



Introduction to DSP

The BORES Signal Processing DSP course - Introduction to DSP - is free of charge on line.

- [basics](#) - sampling, aliasing, reconstruction and quantisation
- [time domain processing](#) - correlation and convolution
- [frequency analysis](#) - Fourier transforms, resolution, spectral leakage and windowing
- [filtering](#) - including FIR filters
- [IIR filters](#) - design, realisation, and quantisation effects
- [DSP processors](#) - real world requirements, special features
- [programming a DSP processor](#)

These courses are for individual study over the Internet only. All material is copyright: you are not permitted to make copies or print out material, for personal use or for teaching.

Our DSP training classes offer intensive and highly practical training in DSP and Media Processing. Contact us via email - bores@bores.com or telephone +44 (0)1483 740138 for details.

Introduction to DSP - basics

This is the first module of the BORES Signal Processing DSP course - Introduction to DSP. It covers the following subjects:

- [what is DSP?](#)
- [converting analogue signals to digital](#)
- [aliasing and the sampling theroem](#)
- [antialiasing and signal reconstruction](#)
- [frequency resolution](#)
- [quantisation error](#)

These courses are for individual study over the Internet only. All material is copyright: you are not permitted to make copies or print out material, for personal use or for teaching.

Our DSP training classes offer intensive and highly practical training in DSP and Media Processing. Contact us via email - bores@bores.com or telephone +44 (0)1483 740138 for details.

Basics: What is DSP?

Digital Signal Processing (DSP) is used in a wide variety of applications, and it is hard to find a good definition that is general.

We can start by dictionary definitions of the words:

Digital

operating by the use of discrete signals to represent data in the form of numbers

Signal

a variable parameter by which information is conveyed through an electronic circuit

Processing

to perform operations on data according to programmed instructions

Which leads us to a simple definition of:

Digital Signal processing

changing or analysing information which is measured as discrete sequences of numbers

Note two unique features of Digital Signal processing as opposed to plain old ordinary digital processing:

- **signals come from the real world** - this intimate connection with the real world leads to many unique needs such as the need to react in real time and a need to measure signals and convert them to digital numbers
- **signals are discrete** - which means the information in between discrete samples is lost

The advantages of DSP are common to many digital systems and include:

Versatility:

- digital systems can be reprogrammed for other applications (at least where programmable DSP chips are used)
- digital systems can be ported to different hardware (for example a different DSP chip or board level product)

Repeatability:

- digital systems can be easily duplicated
- digital systems do not depend on strict component tolerances
- digital system responses do not drift with temperature

Simplicity:

- some things can be done more easily digitally than with analogue systems

DSP is used in a very wide variety of applications.



but most share some common features:

- they use a lot of maths (multiplying and adding signals)
- they deal with signals that come from the real world
- they require a response in a certain time

Where general purpose DSP processors are concerned, most applications deal with signal frequencies that are in the audio range.

Basics: Converting analogue signals

Most DSP applications deal with analogue signals.

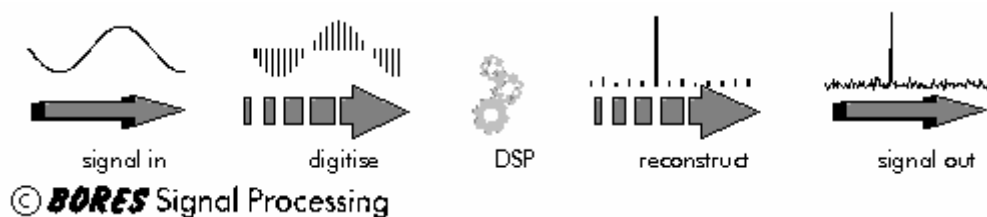
- the analogue signal has to be converted to digital form

The analogue signal - a continuous variable defined with infinite precision - is converted to a discrete sequence of measured values which are represented digitally.

Information is lost in converting from analogue to digital, due to:

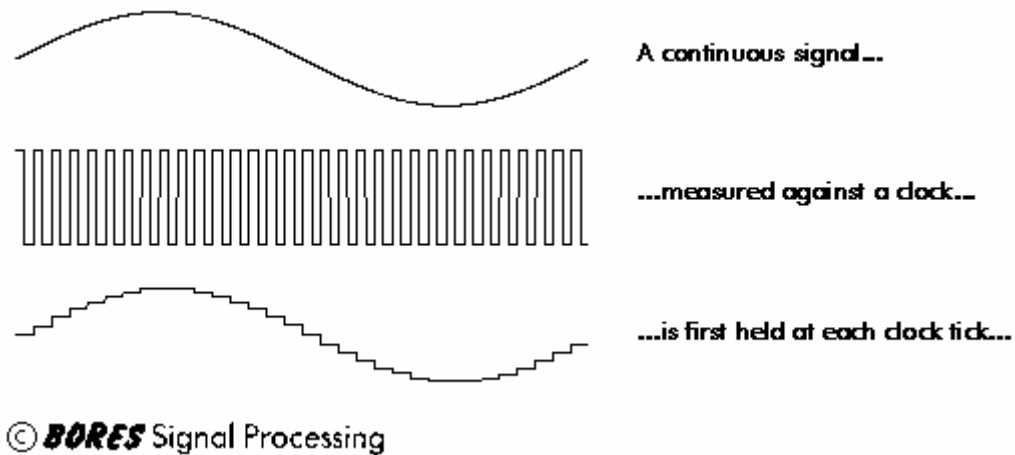
- inaccuracies in the measurement
- uncertainty in timing
- limits on the duration of the measurement

These effects are called quantisation errors.



The continuous analogue signal has to be held before it can be sampled.

Otherwise, the signal would be changing during the measurement.



Only after it has been held can the signal be measured, and the measurement converted to a digital value.

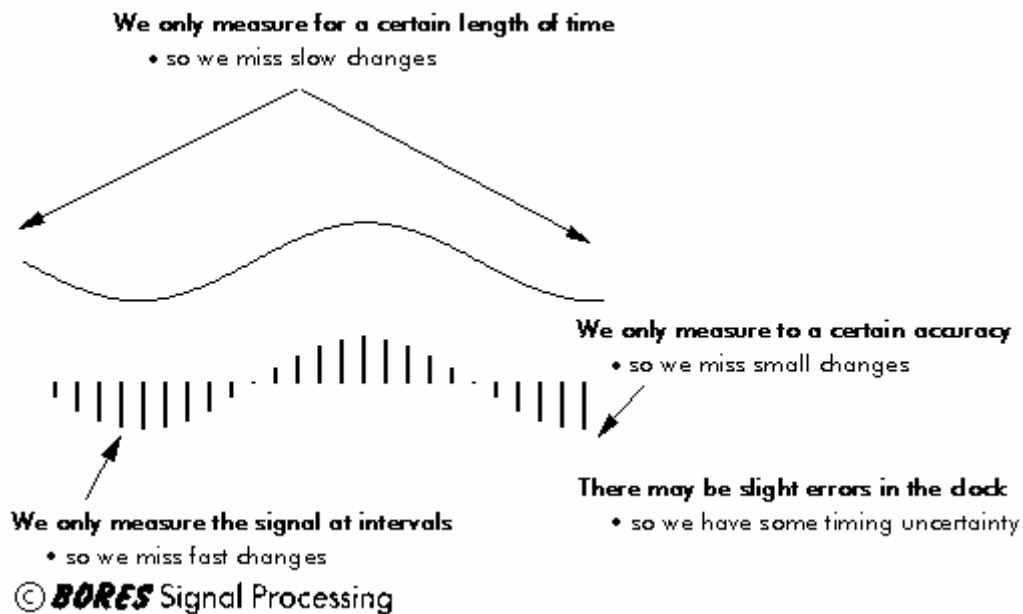


The sampling results in a discrete set of digital numbers that represent measurements of the signal - usually taken at equal intervals of time.

Note that the sampling takes place after the hold. This means that we can sometimes use a slower Analogue to Digital Converter (ADC) than might seem required at first sight. The hold circuit must act fast - fast enough that the signal is not changing during the time the circuit is acquiring the signal value - but the ADC has all the time that the signal is held to make its conversion.

We don't know what we don't measure.

In the process of measuring the signal, some information is lost.



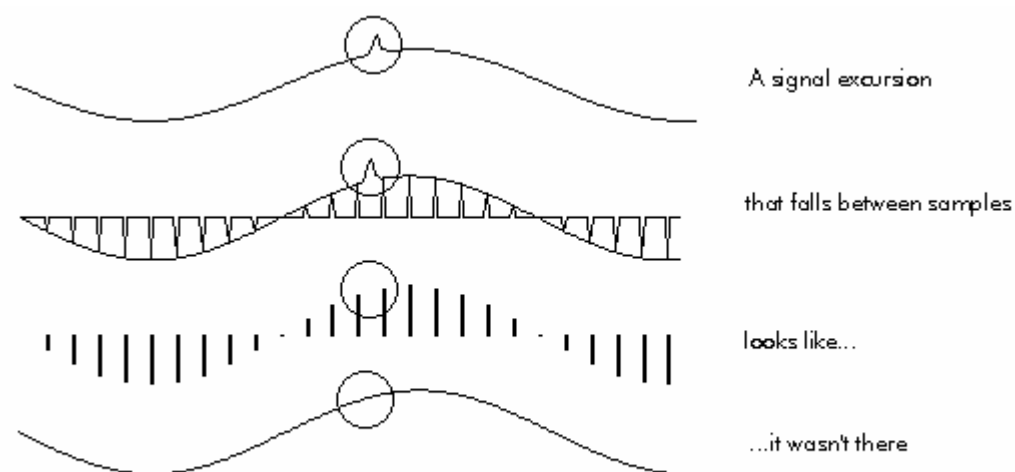
Sometimes we may have some *a priori* knowledge of the signal, or be able to make some assumptions that will let us reconstruct the lost information.

Basics:Aliasing

We only sample the signal at intervals.

We don't know what happened between the samples.

A crude example is to consider a 'glitch' that happened to fall between adjacent samples. Since we don't measure it, we have no way of knowing the glitch was there at all.



© **BORES** Signal Processing

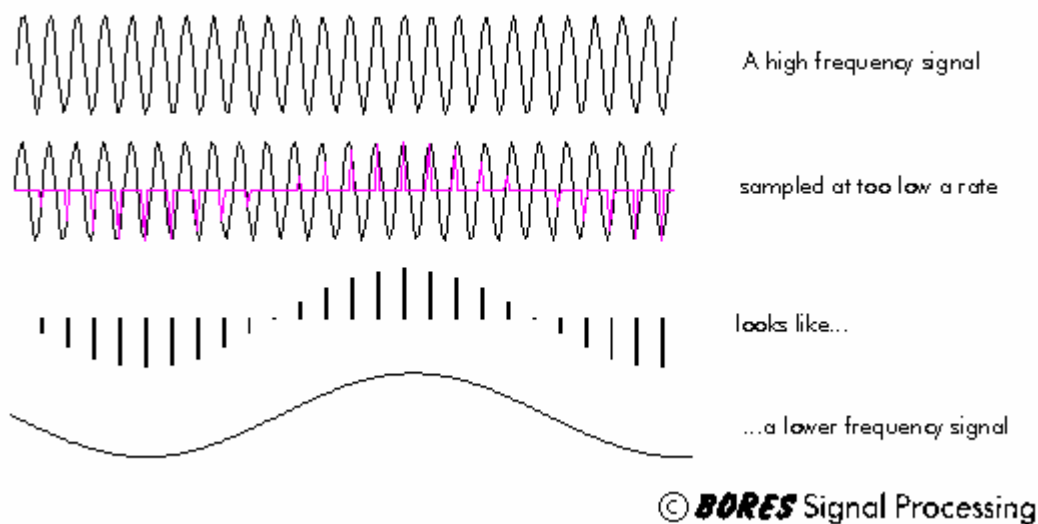
In a less obvious case, we might have signal components that are varying rapidly in between samples. Again, we could not track these rapid inter-sample variations.

We must sample fast enough to see the most rapid changes in the signal.

Sometimes we may have some *a priori* knowledge of the signal, or be able to make some assumptions about how the signal behaves in between samples.

If we do not sample fast enough, we cannot track completely the most rapid changes in the signal.

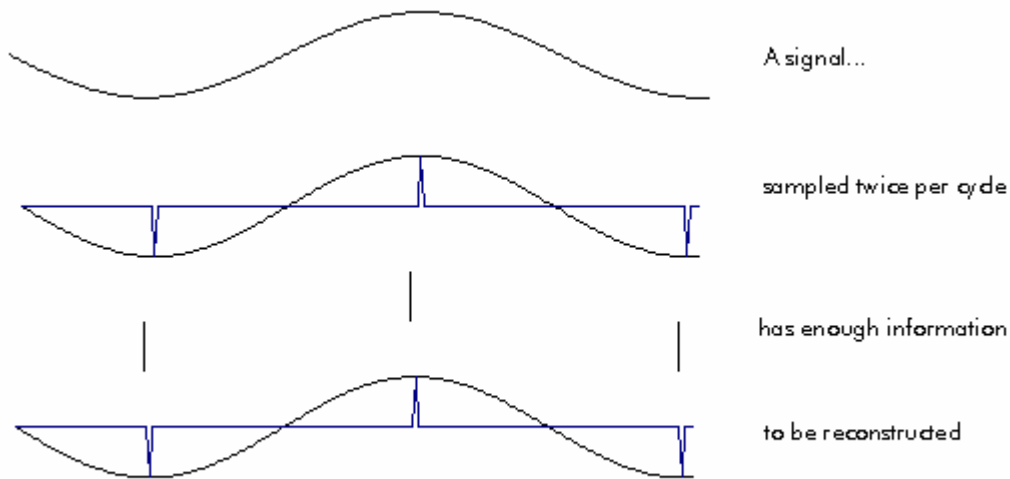
Some higher frequencies can be incorrectly interpreted as lower ones.



In the diagram, the high frequency signal is sampled just under twice every cycle. The result is, that each sample is taken at a slightly later part of the cycle. If we draw a smooth connecting line between the samples, the resulting curve looks like a lower frequency. This is called 'aliasing' because one frequency looks like another.

Note that the problem of aliasing is that we cannot tell which frequency we have - a high frequency looks like a low one so we cannot tell the two apart. But sometimes we may have some *a priori* knowledge of the signal, or be able to make some assumptions about how the signal behaves in between samples, that will allow us to tell unambiguously what we have.

Nyquist showed that to distinguish unambiguously between all signal frequency components we must sample faster than twice the frequency of the highest frequency component.



© **BORES** Signal Processing

In the diagram, the high frequency signal is sampled twice every cycle. If we draw a smooth connecting line between the samples, the resulting curve looks like the original signal. But if the samples happened to fall at the zero crossings, we would see no signal at all - this is why the sampling theorem demands we sample faster than twice the highest signal frequency.

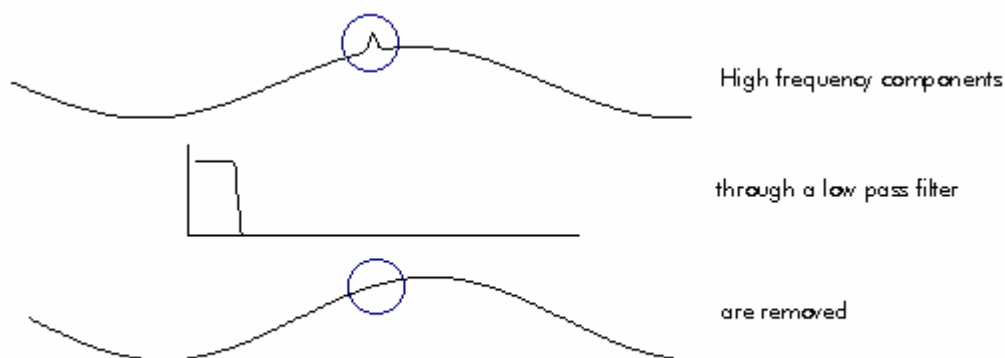
This avoids aliasing.

The highest signal frequency allowed for a given sample rate is called the **Nyquist frequency**.

Actually, Nyquist says that we have to sample faster than the signal **bandwidth**, not the highest frequency. But this leads us into multirate signal processing which is a more advanced subject.

Basics: Antialiasing

Nyquist showed that to distinguish unambiguously between all signal frequency components we must sample at least twice the frequency of the highest frequency component. To avoid aliasing, we simply filter out all the high frequency components before sampling.

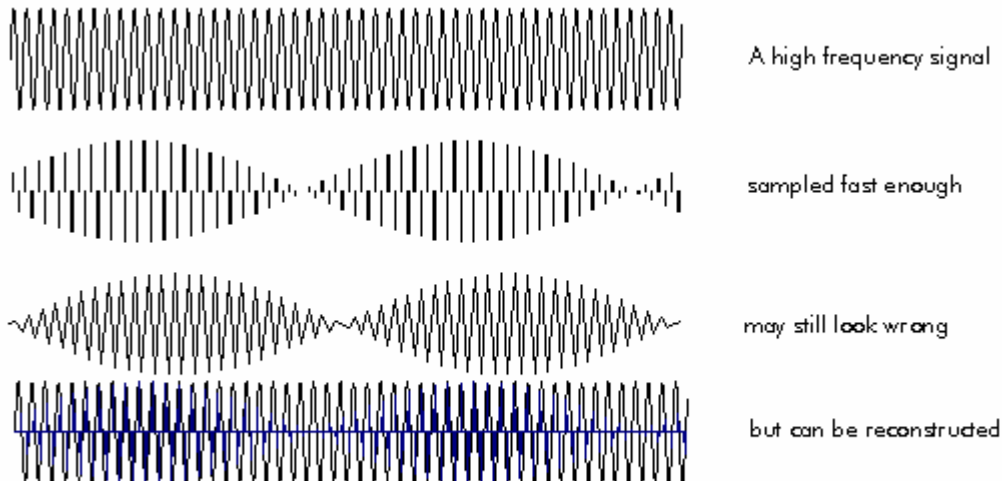


© **BORES** Signal Processing

Note that antialias filters must be analogue - it is too late once you have done the sampling.

This simple brute force method avoids the problem of aliasing. But it does remove information - if the signal had high frequency components, we cannot now know anything about them.

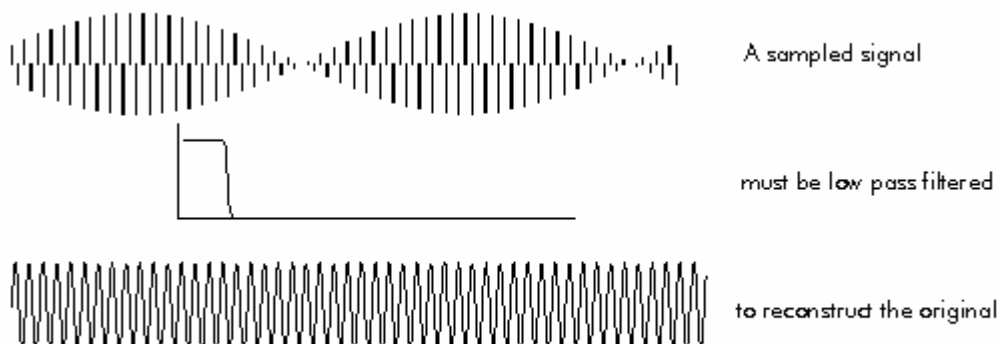
Although Nyquist showed that provide we sample at least twice the highest signal frequency we have all the information needed to reconstruct the signal, the sampling theorem does not say the samples will **look like** the signal.



© **BORES** Signal Processing

The diagram shows a high frequency sine wave that is nevertheless sampled fast enough according to Nyquist's sampling theorem - just more than twice per cycle. When straight lines are drawn between the samples, the signal's frequency is indeed evident - but it looks as though the signal is amplitude modulated. This effect arises because each sample is taken at a slightly earlier part of the cycle. Unlike aliasing, the effect does not change the apparent signal frequency. The answer lies in the fact that the sampling theorem says there is enough information to reconstruct the signal - and the correct reconstruction is not just to draw straight lines between samples.

The signal is properly reconstructed from the samples by low pass filtering: the low pass filter should be the same as the original antialias filter.



© **BORES** Signal Processing

The reconstruction filter interpolates between the samples to make a smoothly varying analogue signal. In the example, the reconstruction filter interpolates between samples in a 'peaky' way that seems at first sight to be strange. The explanation lies in the shape of the reconstruction filter's impulse response.



The impulse response of the reconstruction filter has a classic ' $\sin(x)/x$ ' shape. The stimulus fed to this filter is the series of discrete impulses which are the samples. Every time an impulse hits the filter, we get 'ringing' - and it is the superposition of all these peaky rings that reconstructs the proper signal. If the signal contains frequency components that are close to the Nyquist, then the reconstruction filter has to be very sharp indeed. This means it will have a very long impulse response - and so the long 'memory' needed to fill in the signal even in region of the low amplitude samples.

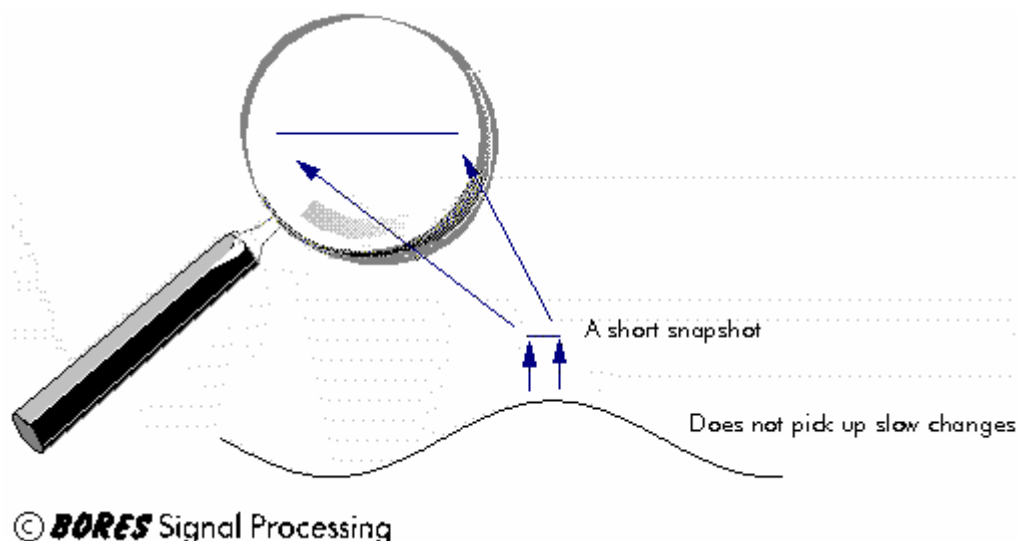
Basics: Frequency resolution

We only sample the signal for a certain time.

