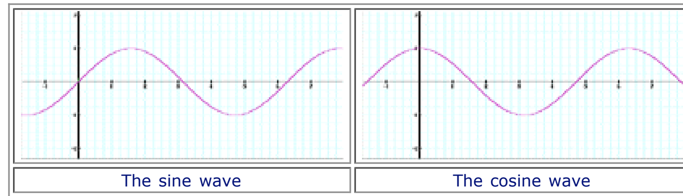
The DFT "à Pied":
Mastering The Fourier Transform in One Day

by Stephan M. Bernsee, http://www.dspdimension.com, *© 1999-2005 all rights reserved* *

If you're into signal processing, you will no doubt say that the headline is a very tall claim. I would second this. Of course you can't learn all the bells and whistles of the Fourier transform in one day without practising and repeating and eventually delving into the maths. However, this online course will provide you with the basic knowledge of how the Fourier transform works, why it works and why it can be very simple to comprehend when we're using a somewhat unconventional approach.The important part: you will learn the basics of the Fourier transform completely without any maths that goes beyond adding and multiplying numbers! I will try to explain the Fourier transform in its practical application to audio signal processing in no more than six paragraphs below.

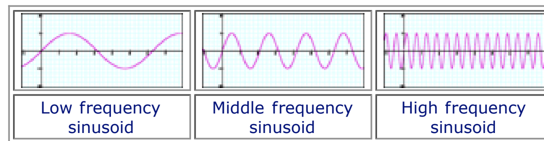
Step 1: Some simple prerequisites

What you need to understand the following paragraphs are essentially four things: how to add numbers, how to multiply and divide them and what a sine, a cosine and a sinusoid is and how they look. Obviously, I will skip the first two things and just explain a bit the last one. You probably remember from your days at school the 'trigonometric functions'[1] that were somehow mysteriously used in conjunction with triangles to calculate the length of its sides from its inner angles and vice versa. We don't need all these things here, we just need to know how the two most important trigonometric functions, the "sine" and "cosine" look like. This is quite simple: they look like very simple waves with peaks and valleys in them that stretch out to infinity to the left and the right of the observer.



The sine wave	The cosine wave

As you can see, both waves are *periodic*, which means that after a certain time, the *period*, they look the same again. Also, both waves look alike, but the cosine wave appears to start at its maximum, while the sine wave starts at zero. Now in practice, how can we tell whether a wave we observe at a given time started out at its maximum, or at zero? Good question: we can't. There's no way to discern a sine wave and a cosine wave in practice, thus we call any wave that looks like a sine or cosine wave a "*sinusoid*", which is Greek and translates to "sinus-like". An important property of sinusoids is "*frequency*", which tells us how many peaks and valleys we can count in a given period of time. High frequency means many peaks and valleys, low frequency means few peaks and valleys:

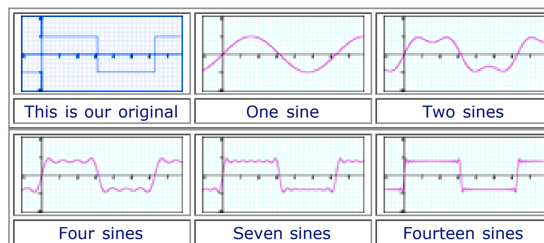


Low frequency sinusoid	Middle frequency sinusoid	High frequency sinusoid

Step 2: Understanding the Fourier Theorem

Jean-Baptiste Joseph Fourier was one of those children parents are either proud or ashamed of, as he started throwing highly complicated mathematical terms at them at the age of fourteen. Although he did a lot of important work during his lifetime, the probably most significant thing he discovered had to do with the conduction of heat in materials. He came up with an equation that described how the heat would travel in a certain medium, and solved this equation with an infinite series of trigonometric functions (the sines and cosines we have discussed above). Basically, and related to our topic, what Fourier discovered boils down to the general rule that every signal, however complex, can be represented by a sum of sinusoid functions that are individually mixed.

