing forever, we would always end up with an infinite number of constituent waves for every signal, which, obviously, would be incorrect.

In our synthesis "Σ" sum formula, we definitely do not want "k" to count from 0 to infinity. We only need it to count to half the sample rate. We can alter our formula to take this into account. We want the frequencies in the "Σ" sum never to rise above this value. We can write this desire in a *slightly* mathematical way as so:

This must be the case:
$maximum\ constituent\ frequency = 0.5f_s$
... where:
"f_s" is the sample rate

The "Σ" sum formula that we are using bases the constituent frequencies on integer multiples of the fundamental frequency of the original signal. The letter "f" (with no subscript) represents the fundamental frequency of the signal. We calculate integer multiples of this by multiplying it by "k". Therefore, the actual frequency of each constituent wave is given by "f * k". We never want the result of "f * k" to be above half the sample rate. Therefore, the maximum frequency can be expressed as the moment when:

$f * k = 0.5f_s$

We can put this into our "Σ" sum as so: [It is easier to read in a bigger font.]

$$f[n] = \sum_{k=0}^{f*k=0.5f_s} c_k\ sin\ ((2\pi * f * k * T_s * n) + \phi_k)$$

This formula means:

> "The discrete periodic signal that we are discussing, which uses the variable "n", is equal to a sum of discrete Sine waves of various unspecified amplitudes and phases, and with frequencies that are integer multiples of the fundamental frequency. The integer multiples of the fundamental frequency range from 0 cycles per second up to, and including, half the sample rate."

[Note that this describes a discrete signal as being the sum of discrete waves – it does not mention anything to do with how they might be analysed yet. Another point is that a *sensibly* sampled signal would not contain frequencies equal to half the sample rate, but we cannot know if a signal was sampled sensibly or not.]

Another improvement

We can tidy up the part above the "Σ". At the moment, we have:
$f * k = 0.5f_s$
... which means that we stop counting waves when "k" multiplied by the fundamental frequency is equal to half the sample rate. We can rearrange it to make it clearer:
$k = 0.5f_s / f$

That means that we stop when "k" is equal to half the sample rate divided by the fundamental frequency (or just before if there is no value of "k" for which this is true). We can put this into the synthesis formula (where we do not need to include the "k =" part because it is implied):

$$f[n] = \sum_{k=0}^{0.5f_S / f} c_k \, sin\left((2\pi * f * k * T_s * n) + \phi_k\right)$$

[Note that this is still not our finished formula. We can improve it, as we will see later.]

A different maximum

Sometimes you will see the synthesis formula written so that "k" stops *before* the frequency equal to half the sample rate. This implies that any periodic discrete signal is equal to a sum of waves with frequencies *below* half the sample rate, which is not true. We saw examples in the previous chapter, and we will see examples later in this chapter. A periodic signal might contain a frequency equal to half the sample rate, and we must test for the frequency equal to half the sample rate too. I think the mistake comes from assuming that every continuous signal is sampled correctly when it is turned into a discrete signal. If we only sample continuous signals so that the sample rate is always above half the maximum constituent frequency, then we would never need to test for frequencies equal to half the sample rate. In which case, the incorrect formula would work. However, if we want to calculate the sum of waves that has the same samples as *any* discrete signal, then we have to test for the frequency equal to half the sample rate. We might not find the exact constituent wave, but we will find a wave with the same samples.

Yet another improvement

At the moment, our synthesis formula is as so:

$$f[n] = \sum_{k=0}^{0.5f_s \,/\, f} c_k \, sin\left((2\pi * f * k * T_s * n) + \phi_k\right)$$

We can make the part above the "Σ" more succinct by realising an interesting fact about periodic discrete signals and waves:

> "If we divide the sample rate of a signal by its fundamental frequency, we will end up with the number of samples in one period of the signal."

The sample rate is the number of samples in one second. The fundamental frequency is the number of cycles in one second. Therefore, if we divide the sample rate by the fundamental frequency, we will end up with the number of samples in one cycle (one period). [This may or may not be obvious.] We will look at some examples.

If the sample rate is 10 samples per second, and the signal being analysed has a fundamental frequency of 1 cycle per second, then there are:
10 ÷ 1 = 10 samples in one cycle of that signal.

If the sample rate is 8 samples per second, and the signal being analysed has a fundamental frequency of 0.5 cycles per second, then there will be:
8 ÷ 0.5 = 16 samples in one cycle of that signal.

If the sample rate is 12 samples per second, and the signal being analysed has a fundamental frequency of 0.2 cycles per second, then the number of samples in one cycle of the signal will be:
12 ÷ 0.2 = 24 samples.

These examples confirm that the sample rate divided by the fundamental frequency of the signal gives the number of samples in one period of the signal. On the top of the "Σ" in our synthesis formula's "Σ" sum, we have the sample rate divided by the fundamental frequency in the form of "f_s / f". Therefore, we can swap the "f_s / f" part for the number of samples in one period of the signal. The conventional way of portraying the number of samples in one period of the signal is to use the letter "N". [This might be confusing to start with because we are already using a lower-case "n" to identify the different samples. As always, such

things will become less confusing as you become more used to them.] Our synthesis formula becomes:

$$f[n] = \sum_{k=0}^{0.5N} c_k \, sin\left((2\pi * f * k * T_s * n) + \phi_k\right)$$

This makes our formula slightly more succinct and less clumsy. The formula now states that we count from "k = 0" up to "k" being equal to half the number of samples within one period of the signal that we are analysing. This has exactly the same overall meaning as our previous synthesis formula, but it is a different way of thinking about what we are doing.

One interesting consequence of our new formula is that it means that any discrete periodic signal is equal to a mean level added to half as many constituent waves as there are samples within one period of the signal. To put this in terms of the analysis, for any periodic signal we only need to test for the mean level, and then half as many waves as there are samples in one period.

If there are 40 samples in one period of the signal, then we only need to calculate the mean level and then test for 20 waves. The maximum number of (non-mean-level) constituent waves that could be in that discrete signal is 20. There could not be any more.

If there are 26 samples in one cycle of a signal, then we only need to calculate the mean level and then test for 13 waves. There cannot be more than 13 (non-mean-level) constituent waves in the signal.

A summary of this important idea is:

> "Any periodic discrete signal can be described as the sum of a mean level and half as many constituent waves as there are samples in one period."

Note that the mean level might be zero units, and some of the constituent waves might have zero amplitudes. Supposing there were an odd number of samples in one period, then we would calculate the mean level, and then test for as many waves as the highest integer below half the number of samples in one period. For example, if there were 33 samples in one period, we would calculate the mean level, and then test for 16 constituent waves.

Although this idea is based on the "f * k = $0.5f_s$" idea, it is quite profound. Without seeing the path to get to the idea, one might not expect that the number of samples in one cycle of a signal would have any relation to the number of non-zero-frequency constituent waves that make up that signal.

The other type of synthesis formula

The above synthesis formula specifically calculates the frequency of each constituent wave. The other type of synthesis formula merely has "f" with a subscript of "k". [This was explained in Chapter 18, in which I used the subscript "n", and in Chapter 30, in which I used the subscript "k".] In such formulas, the frequencies are not specified, and the "k" just identifies different frequencies. Ultimately, "f_k" means "the frequency of constituent wave number 'k', whatever that frequency might be." If we used this vaguer synthesis formula, we would have:

$$f[n] = \sum_{k=0}^{0.5N} c_k \, sin\left((2\pi * f_k * T_s * n) + \phi_k\right)$$

The formula means:

> "The periodic discrete signal that we are discussing, which uses the variable "n", is equal to a sum of discrete Sine waves of various unspecified amplitudes, frequencies and phases, with the unstated rule that the frequencies are different. We will need to test for 0.5N + 1 waves (where "N" is the number of samples in one period) to find the constituent waves."

Personally, I think this form of synthesis formula is unwisely vague, so we will ignore it and concentrate on the other form, where "f * k" specifies the actual frequencies in the sum.

The analysis formulas

The synthesis formula describes the nature of a periodic signal. It does not help us calculate the amplitude or phase of each wave. To know how to calculate these, we need to combine the formula with what are called the "analysis" formulas. These formulas explain the process for calculating the constituent waves of a periodic signal.

As we have already been seen in this chapter, if we use "k" to count through the different constituent waves, then the steps for analysing a *continuous* periodic signal as described with formulas are as so:

$$c_0 = \frac{1}{T}\int_0^T f(t)\ dt$$

$$\phi_0 = 0.5\pi$$

$$g(t) = f(t) - c_0\ sin\ ((2\pi * f * 0 * t) + \phi_0)$$

$$a_k = \frac{2}{T}\int_0^T g(t) * (sin\ (2\pi f k t))\ dt$$

$$b_k = \frac{2}{T}\int_0^T g(t) * (cos\ (2\pi f k t))\ dt$$

$$c_k = \sqrt{(a_k)^2 + (b_k)^2}$$

$$\phi_k = arctan\ (b_k\ /\ a_k)$$

When it comes to discrete signals, we still need to calculate the mean levels (a_k and b_k). The "calculus" way of doing this involves finding the area under the curve for one period (one cycle), and then dividing it by the length of that period. As we are using discrete signals, a method using the same idea would involve adding up the areas of the *rectangles* made up by the samples over one period, and then dividing the total area by the length of that period. Each rectangle will have a base equal to the sampling period, which is the amount of time between the samples.

If we have a signal consisting of these samples over one period:
0, 1, 3, 9, 5, −4, −8, −2, −1, −0.5
... and the sampling period is 0.1 seconds, then the total area under the "curve" would be:
0 * 0.1
+
1 * 0.1
+
3 * 0.1
+
9 * 0.1
+
5 * 0.1
+
−4 * 0.1
+
−8 * 0.1
+
−2 * 0.1
+
−1 * 0.1
+
−0.5 * 0.1

The total area is 0.25 square units. The mean level will be this area divided by the length of the period. [The sampling period is 0.1 seconds. There are 10 samples in a cycle. Therefore, one period or cycle lasts for 10 * 0.1 = 1 second. Therefore, the mean level is:
0.25 ÷ 1 = 0.25 units

It might be clear that, in this situation, there is no advantage in using calculus or in thinking in terms of calculus. It is quicker to calculate the mean levels by finding the average value of the samples over one period. This is the sum of the samples over one period divided by the number of samples. We do not need to know the sampling period (or the sample rate) to calculate this. For the above list of samples, we would calculate this:
(0 + 1 + 3 + 9 + 5 − 4 − 8 − 2 − 1 − 0.5) ÷ 10 = 0.25 units.