We cannot see slow changes in the signal if we don't wait long enough.

In fact we must sample for long enough to detect not only low frequencies in the signal, but also small differences between frequencies. The length of time for which we are prepared to sample the signal determines our ability to resolve adjacent frequencies - the frequency resolution.

We must sample for at least one complete cycle of the lowest frequency we want to resolve.



We can see that we face a forced compromise. We must sample fast to avoid and for a long time to achieve a good frequency resolution. But sampling fast for a long time means we will have a lot of samples - and lots of samples means lots of computation, for which we generally

don't have time. So we will have to compromise between resolving frequency components of the signal, and being able to see high frequencies.

Basics: Quantisation

When the signal is converted to digital form, the precision is limited by the number of bits available.

The diagram shows an analogue signal which is then converted to a digital representation - in this case, with 8 bit precision.

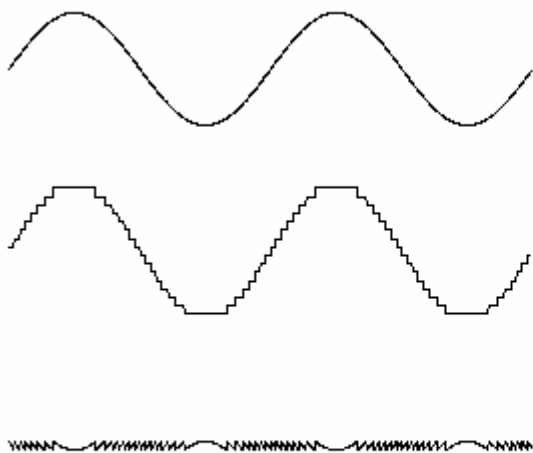
The smoothly varying analogue signal can only be represented as a 'stepped' waveform due to the limited precision.

Sadly, the errors introduced by digitisation are both non linear and signal dependent.

Non linear means we cannot calculate their effects using normal maths.

Signal dependent means the errors are coherent and so cannot be reduced by simple means.

This is a common problem in DSP. The errors due to limited precision (ie word length) are non linear (hence incalculable) and signal dependent (hence coherent). Both are bad news, and mean that we cannot really calculate how a DSP algorithm will perform in limited precision - the only reliable way is to implement it, and test it against signals of the type expected. The non linearity can also lead to instability - particularly with IIR filters.



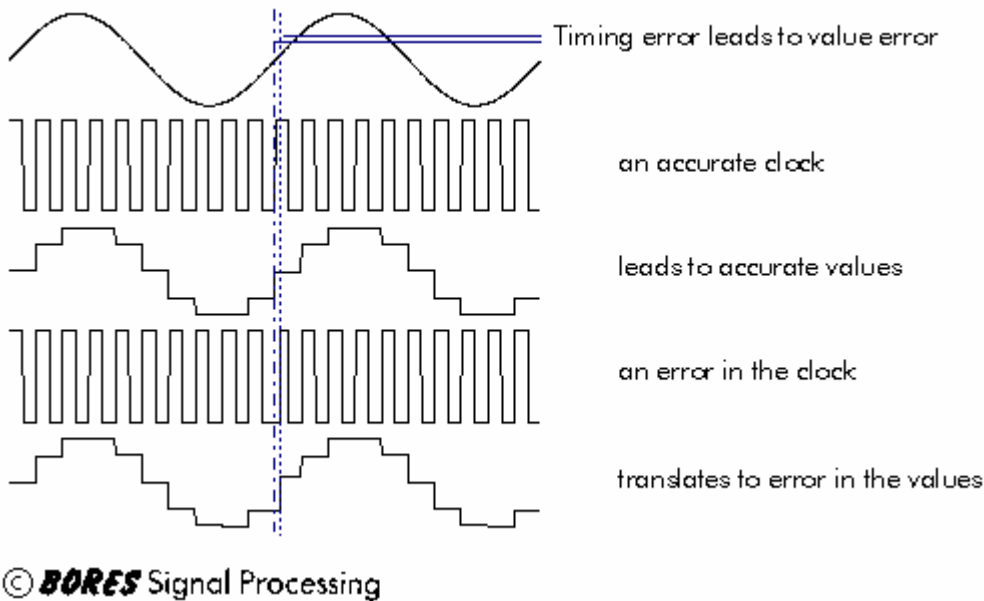
limited precision leads to errors...

which are signal dependent

© **BORES** Signal Processing

The word length of hardware used for DSP processing determines the available precision and dynamic range.

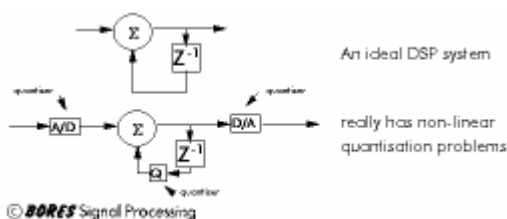
Uncertainty in the clock timing leads to errors in the sampled signal.



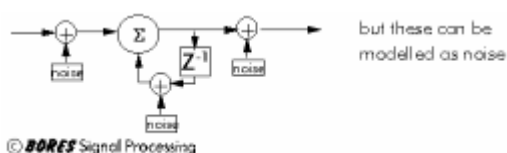
The diagram shows an analogue signal which is held on the rising edge of a clock signal. If the clock edge occurs at a different time than expected, the signal will be held at the wrong value. Sadly, the errors introduced by timing error are both non linear and signal dependent.

A real DSP system suffers from three sources of error due to limited word length in the measurement and processing of the signal:

- limited precision due to word length when the analogue signal is converted to digital form
- errors in arithmetic due to limited precision within the processor itself
- limited precision due to word length when the digital samples are converted back to analogue form

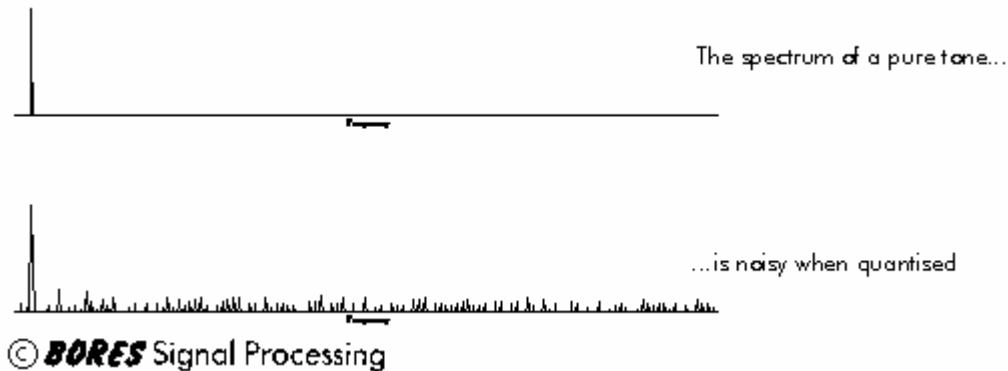


These errors are often called 'quantisation error'. The effects of quantisation error are in fact both non linear and signal dependent. Non linear means we cannot calculate their effects using normal maths. Signal dependent means that even if we could calculate their effect, we would have to do so separately for every type of signal we expect. A simple way to get an idea of the effects of limited word length is to model each of the sources of quantisation error as if it were a source of random noise.



The model of quantisation as injections of random noise is helpful in gaining an idea of the effects. But it is not actually accurate, especially for systems with feedback like IIR filters.

The effect of quantisation error is often similar to an injection of random noise.



The diagram shows the spectrum calculated from a pure tone

- the top plot shows the spectrum with high precision (double precision floating point)
- the bottom plot shows the spectrum when the sine wave is quantised to 16 bits

The effect looks very like low level random noise. The signal to noise ratio is affected by the number of bits in the data format, and by whether the data is fixed point or floating point.

Basics: Summary

A DSP system has three fundamental sources of limitation:

- loss of information because we only take samples of the signal at intervals
- loss of information because we only sample the signal for a certain length of time
- errors due to limited precision (ie word length) in data storage and arithmetic

The effects of these limitations are as follows:

- aliasing is the result of sampling, which means we cannot distinguish between high and low frequencies
- limited frequency resolution is the result of limited duration of sampling, which means we cannot distinguish between adjacent frequencies
- quantisation error is the result of limited precision (word length) when converting between analogue and digital forms, when storing data, or when performing arithmetic

Aliasing and frequency resolution are fundamental limitations - they arise from the mathematics and cannot be overcome. They are limitations of any sampled data system, not just digital ones.

Quantisation error is an artifact of the imperfect precision, and can be improved upon by using an increased word length. It is a feature peculiar to digital systems. Its effects are non linear

and signal dependent, but can sometimes be acceptably modelled as injections of random noise.

Time domain processing

This is the second module of the BORES Signal Processing DSP course - Introduction to DSP.

It covers the following subjects:

- correlation
- autocorrelation to extract a signal from noise
- cross correlation to locate a know signal
- cross correlation to identify a signal
- convolution

These courses are individual study over the Internet only. All material is copyright: you are not permitted to make copies or print out material, for personal use or for teaching.

Our DSP training classes offer intensive and highly practical training in DSP and Media Processing. Contact us via email - bores@bores.com or telephone +44 (0)1483 740138 for details.

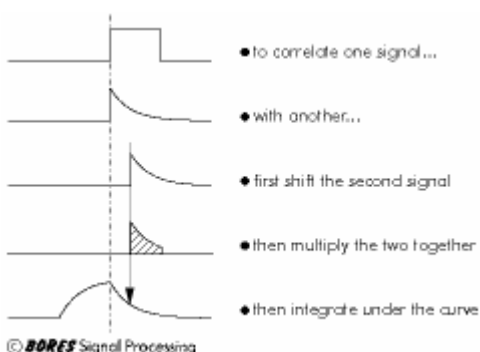
Time domain processing: Correlation

Correlation is a weighted moving average:

$$r[n] = \sum x[k] * y[n + k]$$

© BORES Signal Processing

One signal provides the weighting function.



The diagram shows how a single point of the correlation function is calculated:

- first, one signal is shifted with respect to the other
- the amount of the shift is the position of the correlation function point to be calculated
- each element of one signal is multiplied by the corresponding element of the other

- the area under the resulting curve is integrated

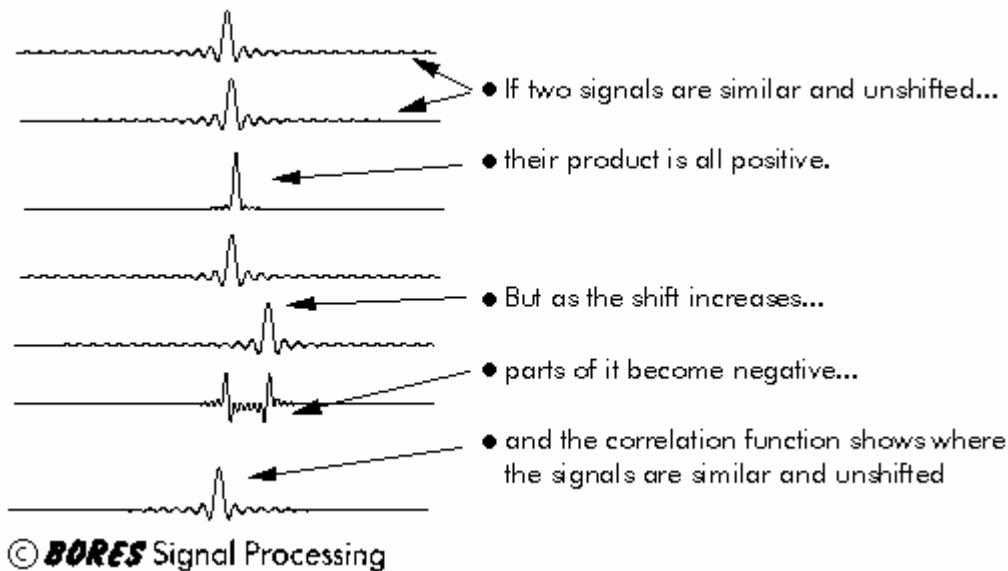
Correlation requires a lot of calculations. If one signal is of length M and the other is of length N , then we need $(N * M)$ multiplications, to calculate the whole correlation function.

Note that really, we want to multiply and then accumulate the result - this is typical of DSP operations and is called a 'multiply/accumulate' operation. It is the reason that DSP processors can do multiplications and additions in parallel.

Time domain processing: Correlation

Correlation is a maximum when two signals are similar in shape, and are in phase (or 'unshifted' with respect to each other).

Correlation is a measure of the similarity between two signals as a function of time shift between them



The diagram shows two similar signals.

When the two signals are similar in shape and unshifted with respect to each other, their product is all positive. This is like constructive interference, where the peaks add and the troughs subtract to emphasise each other. The area under this curve gives the value of the correlation function at point zero, and this is a large value.

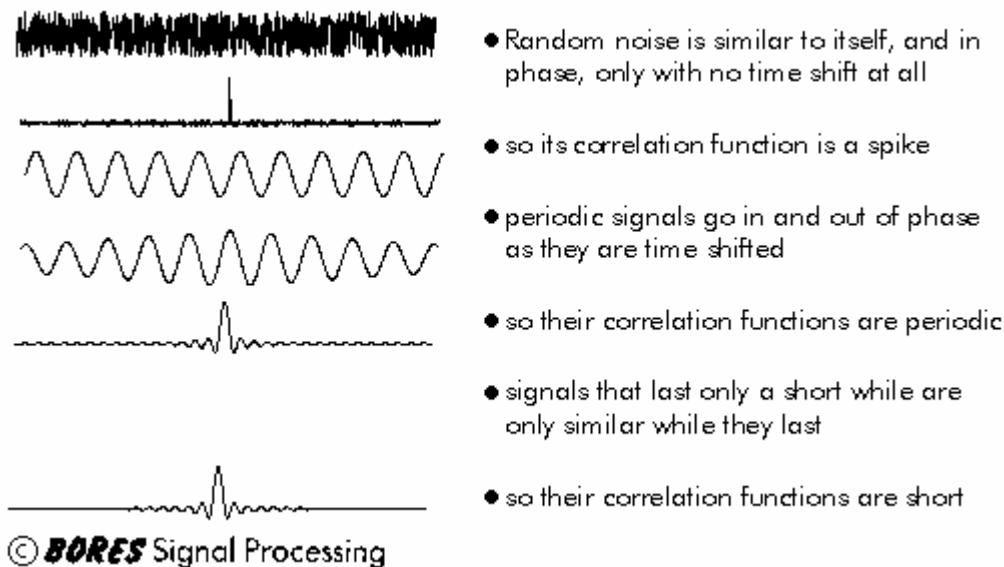
As one signal is shifted with respect to the other, the signals go out of phase - the peaks no longer coincide, so the product can have negative going parts. This is a bit like destructive interference, where the troughs cancel the peaks. The area under this curve gives the value of the correlation function at the value of the shift. The negative going parts of the curve now cancel some of the positive going parts, so the correlation function is smaller.

The largest value of the correlation function shows when the two signals were similar in shape and unshifted with respect to each other (or 'in phase'). The breadth of the correlation function - where it has significant value - shows for how long the signals remain similar.

Time domain processing: Correlation functions

The correlation function shows how similar two signals are, and for how long they remain similar when one is shifted with respect to the other.

Correlating a signal with itself is called autocorrelation. Different sorts of signal have distinctly different autocorrelation functions. We can use these differences to tell signals apart.



The diagram shows three different types of signal:

- Random noise is defined to be uncorrelated - this means it is only similar to itself with no shift at all. Even a shift of one sample either way means there is no correlation at all, so the correlation function of random noise with itself is a single sharp spike at shift zero.
- Periodic signals go in and out of phase as one is shifted with respect to the other. So they will show strong correlation at any shift where the peaks coincide. The autocorrelation function of a periodic signal is itself a periodic signal, with a period the same as that of the original signal.
- Short signals can only be similar to themselves for small values of shift, so their autocorrelation functions are short.

The three types of signal have easily recognisable autocorrelation functions.

Time domain processing: Autocorrelation

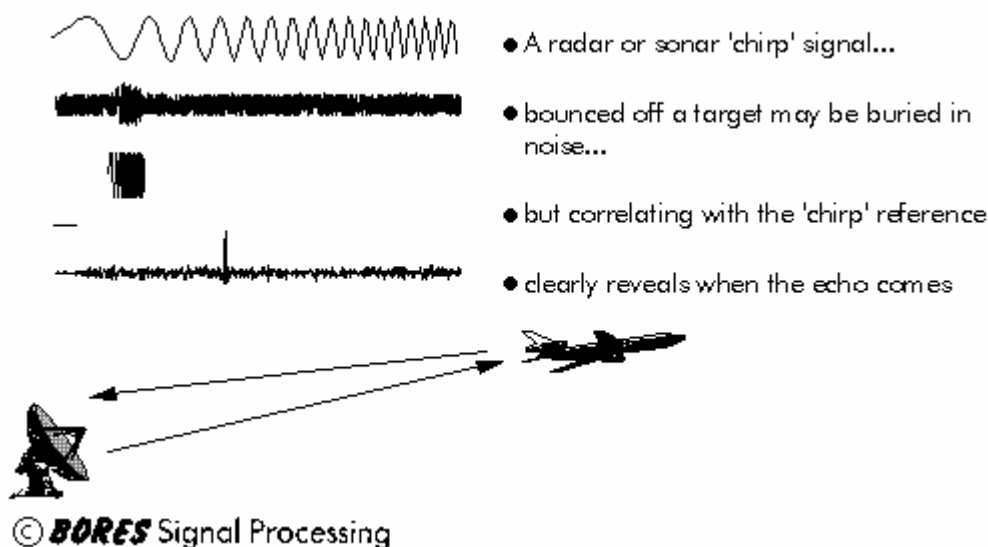
Autocorrelation (correlating a signal with itself) can be used to extract a signal from noise.

The diagram shows how the signal can be extracted from the noise:

- Random noise has a distinctive 'spike' autocorrelation function.
- A sine wave has a periodic autocorrelation function
- So the autocorrelation function of a noisy sine wave is a periodic function with a single spike which contains all the noise power.

The separation of signal from noise using autocorrelation works because the autocorrelation function of the noise is easily distinguished from that of the signal.

Cross correlation (correlating a signal with another) can be used to detect and locate known reference signal in noise.

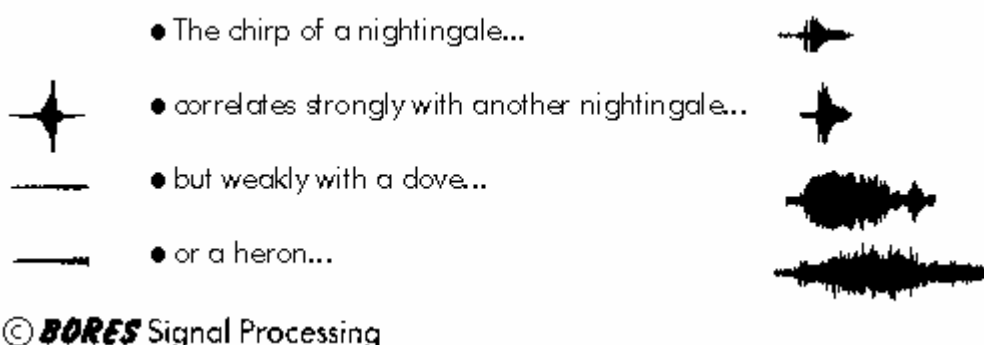


The diagram shows how the signal can be located within the noise.

- A copy of the known reference signal is correlated with the unknown signal.
- The correlation will be high when the reference is similar to the unknown signal.
- A large value for correlation shows the degree of confidence that the reference signal is detected.
- The large value of the correlation indicates when the reference signal occurs.

Time domain processing: Cross correlation to identify a signal

Cross correlation (correlating a signal with another) can be used to identify a signal by comparison with a library of known reference signals.



The diagram shows how the unknown signal can be identified.

- A copy of a known reference signal is correlated with the unknown signal.
- The correlation will be high if the reference is similar to the unknown signal.
- The unknown signal is correlated with a number of known reference functions.
- A large value for correlation shows the degree of similarity to the reference.
- The largest value for correlation is the most likely match.

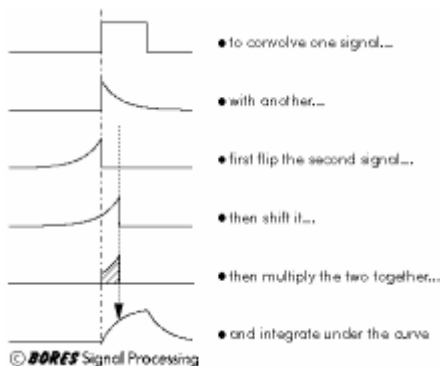
Cross correlation is one way in which sonar can identify different types of vessel.

- Each vessel has a unique sonar 'signature'.
- The sonar system has a library of pre-recorded echoes from different vessels.
- An unknown sonar echo is correlated with a library of reference echoes.
- The largest correlation is the most likely match.

Time domain processing: Convolution

Convolution is a weighted moving average with one signal flipped back to front:

The equation is the same as for correlation except that the second signal ($y[k - n]$) is flipped back to front.



The diagram shows how the unknown signal can be identified.