An example of this:

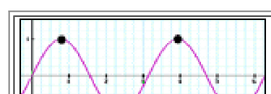


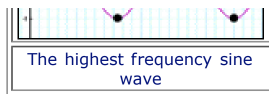
This is our original	One sine	Two sines
Four sines	Seven sines	Fourteen sines

What you see here is our original signal, and how it can be approximated by a mixture of sines (we will call them *partials*) that are mixed together in a certain relationship (a 'recipe'). We will talk about that recipe shortly. As you can see, the more sines we use the more accurately does the result resemble our original waveform. In the 'real' world, where signals are continuous, ie. you can measure them in infinitely small intervals at an accuracy that is only limited by your measurement equipment, you would need infinitely many sines to perfectly build any given signal. Fortunately, as DSPers we're not living in such a world. Rather, we are dealing with samples of such 'realworld' signals that are measured at regular intervals and only with finite precision. Thus, we don't need infinitely many sines, we just need a lot. We will also talk about that 'how much is a lot' later on. For the moment, it is important that you can imagine that every signal you have on your computer can be put together from simple sine waves, after some cooking recipe.

Step 3: How much is "a lot"

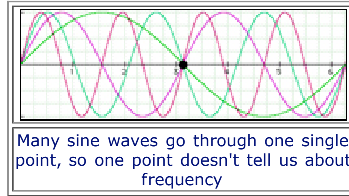
As we have seen, complex shaped waveforms can be built from a mixture of sine waves. We might ask how many of them are needed to build any given signal on our computer. Well, of course, this may be as few as one single sine wave, provided we know how the signal we are dealing with is made up. In most cases, we are dealing with realworld signals that might have a very complex structure, so we do not know in advance how many 'partial' waves there are actually present. In this case, it is very reassuring to know that if we don't know how many sine waves constitute the original signal there is an upper limit to how many we will need. Still, this leaves us with the question of how many there actually are. Let's try to approach this intuitively: assume we have 1000 samples of a signal. The sine wave with the shortest period (ie. the most peaks and valleys in it) that can be present has alternating peaks and valleys for every sample. So, the sine wave with the highest frequency has 500 peaks and 500 valleys in our 1000 samples, with every other sample being a peak. The black dots in the following diagram denote our samples, so the sine wave with the highest frequency looks like this:





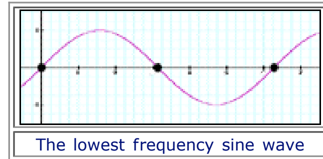
The highest frequency sine wave

Now let's look how low the lowest frequency sine wave can be. If we are given only one single sample point, how would we be able to measure peaks and valleys of a sine wave that goes through this point? We can't, as there are many sine waves of different periods that go through this point.



Many sine waves go through one single point, so one point doesn't tell us about frequency

So, a single data point is not enough to tell us anything about frequency. Now, if we were given two samples, what would be the lowest frequency sine wave that goes through these two points? In this case, it is much simpler. There is one very low frequency sine wave that goes through the two points. It looks like this:



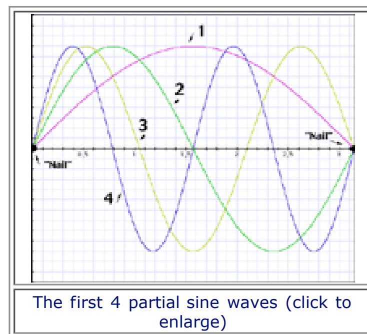
The lowest frequency sine wave

Imagine the two leftmost points being two nails with a string spanned between them (the diagram depicts three data points as the sine wave is periodic, but we really only need the leftmost two to tell its frequency). The lowest frequency we can see is the string swinging back and forth between the two nails, like our sine wave does in the diagram between the two points to the left. If we have 1000 samples, the two 'nails' would be the first and the last sample, ie. sample number 1 and sample number 1000. We know from our experience with musical instruments that the frequency of a string goes down when its length increases. So we would expect that our lowest sine wave gets lower in frequency when we move our nails farther away from each other. If we choose 2000 samples, for instance, the lowest sine wave will be much lower since our 'nails' are now sample number 1 and sample number 2000. In fact, it will be twice as low, since our nails are now twice as far away as in the 1000 samples. Thus, if we have more samples, we can discern sine waves of a lower frequency since their zero crossings (our 'nails') will move farther away. This is very important to understand for the following explanations.