When analysing a discrete signal, the mean level is just the average of the samples over one period. We do not need to use an integral, but we can express the idea with a "Σ" sum. If, for example, there were ten samples in one cycle, we could express this mathematically as so:

$$mean\ level = \frac{1}{10} * \sum_{n=0}^{9} f[n]$$

This means that we add up ten samples from our signal from n = 0 up to n = 9, and then divide the total by 10 to get the average. [Remember that we start counting from 0, so the tenth sample is sample number 9.] The sum is shorthand for:

f[0] + f[1] + f[2] + f[3] + f[4] + f[5] + f[6] + f[7] + f[8] + f[9]
... all multiplied by "1 / 10" (so divided by 10).

We can also think of it as meaning:

sample[0] + sample[1] + sample[2] + sample[3] + sample[4]
+ sample[5] + sample[6] + sample[7] + sample[8] + sample[9]
... all divided by 10.

We can make this formula more general by identifying the number of samples in a period with the letter "N". The formula becomes:

$$mean\ level = \frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

... where:

- "N" is the number of samples in one period.
- "N – 1" is the final sample number in one period. This is one less than "N" because we start counting at zero.
- "n" is the counter that counts through each sample.

The sum adds up each sample of the signal from when "n" is zero up to, and including, when "n" is "N – 1", which is the last sample of the first cycle (the last sample of the first period). After adding up the samples, we divide them by the number of samples – we find the average. Therefore, if the last sample is "sample number N – 1", there must be "N" samples in the sum, and we need to divide the sum by "N" to get the average.

Going back to our analysis, if we start with the discrete signal "f[n]", and there are "N" samples in one cycle, then the mean level of the whole signal, and therefore, the amplitude of the zero-frequency wave will be:

$$\frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

We now know enough to give a formula that calculates the zero-frequency wave's amplitude and phase:

$$c_0 = \frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

$$\phi_0 = 0.5\pi$$

[The amplitude is the mean level, the frequency is zero, and the phase is 0.5π radians. This means that we are using a long-winded way of writing out the mean level by doing it in the form of a wave. If we were using Cosine waves, the phase would be zero radians.]

The general wave formula for the zero-frequency wave (the mean level) is:

$$c_0 \, sin \, ((2\pi * f * 0 * T_s * n) + \phi_0)$$

As we know the phase will be 0.5π radians, we could give it slightly more specifically as:

$$c_0 \, sin \, ((2\pi * f * 0 * T_s * n) + 0.5\pi)$$

However, for the sake of consistency with the layout of the wave formula in the synthesis formula, we will keep it vague and use "ϕ_k".

We then remove this mean level from the signal to produce a second signal, which we will call "g[n]".

$$g[n] = f[n] - \, c_0 \, sin \, ((2\pi * f * 0 * T_s * n) + \phi_0)$$

This means that for every sample in "f[n]", we subtract the corresponding sample of the wave portraying the mean level, and set the corresponding sample of "g[n]" to the result. Every sample in the wave portraying the mean level will have the

same value – they will each have the value "c_0". Therefore, if we were not insisting on portraying the mean level as a wave, we could say:

Every value of "g[n]" is the corresponding value of "f[n]" minus "c_0".

In other words:
g[0] = f[0] – c_0
g[1] = f[1] – c_0
g[2] = f[2] – c_0
g[3] = f[3] – c_0
... and so on.

One good reason for keeping the mean level in terms of a wave is that it reinforces how we are dealing with individual values spaced at intervals of the sampling period. Although such an idea is implied by how "g[n]" and "f[n]" use square brackets, having a wave formula that mentions the sampling period makes the meaning clearer.

The signal "g[n]" will be identical in shape to "f[n]", but it will have a mean level of zero units.

Next, for every frequency that we are testing, we multiply the new signal, "g[n]", by our two test waves and find the mean levels. Put mathematically, this becomes:

$$a_k = \frac{2}{N} * \sum_{n=0}^{N-1} g[n] * sin\,(2\pi f * \ k * T_s * n)$$

$$b_k = \frac{2}{N} * \sum_{n=0}^{N-1} g[n] * cos\,(2\pi f * \ k * T_s * n)$$

Notes:

- "k" is the counter from the synthesis formula. It counts through the different potential constituent waves. It will be the same value for each pair of "a_k" and "b_k". This idea will be clearer when we put the synthesis and analysis formulas together later.
- The samples within the signal are indicated by "g[n]". Where "g[0]" is the first sample, "g[1]" is the second sample and so on.
- Do not confuse the meanings of "n" and "k". The counter "n" identifies every sample in each signal or wave. It starts at zero because the first sample in any signal or wave is sample number zero. The counter "k" identifies every

frequency in each constituent wave or test wave. [It also starts at zero because the first constituent wave will have a frequency of zero cycles per second. However, at this stage, we will have already calculated the details of the constituent wave with a zero frequency. Therefore, "k" will not be zero at this stage.]

- For the waves that we multiply by the signal, we have to calculate their samples as we go. Therefore, the first sample of the Sine wave is the result of "sin ($2\pi f * k * T_s * 0$)". The second sample of the Sine wave is the result of "sin ($2\pi f * k * T_s * 1$)", and so on.
- Multiplying a discrete signal by a discrete wave involves multiplying each corresponding sample by each other. In other words, we multiply the "Signal[0]" sample by the "wave[0]" sample, and then the "Signal[1]" sample by the "wave[1]" sample, and then the "Signal[2]" sample by the "wave[2]" sample, and so on.
- Note how we put in the multiplications by 2, which are necessary for the multiplications by the test waves to produce the correct results. See Chapter 16 on the multiplication of waves if you have forgotten why they would be needed. We could put the multiplications by 2 as the amplitudes of the waves, but it makes no difference to the result whether they are outside or inside the "Σ" sum.

The rest of the formulas are the same as for a continuous periodic signal:

$$c_k = \sqrt{(a_k)^2 + (b_k)^2}$$

$$\phi_k = arctan\ (b_k\ /\ a_k)$$

[Remember that these two calculations are essentially finding the position of a phase point on a circle at the coordinates (a, b). Therefore, we can know some results without using Pythagoras's theorem or arctan, and we can know if we have the correct arctan result of the two possible ones by imagining the circle.]

The final idea we need to express is that the constituent wave with a frequency equal to half the sample rate must have its amplitude halved. Outside of using Fourier formulas, we can achieve this by using test waves that have an amplitude of 1 unit instead of 2 units. When using Fourier formulas, we can do this by not having the multiplications by 2. We can identify the test waves that have frequencies equal to half the sample rate by giving them the subscript "k = 0.5N", or better still, just "0.5N", where "N" is the number of samples in one cycle (one period) of the signal being analysed. [The constituent wave with a frequency equal

to half the sample rate will be wave number "0.5N".] The two formulas that account for frequencies equal to half the sample rate will be:

$$a_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} g[n] * sin\,(2\pi f * k * T_s * n)$$

$$b_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} g[n] * cos\,(2\pi f * k * T_s * n)$$

... where "N" is the number of samples in one cycle of the signal being analysed.

Test waves for half the sample rate

If the signal had been properly sampled, then there would be no need to test for the frequency equal to half the sample rate because those waves would not be in the signal. In such cases, we would not need to calculate "$a_{0.5N}$" and "$b_{0.5N}$".

The test waves will be integer multiples of the fundamental frequency of the discrete signal that we are analysing. The frequency equal to half the sample rate might not necessarily be an integer multiple of the fundamental frequency. In such cases, we would be unable to test for the frequency equal to half the sample rate, and we would be unable to calculate "$a_{0.5N}$" and "$b_{0.5N}$". As it might not be obvious that such situations can occur, we will look at some examples.

If we have a continuous wave with a frequency of 5 cycles per second, and we sample it at 15 samples per second, the discrete signal will have a frequency of 5 cycles per second. The frequency of half the sample rate will be 15 ÷ 2 = 7.5 cycles per second, which is not an integer multiple of 5 cycles per second. For such a situation, we would never get to test for the frequency equal to half the sample rate.

For a 4-cycle-per-second continuous wave sampled at 23 samples per second, the discrete signal would have a frequency of 1 cycle per second. Half the sample rate is 11.5 cycles per second, which is not an integer multiple of 1 cycle per second. The test waves would be 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and 11 cycles per second. We would not be able to (or need to) test the frequency equal to half the sample rate.

The full set of formulas

We can now give the complete set of formulas for performing discrete Fourier series analysis:

$$f[n] = \sum_{k=0}^{0.5N} c_k \, sin\,((2\pi * f * k * T_s * n) + \phi_k)$$

... where:

$$c_0 = \frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

$$\phi_0 = 0.5\pi$$

$$g[n] = f[n] - \, c_0 \, sin\,((2\pi f * 0 * T_s * n) + \phi_0)$$

$$a_k = \frac{2}{N} * \sum_{n=0}^{N-1} g[n] * sin\,(2\pi f * \, k * T_s * n)$$

$$b_k = \frac{2}{N} * \sum_{n=0}^{N-1} g[n] * cos\,(2\pi f * \, k * T_s * n)$$

$$a_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} g[n] * sin\,(2\pi f * \, k * T_s * n)$$

$$b_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} g[n] * cos\,(2\pi f * \, k * T_s * n)$$

$$c_k = \sqrt{(a_k)^2 + (b_k)^2}$$

$$\phi_k = arctan\,(b_k \,/\, a_k)$$

Notes for the formulas are as follows:

- "f[n]" is the discrete signal that we are analysing.
- "N" is the number of samples in one cycle (one period) of the discrete signal.
- We count through the constituent waves with "k". We start with "k" as zero, and we continue until "k" is equal to half the number of samples in one cycle. This will be the point when "f * k" is equal to half the sample rate. [We might not be able to test for the frequency equal to half the sample rate.]
- "C_k" refers to the amplitude of the wave we are finding.
- "f" is the fundamental frequency of the discrete signal that we are analysing.
- "T_s" is the sampling period, which is the time between samples.
- "n" counts through the samples of the constituent and test waves.
- "C_0" is the mean level.
- "g[n]" is the discrete signal after the mean level has been removed.
- "$a_{0.5N}$" and "$b_{0.5N}$" are the test waves for the frequency equal to half the sample rate.

Remember that we are using "k" in the synthesis formula to count between different constituent waves, but to calculate the mean levels for the steps, we count with "n" to go through each sample of each signal.