The diagram shows how a single point of the convolution function is calculated:

- first, one signal is flipped back to front
- then, one signal is shifted with respect to the other
- the amount of the shift is the position of the convolution function point to be calculated
- each element of one signal is multiplied by the corresponding element of the other
- the area under the resulting curve is integrated

Convolution requires a lot of calculations. If one signal is of length M and the other is of length N , then we need $(N * M)$ multiplications, to calculate the whole convolution function.

Note that really, we want to multiply and then accumulate the result - this is typical of DSP operations and is called a 'multiply/accumulate' operation. It is the reason that DSP processors can do multiplications and additions in parallel.

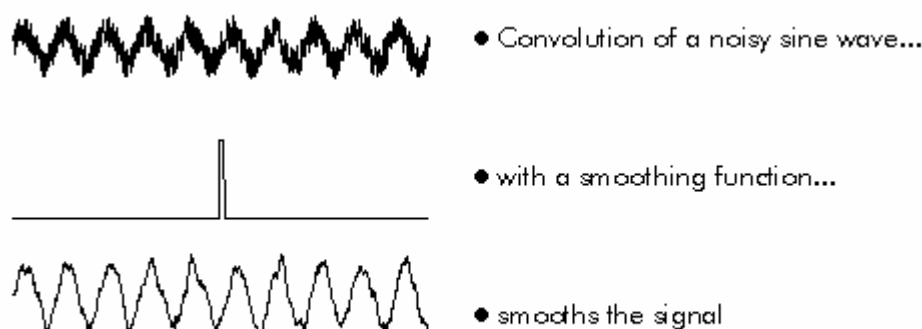
Convolution is used for digital filtering.

The reason convolution is preferred to correlation for filtering has to do with how the frequency spectra of the two signals interact. Convolving two signals is equivalent to multiplying the frequency spectra of the two signals together - which is easily understood, and is what we mean by filtering. Correlation is equivalent to multiplying the **complex conjugate** of the frequency spectrum of one signal by the frequency spectrum of the other. Complex conjugation is not so easily understood and so convolution is used for digital filtering. Convolving by multiplying frequency spectra is called fast convolution.

Time domain processing: Convolution to smooth a signal

Convolution is a weighted moving average with one signal flipped back to front.

Convolving a signal with a smooth weighting function can be used to smooth a signal:



© **BORES** Signal Processing

The diagram shows how a noisy sine wave can be smoothed by convolving with a rectangular smoothing function - this is just a moving average.

The smoothing property leads to the use of convolution for digital filtering.

Time domain processing: Convolution and correlation

Correlation is a weighted moving average:

$$r[n] = \sum x[k] * y[n + k]$$

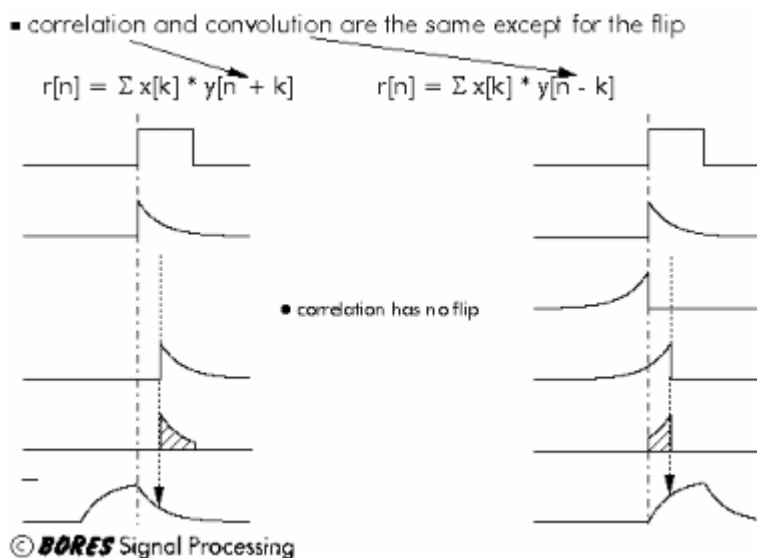
© **BORES** Signal Processing

Convolution is a weighted moving average with one signal flipped back to front:

$$r[n] = \sum x[k] * y[n - k]$$

© **BORES** Signal Processing

Convolution and correlation are the same except for the flip:

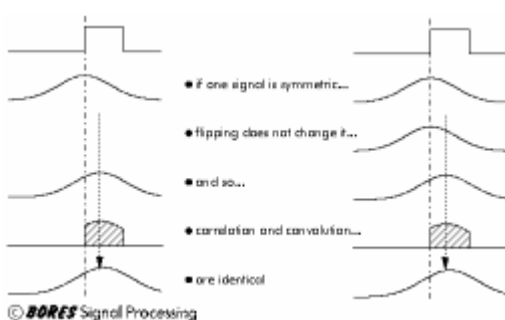


Convolution is used for digital filtering.

The reason convolution is preferred to correlation for filtering has to do with how the frequency spectra of the two signals interact. Convolution of two signals is equivalent to multiplying the frequency spectra of the two signals together - which is easily understood, and is what we mean by filtering. Correlation is equivalent to multiplying the **complex conjugate** of the frequency spectrum of one signal by the frequency spectrum of the other. Complex conjugation is not so easily understood and so convolution is used for digital filtering.

Time domain processing: Convolution and correlation

If one signal is symmetric, convolution and correlation are identical:



If one signal is symmetric, then flipping it back to front does not change it. So convolution and correlation are the same.

We hope you have found this course module to be helpful. Please contact us with your comments or suggestions.

Introduction to DSP - frequency

This is the third module of the BORES Signal Processing DSP course - Introduction to DSP.

It covers the following subjects:

- Fourier transforms
- convolution in the frequency domain
- short time Fourier transforms
- frequency leakage
- windowing
- other transforms

These courses are individual study over the Internet only. All material is copyright: you are not permitted to make copies or print out material, for personal use or for teaching.

Our DSP training classes offer intensive and highly practical training in DSP and Media Processing. Contact us via email - bores@bores.com or telephone +44 (0)1483 740138 for details.

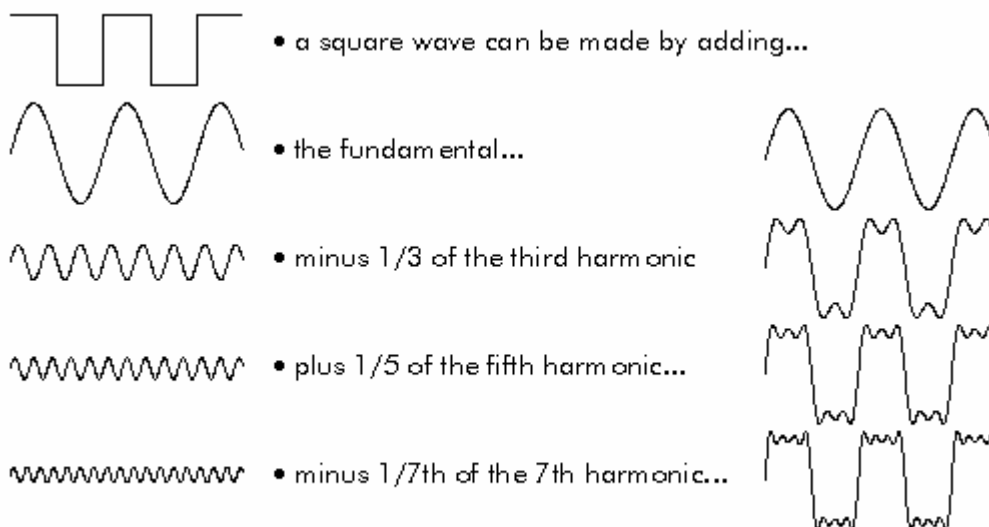
Frequency analysis: Fourier transforms

Jean Baptiste Fourier showed that any signal or waveform could be made up just by adding together a series of pure tones (sine waves) with appropriate amplitude and phase.

This is a rather startling theory, if you think about it. It means, for instance, that by simply turning on a number of sine wave generators we could sit back and enjoy a Beethoven symphony.

Of course we would have to use a very large number of sine wave generators, and we would have to turn them on at the time of the Big Bang and leave them on until the heat death of the universe.

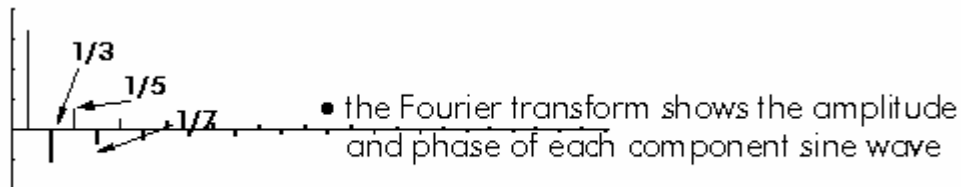
Fourier's theorem assumes we add sine waves of infinite duration.



The diagram shows how a square wave can be made up by adding together pure sine waves at the harmonics of the fundamental frequency.

Any signal can be made up by adding together the correct sine waves with appropriate amplitude and phase.

The Fourier transform is an equation to calculate the frequency, amplitude and phase of each sine wave needed to make up any given signal.



© **BORES** Signal Processing

- The Fourier Transform (FT) is a mathematical formula using integrals
- The Discrete Fourier Transform (DFT) is a discrete numerical equivalent using sums instead of integrals
- The Fast Fourier Transform (FFT) is just a computationally fast way to calculate the DFT

The Discrete Fourier Transform involves a summation:

$$H(f) = \sum c[k] * \exp(-2\pi j k(f\Delta))$$

© **BORES** Signal Processing

Where j is the square root of minus one (defined as a number whose sole property is that its square is minus one).

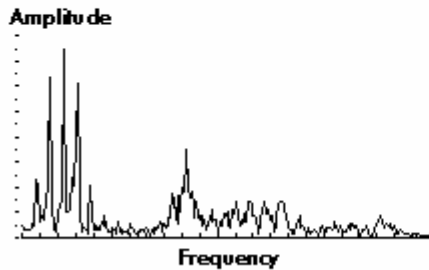
Note that the DFT and the FFT involve a lot of multiplying and then accumulating the result - this is typical of DSP operations and is called a 'multiply/accumulate' operation. It is the reason that DSP processors can do multiplications and additions in parallel.

Frequency analysis: Frequency spectra

Using the Fourier transform, any signal can be analysed into its frequency components.



• a recording of speech...



• can be analysed to show the spectrum

© **BORES** Signal Processing

The diagram shows a recording of speech, and its analysis into frequency components.

With some signals it is easy to see that they are composed of different frequencies: for instance a chord played on the piano is obviously made up of the different pure tones generated by the keys pressed. For other signals the connection to frequency is less obvious: for example a hand clap has a frequency spectrum but it is less easy to see how the individual frequencies are generated.

You can use a piano as an acoustic spectrum analyser to show that a hand clap has a frequency spectrum:

- open the lid of the piano and hold down the 'loud' pedal
- clap your hands loudly over the piano
- you will hear (and see) the strings vibrate to echo the clap sound
- the strings that vibrate show the frequencies
- the amount of vibration shows the amplitude

Each string of the piano acts as a finely tuned resonator.

Frequency analysis: Frequency spectra

Using the Fourier transform, any signal can be analysed into its frequency components.

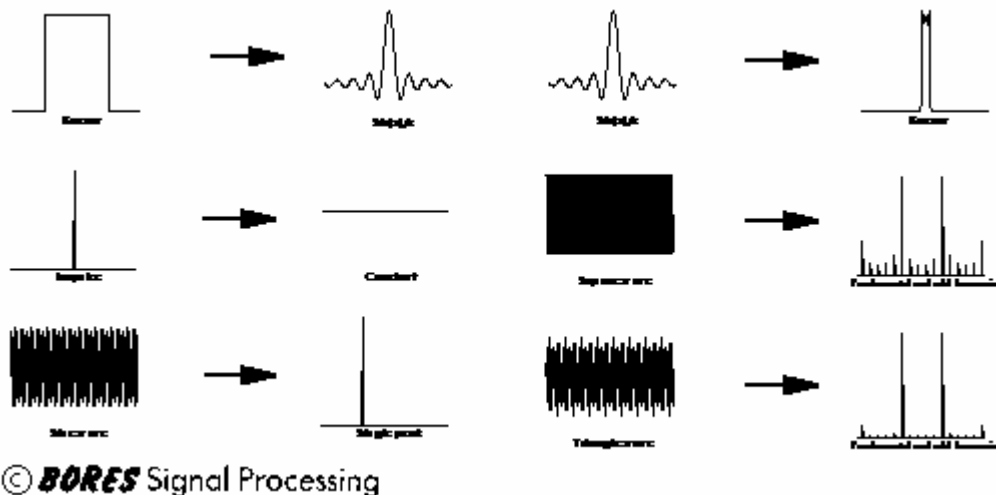
Every signal has a frequency spectrum.

- the signal defines the spectrum
- the spectrum defines the signal

We can move back and forth between the time domain and the frequency domain without losing information.

The above statement is true mathematically, but is quite incorrect in any practical sense - since we will lose information due to errors in the calculation, or due to deliberately missing

out some information that we can't measure or can't compute. But the basic idea is a good one when visualising time signals and their frequency spectra.



The diagram shows a number of signals and their frequency spectra.

Understanding the relation between time and frequency domains is useful:

- some signals are easier to visualise in the frequency domain
- some signals are easier to visualise in the time domain
- some signals take less information to define in the time domain
- some signals take less information to define in the frequency domain

For example a sine wave takes a lot of information to define accurately in the time domain: but in the frequency domain we only need three data - the frequency, amplitude and phase.

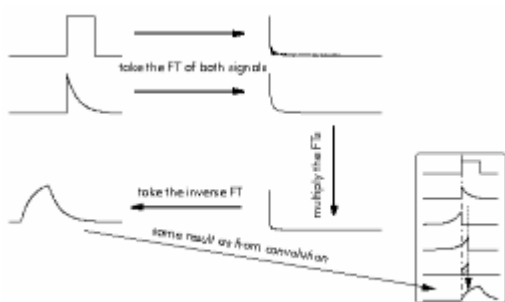
Frequency analysis: Convolution

Convolution is a weighted moving average with one signal flipped back to front:

$$r[n] = \sum x[k] * y[k - n]$$

© **BORES** Signal Processing

Convolution is the same as multiplying frequency spectra.



© **BORES** Signal Processing

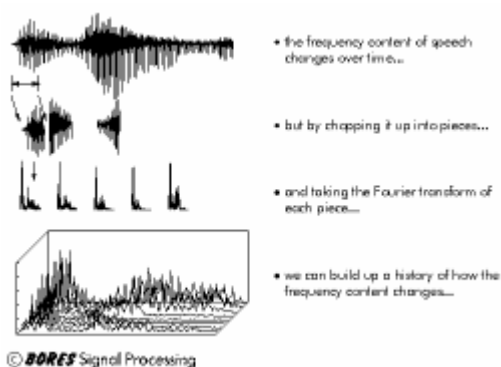
Convolution by multiplying frequency spectra can take advantage of the Fast Fourier Transform - which is a computationally efficient algorithm. So this can be faster than convolution in the time domain, and is called Fast Convolution.

Frequency analysis: Short term Fourier transform

The Fourier transform assumes the signal is analysed over all time - an infinite duration.

This means that there can be no concept of time in the frequency domain, and so no concept of a frequency changing with time. Mathematically, frequency and time are orthogonal - you cannot mix one with the other. But we can easily understand that some signals do have frequency components that change with time. A piano tune, for example, consists of different notes played at different times: or speech can be heard as having pitch that rises and falls over time.

The Short Time Fourier Transform (STFT) tries to evaluate the way frequency content changes with time:



The diagram shows how the Short Time Fourier Transform works:

- the signal is chopped up into short pieces
- and the Fourier transform is taken of each piece

Each frequency spectrum shows the frequency content during a short time, and so the successive spectra show the evolution of frequency content with time. The spectra can be plotted one behind the other in a 'waterfall' diagram as shown.

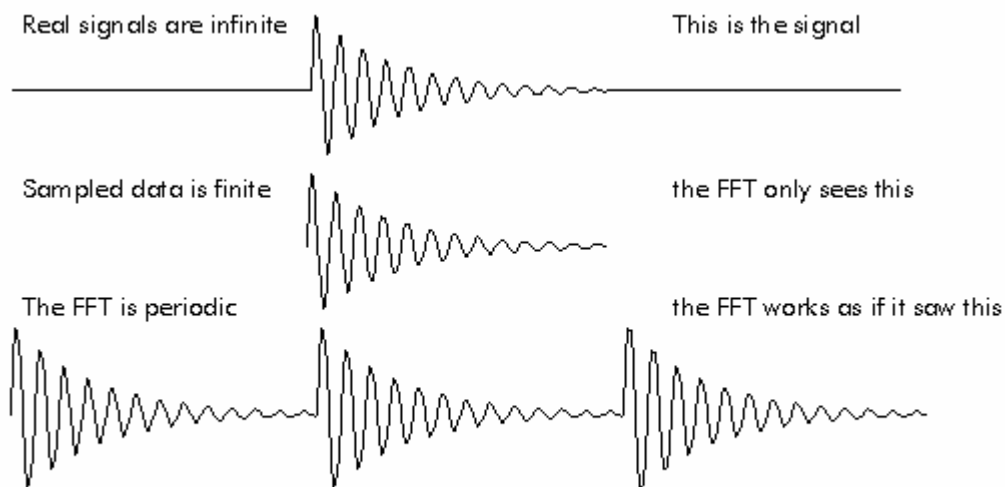
It is important to realise that the Short Time Fourier Transform involves accepting a contradiction in terms because frequency only has a meaning if we use infinitely long sine waves - and so we cannot apply Fourier Transforms to short pieces of a signal.

Frequency analysis: Short signals

The Fourier Transform works on signals of infinite duration.

But if we only measure the signal for a short time, we cannot know what happened to the signal before and after we measured it. The Fourier Transform has to make an assumption about what happened to the signal before and after we measured it.

The Fourier Transform assumes that any signal can be made by adding a series of sine waves of infinite duration. Sine waves are periodic signals. So the Fourier Transform works as if the data, too, were periodic for all time.

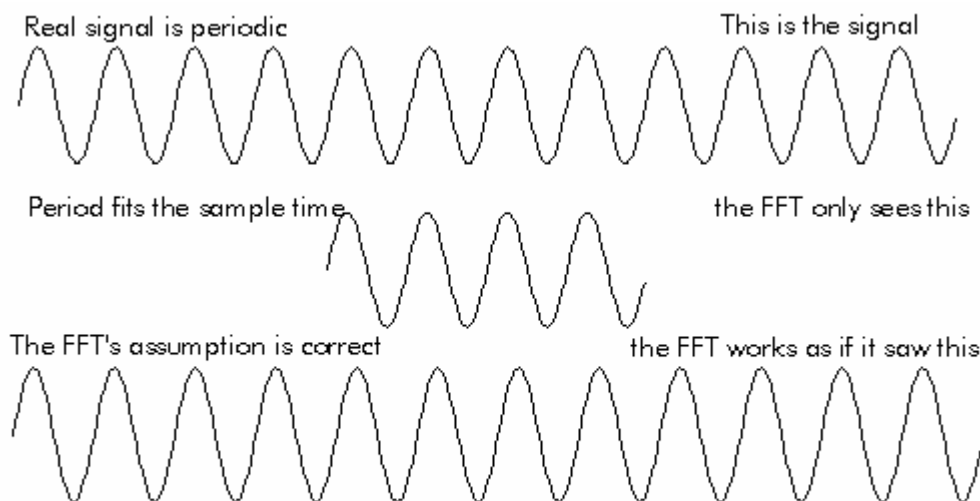


© **BORES** Signal Processing

Frequency analysis: Short signals

If we only measure the signal for a short time, the Fourier Transform works as if the data were periodic for all time.

Sometimes this assumption can be correct:



© **BORES** Signal Processing

The diagram shows what happens if we only measure a signal for a short time: the Fourier Transform works as if the data were periodic for all time.

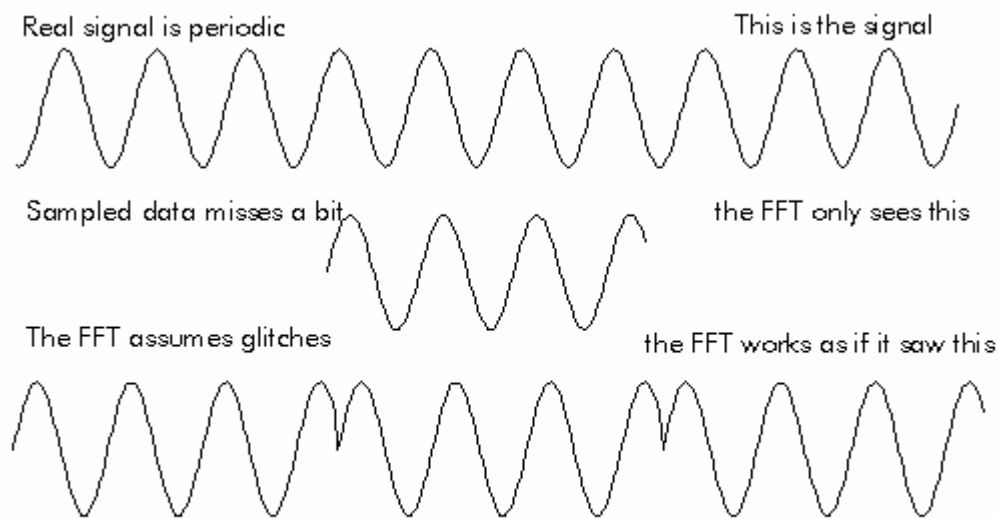
In the case chosen it happens that the signal is periodic - and that an integral number of cycles fit into the total duration of the measurement.

This means that when the Fourier Transform assumes the signal repeats, the end of one signal segment connects smoothly with the beginning of the next - and the assumed signal happens to be exactly the same as the actual signal.