As we can also see, after two 'nails' our wave starts to repeat with the ascending slope (the first and the third nail are identical). This means that any two adjacent nails embrace exactly one half of the complete sine wave, or in other words either one peak or one valley, or 1/2 period.

Summarizing what we have just learned, we see that the *upper frequency* of a sampled sine wave is every other sample being a peak and a valley and the *lower frequency* bound is half a period of the sine wave which is just fitting in the number of samples we are looking at. But wait - wouldn't this mean that while the upper frequency remains fixed, the lowest frequency would drop when we have more samples? Exactly! The result of this is that we will need more sine waves when we want to put together longer signals of unknown content, since we start out at a lower frequency.

All well and good, but still we don't know how many of these sine waves we finally need. As we now know the lower and upper frequency any partial sine wave can have, we can calculate how many of them fit in between these two limits. Since we have nailed our lowest partial sine wave down to the leftmost and rightmost samples, we require that all other sine waves use these nails as well (why should we treat them differently? All sine waves are created equal!). Just imagine the sine waves were strings on a guitar attached to two fixed points. They can only swing between the two nails (unless they break), just like our sine waves below. This leads to the relationship that our lowest partial (1) fits in with 1/2 period, the second partial (2) fits in with 1 period, the third partial (3) fits in with 1 1/2 period asf. into the 1000 samples we are looking at. Graphically, this looks like this:



The first 4 partial sine waves (click to enlarge)

Now if we count how many sine waves fit in our 1000 samples that way, we will find that we need exactly 1000 sine waves added together to represent the 1000 samples. In fact, we will always find that we need as many sine waves as we had samples.

Step 4: About cooking recipes

In the previous paragraph we have seen that any given signal on a computer can be built from a mixture of sine waves. We have considered their frequency and what frequency the lowest and highest sine waves need to have to perfectly reconstruct any signal we analyze. We have seen that the number of samples we are looking at is important for determining the lowest partial sine wave that is needed, but we have not yet discussed how the actual sine waves have to be mixed to yield a certain result. To make up any given signal by adding sine waves, we need to measure one additional aspect of them. As a matter of fact, frequency is not the only thing we need to know. We also need to know the *amplitude* of the sine waves, ie. how much of each sine wave we need to mix together to reproduce our input signal. The amplitude is the height of the peaks of a sine wave, ie. the distance between the peak and our zero line. The higher the amplitude, the louder it will sound when we listen to it. So, if you have a signal that has lots of bass in it you will no doubt expect that there must be a greater portion of lower frequency sine waves in the mix than there are higher frequency sine waves. So, generally, the low frequency sine waves in a bassy sound will have a higher amplitude than the high frequency sine waves. In our analysis, we will need to determine the amplitude of each partial sine wave to complete our recipe.

Step 5: About apples and oranges

If you are still with me, we have almost completed our journey towards the Fourier transform. We have learned how many sine waves we need, that this number depends on the number of samples we are looking at, that there is a lower and upper frequency boundary and that we somehow need to determine the amplitude of the individual partial waves to complete our recipe. We're still not clear, however, on how we can determine the actual recipe from our samples. Intuitively, we would say that we could find the amplitudes of the sine waves somehow by *comparing* a sine wave of known frequency to the samples we have measured and find out how 'equal' they are. If they are exactly equal, we know that the sine wave must be present at the same amplitude, if we find our signal to not match our reference sine wave at all we would expect this frequency not to be present. Still, how could we effectively compare a known sine wave with our sampled signal? Fortunately, DSPers have already figured out how to do this for you. In fact, this is as easy as multipling and adding numbers - we take the 'reference' sine wave of known frequency and unit amplitude (this just

means that it has an amplitude of 1, which is exactly what we get back from the `sin()` function on our pocket calculator or our computer) and multiply it with our signal samples. After adding the result of the multiplication together, we will obtain the amplitude of the partial sine wave at the frequency we are looking at. To illustrate this, here's a simple C code fragment that does this:

Listing 1.1: The direct realization of the Discrete Sine Transform (DST):

```
#define M_PI 3.14159265358979323846

	long bin,k;
	double arg;
	for (bin = 0; bin < transformLength; bin++) {

		transformData[bin] = 0.;
		for (k = 0; k < transformLength; k++) {

			arg = (float)bin * M_PI *(float)k / (float)transformLength;
			transformData[bin] += inputData[k] * sin(arg);

		}

	}
```

This code segment transforms our measured sample points that are stored in `inputData[0...transformLength-1]` into an array of amplitudes of its partial sine waves `transformData[0...transformLength-1]`. According to common terminology, we call the frequency steps of our reference sine wave *bins*, which means that they can be thought of as being 'containers' in which we put the amplitude of any of the partial waves we evaluate. The Discrete Sine Transform (DST) is a generic procedure that assumes we have no idea what our signal looks like, otherwise we could use a more efficient method for determining the amplitudes of the partial sine waves (if we, for example, know beforehand that our signal is a single sine wave of known frequency we could directly check for its amplitude without calculating the whole range of sine waves. An efficient approach for doing this based on the Fourier theory can be found in the literature under the name the "Goertzel" algorithm).

For those of you who insist on an explanation for why we calculate the sine transform that way: As a very intuitive approach to why we multiply with a sine wave of known frequency, imagine that this corresponds roughly to what in the physical world happens when a 'resonance' at a given frequency takes place in a system. The `sin(arg)` term is essentially a resonator that gets excited by our input waveform. If the input has a partial at the frequency we're looking at, its output will be the amplitude of the resonance with the reference sine wave. Since our reference wave is of unit amplitude, the output is a direct measure of the actual amplitude of the partial at that frequency. Since a resonator is nothing but a simple filter, the transform can (admittedly under somewhat relaxed conditions) be seen as a having the features of a bank of very narrow band pass filters that are centered around the frequencies we're evaluating. This helps explaining the fact why the Fourier transform provides an efficient tool for performing filtering of signals.

Just for the sake of completeness: of course, the above routine is invertible, our signal can (within the limits of our numerical precision) be perfectly reconstructed when we know its partial sine waves, by simply adding sine waves together. This is left as an exercise to the reader. The same routine can be changed to work with cosine waves as basis functions - we simply need to change the `sin(arg)` term to `cos(arg)` to obtain the direct realization of the Discrete Cosine Transform (DCT).

Now, as we have discussed in the very first paragraph of this article, in practice we have no way to classify a measured sinus-like function as sine wave or cosine wave. Instead we are always measuring *sinusoids*, so both the sine and cosine transform are of no great use when we are applying them in practice, except for some special cases (like image compression where each image might have features that are well modelled by a cosine or sine basis function, such as large areas of the same color that are well represented by the cosine basis function). A sinusoid is a bit more general than the sine or cosine wave in that it can start at an arbitrary position in its period. We remember that the sine wave always starts out at zero, while the cosine wave starts out at one. When we take the sine wave as reference, the cosine wave starts out 1/4th later in its period. It is common to measure this *offset* in *degree* or *radians*, which are two units commonly used in conjunction with trigonometric functions. One complete period equals 360° (pron. "degree") or 2π radian (pron. "two pi" with "pi" pronounced like the word "pie". π is a Greek symbol for the number 3.14159265358979323846... which has some significance in trigonometry). The cosine wave thus has an offset of 90° or π/2. This offset is called the *phase* of a sinusoid, so looking at our cosine wave we see that it is a sinusoid with a phase offset of 90° or π/2 relative to the sine wave.