Note that some people will give this set of formulas with the understanding that the original continuous signal was correctly sampled to become the discrete signal. Therefore, they will miss out "$a_{0.5N}$" and "$b_{0.5N}$" and stop before the frequency equal to half the sample rate. Such formulas would not work for every discrete signal. It is also very common to see the formulas mistakenly neglect to remove the mean level from the signal, in which case "g[n]" will not be calculated, and "a_k" and "b_k" will be based on "f[n]" and not "g[n]".

Thoughts

As I have already said, the process for analysing a signal is expressed more easily and concisely with words than with formulas. The formulas require an explanation for them to make sense. It would be unlikely that someone who knew nothing about waves and discrete waves could understand what the formulas meant. In a way, the formulas are a result of shoehorning the process into the currently used language of maths. The available mathematical language is a socially constructed and socially limited system. Many people feel obliged to constrain themselves to its awkward rules. An analogous situation would be if every cookery book had to be written in first-century Latin, and no one was allowed to create new words such as "microwave". You could come up with a new recipe, but you would feel pressured

to phrase your recipe in the standard Latin form, despite how inappropriate such a thing would be.

In computer programming, there are far fewer constraints than in maths, and as a consequence, it is usually easier to understand a process from computer code than it is from a mathematical formula. Computer programming has progressed as ideas have progressed, but maths has generally stayed stuck in the past.

It is still useful to understand the processes as portrayed through maths because they enable us to move onto more complicated ideas whilst knowing that those ideas are correct. However, it always pays to be aware that mathematical formulas are not necessarily the best way to portray an idea or a process.

Symbol thoughts

When you first become acquainted with both “f(t)” and “f[n]”, it can be difficult to notice the difference between the brackets. Rounded brackets imply a continuous signal, and in the world of waves and signals, they will usually mean that we are basing everything on time. Square brackets imply a discrete signal. The contents of the square brackets will increase by 1, and will refer to items in a list.

If you know how to program arrays, the concept of “f[n]” might be easier to grasp because it works in the same way as identifying items in an array in many programming languages. We might have an array called “myarray”, in which case, we would identify items in the array by saying such things as “myarray[0] = 4.2”, “myarray[1] = 3.4” and similar. Alternatively, if you do not know how to program or you do not know how to program arrays, then learning about discrete waves and signals will help you understand arrays in programming. An array is just a list of items stored consecutively. First, an array is given an arbitrary name. Then, we can write values to the list or read items from the list by giving the name of the array and identifying where in the list the item is situated. In this way, we could create an array called “mylist”, and identify any item in that list by its position in the list, where the first item is stored in “mylist[0]”. In the C programming language, among others, we identify items in an array by using square brackets.

Note that some authors do not use square brackets when discussing discrete signals but still use round brackets. It will still be straightforward to get used to their methods.

Much of the study of waves and signals involves learning to distinguish between items to which we would not normally pay attention. For example, it can be confusing when you first see a formula that involves "T" meaning period, and "t" meaning time. This is because up until that point, you will not have needed to distinguish between "T" and "t". A similar confusion occurs when a formula contains both "f_s" to indicate the sampling frequency (sample rate) and "f" to indicate the cycles-per-second frequency. Other examples are "n" and "N", and "f(t)" and "f[n]". The more you see these similar symbols, the more easily you will be able to distinguish between them. With time, you will forget ever having a problem with them. If you think they are too difficult or too confusing, just persist, and everything will become easier.

Instead of using "f[n]" or "g[n]" Some people use "x[n]" and "y[n]". These mean the same thing, and might be less prone to being confused with "f(n)" and "g(n)". Strictly speaking, we could use any prefix to the square brackets, such as "mysignal[n]", but mathematicians prefer single characters to words, and they often prefer "x" to other letters.

Alternative formulas

The *continuous* Fourier series formulas can be given in many different ways with countless variations. Some of these variations are:

- The process using just "f(t)", instead of "f(t)" and "g(t)", in which case, it is not explicitly stated that the mean level is removed from the signal to calculate the non-zero-frequency constituent waves.
- Giving each constituent wave in the synthesis formula as a zero-phase Sine wave added to a zero-phase Cosine wave.
- Not actually specifying the frequencies in the synthesis formula, and just identifying them as "f_0", "f_1", f_2" and so on.

All the different ways of portraying the continuous Fourier series formulas can be adapted to the discrete Fourier series formulas. The most common difference that you will see is a formula that does not bother checking for the frequency equal to half the sample rate, with the idea that every discrete signal will have been sensibly sampled. There are countless variations of the set of discrete Fourier series formulas, but in my opinion, the one used in this chapter is the simplest and clearest. You will often see formulas that are unnecessarily complicated and bordering on being incomprehensible.

For interest's sake, I will give a common version of the formulas that uses zero-phase Sine and Cosine waves in the synthesis formula, and does not indicate that the mean level is removed from the signal before the non-zero-frequency waves are calculated.

$$f[n] = \sum_{k=0}^{0.5N} a_k \, sin\,(2\pi fkT_s * n) + b_k \, cos\,(2\pi fkT_s * n)$$

... where:

$$a_0 = 0$$

$$b_0 = \frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

$$a_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} f[n] * sin\,(2\pi fkT_s * n)$$

$$b_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} f[n] * cos\,(2\pi fkT_s * n)$$

$$a_k = \frac{2}{N} * \sum_{n=0}^{N-1} f[n] * sin\,(2\pi fkT_s * n)$$

$$b_k = \frac{2}{N} * \sum_{n=0}^{N-1} f[n] * cos\,(2\pi fkT_s * n)$$

Note that:

- "k" counts through the different pairs of Sine and Cosine waves.
- "n" counts through each sample in the discrete waves and signals.
- "a_0" is zero because the mean level can be expressed using just a Cosine wave with zero phase.
- It is not stated in the formulas that the mean level is removed before we calculate the waves from k = 1 onwards. Without removing the mean level, the calculations will not work (unless there is a zero mean level).

An in-depth example

We will use the full set of formulas mentioned earlier to calculate the constituent discrete waves of a signal. We will start with the continuous signal created by adding:
"y = 2"
"y = 3 sin ((2π * 2t) + 0.2π)"
"y = sin (2π * 3t)"
"y = 1.1 sin (2π * 4t)"

The continuous signal has the formula:
"y = 2 + 3 sin ((2π * 2t) + 0.2π) + sin (2π * 3t) + 1.1 sin (2π * 4t)"

We will sample it with a sample rate of 20 samples per second. Its sampling period will be 0.05 seconds. Therefore, it will be equivalent to this discrete signal:

f[n] = 2 + 3 sin ((2π * 2 * 0.05n) + 0.2π) + sin (2π * 3 * 0.05n)
+ 1.1 sin (2π * 4 * 0.05n)

We will pretend that we have forgotten the details of the signal. We will say that all we know about the signal is the sample rate and the following list of samples that are given to 4 decimal places.

Time	**Sample number**	**Sample Value**
t = 0	0	3.7634
t = 0.05	1	6.7083
t = 0.10	2	6.4508
t = 0.15	3	3.4258
t = 0.20	4	0.3661
t = 0.25	5	−0.7634
t = 0.30	6	−0.3948
t = 0.35	7	0.1024
t = 0.40	8	0.5411
t = 0.45	9	1.7629
t = 0.50	10	3.7634
t = 0.55	11	5.0903
t = 0.60	12	4.5487
t = 0.65	13	2.8078
t = 0.70	14	1.5416

t = 0.75	15	1.2366
t = 0.80	16	0.7808
t = 0.85	17	−0.5156
t = 0.90	18	−1.3610
t = 0.95	19	0.1448

The rest of the samples in the signal would be repeats of the samples from sample number 0 to sample number 19. One cycle of the signal is 20 samples long. If we did not know the sample rate, we could say that the signal has a frequency of 1 ÷ 20 = 0.05 cycles per sample. As we *do* know the sample rate – it is 20 samples per second – we know that the signal has a fundamental frequency of 1 cycle per second.

For what we are doing, we need to know only the first 20 samples of the signal. These go from sample number 0 to sample number 19.

The general synthesis formula that states that a discrete periodic signal is made up of discrete waves is as so:

$$f[n] = \sum_{k=0}^{0.5N} c_k \sin((2\pi * f * k * T_s * n) + \phi_k)$$

The value of "k" increases from 0 upwards, stopping when it is equal to half the number of samples in one period of the signal being analysed. As there are 20 samples in one period of the signal, we stop when "k" is equal to 10. In other words, we test for a mean level and 10 test waves.

We can fill in what we know about our signal into the synthesis formula:

$$f[n] = \sum_{k=0}^{10} c_k \, sin \, ((2\pi * 1 * k * 0.05 * n) + \phi_k)$$

... which is:

$$f[n] = \sum_{k=0}^{10} c_k \, sin \, ((2\pi * k * 0.05n) + \phi_k)$$

The first step is to calculate the mean level of our discrete signal. The calculation that we will be performing is this:

$$\frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

As we know that there are 20 samples in one period of the signal, we can make this more specific:

$$\frac{1}{20} * \sum_{n=0}^{19} f[n]$$

This ultimately means that we add up all the samples for one period (from sample number 0 to sample number 19), and then divide by the number of samples (20). To put this more simply, we just find the average of the samples for one period.

The total of the samples is 40.000010. The average is therefore:
40.000010 ÷ 20 = 2.000001 units.

We will round 2.000001 units to 4 decimal places to match the accuracy that we have been using with our samples:
2.0000

Therefore, our mean level is 2 units. We can portray this as a zero-frequency wave:
"y = 2 sin ((2π * 1 * 0 * 0.05 * n) + 0.5π)"
... where:

- 2 is the amplitude, which we have just calculated.
- 1 is the fundamental frequency of the original signal.
- 0 is the value of "k", and "k" is being used to count through the different constituent waves.
- 0.05 is the sampling period.
- "n" is the range of sample numbers for this wave.
- 0.5π is the phase.

We can make the formula more concise as:
"y = 2 sin (0n + 0.5π)"
... or:
"y = 2 sin (0.5π)
... or even just:
"y = 2"

... although this stops the formula being a wave. Which of the results we choose depends on whether we want the result to be a wave or not, and if we do want it to be a wave, how much we want its parts to match those of the standard discrete wave formula. Keeping it as a wave reinforces the idea that we are dealing with discrete signals.