Frequency analysis: Short signals

If we only measure the signal for a short time, the Fourier Transform works as if the data were periodic for all time.

Sometimes this assumption can be wrong:



© **BORES** Signal Processing

The diagram shows what happens if we only measure a signal for a short time: the Fourier Transform works as if the data were periodic for all time.

In the case chosen it happens that the signal is periodic - but that not quite an integral number of cycles fit into the total duration of the measurement.

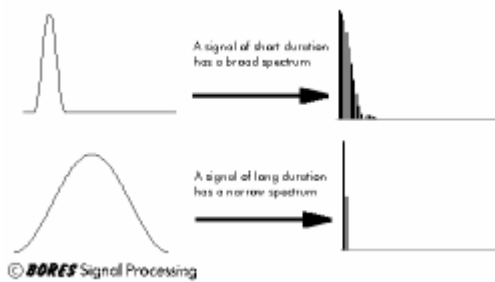
This means that when the Fourier Transform assumes the signal repeats, the end of one signal segment does not connect smoothly with the beginning of the next - the assumed signal is similar to the actual signal, but has little 'glitches' at regular intervals.

The assumed signal is not the same as the actual signal.

Frequency analysis: Frequency leakage

There is a direct relation between a signal's duration in time and the width of its frequency spectrum:

- short signals have broad frequency spectra
- long signals have narrow frequency spectra



Frequency analysis: Frequency leakage

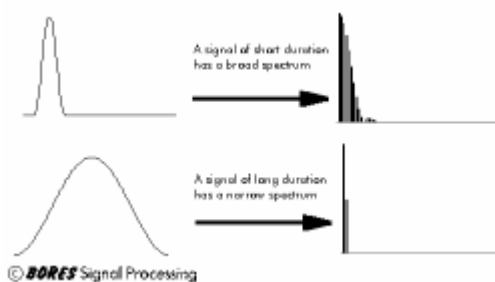
If we only measure the signal for a short time, the Fourier Transform works as if the data were periodic for all time.

If the signal is periodic, two cases arise:

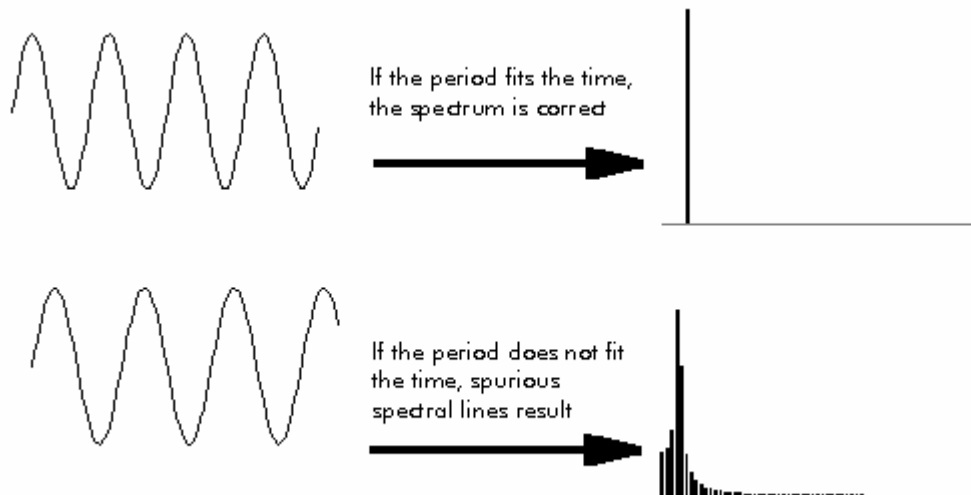
- If an integral number of cycles fit into the total duration of the measurement, then when the Fourier Transform assumes the signal repeats, the end of one signal segment connects smoothly with the beginning of the next - and the assumed signal happens to be exactly the same as the actual signal.
- If not quite an integral number of cycles fit into the total duration of the measurement, then when the Fourier Transform assumes the signal repeats, the end of one signal segment does not connect smoothly with the beginning of the next - the assumed signal is similar to the actual signal, but has little 'glitches' at regular intervals.

There is a direct relation between a signal's duration in time and the width of its frequency spectrum:

- short signals have broad frequency spectra
- long signals have narrow frequency spectra



The 'glitches' are short signals. So they have a broad frequency spectrum. And this broadening is superimposed on the frequency spectrum of the actual signal:



© **BORES** Signal Processing

- if the period exactly fits the measurement time, the frequency spectrum is correct
- if the period does not match the measurement time, the frequency spectrum is incorrect - it is broadened

This broadening of the frequency spectrum determines the frequency resolution - the ability to resolve (that is, to distinguish between) two adjacent frequency components.

Only the one happy circumstance where the signal is such that an integral number of cycles exactly fit into the measurement time gives the expected frequency spectrum. In all other cases the frequency spectrum is broadened by the 'glitches' at the ends. Matters are made worse because the size of the glitch depends on when the first measurement occurred in the cycle - so the broadening will change if the measurement is repeated.

For example a sine wave 'should' have a frequency spectrum which consists of one single line. But in practice, if measured say by a spectrum analyser, the frequency spectrum will be a broad line - with the sides flapping up and down like Batman's cloak. When we see a perfect single line spectrum - for example in the charts sometimes provided with analogue to digital converter chips - this has in fact been obtained by tuning the signal frequency carefully so that the period exactly fits the measurement time and the frequency spectrum is the best obtainable.

Frequency analysis: Windowing

If we only measure the signal for a short time, the Fourier Transform works as if the data were periodic for all time.

If not quite an integral number of cycles fit into the total duration of the measurement, then when the Fourier Transform assumes the signal repeats, the end of one signal segment does not connect smoothly with the beginning of the next - the assumed signal is similar to the actual signal, but has little 'glitches' at regular intervals.

The glitches can be reduced by shaping the signal so that its ends match more smoothly.

Since we can't assume anything about the signal, we need a way to make any signal's ends connect smoothly to each other when repeated.

One way to do this is to multiply the signal by a 'window' function:



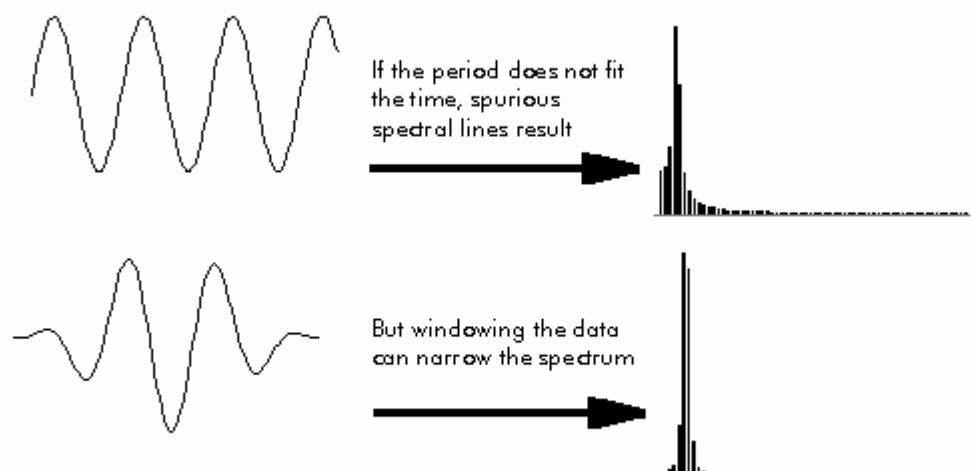
© **BORES** Signal Processing

The easiest way to make sure the ends of a signal match is to force them to be zero at the ends: that way, their value is necessarily the same.

Actually, we also want to make sure that the signal is going in the right direction at the ends to match up smoothly. The easiest way to do this is to make sure neither end is going anywhere - that is, the slope of the signal at its ends should also be zero.

Put mathematically, a window function has the property that its value and all its derivatives are zero at the ends.

Multiplying by a window function (called 'windowing') suppresses glitches and so avoids the broadening of the frequency spectrum caused by the glitches.



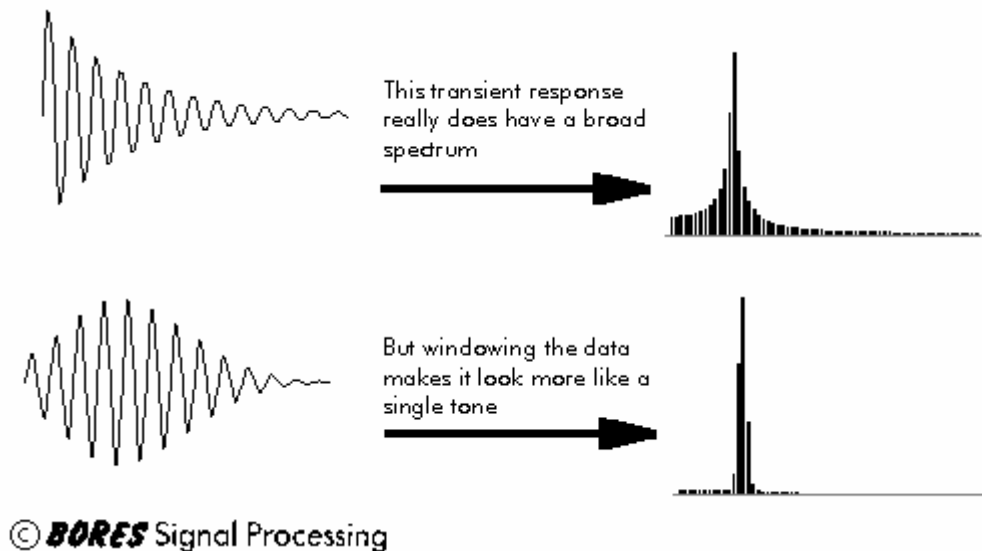
© **BORES** Signal Processing

Windowing can narrow the spectrum and make it closer to what was expected.

Frequency analysis: Windowing

Multiplying by a window function (called 'windowing') suppresses glitches and so avoids the broadening of the frequency spectrum caused by the glitches.

But it is important to remember that windowing is really a distortion of the original signal:



The diagram shows the result of applying a window function without proper thought.

The transient response really does have a broad frequency spectrum - but windowing forces it to look more as if it had a narrow frequency spectrum instead.

Worse than this, the window function has attenuated the signal at the point where it was largest - so has suppressed a large part of the signal power. This means the overall signal to noise ratio has been reduced.

Applying a window function narrows the frequency spectrum at the expense of distortion and signal to noise ratio. Many window function have been designed to trade off frequency resolution against signal to noise and distortion. Choice among them depends on knowledge of the signal and what you want to do with it.

Frequency analysis: Wavelets

Fourier's theorem assumes we add sine waves of infinite duration.

As a consequence, the Fourier Transform is good at representing signals which are long and periodic.

But the Fourier Transform has problems when used with signals which are short, and not periodic.

Other transforms are possible - fitting the data with different sets of functions than sine waves.

The trick is, to find a transform whose base set of functions look like the signal with which we are dealing.



The Fourier Transform fits the data with sine waves
- which are continuous for all time



But some signals don't look like sine waves
- sometimes they only last a short time
- or their frequency content changes with time



The Wavelet Transform fits the data with wavelets
- which are a better fit for short signals
- and for analysing 'snapshots' of longer signals

© **BORES** Signal Processing

The diagram shows a signal that is not a long, periodic signal but rather a periodic signal with a decay over a short time. This is not very well matched by the Fourier Transform's infinite sine waves. But it might be better matched by a different set of functions - say, decaying sine waves. Such functions are called 'wavelets' and can be used in the 'wavelet transform'.

Note that the wavelet transform cannot really be used to measure frequency, because frequency only has meaning when applied to infinite sine waves. But, as with the Short Time Fourier Transform, we are always willing to stretch a point in order to gain a useful tool.

The Fourier Transform's real popularity derives not from any particular mathematical merit, but from the simple fact that some one (Cooley and Tukey) managed to write an efficient program to implement it - called the Fast Fourier Transform (FFT). And now there are lots of FFT programs around for all sorts of processors, so it is likely the FFT will remain the most popular method for many years because of its excellent support.

We hope you have found this course module to be helpful. Please contact us with your comments or suggestions.

Filters

This is the fourth module of the BORES Signal Processing DSP course - Introduction to DSP.

It covers the following subjects:

- [Filtering as a frequency selective process](#)
- [filter specification](#)
- [filtering in the frequency domain](#)
- [filter equations and frequency response](#)
- [FIR filters](#)
- [window filter design](#)

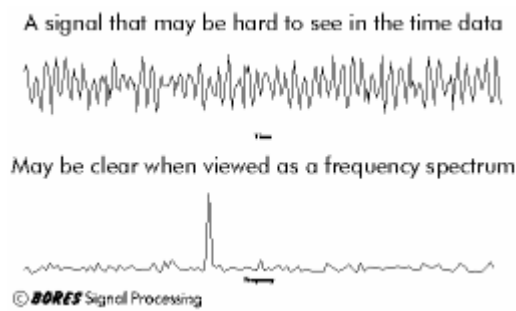
- equiripple filter design

Filtering: Filtering as a frequency selective process

Filtering is a process of selecting, or suppressing, certain frequency components of a signal.

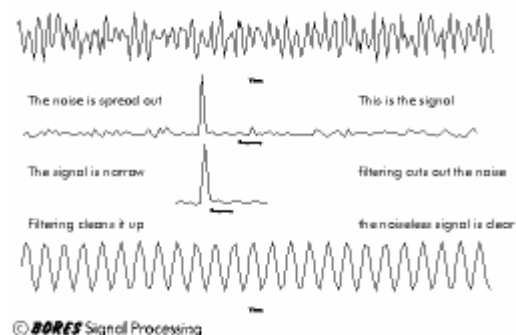
A coffee filter allows small particles to pass while trapping the larger grains. A digital filter does a similar thing, but with more subtlety. The digital filter allows to pass certain frequency components of the signal: in this it is similar to the coffee filter, with frequency standing in for particle size. But the digital filter can be more subtle than simply trapping or allowing through: it can attenuate, or suppress, each frequency components by a desired amount. This allows a digital filter to shape the frequency spectrum of the signal.

Filtering is often, though not always, done to suppress noise. It depends on the signal's frequency spectrum being different from that of the noise:



The diagram shows how how a noisy sine wave viewed as a time domain signal cannot be clearly distinguished from the noise. But when viewed as a frequency spectrum, the sine wave shows as a single clear peak while the noise power is spread over a broad frequency spectrum.

By selecting only frequency components that are present in the signal, the noise can be selectively suppressed:



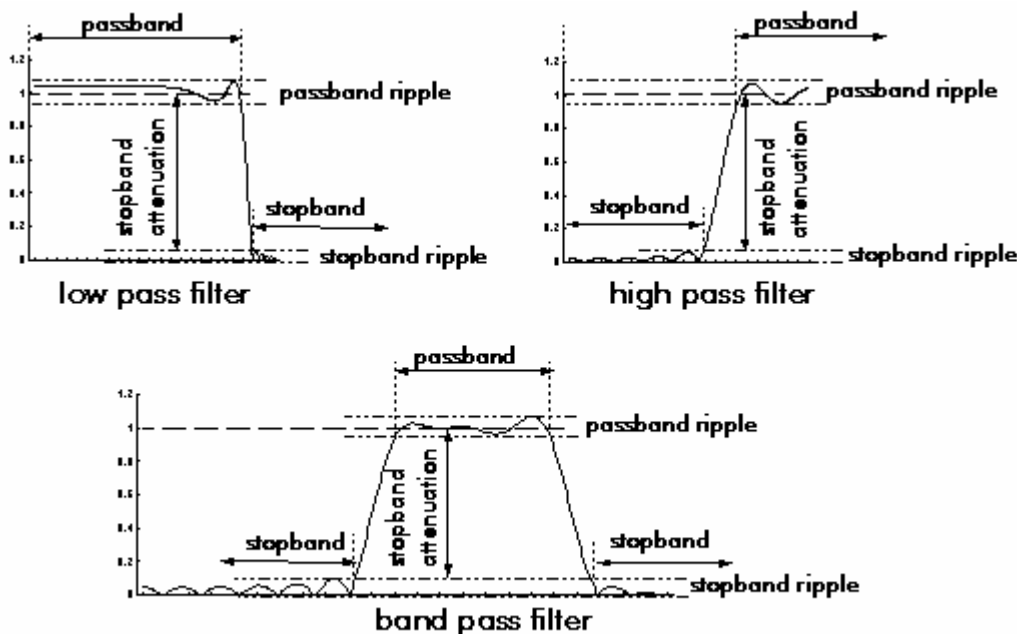
The diagram shows how a noisy sine wave may be 'cleaned up' by selecting only a range of frequencies that include signal frequency components but exclude much of the noise:

- the noisy sine wave (shown as a time signal) contains narrow band signal plus broad band noise

- the frequency spectrum is modified by suppressing a range outside the signal's frequency components
- the resulting signal (shown in the time domain again) looks much cleaner

Filtering: Digital filter specifications

Digital filters can be more subtly specified than analogue filters, and so are specified in a different way:



© **BORES** Signal Processing

Whereas analogue filters are specified in terms of their '3dB point' and their 'rolloff', digital filters are specified in terms of desired attenuation, and permitted deviations from the desired value in their frequency response:

passband

the band of frequency components that are allowed to pass

stopband

the band of frequency components that are suppressed

passband ripple

the maximum amount by which attenuation in the passband may deviate from nominal gain

stopband attenuation

the minimum amount by which frequency components in the stopband are attenuated

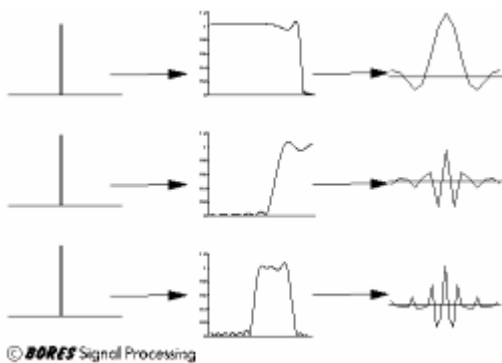
The passband need not necessarily extend to the 3 dB point: for example, if passband ripple is specified as 0.1 dB, then the passband only extends to a point at which attenuation has increased to 0.1 dB.

Between the passband and the stopband lies a transition band where the filter's shape may be unspecified.

Note that the stopband attenuation is formally specified as the attenuation to the top of the first sidelobe of the filter's frequency response.

Digital filters can also have an 'arbitrary response': meaning, the attenuation is specified at certain chosen frequencies, or for certain frequency bands.

Digital filters are also characterised by their response to an impulse: a signal consisting of a single value followed by zeroes:

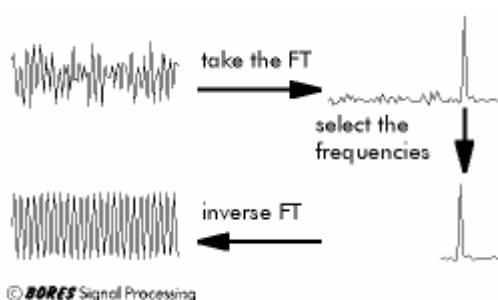


The impulse response is an indication of how long the filter takes to settle into a steady state: it is also an indication of the filter's stability - an impulse response that continues oscillating in the long term indicates the filter may be prone to instability.

The impulse response defines the filter just as well as does the frequency response.

Filtering: Filtering in the frequency domain

Filtering can be done directly in the frequency domain, by operating on the signal's frequency spectrum:



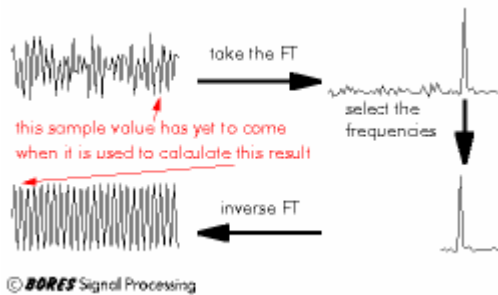
The diagram shows how a noisy sine wave can be cleaned up by operating directly upon its frequency spectrum to select only a range of frequencies that include signal frequency components but exclude much of the noise:

- the noisy sine wave (shown as a time signal) contains narrow band signal plus broad band noise
- the frequency spectrum is calculated
- the frequency spectrum is modified by suppressing a range outside the signal's frequency components

- the time domain signal is calculated from the frequency spectrum
- the resulting signal (shown in the time domain again) looks much cleaner

Filtering in the frequency domain is efficient, because every calculated sample of the filtered signal takes account of all the input samples.

Filtering in the frequency domain is sometimes called 'acausal' filtering because (at first sight) it violates the laws of cause and effect.



Because the frequency spectrum contains information about the whole of the signal - for all time values - samples early in the output take account of input values that are late in the signal, and so can be thought of as still to happen. The frequency domain filter 'looks ahead' to see what the signal is going to do, and so violates the laws of cause and effect. Of course this is nonsense - all it means is we delayed a little until the whole signal had been received before starting the filter calculation - so filtering directly in the frequency domain is perfectly permissible and in fact often the best method. It is often used in image processing.

There are good reasons why we might not be able to filter in the frequency domain:

- we might not be able to afford to wait for future samples - often, we need to deliver the next output as quickly as possible, usually before the next input is received
- we might not have enough computational power to calculate the Fourier transform
- we might have to calculate on a continuous stream of samples without the luxury of being able to chop the signal into convenient lumps for the Fourier transform
- we might not be able to join the edges of the signals smoothly after transforming back from the frequency domain

None of the above reasons should make us ignore the possibility of frequency domain filtering, which is very often the best method. It is often used in image processing, or certain types of experiment where the data necessarily comes in bursts, such as NMR or infra red spectroscopy.