So what's this phase business all about. As we can't restrict our signal to start out at zero phase or 90° phase all the time (since we are just observing a signal which might be beyond our control) it is of interest to determine its frequency, amplitude and phase to uniquely describe it at any one time instant. With the sine or cosine transform, we're restricted to zero phase or 90° phase and any sinusoid that has an arbitrary phase will cause adjacent frequencies to show spurious peaks (since they try to 'help' the analysis to force-fit the measured signal to a sum of zero or 90° phase functions). It's a bit like trying to fit a round stone into a square hole: you need smaller round stones to fill out the remaining space, and even more even smaller stones to fill out the space that is still left empty, and so on. So what we need is a transform that is general in that it can deal with signals that are built of sinusoids of arbitrary phase.

Step 6: The Discrete Fourier transform.

The step from the sine transform to the Fourier transform is simple, making it in a way more 'general'. While we have been using a sine wave for each frequency we measure in the sine transform, we use both a sine and a cosine wave in the Fourier transform. That is, for any frequency we are looking at we 'compare' (or 'resonate') our measured signal with both a cosine and a sine wave of the same frequency. If our signal looks much like a sine wave, the sine portion of our transform will have a large amplitude. If it looks like a cosine wave, the cosine part of our transform will have a large amplitude. If it looks like the opposite of a sine wave, that is, it starts out at zero but drops to -1 instead of going up to 1, its sine portion will have a large *negative* amplitude. It can be shown that the + and - *sign* together with the *sine and cosine phase* can represent any sinusoid at the given frequency[2].

Listing 1.2: The direct realization of the Discrete Fourier Transform[3]:

```
#define M_PI 3.14159265358979323846

	long bin, k;
	double arg, sign = -1.; /* sign = -1 -> FFT, 1 -> iFFT */

	for (bin = 0; bin <= transformLength/2; bin++) {

		cosPart[bin] = (sinPart[bin] = 0.);
		for (k = 0; k < transformLength; k++) {

			arg = 2.*(float)bin*M_PI*(float)k/(float)transformLength;

			sinPart[bin] += inputData[k] * sign * sin(arg);
			cosPart[bin] += inputData[k] * cos(arg);

		}

	}
```

We're still left with the problem of how to get something useful out of the Fourier Transform. I have claimed that the benefit of the Fourier transform over the Sine and Cosine transform is that we are working with sinusoids. However, we don't see any sinusoids yet, there are still only sines and cosines. Well, this requires an additional processing step:

Listing 1.3: Getting sinusoid frequency, magnitude and phase from the Discrete Fourier Transform:

```
#define M_PI 3.14159265358979323846

        long bin;
        for (bin = 0; bin <= transformLength/2; bin++) {

                /* frequency */
                frequency[bin] = (float)bin * sampleRate / (float)transformLength;

                /* magnitude */
                magnitude[bin] = 20. * log10( 2. * sqrt(sinPart[bin]*sinPart[bin] + cosPart[bin]*cosPart[bin]) / (float)transformLength);

                /* phase */
                phase[bin] = 180.*atan2(sinPart[bin], cosPart[bin]) / M_PI - 90.;

        }
```

After running the code fragment shown in Listing 1.3 on our DFT output, we end up with a representation of the input signal as a sum of sinusoid waves. The k-th sinusoid is described by `frequency[k]`, `magnitude[k]` and `phase[k]`. Units are Hz (Hertz, periods per seconds), dB (Decibel) and ° (Degree). Please note that after the post-processing of Listing 1.3 that converts the sine and cosine parts into a single sinusoid, we name the amplitude of the k-th sinusoid the DFT bin "*magnitude*", as it will now always be a positive value. We could say that an amplitude of -1.0 corresponds to a magnitude of 1.0 and a phase of either + or -180°. In the literature, the array `magnitude[]` is called the *Magnitude Spectrum* of the measured signal, the array `phase[]` is called the *Phase Spectrum* of the measured signal at the time where we take the Fourier transform.