The next step is this part:
$g[n] = f[n] - c_0 \, sin \, ((2\pi f * 0 * T_s * n) + \phi_0)$
... which, given that we know the amplitude, frequency and phase of the zero-frequency wave, as well as its sample rate, can be given as so:
$g[n] = f[n] - 2 \, sin \, ((2\pi * 1 * 0 * 0.05n) + 0.5\pi)$

This means that we subtract the mean level from every sample in "f[n]", and put the result in the corresponding part of "g[n]". For our example, we subtract 2 from each sample in "f[n]". The result of this is that "g[n]" contains our signal with the mean level removed. The signal "g[n]" has a zero mean level. Our list of samples becomes:

Sample number	**Value**
0	1.7634
1	4.7083
2	4.4508
3	1.4258
4	−1.6339
5	−2.7634
6	−2.3948
7	−1.8976
8	−1.4589
9	−0.2371
10	1.7634
11	3.0903
12	2.5487
13	0.8078
14	−0.4584
15	−0.7634
16	−1.2192
17	−2.5156
18	−3.3610
19	−1.8552

The above list of samples will be identified as the signal “g[n]”.

The next stages are to test for the constituent waves for every value of “k” from 1 upwards, stopping after we have tested for the wave for when “k” is 10. In other words, we test for integer multiples of the signal’s frequency from a multiple of 1 up to a multiple of 10. As our signal’s frequency is 1 cycle per second, we can say that we will be testing frequencies from 1 cycle per second up to 10 cycles per second.

The general formulas to calculate the constituent waves are as so:

$$a_k = \frac{2}{N} * \sum_{n=0}^{N-1} g[n] * sin\ (2\pi f *\ k * T_s * n)$$

$$b_k = \frac{2}{N} * \sum_{n=0}^{N-1} g[n] * cos\ (2\pi f *\ k * T_s * n)$$

$$a_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} g[n] * sin\ (2\pi f *\ k * T_s * n)$$

$$b_{0.5N} = \frac{1}{N} * \sum_{n=0}^{N-1} g[n] * cos\ (2\pi f *\ k * T_s * n)$$

$$c_k = \sqrt{(a_k)^2 + (b_k)^2}$$

$$\phi_k = arctan\ (b_k\ /\ a_k)$$

As always, it is important not to confuse “k” and “n”. The above “a_k” and “b_k” formulas use “n” to count through the samples in each signal. We use the letter “k” to count through the different potential constituent waves. In our example, “n” will count from zero to 19 (because we have 20 samples in each signal or wave), and “k” will count from zero up to 10 (because we are testing for the mean level and 10 constituent waves).

Given that we know the fundamental frequency of the signal, and that we know the sample rate and the number of samples in one cycle, we can make the "a_k" and "b_k" formulas more specific:

$$a_k = \frac{2}{20} * \sum_{n=0}^{19} g[n] * sin\ (2\pi * k * 0.05n)$$

$$b_k = \frac{2}{20} * \sum_{n=0}^{19} g[n] * cos\ (2\pi * k * 0.05n)$$

$$a_{10} = \frac{1}{20} * \sum_{n=0}^{19} g[n] * sin\ (2\pi * k * 0.05n)$$

$$b_{10} = \frac{1}{20} * \sum_{n=0}^{19} g[n] * cos\ (2\pi * k * 0.05n)$$

We have calculated the wave for when "k" was zero (it was the mean level). Now we calculate the wave for when "k" is 1. In all these examples, we will work with our original samples, which were given to 4 decimal places, and we will use test waves that are rounded to 4 decimal places. This is for consistency's sake.

To start, we will calculate a_1, which will involve this calculation:

$$a_1 = \frac{2}{20} * \sum_{n=0}^{19} g[n] * sin\ (2\pi * 1 * 0.05n)$$

... which can be slightly simplified as so:

$$a_1 = 0.1 * \sum_{n=0}^{19} g[n] * sin\ (2\pi * 0.05n)$$

[This means that we multiply our signal "g[n]" by our first Sine test wave, and then add up the samples and multiply them by "2 / 20".]

To obtain "a_1", we first need to calculate the samples for the discrete wave with this formula:
"y = sin (2π * 0.05n)"
... for 20 samples (in other words, for values of "n" from 0 up to 19). We will round up each sample to 4 decimal places so that the accuracy matches that of the samples of our signal – we do not have to do this, but we will do it here for the sake of consistency. Then, we multiply each sample from our "g[n]" signal by the corresponding sample from the discrete Sine wave. In other words, we multiply the first sample from our "g[n]" signal by the first sample from our discrete Sine wave. Then, we multiply the second sample from our "g[n]" signal by the second sample from our discrete Sine wave, and so on. Doing this will create a new signal containing 20 samples. We add up those samples and multiply them by 0.1 [which is 2 ÷ 20.] We are ultimately finding the average and multiplying it by 2. The result will be the value of "a_1".

[As a reminder, in this situation, we are still ultimately multiplying the signal by Sine and Cosine waves with amplitudes of 2 units, and then finding the average of each result. However, the way the formulas have been set up means that the Sine and Cosine waves have amplitudes of 1 unit, and we then find the average and multiply it by 2. This has the same result. It is slightly less intuitive, but it is simpler in a formula.]

The following table shows the multiplications:

Sample number	Sample from our g[n] signal	Sample from our Sine wave	Result of multiplying them together
0	1.7634	0.0000	0.0000
1	4.7083	0.3090	1.4549
2	4.4508	0.5878	2.6162
3	1.4258	0.8090	1.1535
4	−1.6339	0.9511	−1.5540
5	−2.7634	1.0000	−2.7634
6	−2.3948	0.9511	−2.2777
7	−1.8976	0.8090	−1.5352
8	−1.4589	0.5878	−0.8575
9	−0.2371	0.3090	−0.0733
10	1.7634	0.0000	0.0000
11	3.0903	−0.3090	−0.9549
12	2.5487	−0.5878	−1.4981
13	0.8078	−0.8090	−0.6535
14	−0.4584	−0.9511	0.4360
15	−0.7634	−1.0000	0.7634
16	−1.2192	−0.9511	1.1596
17	−2.5156	−0.8090	2.0351
18	−3.3610	−0.5878	1.9756
19	−1.8552	−0.3090	0.5733

As we can see, the signal created from the multiplication has the samples:
0.0000, 1.4549, 2.6162, 1.1535, and so on.

If we round up each sample to 4 decimal places, the sum of all the samples is 0 units. We multiply this by 0.1 and we have the value of "a_1" as 0. [In other words, the average of the samples is zero.]

We then calculate "b_1" in the same way, but using Cosine. We will skip the details because the process is essentially the same. We end up with zero again. This means that there is no constituent wave with a frequency of 1 cycle per second in the signal. To be consistent with our synthesis formula, we should still give a formula for this result. It will be a wave with a frequency of one cycle per second and an amplitude of zero units. For consistency's sake, we could write it as so:
"$y = 0 \sin (2\pi * 1 * 0.05n)$"

[Normally, there would be no point in providing a zero-amplitude wave – we could just say that that frequency does not exist in the signal. However, because we are following the layout of our particular synthesis formula, it is consistent to do so. In this example, we will write zero-amplitude waves, but most people would not bother.]

When "k" is 2, we end up with:
"a_2" as 0.1 * 24.2717 = 2.4272
... and:
"b_2" as 0.1 * 17.6333 = 1.7633

The fact that at least one of these is non-zero means that there is definitely a constituent wave with this frequency (2 cycles per second). We calculate its amplitude "c_2" and phase "ϕ_2" with the formulas:

$$c_2 = \sqrt{(a_2)^2 + (b_2)^2}$$

$$\phi_2 = arctan\ (b_2 \ / \ a_2)$$

The amplitude is:
$\sqrt{2.4272^2 + 1.7633^2}$
= 3.0001 [to 4 decimal places]

The phase is:
arctan (1.7633 / 2.4272)
= 0.6283 radians
= 0.2000π radians [to 4 decimal places].

We check this is the correct answer to arctan by imagining (2.4272, 1.7633) as the coordinates of a phase point on a circle. [The x-axis coordinate would be "a_k"; the y-axis coordinate would be "b_k".] The point would be in the upper right-hand quarter of the circle, so 0.2000π is the result that we want out of the two possible ones.

The discrete constituent wave has the formula:
"y = 3.0001 sin ((2π * 2 * 0.05n) + 0.2000π)"
... which, given the accuracy we have been using, we will write as:
"y = 3 sin ((2π * 2 * 0.05n) + 0.2π)"

This matches the 2-cycle-per-second discrete wave that was actually used to create the signal in the first place.

When “k” is 3, we end up with:
“a_3” as 0.1 * 10.0001 = 1.0000 [to 4 decimal places]
... and:
“b_3” as 0.1 * −0.0003 = −0.0000 [to 4 decimal places]

These give an amplitude of 1 unit and a phase of 0 radians. The discrete wave has the formula:
“y = 1 sin (2π * 3 * 0.05n)”

When “k” is 4, we end up with:
“a_4” as 0.1 * 11.0004 = 1.1000 [to 4 decimal places]
... and:
“b_4” as 0.1 * 0 = 0

These give an amplitude of 1.1 units and a phase of 0 radians. The discrete wave has the formula:
“y = 1.1 sin (2π * 4 * 0.05n)”

When “k” is 5, we end up with:
“a_5” as 0.1 * 0.0001 = 0.0000 [to 4 decimal places]
... and:
“b_5” as 0.1 * 0.0001 = 0.0000 [to 4 decimal places]

These imply that there is no wave of 5 cycles per second. To fit in with the synthesis formula, we will express this as a wave with zero amplitude:
“y = 0 sin (2π * 5 * 0.05n)”

When “k” is 6, we end up with:
“a_6” as 0.1 * 0.0002 = 0.0000 [to 4 decimal places]
... and:
“b_6” as 0.1 * 0.0006 = 0.0001 [to 4 decimal places]

These are close enough to zero that we can assume that there is no wave with a frequency of 6 cycles per second in the signal. If we calculated the details anyway, we would have an amplitude of 0.0001 units and a phase of 0.5π radians. We will say that there is no wave with this frequency. We will express the fact with a wave with zero amplitude:
“y = 0 sin (2π * 6 * 0.05n)”

When “k” is 7, we end up with:
“a_7” as 0.1 * −0.0003 = −0.0000 [to 4 decimal places]
... and:
“b_7” as 0.1 * −0.0001 = −0.0000 [to 4 decimal places]

Therefore, there is no wave with a frequency of 7cycles per second, and we will express this with a zero-amplitude wave:
“y = 0 sin (2π * 7 * 0.05n)”

When “k” is 8, we end up with:
“a_8” as 0.1 *−0.0003 = −0.0000 [to 4 decimal places]
... and:
“b_8” as 0.1 * −0.0001 = −0.0000 [to 4 decimal places]

Therefore, there is no wave with a frequency of 8 cycles per second. We will express this as so:
“y = 0 sin (2π * 8 * 0.05n)”

When “k” is 9, we end up with:
“a_9” as 0.1 * −0.0002 = −0.0000 [to 4 decimal places]
... and:
“b_9” as 0.1 * 0.0002 = 0.0000 [to 4 decimal places]

Therefore, there is no wave with a frequency of 9 cycles per second. We express this as so:
“y = 0 sin (2π * 9 * 0.05n)”

When “k” is 10, we use the pair of formulas specifically for when the test frequencies are equal to half the sample rate:

$$a_{10} = \frac{1}{20} * \sum_{n=0}^{19} g[n] * sin\ (2\pi * k * 0.05n)$$

$$b_{10} = \frac{1}{20} * \sum_{n=0}^{19} g[n] * cos\ (2\pi * k * 0.05n)$$

As we know, these two formulas are the same as usual, but without the multiplication by 2. Therefore, the resulting wave for this frequency will have the correct amplitude. Remember that the discovered wave, if any, might not be the same wave that was in the signal. However, it will have the same samples. A well-

sampled signal would not contain a frequency equal to half the sample rate, but we will say that we do not know if this signal was sampled well or not.