Filtering: Digital filter equation

Output from a digital filter is made up from previous inputs and previous outputs, using the operation of convolution:

Output previous input previous output

$$y[n] = \sum c[k] * x[n-k] + \sum d[i] * y[n-i]$$

coefficients

© BOREF Signal Processing

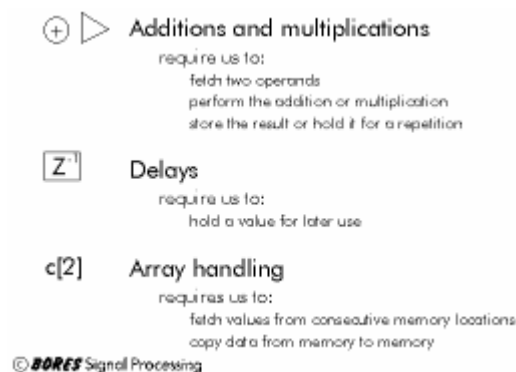
Two convolutions are involved: one with the previous inputs, and one with the previous outputs. In each case the convolving function is called the filter **coefficients**.

The filter can be drawn as a block diagram:

Two convolutions are involved: one with the previous inputs, and one with the previous outputs. In each case the convolving function is called the filter **coefficients**.

The filter can be drawn as a block diagram:

The filter diagram can show what hardware elements will be required when implementing the filter:



The left hand side of the diagram shows the direct path, involving previous inputs: the right hand side shows the feedback path, operating upon previous outputs.

Filtering: Filter frequency response

Since filtering is a frequency selective process, the important thing about a digital filter is its frequency response.

The filter's frequency response can be calculated from its filter equation:

$$H(f) = \frac{\sum c[k] * \exp(-2\pi j k(f\Delta))}{1 - \sum d[i] * \exp(-2\pi j i(f\Delta))}$$

© BOREF Signal Processing

Where j is the square root of minus one (defined as a number whose sole property is that its square is minus one).

The frequency response $H(f)$ is a continuous function, even though the filter equation is a discrete summation.

Whilst it is nice to be able to calculate the frequency response given the filter coefficients, when designing a digital filter we want to do the inverse operation: that is, to calculate the filter coefficients having first defined the desired frequency response. So we are faced with an inverse problem.

Sadly, there is no general inverse solution to the frequency response equation.

To make matters worse, we want to impose an additional constraint on acceptable solutions. Usually, we are designing digital filters with the idea that they will be implemented on some piece of hardware. This means we usually want to design a filter that meets the requirement but which requires the least possible amount of computation: that is, using the smallest number of coefficients. So we are faced with an insoluble inverse problem, on which we wish to impose additional constraints.

This is why digital filter design is more an art than a science: the art of finding an acceptable compromise between conflicting constraints.

If we have a powerful computer and time to take a coffee break while the filter calculates, the small number of coefficients may not be important - but this is a pretty sloppy way to work and would be more of an academic exercise than a piece of engineering.

Filtering: FIR filters

It is much easier to approach the problem of calculating filter coefficients if we simplify the filter equation so that we only have to deal with previous inputs (that is, we exclude the possibility of feedback). The filter equation is then simplified:

$$y[n] = \sum c[k] * x[n-k] + \sum d[j] * y[n-j]$$

© **BORES** Signal Processing

If such a filter is subjected to an impulse (a signal consisting of one value followed by zeroes) then its output must necessarily become zero after the impulse has run through the summation. So the impulse response of such a filter must necessarily be finite in duration. Such a filter is called a Finite Impulse Response filter or FIR filter.



© **BORES** Signal Processing

The filter's frequency response is also simplified, because all the bottom half goes away:

$$H(f) = \frac{\sum c[k] * \exp(-2\pi j k(f\Delta))}{1 - \sum d[j] * \exp(-2\pi j k(f\Delta))}$$

© **BORES** Signal Processing

It so happens that this frequency response is just the Fourier transform of the filter coefficients.

The inverse solution to a Fourier transform is well known: it is simply the inverse Fourier transform.

So the coefficients for an FIR filter can be calculated simply by taking the inverse Fourier transform of the desired frequency response.

Here is a recipe for calculating FIR filter coefficients:

- decide upon the desired frequency response
- calculate the inverse Fourier transform
- use the result as the filter coefficients

BUT...

Filtering: FIR filter design by the window method

So the filter coefficients for an FIR filter can be calculated simply by taking the inverse Fourier transform of the desired frequency response.

BUT...

- The inverse Fourier transform has to take samples of the continuous desired frequency response.
- to define a sharp filter needs closely spaced frequency samples - so a lot of them
- so the inverse Fourier transform will give us a lot of filter coefficients
- but we don't want a lot of filter coefficients

We can do a better job by noting that:

- the filter coefficients for an FIR filter are also the impulse response of the filter
- the impulse response of an FIR filter dies away to zero
- so many of the filter coefficients for an FIR filter are small
- and perhaps we can throw away these small values as being less important



© **BORES** Signal Processing

Here is a better recipe for calculating FIR filter coefficients based on throwing away the small ones:

- pretend we don't mind lots of filter coefficients
- specify the desired frequency response using lots of samples
- calculate the inverse Fourier transform
- this gives us a lot of filter coefficients
- so truncate the filter coefficients to give us less
- then calculate the Fourier transform of the truncated set of coefficients to see if it still matches our requirement

BUT...

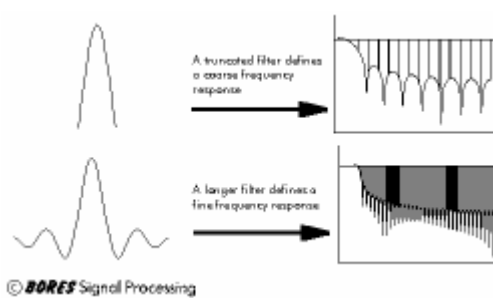
Filtering: FIR filter design by the window method

FIR filter coefficients can be calculated by taking the inverse Fourier transform of the desired frequency response and throwing away the small values:

- pretend we don't mind lots of filter coefficients
- specify the desired frequency response using lots of samples
- calculate the inverse Fourier transform
- this gives us a lot of filter coefficients
- so truncate the filter coefficients to give us less
- then calculate the Fourier transform of the truncated set of coefficients to see if it still matches our requirement

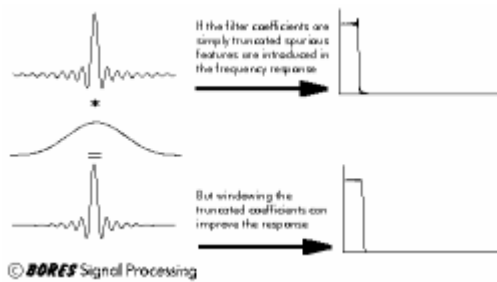
BUT...

Truncating the filter coefficients means we have a truncated signal. And a truncated signal has a broad frequency spectrum:



So truncating the filter coefficients means the filter's frequency response can only be defined coarsely.

Luckily, we already know a way to sharpen up the frequency spectrum of a truncated signal, by applying a window function. So after truncation, we can apply a window function to sharpen up the filter's frequency response:



So here is an even better recipe for calculating FIR filter coefficients:

- pretend we don't mind lots of filter coefficients
- specify the desired frequency response using lots of samples
- calculate the inverse Fourier transform
- this gives us a lot of filter coefficients
- so truncate the filter coefficients to give us less
- apply a window function to sharpen up the filter's frequency response
- then calculate the Fourier transform of the truncated set of coefficients to see if it still matches our requirement

This is called the window method of FIR filter design.

BUT...

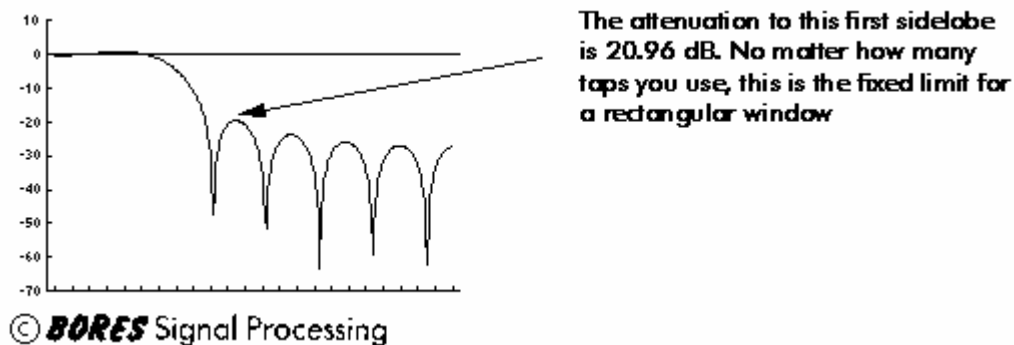
Filtering: FIR filter design by the window method

FIR filter coefficients can be calculated using the window method:

- pretend we don't mind lots of filter coefficients
- specify the desired frequency response using lots of samples
- calculate the inverse Fourier transform
- this gives us a lot of filter coefficients
- so truncate the filter coefficients to give us less
- apply a window function to sharpen up the filter's frequency response
- then calculate the Fourier transform of the truncated set of coefficients to see if it still matches our requirement

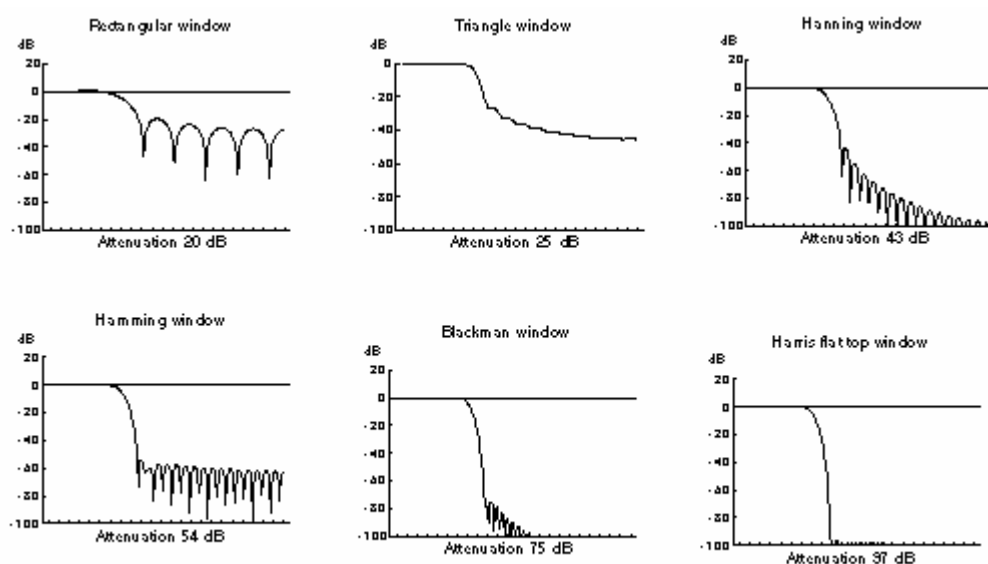
BUT...

Most window functions have a fixed attenuation to the top of their first sidelobe:



No matter how many filter coefficients you throw at it, you cannot improve on a fixed window's attenuation.

This means that the art of FIR filter design by the window method lies in an appropriate choice of window function:



© **BORES** Signal Processing

For example, if you need an attenuation of 20 dB or less, then a rectangle window is acceptable. If you need 43 dB you are forced to choose the Hanning window, and so on.

Sadly, the better window functions need more filter coefficients before their shape can be adequately defined. So if you need only 25 dB of attenuation you should choose a triangle window functions which will give you this attenuation: the Hamming window, for example, would give you more attenuation but require more filter coefficients to be adequately defined - and so would be wasteful of computer power.

The art of FIR filter design by the window method lies in choosing the window function which meets your requirement with the minimum number of filter coefficients.

You may notice that if you want an attenuation of 30 dB you are in trouble: the triangle window is not good enough but the Hanning window is too good (and so uses more coefficients than you need). The Kaiser window function is unique in that its shape is variable. A variable

parameter defines the shape, so the Kaiser window is unique in being able to match precisely the attenuation you require without overperforming.

Filtering: FIR filter design by the equiripple method

FIR filter coefficients can be calculated using the window method.

But the window method does not correspond to any known form of optimisation. In fact it can be shown that the window method is not optimal - by which we mean, it does not produce the lowest possible number of filter coefficients that just meets the requirement.

The art of FIR filter design by the window method lies in choosing the window function which meets your requirement with the minimum number of filter coefficients.

If the window method design is not good enough we have two choices:

- use another window function and try again
- do something clever

The Remez Exchange algorithm is something clever. It uses a mathematical optimisation method.

The following explanation is not mathematically correct, but since we are trying to get an idea of what is going on, and not trying to duplicate the thinking of geniuses, it is worth going through anyway.

Using the window method to design a filter we might proceed manually as follows:

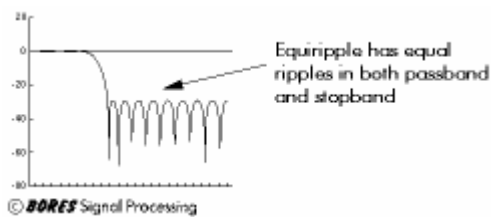
- choose a window function that we think will do
- calculate the filter coefficients
- check the actual filter's frequency response against the design goal
- if it overperforms, reduce the number of filter coefficients or relax the window function design
- try again until we find the filter with the lowest number of filter coefficients possible

In a way, this is what the Remez Exchange algorithm does automatically. It iterates between the filter coefficients and the actual frequency response until it finds the filter that just meets the specification with the lowest possible number of filter coefficients. Actually, the Remez Exchange algorithm never really calculates the frequency response: but it does keep comparing the actual with the design goal.

Remez was a Russian. Two Americans - Parks and McLellan - wrote a FORTRAN program to implement the Remez algorithm. So this type of filter design is often called a Parks McLellan filter design.

The Remez/Parks McLellan method produces a filter which just meets the specification without overperforming. Many of the window method designs actually perform better as you move further away from the passband: this is wasted performance, and means they are using more

filter coefficients than they need. Similarly, many of the window method designs actually perform better than the specification within the passband: this is also wasted performance, and means they are using more filter coefficients than they need. The Remez/Parks McLellan method performs just as well as the specification but no better: one might say it produces the worst possible design that just meets the specification at the lowest possible cost - almost a definition of practical engineering. So Remez/Parks McLellan designs have equal ripple - up to the specification but no more - in both passband and stopband. This is why they are often called equiripple designs.



The equiripple design produces the most efficient filters - that is, filters that just meet the specification with the least number of coefficients. But there are reasons why they might not be used in all cases:

- a particular filter shape may be desired, hence a choice of a particular window function
- equiripple is very time consuming - a design that takes a few seconds to complete using the window method can easily take ten or twenty minutes with the equiripple method
- the window method is very simple and easy to include in a program - for example, where one had to calculate a new filter according to some dynamically changing parameters
- there is no guarantee that the Remez Exchange algorithm will converge - it may converge to a false result (hence equiripple designs should always be checked): or it may not converge ever (resulting in hung computers, divide by zero errors and all sorts of other horrors)

We hope you have found this course module to be helpful. Please contact us with your comments or suggestions.

IIR Filters

This is the fifth module of the BORES Signal Processing DSP course - Introduction to DSP. It covers the following subjects:

- IIR filter equations and frequency response
- the z transform
- the meaning of z
- poles and zeroes
- IIR filter design by impulse invariance
- IIR filter design by the bilinear transform
- direct form I and II filters
- quantisation in IIR filters
- IIR filter implementation structures

IIR filters: Digital filter equation

Output from a digital filter is made up from previous inputs and previous outputs, using the operation of convolution:

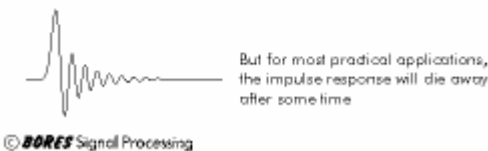
$$y[n] = \sum c[k] * x[n-k] + \sum d[i] * y[n-i]$$

Diagram illustrating the digital filter equation. The equation is $y[n] = \sum c[k] * x[n-k] + \sum d[i] * y[n-i]$. Arrows point from the text labels to the corresponding parts of the equation: 'Output' points to $y[n]$, 'previous input' points to $x[n-k]$, 'previous output' points to $y[n-i]$, and 'coefficients' points to $c[k]$ and $d[i]$.

© BORES Signal Processing

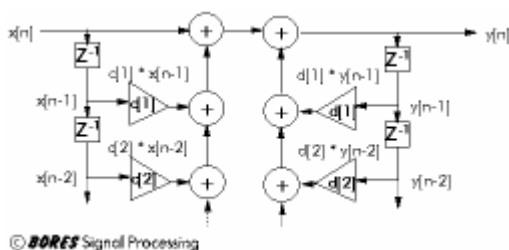
Two convolutions are involved: one with the previous inputs, and one with the previous outputs. In each case the convolving function is called the filter coefficients.

If such a filter is subjected to an impulse (a signal consisting of one value followed by zeroes) then its output need not necessarily become zero after the impulse has run through the summation. So the impulse response of such a filter can be infinite in duration. Such a filter is called an Infinite Impulse Response filter or IIR filter.



Note that the impulse response need not necessarily be infinite: if it were, the filter would be unstable. In fact for most practical filters, the impulse response will die away to a negligibly small level. One might argue that mathematically the response can go on for ever, getting smaller and smaller: but in a digital world once a level gets below one bit it might as well be zero. The Infinite Impulse Response refers to the ability of the filter to have an infinite impulse response and does not imply that it necessarily will have one: it serves as a warning that this type of filter is prone to feedback and instability.

The filter can be drawn as a block diagram:



The filter diagram can show what hardware elements will be required when implementing the filter:



Additions and multiplications

require us to:
fetch two operands
perform the addition or multiplication
store the result or hold it for a repetition



Delays

require us to:
hold a value for later use



Array handling

requires us to:
fetch values from consecutive memory locations
copy data from memory to memory

© BOREF Signal Processing

The left hand side of the diagram shows the direct path, involving previous inputs: the right hand side shows the feedback path, operating upon previous outputs.

IIR filters: Filter frequency response

Since filtering is a frequency selective process, the important thing about a digital filter is its frequency response.

The filter's frequency response can be calculated from its filter equation:

$$H(f) = \frac{\sum c[k] * \exp(-2\pi j k(f\Delta))}{1 - \sum d[i] * \exp(-2\pi j i(f\Delta))}$$

© BOREF Signal Processing

Where j is the square root of minus one (defined as a number whose sole property is that its square is minus one).

The frequency response $H(f)$ is a continuous function, even though the filter equation is a discrete summation.

Whilst it is nice to be able to calculate the frequency response given the filter coefficients, when designing a digital filter we want to do the inverse operation: that is, to calculate the filter coefficients having first defined the desired frequency response. So we are faced with an inverse problem.

Sadly, there is no general inverse solution to the frequency response equation.

To make matters worse, we want to impose an additional constraint on acceptable solutions. Usually, we are designing digital filters with the idea that they will be implemented on some piece of hardware. This means we usually want to design a filter that meets the requirement but which requires the least possible amount of computation: that is, using the smallest number of coefficients. So we are faced with an insoluble inverse problem, on which we wish to impose additional constraints.

This is why digital filter design is more an art than a science: the art of finding an acceptable compromise between conflicting constraints.

If we have a powerful computer and time to take a coffee break while the filter calculates, the small number of coefficients may not be important - but this is a pretty sloppy way to work and would be more of an academic exercise than a piece of engineering.

IIR filters: The z transform

The equation for the filter's frequency response can be simplified by substituting a new variable, z :

$$H(f) = \frac{\sum c[k] * \exp(-2\pi j k(f\Delta))}{1 - \sum d[j] * \exp(-2\pi j j(f\Delta))}$$

$z = \exp(2\pi j f\Delta)$
 $H(z) = \frac{\sum c[k] * (1/z)^k}{1 - \sum d[j] * (1/z)^j}$

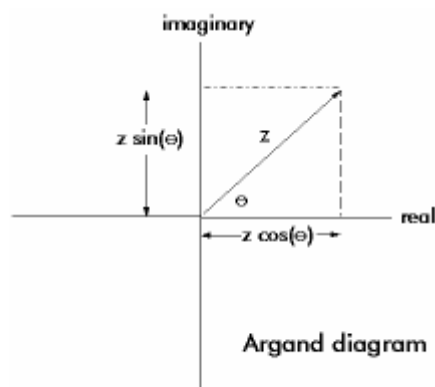
© BOREF Signal Processing

Note that z is a complex number.

$$z = \exp(2\pi j f\Delta)$$

© BOREF Signal Processing

Complex numbers can be drawn using an Argand diagram. This is a plot where the horizontal axis represents the real part, and the vertical axis the imaginary part, of the number.



© BOREF Signal Processing

The complex variable z is shown as a vector on the Argand diagram.

The z transform is defined as a sum of signal values $x[n]$ multiplied by powers of z :

$$X[z] = \sum x[n] * (1/z)^n$$

© BOREF Signal Processing

Which has the curious property of letting us generate an earlier signal value from a present one, because the z transform of $x[n-1]$ is just the z transform of $x[n]$ multiplied by $(1/z)$:

$$X[n-1] = (1/z) \sum x[n] * (1/z)^n = (1/z) X[n]$$

© **BORES** Signal Processing

So the z transform of the last signal value can be obtained by multiplying the z transform of the current value by (1/z). This is why, in the filter diagram, the delay elements are represented formally using the 1/z notation.