As a reference for measuring the bin magnitude in decibels, our input wave is expected to have sample values in the range [-1.0, 1.0), which corresponds to a magnitude of 0dB digital full scale (DFS). As an interesting application of the DFT, listing 1.3 can, for example, be used to write a spectrum analyzer based on the Discrete Fourier Transform.

Conclusion

As we have seen, the Fourier transform and its 'relatives', the discrete sine and cosine transform provide handy tools to decompose a signal into a bunch of partial waves. These are either sine or cosine waves, or sinusoids (described by a combination of sine and cosine waves). The advantage of using both the sine and cosine wave simultaneously in the Fourier transform is that we are thus able to introduce the concept of phase which makes the transform more general in that we can use it to efficiently and clearly analyze sinusoids that are neither a pure sine or cosine wave, and of course other signals as well.

The Fourier transform is independent of the signal under examination in that it requires the same number of operations no matter if the signal we are analyzing is one single sinusoid or something else more complicated. This is the reason why the Discrete Fourier transform is called a *nonparametric* transform, meaning that it is not directly helpful when an 'intelligent' analysis of a signal is needed (in the case where we are examining a signal that we know is a sinusoid, we would prefer just getting information about its phase, frequency and magnitude instead of a bunch of sine and cosine waves at some predefined frequencies).

We now also know that we are evaluating our input signal at a fixed frequency grid (our *bins*) which may have nothing to do with the actual frequencies present in our input signal. Since we choose our reference sine and cosine waves (almost) according to taste with regard to their frequency, the grid we impose on our analysis is artificial. Having said this, it is immediately clear that one will easily encounter a scenario where the measured signal's frequencies may come to lie between the frequencies of our transform bins. Consequently, a sinusoid that has a frequency that happens to lie between two frequency 'bins' will not be well represented in our transform. Adjacent bins that surround the bin closest in frequency to our input wave will try to 'correct' the deviation in frequency and thus the energy of the input wave will be *smeared* over several neighbouring bins. This is also the main reason why the Fourier transform will not readily analyze a sound to return with its *fundamental* and *harmonics* (and this is also why we call the sine and cosine waves *partials*, and not *harmonics*, or *overtones*).

Simply speaking, without further post-processing, the DFT is little more than a bank of narrow, slightly overlapping band pass filters ('channels') with additional phase information for each channel. It is useful for analyzing signals, doing filtering and applying some other neat tricks (changing the pitch of a signal without changing its speed is one of them explained in a different article on DSPdimension.com), but it requires additional post processing for less generic tasks. Also, it can be seen as a special case of a family of transforms that use basis functions other than the sine and cosine waves. Expanding the concept in this direction is beyond the scope of this article.

Finally, it is important to mention that there is a more efficient implementation of the DFT, namely an algorithm called the "Fast Fourier Transform" (FFT) which was originally conceived by Cooley and Tukey in 1969 (its roots however go back to the work of Gauss and others). The FFT is just an efficient algorithm that calculates the DFT in less time than our straightforward approach given above, it is otherwise identical with regard to its results. However, due to the way the FFT is implemented in the Cooley/Tukey algorithm it requires that the transform length be a power of 2. In practice, this is an acceptable constraint for most applications. The available literature on different FFT implementations is vast, so suffice it to say that there are many different FFT implementations, some of which do not have the power-of-two restriction of the classical FFT. An implementation of the FFT is given by the routine `smbFft()` in Listing 1.4 below.