Instead of multiplying the sum of the signals' samples by 2 / 20, which is 0.1, we will be multiplying them by 1 / 20, which is 0.05.

When "k" is 10, we end up with:

"a_{10}" as 0.05 * 0.0000 = 0.0000 [to 4 decimal places]
... and:
"b_{10}" as 0.05 * 0.0002 = 0.0000 [to 4 decimal places]

Therefore, there is no wave with a frequency of 10 cycles per second. We express this as so:
"y = 0 sin (2π * 10 * 0.05n)"

If we continued to test waves past "k = 10", we would just find duplicates of the waves that we had already found.

If we include waves with amplitudes of zero units, our full list of constituent waves ordered in terms of "k" is so:

k = 0: "y = 2 sin ((2π * 0 * 0.05n) + 0.5π)" [This is the mean level.]
k = 1: "y = 0 sin (2π * 1 * 0.05n)"
k = 2: "y = 3 sin ((2π * 2 * 0.05n) + 0.2π)"
k = 3: "y = 1 sin (2π * 3 * 0.05n)"
k = 4: "y = 1.1 sin (2π * 4 * 0.05n)"
k = 5: "y = 0 sin (2π * 5 * 0.05n)"
k = 6: "y = 0 sin (2π * 6 * 0.05n)"
k = 7: "y = 0 sin (2π * 7 * 0.05n)"
k = 8: "y = 0 sin (2π * 8 * 0.05n)"
k = 9: "y = 0 sin (2π * 9 * 0.05n)"
k = 10: "y = 0 sin (2π * 10 * 0.05n)"

If we ignore waves with zero amplitudes, we have:

k = 0: “y = 2 sin ((2π * 0 * 0.05n) + 0.5π)” [This is the mean level.]
k = 1: [No wave]
k = 2: “y = 3 sin ((2π * 2 * 0.05n) + 0.2π)”
k = 3: “y = sin (2π * 3 * 0.05n)”
k = 4: “y = 1.1 sin (2π * 4 * 0.05n)”
k = 5: [No wave]
k = 6: [No wave]
k = 7: [No wave]
k = 8: [No wave]
k = 9: [No wave]
k = 10: [No wave]

We can write out the mean level and constituent waves as we would do normally:
“y = 3 sin ((2π * 2 * 0.05n) + 0.2π)”
“y = sin (2π * 3 * 0.05n)”
“y = 1.1 sin (2π * 4 * 0.05n)”
... and a mean level of 2 units.

These are exactly the same waves that were in the original discrete signal. This shows that the method works, although it is worth noting that we rounded up several values during the calculations. If we had used more accuracy throughout the calculations, we would have ended up with a set of waves that, when added, would have had exactly the same samples as our original signal. Unlike continuous Fourier series analysis, which often results in an approximation, discrete Fourier series analysis finds a set of waves that, when added together, exactly recreate the analysed signal. [The exactness depends on the precision used in the calculations. Incorrect results can only be a consequence of mistakes or rounding errors.]

We could express the discovered waves in terms of the continuous waves that, when sampled at 20 samples per second, would produce those discrete waves:
“y = 3 sin ((2π * 2t) + 0.2π)”
“y = sin (2π * 3t)”
“y = 1.1 sin (2π * 4t)”
... and a mean level of 2 units.

Spreadsheet programs

Instead of using a calculator, it is easiest to perform exercises such as the one we have just seen by using a spreadsheet program such as Microsoft Excel or one of its free alternatives. If you want to know how to use Excel to perform discrete Fourier series analysis, read this section. If not, then feel free to skip it.

I will presume that you know the very basics of Excel and have an empty spreadsheet in front of you.

Creating a signal

To create the initial samples of our signal in the example that we have just seen, we need to continue as follows:

First, we tell Excel that we want all the cells to be numbers. Right-click on the top left corner of the spreadsheet between "1" and "A". In the menu that appears, choose "Format cells...", then for the "Category", choose "number". Then set "Decimal places" to 8. Press the "OK" button. This will make all the cells show numbers to 8 decimal places.

We will create a column with the numbers from 0 to 19. This column will act as the sample numbers (by which I mean the numbers that identify each sample). For this example, we will start the column in the cell "E3". Therefore, enter "0" in cell "E3", "1" in cell "E4", "2" in cell "E5" and "3" in cell "E6". We could enter all the numbers by hand like this, but it is quicker to use a shortcut method. Select the cells containing 1 to 3 (which will appear as 1.00000000 to 3.00000000). Move the mouse cursor over the bottom right-hand corner of the selected cells. It will turn into a thick black cross. Left-click the bottom right-hand corner of the selected cells and, keeping it pressed, drag the mouse vertically downwards to cell "E22". Excel will fill in the rest of the cells for you. If the cells should turn into "######", extend the width of the cell by double clicking on the line between "E" and "F" in the letters at the top of the cell columns.

To the column to the right of our numbers (column "F"), we will create a column listing the mean level – for this example, it will be the number 2 in each cell. Enter "2" in the first cell (cell "F3"), and press "Enter". Then, copy and paste the cell into the cells underneath. It needs to be copied into all the cells that have a number in the column to the left, so in other words, cells "F4" to "F22".

Next to that column (in column “G”), we will create a column listing the samples of the first wave. Enter the following in the first cell (cell “G3”):

```
=3 * SIN( (2*PI() * 2 * 0.05 * E3) + 0.2 * PI() )
```

... where “E3” refers to the cell that contains the number “0” in our list of numbers from 0 to 19. [In Excel, “π” is portrayed by “`PI()`”.] Press “Enter”, and then copy and paste that cell into the cells beneath (down to cell “G22”), and Excel will adjust the cell numbers accordingly. This saves typing out 20 different formulas.

In the first cell of the next column (cell “H3” of column “H”) enter this formula:

```
=SIN(2 * PI() * 3 * 0.05 * E3)
```

... where “E3” still refers to the cell that contains the number “0” in our list of numbers. Press enter, and then copy and paste the cell into the cells below, down to cell “H22”.

In the first cell of the next column (cell “I3”), enter this:

```
=1.1 * SIN(2 * PI() * 4 * 0.05 * E3)
```

Press “Enter”, and copy and paste the completed cell to the cells below, down to cell “I22”.

In the next column, enter the following in the first cell (cell “J3”):

```
=F3 + G3 + H3 + I3
```

... where “F3”, “G3”, “H3” and “I3” contain the first samples of the mean level and each of the waves. Press “Enter”, then copy and paste that cell into the cells beneath it, down to cell “J22”. Excel will fill in the relevant cell numbers. This column now contains the samples of the signal that we are going to analyse.

[We could have achieved the same column of samples by having all the wave formulas and the mean level in one very long formula in a single column instead of four different formulas in four different columns, but doing that makes it much harder to spot mistakes.]

If we wanted, we could write names above the columns to say what is in them. In cell “J2”, we could write “Our signal” to identify the column of samples. We could also highlight the cells of our signal. To do this, select the cells from “J3” to “J22”, and click on the (usually yellow) bucket button at the top of Excel to turn the background yellow.

Analysing a signal

We will pretend that we have forgotten the details of the signal we just created, and all we have is the column of samples (J3 to J22). We will now analyse it using discrete Fourier series analysis.

The first step is to calculate the mean level. In cell "J24", type the following, but *without* pressing "Enter"*:*

```
=SUM(
```

... and then use the mouse to select the cells "J3" to "J22" and press "Enter". Excel will fill in the correct cell numbers, and the cell will have the contents:

```
=SUM(J3:J22)
```

This gives us the total of all the samples of our signal. Click in the cell and add:

```
/20
```

... to the end, and press "Enter". The cell's contents will be:

```
=SUM(J3:J22)/20
```

... and the cell will show the average of the samples in our signal. This is the mean level. If everything has worked, this should show the number "2".

We now want a version of our signal with the mean level removed. In cell "K3", type:

```
=J3-2
```

... and press "Enter". This is our first sample with the value 2 subtracted from it. Copy and paste this cell into the cells below, down to cell "K22", and Excel will fill in the relevant values. The column will be our signal with zero mean level. This will be the signal that we will now analyse.

To keep this explanation short, we will test only one frequency. We will skip one cycle per second and go straight to the frequency of 2 cycles per second. To test for this frequency, we need to create our test Sine wave and our test Cosine wave.

To avoid confusing our columns, we will leave a gap of one column, and we will put our Sine test wave into column "M". In cell "M3", type:

```
=2 * SIN(2 * PI() * 2 * 0.05 * E3)
```

... then press "Enter", and copy and paste the cell into the cells below down to cell "M22". In the formula "`PI()`" is Excel's way of writing "π". The value "2" after the "`PI()`" is the test frequency. The value "0.05" is the sampling period – we are using a sample rate of 20 samples per second. "E3" is the cell with the number 0 in it in our list of numbers from 0 to 19.

The "M" column now contains the samples of our test Sine wave.

In cell “N3”, type:

```
=2 * COS(2 * PI() * 2 * 0.05 * E3)
```

... then press “Enter”, and copy and paste the cell into the cells below down to cell “N22”. The “N” column now contains the samples of our test Cosine wave.

Now, we will multiply our two test waves by our signal (with its mean level removed).