IIR filters: The meaning of z

z is a complex number:

$$z = \exp(2\pi j f \Delta)$$

© **BORES** Signal Processing

When drawn on the Argand diagram, z has the curious property that it can only have a magnitude of 1:

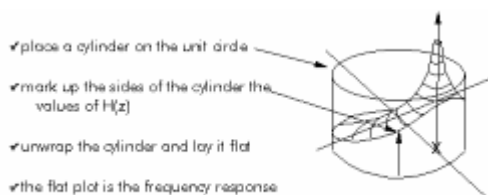
$$\begin{aligned} \checkmark z &= \exp(2\pi j f \Delta) \\ \checkmark \exp(2\pi j f \Delta) &= \cos(2\pi f \Delta) + j \sin(2\pi f \Delta) \\ \checkmark \text{so } z^2 &= \{\cos(2\pi f \Delta) + j \sin(2\pi f \Delta)\}^2 \\ \checkmark \text{and } \{\cos(2\pi f \Delta) + j \sin(2\pi f \Delta)\}^2 &= \cos^2(2\pi f \Delta) + \sin^2(2\pi f \Delta) \\ \checkmark \text{and } \cos^2(2\pi f \Delta) + \sin^2(2\pi f \Delta) &= 1 \\ \checkmark \text{so } |z| &= 1 \end{aligned}$$

© **BORES** Signal Processing

So z, which is the variable used in our frequency response, traces a circle of radius 1 on the Argand diagram. This is called the unit circle.

The map of values in the z plane is called the transfer function H(z).

The frequency response is the transfer function H(z) evaluated around the unit circle on the Argand diagram of z:



© **BORES** Signal Processing

Note that in the sampled data z plane, frequency response maps onto a circle - which helps to visualise the effect of aliasing.

IIR filters: The meaning of z

z is a complex number:

$$z = \exp(2\pi j f \Delta)$$

© **BORES** Signal Processing

When drawn on the Argand diagram, z has the curious property that it can only have a magnitude of 1:

$\checkmark z = \exp(j2\pi f\Delta)$
 $\checkmark \exp(j2\pi f\Delta) = \cos(2\pi f\Delta) + j\sin(2\pi f\Delta)$
 $\checkmark \text{so } z^2 = \{\cos(2\pi f\Delta) + j\sin(2\pi f\Delta)\}^2$
 $\checkmark \text{and } \{\cos(2\pi f\Delta) + j\sin(2\pi f\Delta)\}^2 = \cos^2(2\pi f\Delta) + \sin^2(2\pi f\Delta)$
 $\checkmark \text{and } \cos^2(2\pi f\Delta) + \sin^2(2\pi f\Delta) = 1$
 $\checkmark \text{so } |z| = 1$

© BOREF Signal Processing

So z , which is the variable used in our frequency response, traces a unit circle on the Argand diagram.

At first sight, z can have no value off the unit circle.

But if we use a mathematical fiction for a moment, we can imagine that the frequency f could itself be a complex number:

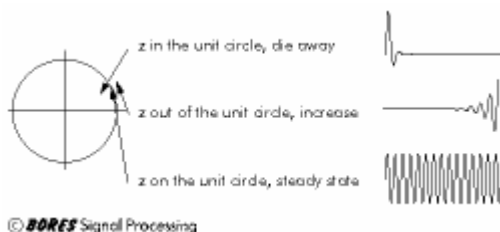
$$f(\text{complex}) = f_r + jf_i$$

In which case, the j from imaginary frequency component can cancel the j in the z term:

$$z = \exp(j2\pi f_r\Delta) * \exp(2\pi f_i\Delta)$$

and the imaginary component of frequency introduces a straightforward exponential decay on top of the complex oscillation: showing that z can be off the unit circle, and that if it is this relates to transient response. The imaginary frequency has to do with transient response, while the real frequency (both real as in actual, and real as in the real part of a complex number) has to do with steady state oscillation.

For real frequencies z lies on the unit circle. Values of the transfer function $H(z)$ for z off the unit circle relate to transient terms:



© BOREF Signal Processing

The position of z , inside or outside the unit circle, determines the stability of transient terms:

- if z is inside the unit circle, the transient terms will die away
- if z is on the unit circle, oscillations will be in a steady state
- if z is outside the unit circle, the transient terms will increase

IIR filters: Poles and zeroes

The IIR filter's transfer function is a ratio of terms.

- if the numerator becomes zero, the transfer function will also become zero - this is called a zero of the function

- if the denominator becomes zero, we have a division by zero - the function can become infinitely large - this is called a pole of the function

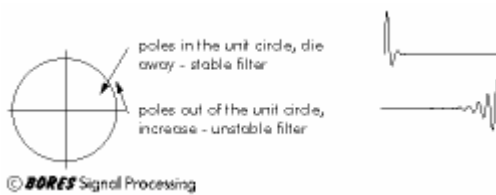
$$H(f) = \frac{\sum c[k] * (1/z)^k}{1 - \sum d[j] * (1/z)^j}$$

If the numerator is zero, the function is zero - a 'zero' of the function

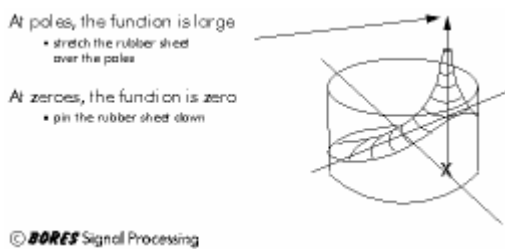
If the denominator is zero, the function can be infinite - a 'pole' of the function

© BOREF Signal Processing

The positions of poles (very large values) affects the stability of the filter:



The shape of the transfer function $H(z)$ is determined by the positions of its poles and zeroes:

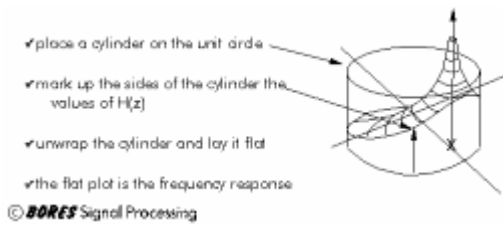


This can be visualised using the rubber sheet analogy:

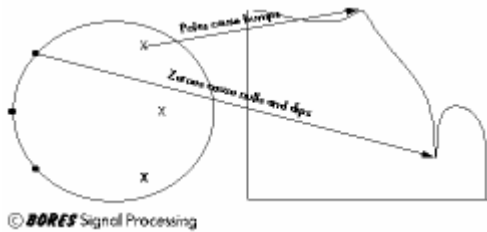
- imagine the Argand diagram laid out on the floor
- place tall vertical poles at the poles of the function
- stretch a rubber sheet over the poles
- at zeroes, pin the rubber sheet to the floor
- the rubber sheet will take up a shape which is determined by the position of the poles and zeroes

Thanks are due to Jim Richardson for the rubber sheet analogy, which came to mind while he was an instructor officer at the Royal Naval Engineering College, Devonport.

Now the frequency response is the transfer function $H(z)$ evaluated around the unit circle on the Argand diagram of z :



and since the shape of the transfer function can be determined from the positions of its poles and zeroes, so can the frequency response.



The frequency response can be determined by tracing around the unit circle on the Argand diagram of the z plane:

- project poles and zeroes radially to hit the unit circle
- poles cause bumps
- zeroes cause dips
- the closer to the unit circle, the sharper the feature

IIR filters: IIR filter design by impulse invariance

Direct digital IIR filter design is rarely used, for one very simple reason:

- nobody knows how to do it

While it is easy to calculate the filter's frequency response, given the filter coefficients, the inverse problem - calculating the filter coefficients from the desired frequency response - is so far an insoluble problem. Not many text books admit this.

Because we do not know how to design digital IIR filters, we have to fall back on analogue filter designs (for which the mathematics is well understood) and then transform these designs into the sampled data z plane Argand diagram.

Note that the filter's impulse response defines it just as well as does its frequency response.

Here is a recipe for designing an IIR digital filter:

- decide upon the desired frequency response
- design an appropriate analogue filter
- calculate the impulse response of this analogue filter
- sample the analogue filter's impulse response
- use the result as the filter coefficients

calculate the impulse response of an analogue filter



and sample it



and use the samples to define the digital filter



© **BORES** Signal Processing

This process is called the method of impulse invariance.

The method of impulse invariance seems simple: but it is complicated by all the problems inherent in dealing with sampled data systems. In particular the method is subject to problems of aliasing and frequency resolution.

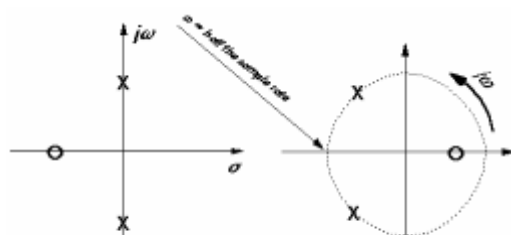
IIR filters: IIR filter design by the bilinear transform

The method of filter design by impulse invariance suffers from aliasing.

The aliasing will be a problem if the analogue filter prototype's frequency response has significant components at or beyond the Nyquist frequency.

The problem with which we are faced is to transform the analogue filter design into the sampled data z plane Argand diagram. The problem of aliasing arises because the frequency axis in the sampled data z plane Argand diagram is a circle:

- in the analogue domain the frequency axis is an infinitely long straight line
- in the sampled data z plane Argand diagram the frequency axis is a circle



In the analogue s plane the frequency axis is straight and infinite

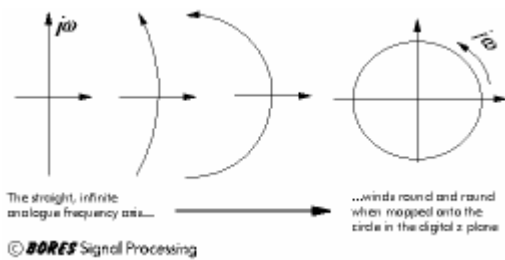
In the digital z plane the frequency axis is the unit circle

© **BORES** Signal Processing

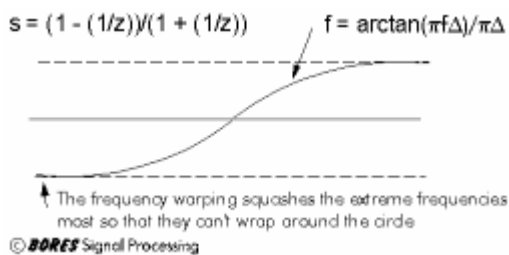
Note also that:

- in the analogue domain transient response is shown along the horizontal axis
- in the sampled data z plane Argand diagram transient response is shown radially outwards from the centre

The problem of aliasing arises because we wrap an infinitely long, straight frequency axis around a circle. So the frequency axis wraps around and around, and any components above the Nyquist frequency get wrapped back on top of other components.



The bilinear transform is a method of squashing the infinite, straight analogue frequency axis so that it becomes finite. This is like squashing a concertina or accordion. To avoid squashing the filter's desired frequency response too much, the bilinear transform squashes the far ends of the frequency axis the most - leaving the middle portion relatively unsquashed:

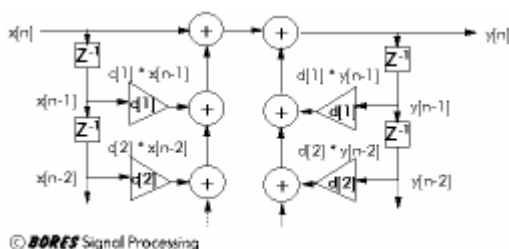


The infinite, straight analogue frequency axis is squashed so that it becomes finite - in fact just long enough to wrap around the unit circle once only. This is also sometimes called frequency warping

Sadly, frequency warping does change the shape of the desired filter frequency response. In particular, it changes the shape of the transition bands. This is a pity, since we went to a lot of trouble designing an analogue filter prototype that gave us the desired frequency response and transition band shapes. One way around this is to warp the analogue filter design before transforming it to the sampled data z plane Argand diagram: this warping being designed so that it will be exactly undone by the frequency warping later on. This is called prewarping.

IIR filters: Direct form I

Filters can be drawn as diagrams:



This particular diagram is called the direct form 1 because the diagram can be drawn directly from the filter equation.

The filter diagram can show what hardware elements will be required when implementing the filter:

\oplus \triangleright **Additions and multiplications**
 require us to:
 fetch two operands
 perform the addition or multiplication
 store the result or hold it for a repetition

Z^{-1} **Delays**
 require us to:
 hold a value for later use

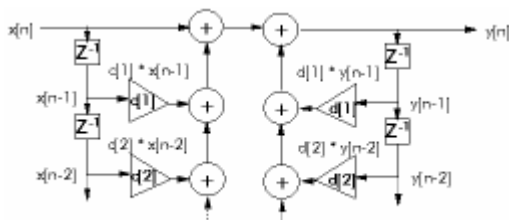
$c[2]$ **Array handling**
 requires us to:
 fetch values from consecutive memory locations
 copy data from memory to memory

© **BORES** Signal Processing

The left hand side of the diagram shows the direct path, involving previous inputs: the right hand side shows the feedback path, operating upon previous outputs.

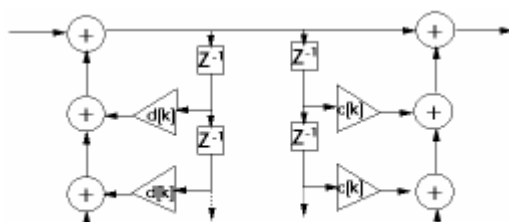
IIR filters: Direct form II

The filter diagram for direct form 1 can be drawn direct from the filter equation:



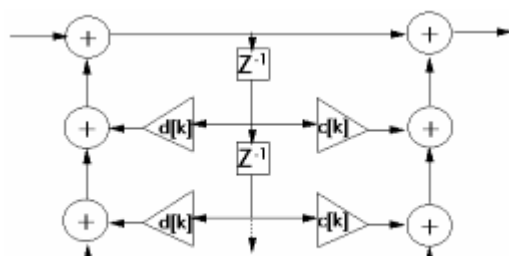
© **BORES** Signal Processing

The block diagram is in two halves: and since the results from each half are simply added together it does not matter in which order they are calculated. So the order of the halves can be swapped:



© **BORES** Signal Processing

Now, note that the result after each delay is the same for both branches. So the delays down the centre can be combined:



© **BORES** Signal Processing

This is called direct form 2. Its advantage is that it needs less delay elements. And since delay elements require hardware (for example, processor registers) the direct form 2 requires less hardware and so is more efficient than direct form 1.

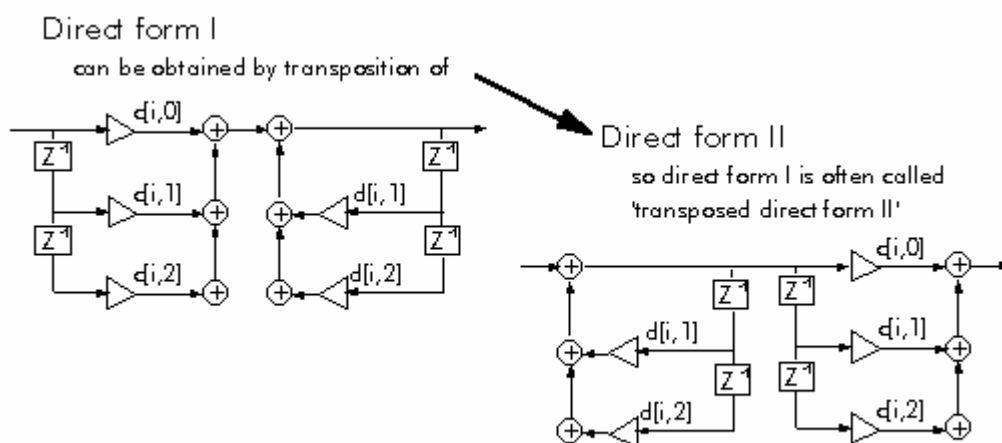
direct form 2 is also called canonic, which simply means 'having the minimum number of delay elements'.

IIR filters: Direct form II

The transposition theorem says that if we take a filter diagram and reverse all the elements - swapping the order of execution for every element, and reversing the direction of all the flow arrows - then the result is the same:

- if everything is turned back to front, it all works just the same

This means that the direct form 1 diagram can be obtained by transposition of the direct form 2 diagram:



© **BORES** Signal Processing

For this reason, direct form 1 is often called transposed direct form 2.

Don't ask me why these terms seem to be as confusing as they possibly could be - I didn't make them up. I imagine mathematicians sit around at coffee break and come up with new ways to spread despondency amongst us lesser mortals. Here are the two main sources of confusion:

direct form 1

so called because it can be drawn direct from the filter equation

direct form 2

so called because it can be derived by changing the diagram of direct form 1

transposed

so called because it is obtained by transposition of direct form 2 - but really, this is just direct form 1

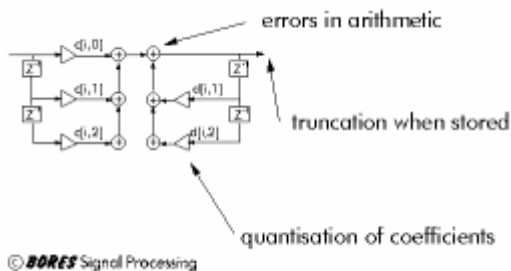
canonic

so called because it has the minimum number of delay elements - but really, it is just direct form 2

IIR filters: Quantisation in IIR filters

Digital filters are examples of sampled data systems.

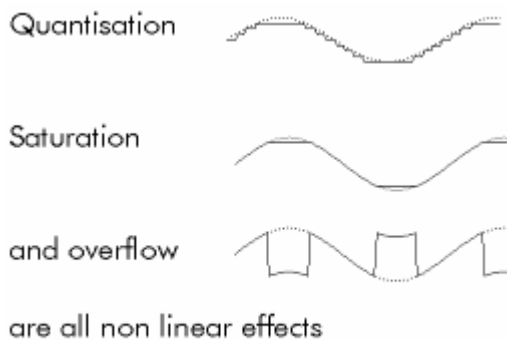
Sampled data systems suffer from problems of limited precision which lead to quantisation errors. Apart from errors when measuring signals, these arise within the hardware used for processing:



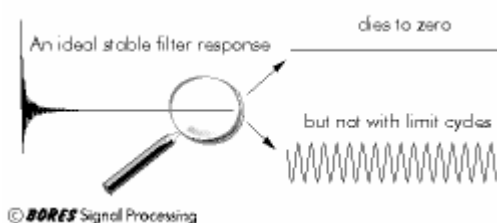
Primary sources of quantisation error are:

- errors in arithmetic within the hardware (for example 16 bit fixed point roundoff)
- truncation when results are stored (most DSP processors have extended registers internally, so truncation usually occurs when results are stored to memory)
- quantisation of filter coefficients which have to be stored in memory

The effects of quantisation, saturation and overflow are all non linear, signal dependent errors. This is bad news because non linear effects cannot be calculated using normal mathematics: and because signal dependent (coherent) errors cannot be treated statistically using the usual assumptions about randomness of noise - they will depend on what signals are being processed.



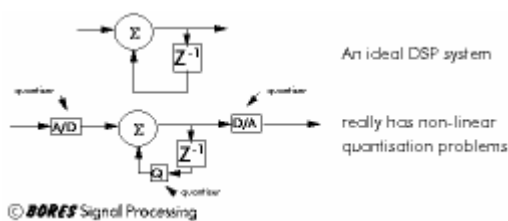
One example of a strange effect of non linear systems is limit cycles:



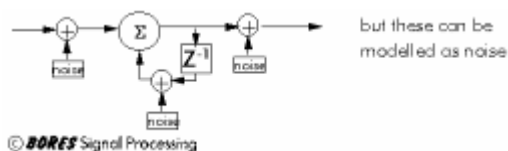
A non linear system can oscillate at a low bit level, even when there is no input. This is not the same as the impulse response being infinite - it is a result of a non linear (or chaotic) system.

Limit cycles can sometimes be heard when listening to modern 'sigma delta' digital to analogue converters. These chips use long digital filters which are subject to non linear errors - and you can sometimes hear the effect of limit cycles as quiet hisses or pops even when the digital output to the DAC is held steady.

When treating quantisation effects we usually acknowledge that these are non linear, signal dependent errors:



but we often model these as if they were injections of random noise:



Sadly, with IIR filters the non linear, signal dependent effects dominate and the model of quantisation as random noise is completely inadequate. The effects will also depend on the hardware used to implement the filter: for example most DSP processors have extended registers internally - whether these are used or not will affect the quantisation error crucially.

It is not possible to model the effects of quantisation in an IIR filter using simple random noise models.

Some idea of quantisation effects in IIR filters can be gained using complex statistical models: but really the only way to evaluate the effects of quantisation in an IIR filter is to:

- implement the filter on the target hardware
- test it with signals of the sort expected

IIR filters: Quantisation in IIR filters

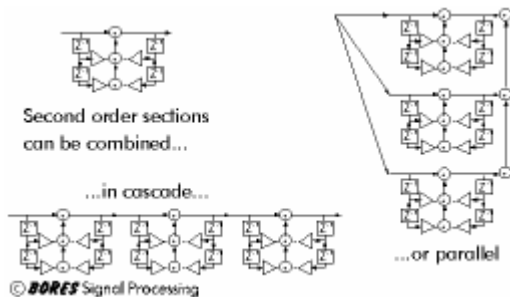
IIR filters are very sensitive to quantisation errors.

The higher the order of the filter, the more it suffers from quantisation effects: because the filter is more complex, and so the errors accumulate more.

In fact, IIR filters are so sensitive to quantisation errors that it is generally unrealistic to expect anything higher than a second order filter to work.

This is why IIR filters are usually realised as second order sections. Most analogue filters (except for Bessel filters) are also usually realised as second order sections, which is a convenient excuse but not the real one.

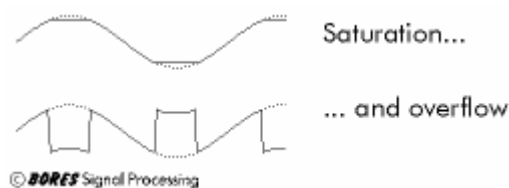
Second order sections can be combined to create higher order filters:



IIR filters: Quantisation in IIR filters

Quantisation errors can be minimised by keeping values large - so that the maximum number of bits is used to represent them.

There is a limit to how large numbers can be, determined by the precision of the hardware used for processing. If the maximum number size is exceeded, the hardware may allow overflow or saturation:



Saturation and overflow are both non linear quantisation errors.