Listing 1.4: The Discrete Fast Fourier Transform (FFT):

```
#define M_PI 3.14159265358979323846

void smbFft(float *fftBuffer, long fftFrameSize, long sign)

/*

        FFT routine, (C)1996 S.M.Bernsee. Sign = -1 is FFT, 1 is iFFT (inverse)

        Fills fftBuffer[0...2*fftFrameSize-1] with the Fourier transform of the
        time domain data in fftBuffer[0...2*fftFrameSize-1]. The FFT array takes
        and returns the cosine and sine parts in an interleaved manner, ie.
        fftBuffer[0] = cosPart[0], fftBuffer[1] = sinPart[0], asf. fftFrameSize
        must be a power of 2. It expects a complex input signal (see footnote
        2), ie. when working with 'common' audio signals our input signal has to
        be passed as {in[0],0.,in[1],0.,in[2],0.,...} asf. In that case, the
        transform of the frequencies of interest is in
        fftBuffer[0...fftFrameSize].

*/

{

        float wr, wi, arg, *p1, *p2, temp;
        float tr, ti, ur, ui, *p1r, *p1i, *p2r, *p2i;
        long i, bitm, j, le, le2, k, logN;
        logN = (long)(log(fftFrameSize)/log(2.)+.5);

        for (i = 2; i < 2*fftFrameSize-2; i += 2) {

                for (bitm = 2, j = 0; bitm < 2*fftFrameSize; bitm <<= 1) {

                        if (i & bitm) j++;
                        j <<= 1;

                }

                if (i < j) {
```

```
                    p1 = fftBuffer+i; p2 = fftBuffer+j;
                    temp = *p1; *(p1++) = *p2;
                    *(p2++) = temp; temp = *p1;
                    *p1 = *p2; *p2 = temp;

            }

        }

        for (k = 0, le = 2; k < logN; k++) {

            le <<= 1;
            le2 = le>>1;
            ur = 1.0;
            ui = 0.0;
            arg = M_PI / (le2>>1);
            wr = cos(arg);
            wi = sign*sin(arg);

            for (j = 0; j < le2; j += 2) {

                    p1r = fftBuffer+j; p1i = p1r+1;
                    p2r = p1r+le2; p2i = p2r+1;

                    for (i = j; i < 2*fftFrameSize; i += le) {

                            tr = *p2r * ur - *p2i * ui;
                            ti = *p2r * ui + *p2i * ur;
                            *p2r = *p1r - tr; *p2i = *p1i -
                            ti;
                            *p1r += tr; *p1i += ti;
                            p1r += le; p1i += le;
                            p2r += le; p2i += le;

                    }

                    tr = ur*wr - ui*wi;
                    ui = ur*wi + ui*wr;
                    ur = tr;

            }

        }

}
```

---

[1] simply speaking, trigonometric functions are functions that are used to calculate the angles in a triangle ("tri-gonos" = Greek for "three corners") from the length of its sides, namely sinus, cosinus, tangent and the arcus tangent. The sinus and cosinus functions are the most important ones, as the tangent and arcus tangent can be obtained from sinus and cosinus relationships alone.

[2] Note that in the literature, due to a generalization that is made for the Fourier transform to work with another type of input signal called a 'complex signal' (complex in this context refers to a certain type of numbers rather than to an input signal that has a complex harmonic structure), you will encounter the sine and cosine part under the name 'real' (for the cosine part) and 'imaginary' part (for the sine part).

[3] if you're already acquainted with the DFT you may have noted that this is actually an implementation of the "real Discrete Fourier Transform", as it uses only real numbers as input and does not deal with negative frequencies: in the real DFT positive and negative frequencies are symmetric and thus redundant. This is why we're calculating only almost half as many bins than in the sine transform (we calculate one additional bin for the highest frequency, for symmetry reasons).

Last change: 12.04.2005, ©1999-2005 S. M. Bernsee, all rights reserved. Content subject to change without notice. Content provided 'as is', see disclaimer. Graphs made using Algebra Graph, MathPad, sonicWORX and other software. Care has been taken to describe everything as simple yet accurate as possible. If you find errors, typos and ambiguous descriptions in this article, please notify me and I will correct or further outline them.

Special thanks to Richard Dobson for providing immensely useful suggestions and corrections to my incomplete knowledge of the English language.