In cell “O3”, we type:

```
=K3 * M3
```

... and press “Enter”. Copy and paste the cell into the cells below down to cell “O22”. This column contains the signal from which we will calculate the first mean level. To calculate the mean level of this signal, we need to add up the samples and divide by the number of samples. Therefore, in cell “O24” type this:

```
=SUM(O3:O22)/20
```

[Be careful to notice which is the letter “O” and which is the number zero “0”.] Excel will calculate the average of the samples in this cell, which will be the “first mean level”. If you have typed in everything correctly, this will be 2.42705098.

We then multiply our Cosine test wave by our signal (with its mean level removed). In cell “P3”, type:

```
=K3 * N3
```

... and press “Enter”. Copy and paste the cell into the cells below, down to cell “P22”. Then, in cell “P24”, type:

```
=SUM(P3:P22)/20
```

The average of the samples, which is the “second mean level”, will now be in cell “P24”. This will be 1.76335576.

As both the “first mean level” and the “second mean level” are non-zero, it means that there is definitely a constituent wave with a frequency of 2 cycles per second in our signal. We can imagine the two mean levels as being the coordinates of a phase point on a circle. The coordinates would be (2.42705098, 1.76335576). The amplitude of the wave will be the distance of this point from the origin of the axes; the phase of the wave will be the angle of this point.

We calculate the amplitude using Pythagoras’s theorem. We can do this in Excel. In cell “O26” (or any empty cell that is out of the way), type:

```
=SQRT(O24^2 + P24^2)
```

Excel will calculate the square root of the sum of the squares of the mean levels. The term "SQRT(...)" finds the square root of the value between the brackets. The symbol "^" means "raised to the power of...", so "O24^2" means the contents of "O24" squared, and "P24^2" means the contents of "P24" squared. The result of the whole calculation will be 3.00000000.

We calculate the phase using arctan. We can do this in two different ways. For the first way, in cell "P27" (or any empty cell that is out of the way), type:
`=ATAN(P24/O24)`
Excel will calculate *one* of the two possible arctan results. "ATAN(...)" is Excel's term for arctan. The result will be: 0.62831853 radians to 8 decimal places. As we know that the phase point of the circle will be in the top right hand quarter of the axes, we can tell that this is the result that we want. The result happens to be 0.2π radians, which we can find out by dividing the result by π.

The second way to calculate the result is to use "ATAN2", which gives the only correct arctan result of the two possible ones. In other words, we do not have to think about whether the result is the one we want out of the two possible results. In cell "P28" (or any empty cell that is out of the way), type:
`=ATAN2(O24,P24)`
Excel will calculate the result again as 0.62831853 radians, which is 0.2π radians.

Our discovered wave has a frequency of 2 cycles per second, an amplitude of 3 units, and a phase of 0.2π radians. Its discrete formula is:
"y = 3 sin ((2π * 2 * 0.05n) + 0.2π)"

Its continuous formula is:
"y = 3 sin ((2π * 2t) + 0.2π)"

This matches the wave that we know exists in the signal.

To find all the waves in the signal, we would need to test the frequencies of 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 cycles per second (while remembering that because 10 cycles per second is equal to half the sample rate, we would halve the amplitude of the discovered wave). We can do all of this in Excel.

By using Microsoft Excel or one of its free alternatives, we can work with low sample rates and we can still see what is happening. If you know how to program arrays in any programming language, you can perform all of this in a less cumbersome way, but it is more effort to examine the results of each stage. Spreadsheet programs are very good for learning about discrete waves.

[Conversely, trying to work with discrete waves is very good for learning how to use spreadsheet programs.]

It would be possible to use the spirit of Microsoft Excel and write out the calculations in neat columns on a piece of paper. This would obviously take longer to do, but it would achieve the same results.

More examples

A wave over half the sample rate

To show what happens when we have a frequency above half the sample rate, we will add a wave of 14 cycles per second to the continuous signal in the last example. We will add the wave:
"y = 2.2 sin (2π * 14t)"

The full formula for the continuous signal will be:
y = 2 + 3 sin ((2π * 2t) + 0.2π) + sin (2π * 3t) + 1.1 sin (2π * 4t) + 2.2 sin (2π * 14t)

The sample rate will stay at 20 cycles per second, and we will say that the samples of the signal are given to 4 decimal places. The full formula of the discrete signal will be:

f[n] = 2 + 3 sin ((2π * 2 * 0.05n) + 0.2π) + sin (2π * 3 * 0.05n) +
1.1 sin (2π * 4 * 0.05n) + 2.2 sin (2π * 14 * 0.05n)

This signal still has a frequency of one cycle per second.

Our synthesis formula will remain as:

$$f[n] = \sum_{k=0}^{10} c_k \, sin \, ((2\pi * 1 * k * 0.05 * n) + \phi_k)$$

... where 1 is the fundamental frequency of the discrete signal.

The mean level of the entire signal is still 2 units [to 4 decimal places]. Therefore, when "k" is zero, the discrete wave representing the mean level is:
"y = 2 sin ((2π * 0 * 0.05n) + 0.5π)"

For the other values of "k", we will skip the details of each calculation and jump straight to the calculated constituent wave. We will round up the amplitudes to make them easier to read. Our full list of calculated waves is as so:

"k = 0": "y = 2 sin ((2π * 0 * 0.05n) + 0.5π)" [This is the mean level.]
"k = 1": [No wave]
"k = 2": "y = 3 sin ((2π * 2 * 0.05n) + 0.2π)"
"k = 3": "y = sin (2π * 3 * 0.05n)"
"k = 4": "y = 1.1 sin (2π * 4 * 0.05n)"
"k = 5": [No wave]
"k = 6": "y = 2.2 sin ((2π * 6 * 0.05n) + π)"
"k = 7": [No wave]
"k = 8": [No wave]
"k = 9": [No wave]
"k = 10": [No wave]

For the "k_6" wave, "a_6" was −2.2001 and "b_6" was 0. Pythagoras's theorem gives the amplitude as $\sqrt{-2.2001^2 + 0^2}$ = 2.2001 units, which we will round up to 2.2. The phase is arctan (0 ÷ −2.2001) = arctan (0) = 0 radians. There are two possible angles that have a gradient of 0. These are 0 radians and π radians (0 degrees and 180 degrees). If we think of "a_6" and "b_6" as coordinates of a phase point on a circle, they indicate the point at: (−2.2001, 0), which is at an angle of 180 degrees (π radians). Therefore, the phase should be π radians. As we were calculating the amplitude and phase with a value that was zero, we could have done this in our heads instead of resorting to Pythagoras's theorem and arctan.

Given that there is no point in testing for waves with frequencies higher than half the sample rate, there was no point in testing for a frequency of 14 cycles per second. However, our 14-cycle-per-second wave has become "aliased" to appear as a 6-cycle-per-second wave with a phase of 180 degrees. It is the case that the samples of:
"y = 2.2 sin (2π * 14 * 0.05n)
... are identical to those of:
"y = 2.2 sin ((2π * 6 * 0.05n) + π)"

We can know why this happens by how a 14 cycle-per-second wave will become sampled as if it were lower by the sample rate. As the sample rate is 20 samples per second, the wave ends up with the same characteristics, but a frequency of 14 – 20 = −6 cycles per second. When we analyse the signal, we find the wave with its frequency made positive. Therefore, it has a frequency of +6 cycles per second and a different phase. This was all explained in the previous chapter.

If we add up the constituent waves that we found when analysing our discrete signal, we would have exactly the same list of samples. Our analysis might not have found the exact sum of waves that were added, but it found a sum of waves that would create exactly the same discrete signal.

A square wave

We will analyse a discrete square wave to see what happens. We will use a sample rate of 20 samples per second, and give the square wave a frequency of 1 cycle per second. Our square wave will fluctuate between 0 and 1, which means that it will consist of groups of ten zeroes, followed by ten ones. We will give it a non-zero phase to make the example more interesting.

Our square wave has the following samples:

Time	Sample number	Value
t = 0.00	0	0
t = 0.05	1	0
t = 0.10	2	0
t = 0.15	3	0
t = 0.20	4	1
t = 0.25	5	1
t = 0.30	6	1
t = 0.35	7	1
t = 0.40	8	1
t = 0.45	9	1
t = 0.50	10	1
t = 0.55	11	1
t = 0.60	12	1
t = 0.65	13	1
t = 0.70	14	0
t = 0.75	15	0
t = 0.80	16	0
t = 0.85	17	0
t = 0.90	18	0
t = 0.95	19	0

Our synthesis formula will be the same as before:

$$f[n] = \sum_{k=0}^{10} c_k \, sin\,((2\pi * 1 * k * 0.05 * n) + \phi_k)$$

... where:

- The number 10 is half the number of samples in one period of the signal. It is also the number that, when multiplied by the fundamental frequency (1 cycle per second) will be equal to half the sample rate. We stop after we have tested for the frequency produced when "k = 10".
- The number 1 is the fundamental frequency of the discrete signal.

This means that we will be trying to create a discrete square wave with just a mean level and 10 discrete pure waves. If we were analysing a *continuous* square wave, we would need an infinite number of waves to get the closest approximation, and we could never get an accurate result. We will see what happens when we analyse this discrete square wave.

The samples in our list of samples add up to 10, so the average is 0.5. Therefore, the mean level is 0.5 units, and the "k_0" wave is:
"y = 0.5 sin ((2π * 0 * 0.05n) + 0.5π)"

We remove this from each sample of our signal, or more simply, remove 0.5 from each sample. Then, we calculate the constituent waves in the normal way. We create a discrete Sine wave and a discrete Cosine wave for each test frequency and multiply the corresponding samples by our zero-mean level signal. Note that, in this example, we will round the samples from the test Sine and Cosine waves to 4 decimal places. This is to emphasise how, generally, there would be a limit to the accuracy of the values in all of our calculations.

For this example, we will skip the laborious calculations, and go straight to the "a_k" and "b_k" results.

For "k_1":
"a_1" is: 0.29022
"b_1" is: −0.56958
Therefore, the amplitude is: 0.6393 units.
The phase is: 5.1836 radians, which is 1.65π radians.

For “k_2”:
“a_2” is: 0
“b_2” is: 0
Therefore, there is no wave with a frequency of 2 cycles per second in the signal.

For “k_3”:
“a_3” is: −0.2176
“b_3” is: 0.0345
Therefore, the amplitude is: 0.2203 units.
The phase is: 2.9844 radians, which is 0.95π radians.

For “k_4”:
“a_4” is: 0
“b_4” is: 0
Therefore, there is no wave with a frequency of 4 cycles per second in the signal.