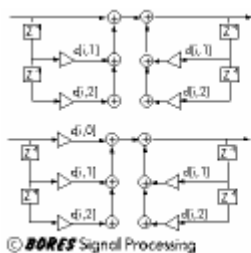
Note that overflow, although looking more drastic than saturation, may be preferred. It is a property of two's complement integer arithmetic that if a series of numbers are added together, even if overflow occurs at intermediate stages, so long as the result is within the range that can be represented the result will be correct.

Overflow or saturation can be avoided by scaling the input to be small enough that overflow does not occur during the next stage of processing. There are two choices:

- scaling the input so that overflow can never occur
- scaling the input so that the biggest reasonably expected signal never overflows

Scaling reduces the number of bits left to represent a signal (dividing down means some low bits are lost), so it increases quantisation errors.

Scaling requires an extra multiplier in the filter, which means more hardware:



The basic direct form second order section has four multipliers...

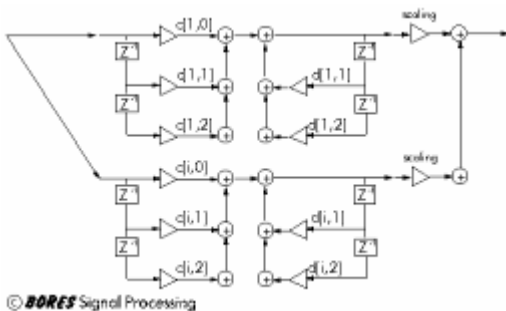
...but scaling introduces an extra multiplier, making five

Note that hardware with higher precision or using floating point arithmetic, may not require scaling and so can implement filters with less operations.

IIR filters: Parallel and cascade IIR structures

Because IIR filters are very sensitive to quantisation errors, they are usually implemented as second order sections.

The parallel form is simple:

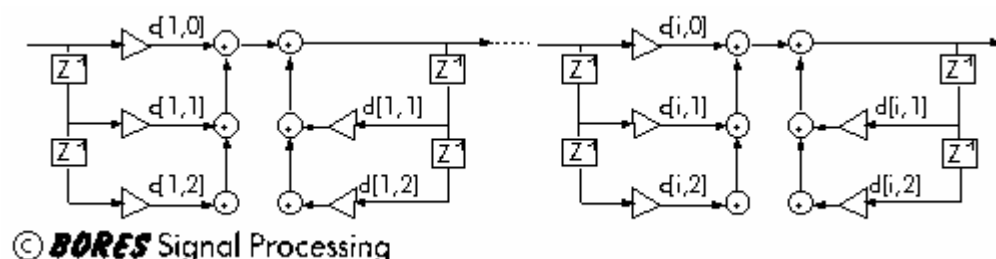


The outputs from each second order section are simply added together.

If scaling is required, this is done separately for each section. It is possible to scale each section appropriately, and by a different scale factor, to minimise quantisation error. In this case another extra multiplier is required for each section, to scale the individual section outputs back to the same common scale factor before adding them.

The order in which parallel sections are calculated does not matter, since the outputs are simply added together at the end.

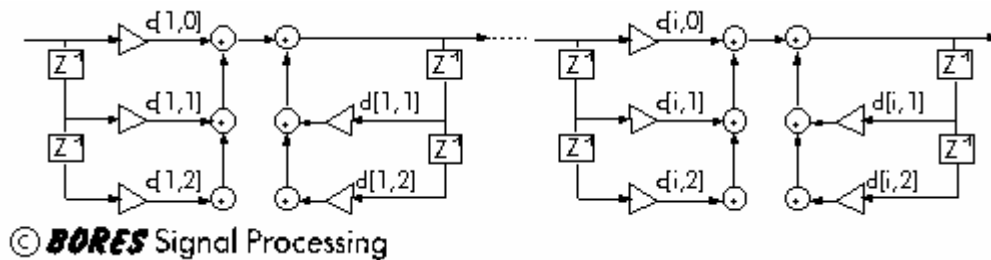
In the cascade form, the output of one section forms the input to the next:



Mathematically, it does not matter in which order the sections are placed - the result will be the same. This assumes that there are no errors. In practice, the propagation of errors is crucial to the success of an IIR filter so the order of the sections in the cascade form is vital.

IIR filters: Cascade IIR structure

In the cascade form, the output of one section forms the input to the next:



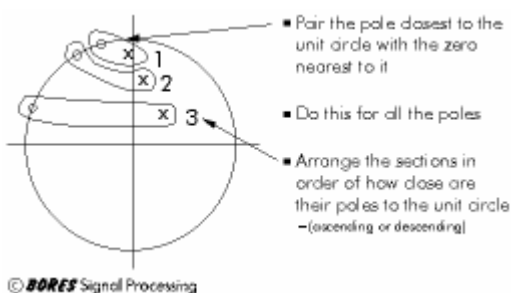
In practice, the propagation of errors is crucial to the success of an IIR filter so the order of the sections in the cascade, and the selection of which filter coefficients to group in each section, is vital:

- sections with high gain are undesirable because they increase the need for scaling and so increase quantisation errors
- it is desirable to arrange sections to avoid excessive scaling

To reduce the gain of each section we note that:

- poles cause high gain (bumps in the frequency response)
- zeroes cause low gain (dips in the frequency response)
- the closer to the unit circle, the greater the effect

This suggests a way to group poles and zeroes in each section to avoid high gain sections:



Note that the pole closest to the unit circle will provide the highest gain because it is a large value close to the unit circle. This can best be countered by pairing it with the zero closest to it. Here is a recipe for grouping poles and zeroes to create sections which avoid high gain:

- pair the pole closest to the unit circle with the zero closest to it (note: not closest to the unit circle)
- do this for all the poles, working up in terms of their distance from the unit circle
- arrange the sections in order of how close their poles are to the unit circle

The question remains, whether to place the high gain sections first or last.

Recall that:

- poles are large values (high gain)
- the closer to the unit circle, the higher the gain
- poles cause bumps in the frequency response
- the closer to the unit circle, the sharper the bump (high Q)
- poles in the early stages affect the input to later stages
- poles at late stages have the last word

So, the section with the pole closest to the unit circle will have the highest gain but also the sharpest shape. As with so much else in digital filter design, we are faced with a compromise between conflicting desires:

- poles close to the unit circle in early stages cause high gain early on, so require more signal scaling and worse quantisation errors later on
- poles close to the unit circle in late stages cause significant noise shaping at a late stage

DSP processors

This is the sixth module of the BORES Signal Processing DSP course - Introduction to DSP.

This module introduces the common features of DSP processors and shows how these relate to, and arise from, the requirements of typical DSP applications. The aim of this module is to convey an understanding of what makes DSP processors special, and of what to look for when evaluating one chip against another. It covers the following subjects:

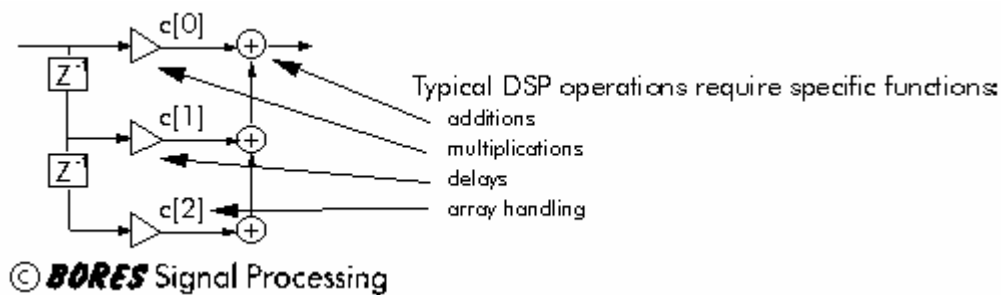
- characteristic features of DSP processors
- special features for arithmetic
- I/O interfaces
- memory architectures
- data formats
- some basic DSP chip designs
- brief overview of some major DSP processors

DSP processors: Characteristics of DSP processors

Although there are many DSP processors, they are mostly designed with the same few basic operations in mind: so they share the same set of basic characteristics. These characteristics fall into three categories:

- specialised high speed arithmetic
- data transfer to and from the real world
- multiple access memory architectures

Typical DSP operations require a few specific operations:



The diagram shows an FIR filter. This illustrates the basic DSP operations:

- additions and multiplications
- delays
- array handling

Each of these operations has its own special set of requirements:



additions and multiplications require us to:

- fetch two operands
- perform the addition or multiplication (usually both)
- store the result or hold it for a repetition



delays require us to:

- hold a value for later use



array handling requires us to:

- fetch values from consecutive memory locations
- copy data from memory to memory

To suit these fundamental operations DSP processors often have:

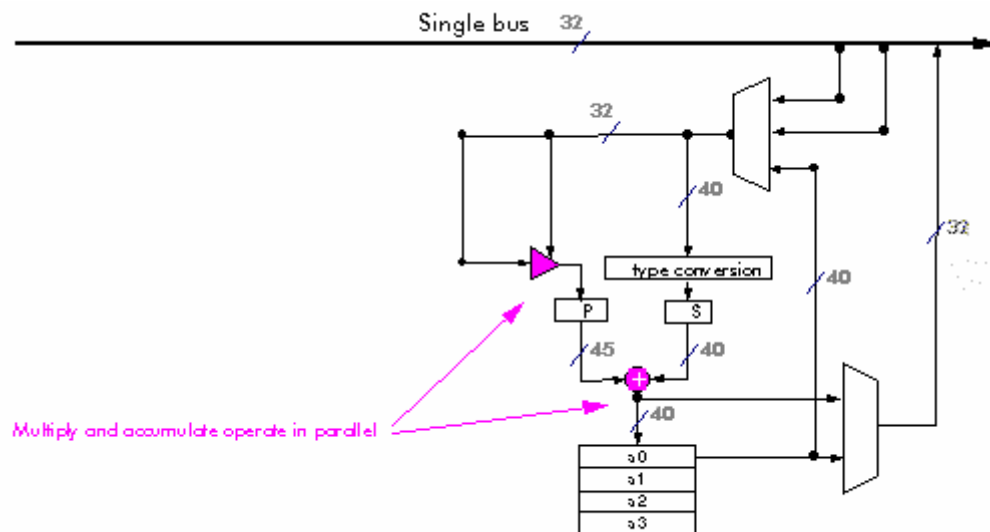
- parallel multiply and add
- multiple memory accesses (to fetch two operands and store the result)
- lots of registers to hold data temporarily
- efficient address generation for array handling
- special features such as delays or circular addressing

DSP processors: Mathematics

To perform the simple arithmetic required, DSP processors need special high speed arithmetic units.



➕ ➤ Most DSP operations require additions and multiplications together. So DSP processors usually have hardware adders and multipliers which can be used in parallel within a single instruction:



© **BORES** Signal Processing

The diagram shows the data path for the Lucent DSP32C processor. The hardware multiply and add work in parallel so that in the space of a single instruction, both an add and a multiply can be completed.



[Z⁻¹] Delays require that intermediate values be held for later use. This may also be a requirement, for example, when keeping a running total - the total can be kept within the processor to avoid wasting repeated reads from and writes to memory. For this reason DSP processors have lots of registers which can be used to hold intermediate values:



Registers may be fixed point or floating point format.



c[2] Array handling requires that data can be fetched efficiently from consecutive memory locations. This involves generating the next required memory address. For this reason DSP processors have address registers which are used to hold addresses and can be used to generate the next needed address efficiently:



The ability to generate new addresses efficiently is a characteristic feature of DSP processors. Usually, the next needed address can be generated during the data fetch or store operation, and with no overhead. DSP processors have rich sets of address generation operations:

*rP	register indirect	read the data pointed to by the address in register rP
*rP++	postincrement	having read the data, postincrement the address pointer to point to the next value in the array
*rP--	postdecrement	having read the data, postdecrement the address pointer to point to the previous value in the array
*rP++rI	register postincrement	having read the data, postincrement the address pointer <i>by the amount held in register rI</i> to point to <i>rI</i> values further down the array
*rP++rIr bit reversed		having read the data, postincrement the address pointer to point to the next value in the array, <i>as if the address bits were in bit reversed order</i>

The table shows some addressing modes for the Lucent DSP32C processor. The assembler syntax is very similar to C language. Whenever an operand is fetched from memory using register indirect addressing, the address register can be incremented to point to the next needed value in the array. This address increment is free - there is no overhead involved in the address calculation - and in the case of the Lucent DSP32C processor up to three such addresses may be generated in each single instruction. Address generation is an important factor in the speed of DSP processors at their specialised operations.

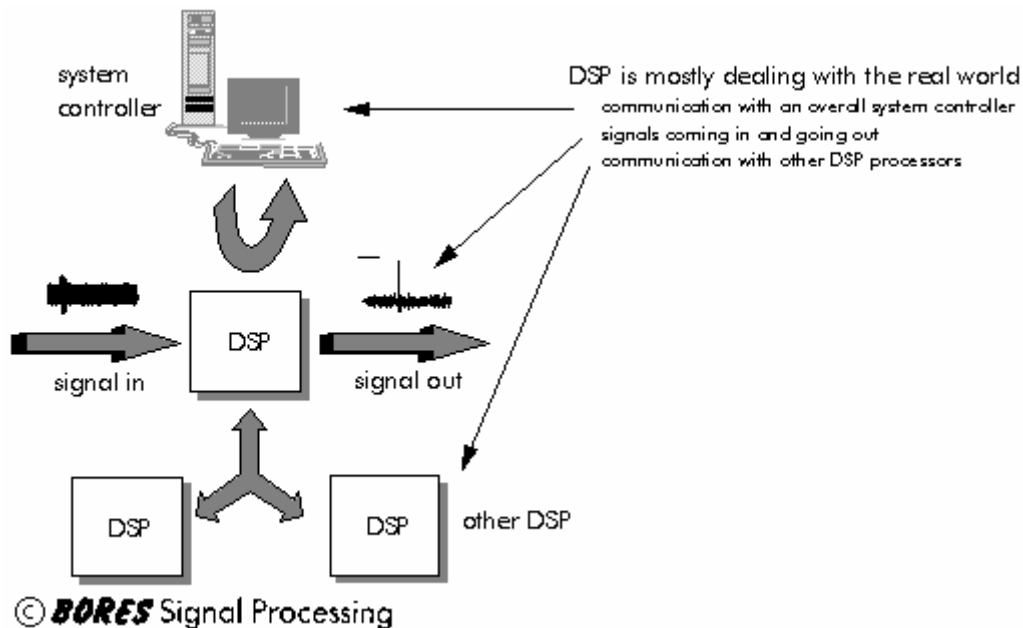
The last addressing mode - bit reversed - shows how specialised DSP processors can be. Bit reversed addressing arises when a table of values has to be reordered by reversing the order of the address bits:

- reverse the order of the bits in each address
- shuffle the data so that the new, bit reversed, addresses are in ascending order

This operation is required in the Fast Fourier Transform - and just about nowhere else. So one can see that DSP processors are designed specifically to calculate the Fast Fourier Transform efficiently.

DSP processors: Input and output interfaces

In addition to the mathematics, in practice DSP is mostly dealing with the real world. Although this aspect is often forgotten, it is of great importance and marks some of the greatest distinctions between DSP processors and general purpose microprocessors:



In a typical DSP application, the processor will have to deal with multiple sources of data from the real world. In each case, the processor may have to be able to receive and transmit data in real time, without interrupting its internal mathematical operations. There are three sources of data from the real world:

- signals coming in and going out
- communication with an overall system controller of a different type
- communication with other DSP processors of the same type

These multiple communications routes mark the most important distinctions between DSP processors and general purpose processors.

When DSP processors first came out, they were rather fast processors: for example the first floating point DSP - the AT&T DSP32 - ran at 16 MHz at a time when PC computer clocks were 5 MHz. This meant that we had very fast floating point processors: a fashionable demonstration at the time was to plug a DSP board into a PC and run a fractal (Mandelbrot) calculation on the DSP and on a PC side by side. The DSP fractal was of course faster. Today, however, the fastest DSP processor is the Texas TMS320C6201 which runs at 200 MHz. This is no longer very fast compared with an entry level PC. And the same fractal today will actually run faster on the PC than on the DSP. But DSP processors are still used - why? The answer lies only partly in that the DSP can run several operations in parallel: a far more basic answer is that the DSP can handle signals very much better than a Pentium. Try feeding eight channels of high quality audio data in and out of a Pentium simultaneously in real time, without impacting on the processor performance, if you want to see a real difference.

The need to deal with these different sources of data efficiently leads to special communication features on DSP processors:

DSP processors: Memory architectures

Typical DSP operations require simple many additions and multiplications.



additions and multiplications require us to:

- fetch two operands
- perform the addition or multiplication (usually both)
- store the result or hold it for a repetition

To fetch the two operands in a single instruction cycle, we need to be able to make two memory accesses simultaneously.

Actually, a little thought will show that since we also need to store the result - and to read the instruction itself - we really need more than two memory accesses per instruction cycle.

For this reason DSP processors usually support multiple memory accesses in the same instruction cycle. It is not possible to access two different memory addresses simultaneously over a single memory bus. There are two common methods to achieve multiple memory accesses per instruction cycle:

- Harvard architecture
- modified von Neuman architecture

The Harvard architecture has two separate physical memory buses. This allows two simultaneous memory accesses:

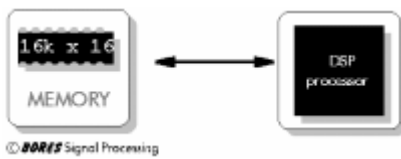


The true Harvard architecture dedicates one bus for fetching instructions, with the other available to fetch operands. This is inadequate for DSP operations, which usually involve at least two operands. So DSP Harvard architectures usually permit the 'program' bus to be used also for access of operands. Note that it is often necessary to fetch three things - the instruction plus two operands - and the Harvard architecture is inadequate to support this: so DSP Harvard architectures often also include a cache memory which can be used to store instructions which will be reused, leaving both Harvard buses free for fetching operands. This extension - Harvard architecture plus cache - is sometimes called an extended Harvard architecture or Super Harvard ARCHitecture (SHARC).

The Harvard architecture requires two memory buses. This makes it expensive to bring off the chip - for example a DSP using 32 bit words and with a 32 bit address space requires at least 64 pins for each memory bus - a total of 128 pins if the Harvard architecture is brought off the chip. This results in very large chips, which are difficult to design into a circuit.

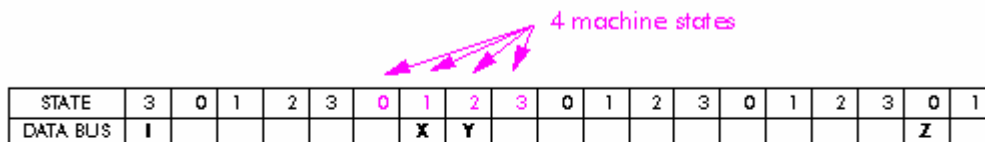
Even the simplest DSP operation - an addition involving two operands and a store of the result to memory - requires four memory accesses (three to fetch the two operands and the instruction, plus a fourth to write the result) This exceeds the capabilities of a Harvard architecture. Some processors get around this by using a modified von Neuman architecture.

The von Neuman architecture uses only a single memory bus:



This is cheap, requiring less pins than the Harvard architecture, and simple to use because the programmer can place instructions or data anywhere throughout the available memory. But it does not permit multiple memory accesses.

The modified von Neuman architecture allows multiple memory accesses per instruction cycle by the simple trick of running the memory clock faster than the instruction cycle. For example the Lucent DSP32C runs with an 80 MHz clock: this is divided by four to give 20 million instructions per second (MIPS), but the memory clock runs at the full 80 MHz - each instruction cycle is divided into four 'machine states' and a memory access can be made in each machine state, permitting a total of four memory accesses per instruction cycle:



in each instruction cycle

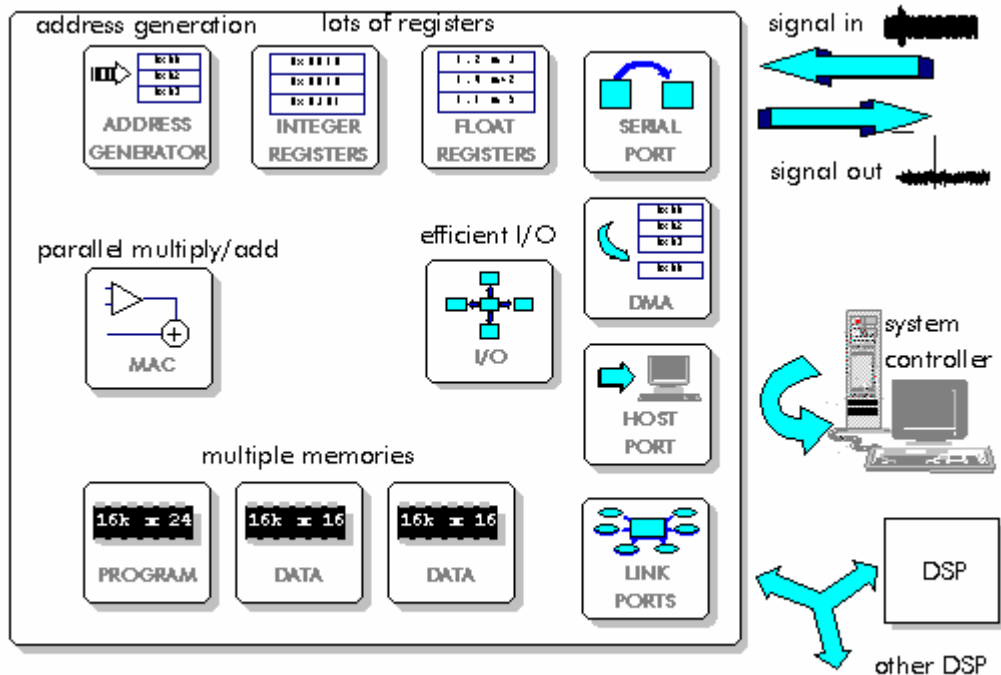
© **BORES** Signal Processing

In this case the modified von Neuman architecture permits all the memory accesses needed to support addition or multiplication: fetch of the instruction; fetch of the two operands; and storage of the result.