For “k_5”:
“a_5” is: 0.1
“b_5” is: 0.1
Therefore, the amplitude is: 0.1414 units.
The phase is: 0.25π radians, which is something we can know without a calculator. [Both the values are positive and the same. Therefore, they must be at 45 degrees, which is 0.25π radians.]

For “k_6”:
“a_6” is: 0
“b_6” is: 0
Therefore, there is no wave with a frequency of 6 cycles per second in the signal.

For “k_7”:
“a_7” is: 0.01756
“b_7” is: −0.11086
Therefore, the amplitude is: 0.1122 units.
The phase is: 4.8695 radians, which is 1.55π radians.

For “k_8”:
“a_8” is: 0
“b_8” is: 0
Therefore, there is no wave with a frequency of 8 cycles per second in the signal.

For “k_9”:
“a_9” is: −0.0902
“b_9” is: 0.04600
Therefore, the amplitude is: 0.1013 units.
The phase is: 2.6700 radians.

[If we had used slightly more accuracy, we would have calculated the phase as 0.85π.]

For “k_{10}”, which is when the test frequency is equal to half the sample rate, the amplitude is calculated as half what it would be for the other test waves. In this particular example, this is irrelevant as there is no wave with a frequency of 10 cycles per second:
“a_{10}” is: 0
“b_{10}” is: 0

To be consistent with our synthesis formula, we will express the waves that do not exist in the signal as waves with zero amplitudes. Our full list of discrete waves is as so:

“k_0”: “y = 0.5 sin ((2π * 0 * 0.05n + 0.5π)” [This is the mean level.]
“k_1”: “y = 0.6393 sin ((2π * 1 * 0.05n) + 1.65π)”
“k_2”: “y = 0 sin (2π * 2 * 0.05n)”
“k_3”: “y = 0.2203 sin ((2π * 3 * 0.05n) + 0.95π)
“k_4”: “y = 0 sin (2π * 4 * 0.05n)”
“k_5”: “y = 0.1414 sin ((2π * 5 * 0.05n) + 0.25π)”
“k_6”: “y = 0 sin (2π * 6 * 0.05n)”
“k_7”: “y = 0.1122 sin ((2π * 7 * 0.05n) + 1.55π)”
“k_8”: “y = 0 sin (2π * 8 * 0.05n)”
“k_9”: “y = 0.1013 sin ((2π * 9 * 0.05n) + 2.67)”
“k_{10}”: “y = 0 sin (2π * 10 * 0.05n)”

If we were being less pedantic, we would ignore the waves that are not there, and give the list as so:

"k_0": "y = 0.5 sin ((2π * 0 * 0.05n + 0.5π)" [This is the mean level.]
"k_1": "y = 0.6393 sin ((2π * 1 * 0.05n) + 1.65π)"
"k_2": [No wave]
"k_3": "y = 0.2203 sin ((2π * 3 * 0.05n) + 0.95π)
"k_4": [No wave]
"k_5": "y = 0.1414 sin ((2π * 5 * 0.05n) + 0.25π)"
"k_6": [No wave]
"k_7": "y = 0.1122 sin ((2π * 7 * 0.05n) + 1.55π)"
"k_8": [No wave]
"k_9": "y = 0.1013 sin ((2π * 9 * 0.05n) + 2.67)"
"k_{10}": [No wave]

If we give the list of constituent waves as we would normally, we would give them as so:

"y = 0.6393 sin ((2π * 1 * 0.05n) + 1.65π)"
"y = 0.2203 sin ((2π * 3 * 0.05n) + 0.95π)"
"y = 0.1414 sin ((2π * 5 * 0.05n) + 0.25π)"
"y = 0.1122 sin ((2π * 7 * 0.05n) + 1.55π)"
"y = 0.1013 sin ((2π * 9 * 0.05n) + 2.67)"
... and a mean level of 0.5 units.

We can test the accuracy of these results by seeing the samples from the signal made from adding these waves (with each sample rounded up to 4 decimal places). As our original signal consisted only of values that were 0 or 1, our signal should have values that are equal to, or very close to, 0 or 1.

Sample number	Sum of the samples of the discovered waves
0	0.0001
1	−0.0003
2	0.0001
3	−0.0001
4	1.0000
5	1.0000
6	1.0002
7	0.9999
8	1.0000
9	1.0001
10	0.9999
11	1.0003
12	0.9999
13	1.0001
14	0.0000
15	0.0000
16	−0.0002
17	0.0001
18	0.0000
19	−0.0001

As we can see, when our samples are rounded up to 4 decimal places, the discrete signal created by adding our discovered constituent discrete waves is almost identical to the original square wave. If we rounded up the results to 3 decimal places, the samples would be correct. This all shows that despite only having a sample rate of 20 samples per second, and despite only being able to use 10 discrete waves and a mean level, we can analyse the most extreme form of discrete signal surprisingly accurately. Given that we were arbitrarily using samples rounded up to 4 decimal places throughout this example, this is a good result. If we had used more decimal places, our results would have been essentially perfect.

If we had analysed a continuous square wave using continuous Fourier series analysis, no matter how many constituent waves we discovered, and no matter how accurately we calculated them, adding them together would never have produced an accurate copy of the original signal. When we analyse a discrete signal with discrete Fourier series analysis, we get accurate results, with the downside that our discrete signal is itself an approximation to a continuous signal.

One advantage of discrete signals from the point of view of the work involved is that there will always be a limit to the number of constituent waves that we have to find. As discussed earlier, this will always be equal to half the number of samples within one period of the signal being analysed.

Random samples

We will analyse a discrete signal that is made up of random samples. This will reinforce how well the method works for discrete signals. Our signal will be made up of the following samples, each of which has been randomly generated by Microsoft Excel. [They are sufficiently random for an example such as this.] We will say that there is a sample rate of 32 samples per second and that the signal has a frequency of 2 cycles per second. Therefore, there are 16 samples in one period of the signal. This implies that the signal will consist of a mean level and 8 constituent waves (although the mean level might be zero, and some of the constituent waves might have zero amplitude.) The randomly created samples are as follows:

Time	**Sample number**	**Value**
t = 0.00000	0	−1.8600
t = 0.03125	1	2.3500
t = 0.06250	2	2.6800
t = 0.09375	3	−4.9400
t = 0.12500	4	−1.2500
t = 0.15625	5	3.9800
t = 0.18750	6	4.6500
t = 0.21875	7	2.8400
t = 0.25000	8	−0.9100
t = 0.28125	9	0.1900
t = 0.31250	10	−2.1900
t = 0.34375	11	3.5900
t = 0.37500	12	0.7200
t = 0.40625	13	−0.5700
t = 0.43750	14	−4.6600
t = 0.46875	15	3.3800

The next sample would be the start of the next cycle:

t = 0.50000	16	−1.8600

We will start with the general synthesis formula:

$$f[n] = \sum_{k=0}^{0.5N} c_k \, sin\,((2\pi * f * k * T_s * n) + \phi_k)$$

We can fill in the details that we know. There are 16 samples in one period of the signal. The sample rate is 32 samples per second, so the sampling period is 1 ÷ 32 = 0.03125 seconds. The fundamental frequency of the signal is 2 cycles per second.

$$f[n] = \sum_{k=0}^{0.5 * 16} c_k \, sin\,((2\pi * 2 * k * 0.03125 * n) + \phi_k)$$

We can tidy that up as:

$$f[n] = \sum_{k=0}^{8} c_k \, sin\,((2\pi * 2k * 0.03125n) + \phi_k)$$

From the formula (or from knowing that there are 16 samples in one period), we know that we will only need to test for the mean level and 8 waves.

The mean level is 0.5 units. We therefore remove this from each of the samples. Our new list of samples, upon which we will perform our calculations, is as so:

−2.36, 1.85, 2.18, −5.44, −1.75, 3.48, 4.15, 2.34, −1.41, −0.31, −2.69, 3.09, 0.22, −1.07, −5.16, 2.88

In this example, we will not round up the values of the test waves, the results of the multiplications, or the values of "a_k" and "b_k". We will round up the final amplitudes and phases to 4 decimal places. We will skip the calculations, and go straight to the "a_k" and "b_k" results, which we will show to 8 decimal places (but we will use as much accuracy as possible in the *calculations* of the phase and amplitudes).

For "k_1":
"a_1" is: 0.62496010
"b_1" is: −0.82507233

The discrete wave is:
"y = 1.0350 sin ((2π * 2 * 0.03125n) + 5.3606)"

For “k_2”:
“a_2” is: −0.68349765
“b_2” is: 0.31220193

The discrete wave is:
“y = 0.7514 sin ((2π * 4 * 0.03125n) + 2.7131)”

For “k_3”:
“a_3” is: 1.87706738
“b_3” is: 1.91339296

The discrete wave is:
“y = 2.6804 sin ((2π * 6 * 0.03125n) + 0.7950)”

For “k_4”:
“a_4” is: 0.13500000
“b_4” is: −0.47250000
[By chance, “a_4” only has 3 decimal places, and “b_4” only has 4 decimal places.]

The discrete wave is:
“y = 0.4914 sin ((2π * 8 * 0.03125n) + 4.9907)”

For “k_5”:
“a_5” is: −1.12212616
“b_5” is: −1.36600443

The discrete wave is:
“y = 1.7678 sin ((2π * 10 * 0.03125n) + 4.0247)”

For “k_6”:
“a_6” is: −0.80849765
“b_6” is: −0.87220193

The discrete wave is:
“y = 1.1893 sin ((2π * 12 * 0.03125n) + 3.9649)”

For "k_7":
"a_7" is: −1.38923344
"b_7" is: −0.19731619

The discrete wave is:
"y = 1.4032 sin ((2π * 14 * 0.03125n) + 3.2827)"

For "k_8":
"a_8" is: 0
"b_8" is: −1.70500000

As we are testing for the frequency equal to half the sample rate, we have to halve the amplitude. Without halving the amplitude, the discrete wave would be:
"y = 1.7050 sin ((2π * 16 * 0.03125n) + 4.7124)"

With the amplitude halved, the discrete wave is:
"y = 0.8525 sin ((2π * 16 * 0.03125n) + 4.7124)"
... which is:
"y = 0.8525 sin ((2π * 16 * 0.03125n) + 1.5π)"

[Remember that when testing for the frequency equal to half the sample rate, if we find a constituent wave, it will always have a phase of 0.5π radians or 1.5π radians. This was explained in the previous chapter.]

Our full list of discrete waves is as so:
"y = 1.0350 sin ((2π * 2 * 0.03125n) + 5.3606)"
"y = 0.7514 sin ((2π * 4 * 0.03125n) + 2.7131)"
"y = 2.6804 sin ((2π * 6 * 0.03125n) + 0.7950)"
"y = 0.4914 sin ((2π * 8 * 0.03125n) + 4.9907)"
"y = 1.7678 sin ((2π * 10 * 0.03125n) + 4.0247)"
"y = 1.1893 sin ((2π * 12 * 0.03125n) +3.9649)"
"y = 1.4032 sin ((2π * 14 * 0.03125n) + 3.2827)"
"y = 0.8525 sin ((2π * 16 * 0.03125n) + 1.5π)"
... and a mean level of 0.5 units, which can be phrased as this discrete wave:
"y = 0.5 sin ((2π * 0 * 0.03125n) + 0.5π)"

We will add up these waves using the amplitudes and phases given to 8 decimal places, and compare the resulting signal's samples (rounded to 8 decimal places) with the samples from the signal with which we started:

Sample number	**Original sample**	**Sample from adding discovered constituent waves**	**Difference**
0	−1.8600	−1.85999999	−0.00000001
1	2.3500	2.35000000	0.00000000
2	2.6800	2.67999999	0.00000001
3	−4.9400	−4.94000000	0.00000000
4	−1.2500	−1.25000001	0.00000001
5	3.9800	3.98000000	0.00000000
6	4.6500	4.65000000	0.00000000
7	2.8400	2.84000001	−0.00000001
8	−0.9100	−0.91000000	0.00000000
9	0.1900	0.19000000	0.00000000
10	−2.1900	−2.19000001	0.00000001
11	3.5900	3.58999999	0.00000001
12	0.7200	0.72000002	−0.00000002
13	−0.5700	−0.57000001	0.00000001
14	−4.6600	−4.66000000	0.00000000
15	3.3800	3.38000001	−0.00000001

As we can see, the difference between the samples from adding up our discovered constituent waves and the original samples is negligible. If we rounded up the discovered samples to 7 decimal places, there would be no difference at all. This shows that not only does the discrete method of finding constituent waves work, but (if we use enough accuracy) it works *exactly*. With *continuous* Fourier series analysis, there are many continuous signals for which we would only find an approximate sum of waves. With discrete Fourier analysis, we find a set of waves that when summed have *exactly* the same samples as the discrete signal being analysed.

In this example, the discrete waves we discovered are not the waves that originally made up the signal because the original signal was not made from adding waves – it was made from random samples. Our list of discrete waves are the waves that when added together create an identical signal to our random-sample signal. Supposing our original signal *had* been created by adding waves, then our discovered constituent waves would all be identical to those that were added,

except for the wave with a frequency equal to half the sample rate. That wave would have had the same samples, but it might not have been the same wave.

If we had not tested for the frequency equal to half the sample rate, the samples from the sum of our discovered constituent waves would have been incorrect by approximately −0.8525, +0.8525, −0.8525, +0.8525, −0.8525, +0.8525 and so on. This means that the signal would be missing a wave that had those samples. Chapter 42 on sampling taught us that if a signal has alternate negative and positive samples of the same absolute value, then it must have had a frequency equal to half the sample rate, and it will be discovered as a wave with a phase of 1.5π radians. Therefore, by comparing our results with what they should be, we can tell that we need to test for the frequency equal to half the sample rate. [As always when testing for the frequency equal to half the sample rate, we must halve the discovered amplitude, and we must be aware that, although the discovered wave will have the correct samples, it will probably not be the original wave, not that there was an "original" wave in this example.]

A well-sampled signal would only contain frequencies below half the sample rate, and for most continuous waves that have been correctly filtered before sampling, this will be true. In this case, our samples are random numbers, which means that they did not come from a correctly sampled wave.

This example confirms that it is possible to find a series of constituent waves that when added have the same samples as the signal we are analysing. In a typical discrete signal, we would hope that there would be no frequencies equal to half the sample rate, but this example shows that, first, we can tell if there is such a frequency, and second, we can find a constituent wave that has the relevant samples.

More random samples

We will analyse another randomly created signal. This time, we will use a sample rate of 11 samples per second.

As this is an odd sample rate, it pays to remember the section on "Odd sample rates" from the previous chapter. In the example that we have just done, we had a sample rate of 32 samples per second and a signal with a fundamental frequency of 2 cycles per second. One period of the signal fitted into 16 samples, so we needed only 16 samples for the analysis. It also turned out that we needed to test for the frequency equal to half the sample rate, which was not a problem. For this

example, it would be simplest to go through the steps of analysing a 1-cycle-per-second signal. To do that, we would need only 11 samples. However, for randomly created samples, we would almost certainly have a constituent wave with a frequency equal to half the sample rate. If that frequency exists in the signal, then the fundamental frequency cannot be 1 cycle per second – the presence of a 5.5-cycle-per-second wave would make the fundamental frequency 0.5 cycles per second. This means that we cannot do a 1-cycle-per-second example with 11 randomly created samples. Instead, we must do a 0.5-cycle-per-second example with 22 randomly created samples. [If you were attempting to practise analysing signals yourself, you might come across this problem.]

Anyway, we will analyse these 22 samples:

Time	**Sample number**	**Value**
t = 0.0000	0	−2.96
t = 0.0909	1	−2.83
t = 0.1818	2	2.13
t = 0.2727	3	−2.09
t = 0.3636	4	4.42
t = 0.4545	5	1.15
t = 0.5455	6	−3.66
t = 0.6364	7	−2.66
t = 0.7273	8	1.14
t = 0.8182	9	1.93
t = 0.9091	10	−4.87
t = 1.0000	11	1.56
t = 1.0909	12	4.91
t = 1.1818	13	1.53
t = 1.2727	14	2.88
t = 1.3636	15	−4.43
t = 1.4545	16	4.98
t = 1.5455	17	−3.10
t = 1.6364	18	−4.21
t = 1.7273	19	−0.64
t = 1.8182	20	2.87
t = 1.9091	21	−4.81

The sample rate is 11 samples per second, and the fundamental frequency is 0.5 cycles per second. The test frequencies will be integer multiples of 0.5 cycles per second. We start with the general synthesis formula:

$$f[n] = \sum_{k=0}^{0.5N} c_k \, sin\left((2\pi * f * k * T_s * n) + \phi_k\right)$$

We fill in the details that we know:

$$f[n] = \sum_{k=0}^{11} c_k \, sin\left((2\pi * 0.5 * k * (1/11) * n) + \phi_k\right)$$

As the formula indicates, we will only need to test for the mean level and 11 constituent waves.

The first step is to calculate the mean level. We add up the samples and divide by the number of samples. The mean level is −0.30727273 to 8 decimal places. We can also go through the less straightforward "formula" way of calculating the mean level. We will fill in the details for this:

$$c_0 = \frac{1}{N} * \sum_{n=0}^{N-1} f[n]$$

It becomes:

$$c_0 = \frac{1}{22} * \sum_{n=0}^{21} f[n]$$

It means that we go through the signal "f[n]" adding up the samples from sample number 0 up to sample number 21, and then multiply the result by "1 / 22".

However we calculate the mean level, we will remove it from the original signal. We then perform the analysis on the new signal, using test frequencies of:
0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, and 5.5 cycles per second.

For this example, we will skip the actual calculations, and go straight to the calculated waves. We will use a high number of decimal places in all our calculations to achieve a good level of accuracy.

For the results shown here, the amplitudes and phases are rounded to 8 decimal places:

k = 0: “y = −0.30727273 sin ((2π * 0t) + 0.5π)”, which is the mean level.
k = 0.5: “y = 1.13188251 sin ((2π * 0.5t) + 4.73830878)”
k = 1.0: “y = 1.31259018 sin ((2π * 1t) + 0.04484107)”
k = 1.5: “y = 1.08444466 sin ((2π * 1.5t) + 4.09633258)”
k = 2.0: “y = 1.05447761 sin ((2π * 2t) + 4.35984915)”
k = 2.5: “y = 1.25982211 sin ((2π * 2.5t) + 3.16228894)”
k = 3.0: “y = 2.58516838 sin ((2π * 3t) + 5.79748573)”
k = 3.5: “y = 0.88940241 sin ((2π * 3.5t) + 5.47071166)”
k = 4.0: “y = 0.89137765 sin ((2π * 4t) + 1.53199820)”
k = 4.5: “y = 1.57687691 sin ((2π * 4.5t) + 3.51317806)”
k = 5.0: “y = 1.29230990 sin ((2π * 5t) + 0.72211303)”
k = 5.5: “y = 1.00090909 sin ((2π * 5.5t) + 0.5π)”

[Remember to halve the discovered amplitude of the frequency equal to half the sample rate. The value above is the correct halved one. One way to be sure that we are calculating in the correct way is to check that the phase of the 5.5-cycle-per-second wave has a phase of either 0.5π or 1.5π radians. The phase of the frequency equal to half the sample rate will always be 0.5π or 1.5π radians, as explained in the last chapter.]

The sum of the above waves has *exactly* the same samples as our original signal. This confirms again that discrete Fourier series analysis works perfectly.

Summary of important ideas

We will look at some important ideas that we have learnt in this chapter.

- If we are analysing a periodic discrete signal, we will be able to find a set of discrete constituent waves that, when added together, have the same samples.

- For a signal created by adding waves, we will be able to find the waves that were added except for the wave with a frequency equal to half the sample rate, if such a wave existed. For that wave, depending on its phase, it might have been lost in the sampling process, it might be recoverable exactly, or a wave with the same samples might be found instead.

- Any discrete periodic signal can be replicated with a mean level and a number of constituent waves equal to half the number of samples in one period of the signal.

- Therefore, we only need to calculate the mean level and then test for a number of waves equal to half the number of samples in one period of the signal. If we test for more waves, we will just end up finding duplicates of the waves that we have already found.

- If a continuous signal was sampled sensibly, in the sense that all the constituent frequencies were lower than half the sample rate, then we do not need to test for the frequency equal to half the sample rate. Some people use formulas based on the idea that every discrete signal was sampled sensibly. Such formulas will fail if the signal was not sampled sensibly.

Conclusion

As with continuous Fourier series analysis, the most complicated part of discrete Fourier series analysis is the range of formulas that explain the process. The process can be explained more simply with words than with formulas, and ultimately the formulas leave out much that is needed to understand them. As with continuous Fourier series analysis, the formulas are a step towards more complicated concepts, so it can be useful to understand them.

Discrete Fourier series analysis has a more complicated-sounding name, but it is actually an easier process than continuous Fourier series analysis. You can see this for yourself by using a spreadsheet program such as Microsoft Excel to analyse a signal with a low sample rate.