Both Harvard and von Neuman architectures require the programmer to be careful of where in memory data is placed: for example with the Harvard architecture, if both needed operands are in the same memory bank then they cannot be accessed simultaneously.

DSP processors: Example processors

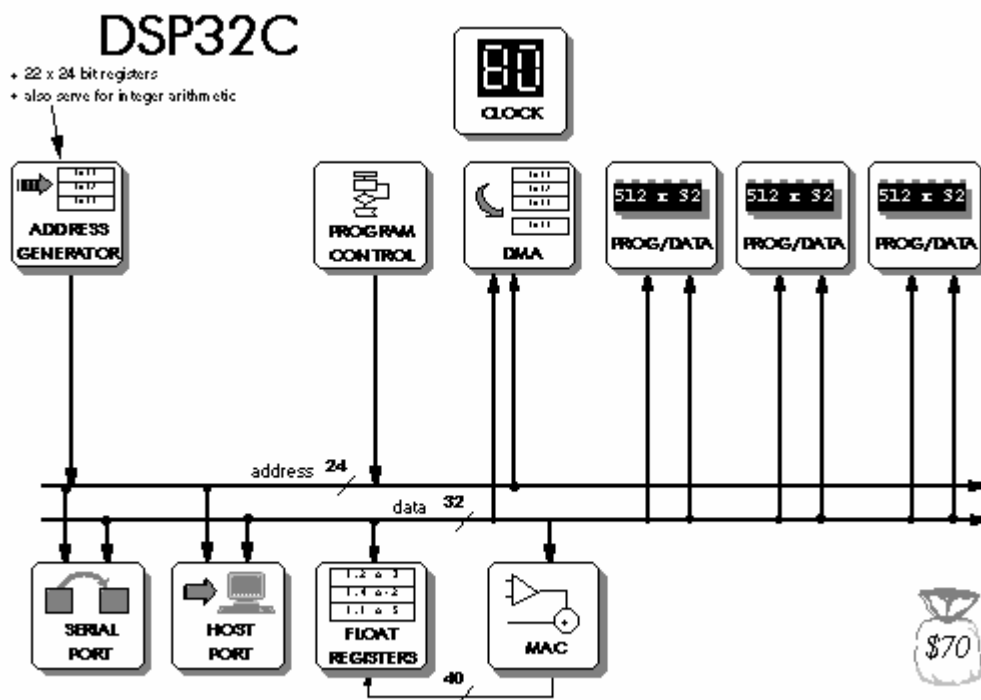
Although there are many DSP processors, they are mostly designed with the same few basic operations in mind: so they share the same set of basic characteristics. This enables us to draw the processor diagrams in a similar way, to bring out the similarities and allow us to concentrate on the differences:



© **BORES** Signal Processing

The diagram shows a generalised DSP processor, with the basic features that are common.

These features can be seen in the diagram for one of the earliest DSP processors - the Lucent DSP32C:

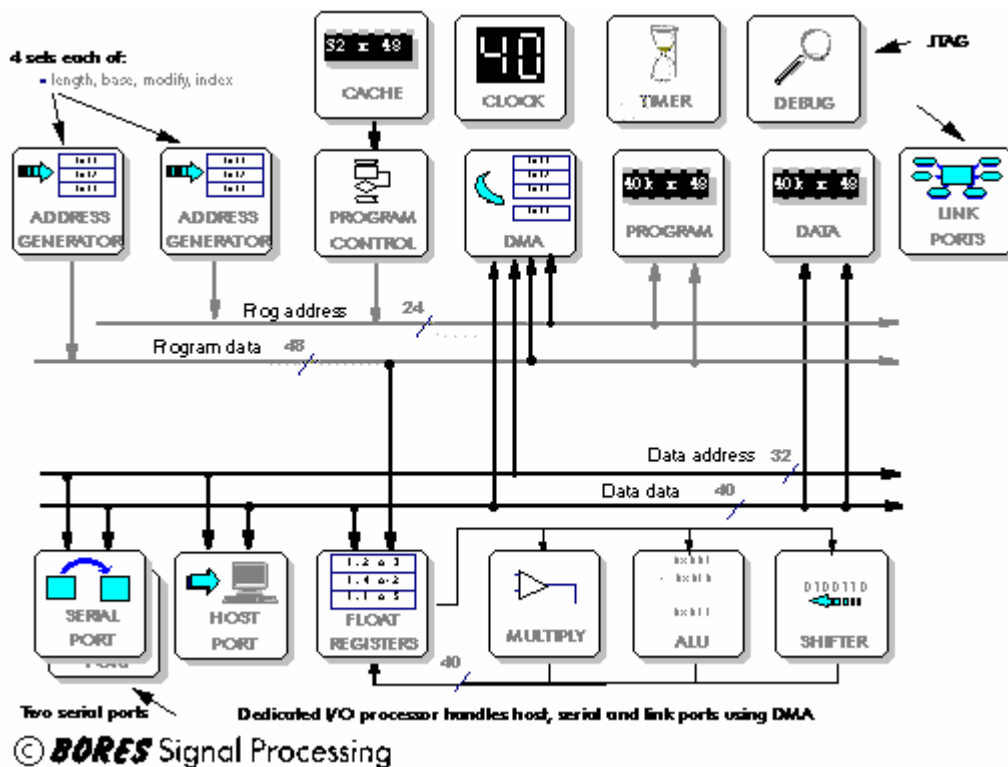


© **BORES** Signal Processing

The Lucent DSP32C has four memory areas (three internal plus one external), and uses a modified von Neuman architecture to achieve four memory accesses per instruction cycle - the

von Neuman architecture is shown by the presence of only a single memory bus. It has four floating point registers: the address generation registers also double as general purpose fixed point registers. The Lucent DSP32C has a host port: showing that this chip is designed to be integrated into systems with another system controller - in this case, a microcontroller or PC (ISA) bus.

Looking at one of the more recent DSP processors - the Analog Devices ADSP21060 - shows how similar are the basic architectures:



The ADSP21060 has a Harvard architecture - shown by the two memory buses. This is extended by a cache, making it a Super Harvard ARCHitecture (SHARC). Note, however, that the Harvard architecture is not fully brought off chip - there is a special bus switch arrangement which is not shown on the diagram. The 21060 has two serial ports in place of the Lucent DSP32C's one. Its host port implements a PCI bus rather than the older ISA bus. Apart from this, the 21060 introduces four features not found on the Lucent DSP32C:

- There are two sets of address generation registers. DSP processors commonly have to react to interrupts quickly - the two sets of address generation registers allow for swapping between register sets when an interrupt occurs, instead of having to save and restore the complete set of registers.
- There are six link ports, used to connect with up to six other 21060 processors: showing that this processor is intended for use in multiprocessor designs.
- There is a timer - useful to implement DSP multitasking operating system features using time slicing.
- There is a debug port - allowing direct non-intrusive debugging of the processor internals.

DSP processors: Data formats

DSP processors store data in fixed or floating point formats.

It is worth noting that fixed point format is not quite the same as integer:

integer

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

$= 2^6 + 2^4 + 2^1 + 2^0 = 64 + 16 + 2 + 1 = 83$

$- 2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$

fixed point

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

$= 2^{-1} + 2^{-3} = 0.5 + 0.125 = 0.625$

$- 2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad 2^{-5} \quad 2^{-6} \quad 2^{-7}$

Radix point is assumed in fixed point format

© **BORES** Signal Processing

The integer format is straightforward: representing whole numbers from 0 up to the largest whole number that can be represented with the available number of bits. Fixed point format is used to represent numbers that lie between 0 and 1: with a 'binary point' assumed to lie just after the most significant bit. The most significant bit in both cases carries the sign of the number.

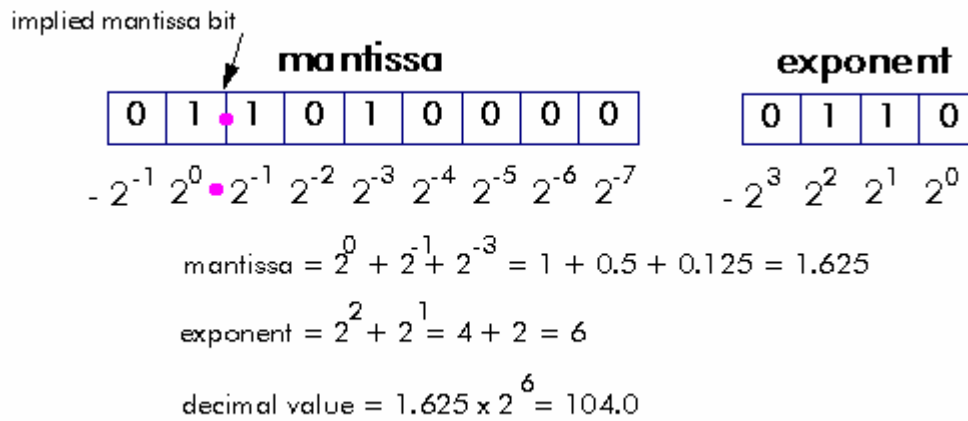
- The size of the fraction represented by the smallest bit is the precision of the fixed point format.
- The size of the largest number that can be represented in the available word length is the dynamic range of the fixed point format

To make the best use of the full available word length in the fixed point format, the programmer has to make some decisions:

- If a fixed point number becomes too large for the available word length, the programmer has to scale the number down, by shifting it to the right: in the process lower bits may drop off the end and be lost
- If a fixed point number is small, the number of bits actually used to represent it is small. The programmer may decide to scale the number up, in order to use more of the available word length

In both cases the programmer has to keep a track of by how much the binary point has been shifted, in order to restore all numbers to the same scale at some later stage.

Floating point format has the remarkable property of automatically scaling all numbers by moving, and keeping track of, the binary point so that all numbers use the full word length available but never overflow:



© **BORES** Signal Processing

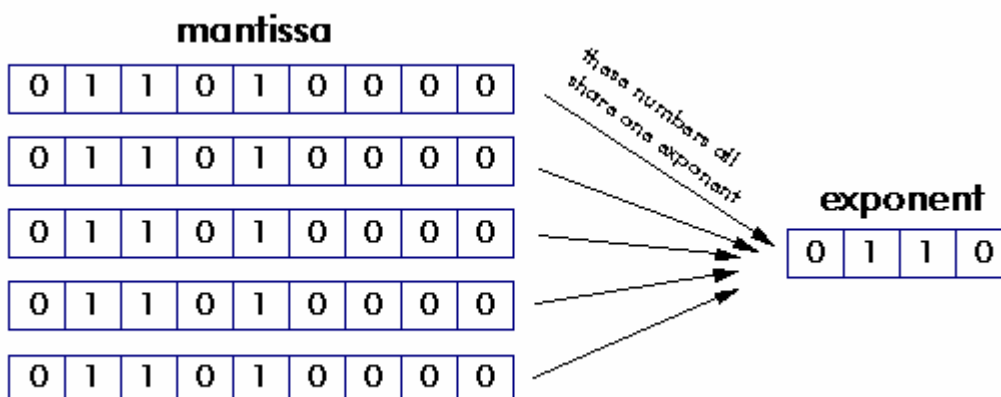
Floating point numbers have two parts: the mantissa, which is similar to the fixed point part of the number, and an exponent which is used to keep track of how the binary point is shifted. Every number is scaled by the floating point hardware:

- If a number becomes too large for the available word length, the hardware automatically scales it down, by shifting it to the right
- If a number is small, the hardware automatically scale it up, in order to use the full available word length of the mantissa

In both cases the exponent is used to count how many times the number has been shifted.

In floating point numbers the binary point comes after the second most significant bit in the mantissa.

The block floating point format provides some of the benefits of floating point, but by scaling blocks of numbers rather than each individual number:



© **BORES** Signal Processing

Block floating point numbers are actually represented by the full word length of a fixed point format.

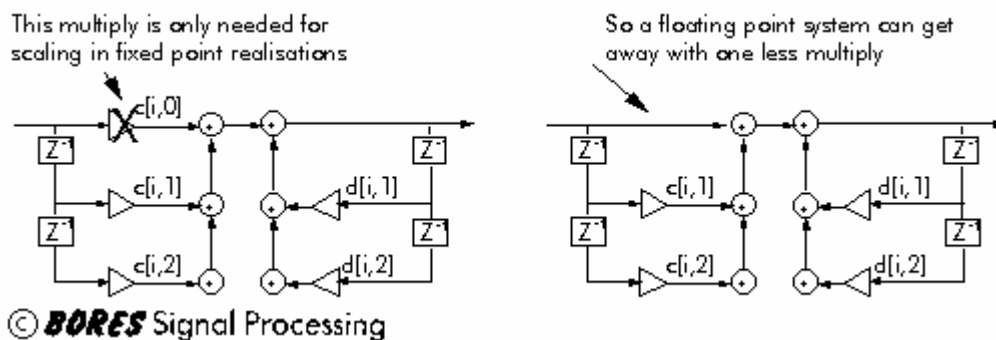
- If any one of a block of numbers becomes too large for the available word length, the programmer scales down all the numbers in the block, by shifting them to the right

- If the largest of a block of numbers is small, the programmer scales up all numbers in the block, in order to use the full available word length of the mantissa

In both cases the exponent is used to count how many times the numbers in the block have been shifted.

Some specialised processors, such as those from Zilog, have special features to support the use of block floating point format: more usually, it is up to the programmer to test each block of numbers and carry out the necessary scaling.

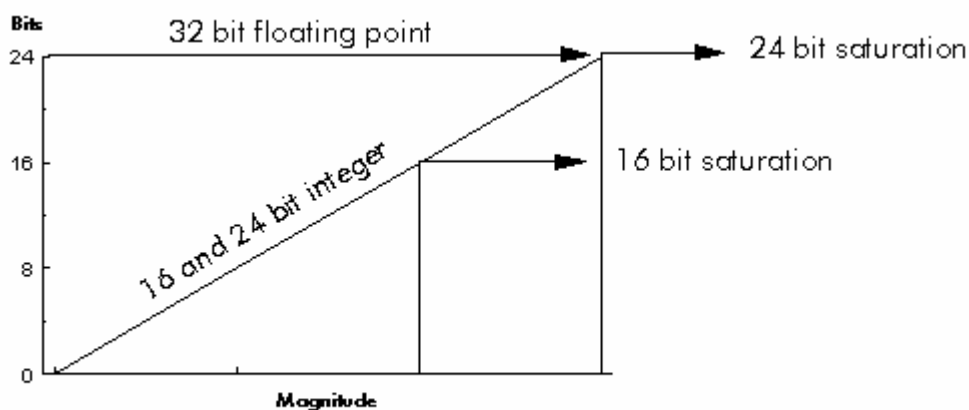
The floating point format has one further advantage over fixed point: it is faster. Because of quantisation error, a basic direct form 1 IIR filter second order section requires an extra multiplier, to scale numbers and avoid overflow. But the floating point hardware automatically scales every number to avoid overflow, so this extra multiplier is not required:



DSP processors: Precision and dynamic range

The precision with which numbers can be represented is determined by the word length in the fixed point format, and by the number of bits in the mantissa in the floating point format.

In a 32 bit DSP processor the mantissa is usually 24 bits: so the precision of a floating point DSP is the same as that of a 24 bit fixed point processor. But floating point has one further advantage over fixed point: because the hardware automatically scales each number to use the full word length of the mantissa, the full precision is maintained even for small numbers:



© **BORES** Signal Processing

There is a potential disadvantage to the way floating point works. Because the hardware automatically scales and normalises every number, the errors due to truncation and rounding depend on the size of the number. If we regard these errors as a source of quantisation noise, then the noise floor is modulated by the size of the signal. Although the modulation can be shown to be always downwards (that is, a 32 bit floating point format always has noise which is less than that of a 24 bit fixed point format), the signal dependent modulation of the noise may be undesirable: notably, the audio industry prefers to use 24 bit fixed point DSP processors over floating point because it is thought by some that the floating point noise floor modulation is audible.

The precision directly affects quantisation error.

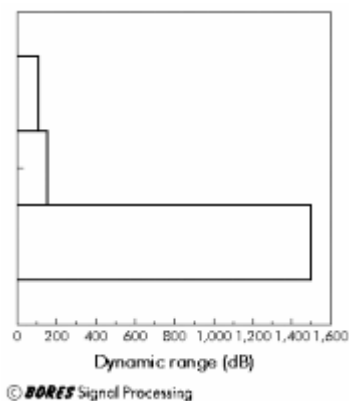
The largest number which can be represented determines the dynamic range of the data format. In fixed point format this is straightforward: the dynamic range is the range of numbers that can be represented in the available word length. For floating point format, though, the binary point is moved automatically to accommodate larger numbers: so the dynamic range is determined by the size of the exponent. For an 8 bit exponent, the dynamic range is close to 1,500 dB:

$$1 \times 2^{-127} \rightarrow 2 \times 2^{127}$$

or:

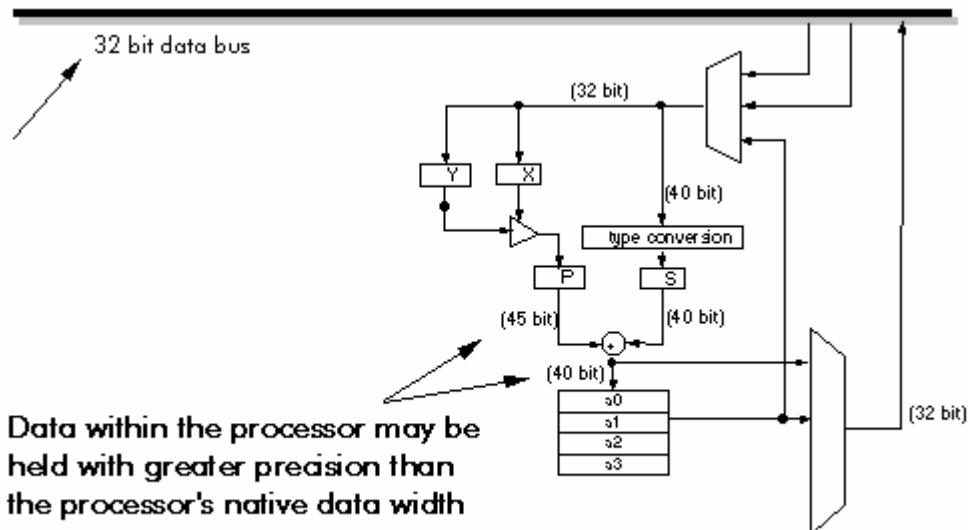
$$5.9 \times 10^{-39} \rightarrow 3.4 \times 10^{38}$$

So the dynamic range of a floating point format is enormously larger than for a fixed point format:



While the dynamic range of a 32 bit floating point format is large, it is not infinite: so it is possible to suffer overflow and underflow even with a 32 bit floating point format. A classic example of this can be seen by running fractal (Mandelbrot) calculations on a 32 bit DSP processor: after quite a long time, the fractal pattern ceases to change because the increment size has become too small for a 32 bit floating point format to represent.

Most DSP processors have extended precision registers within the processor:



© **BORES** Signal Processing

The diagram shows the data path of the Lucent DSP32C processor. Although this is a 32 bit floating point processor, it uses 40 and 45 bit registers internally: so results can be held to a wider dynamic range internally than when written to memory.

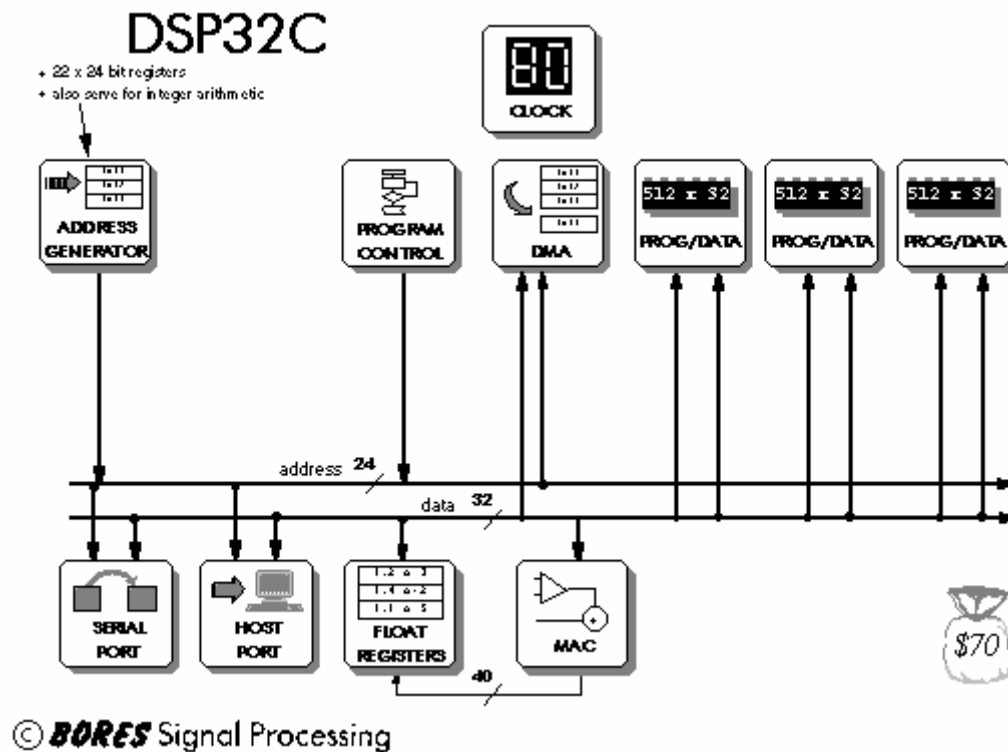
DSP processors: Review of DSP Processors

Although there are many DSP processors, they are mostly designed with the same few basic operations in mind: so they share the same set of basic characteristics. We can learn a lot by considering how each processor differs from its competitors, and so gaining an understanding of how to evaluate one processor against others for particular applications.

A simple processor design like the Lucent DSP32C shows the basic features of a DSP processor:

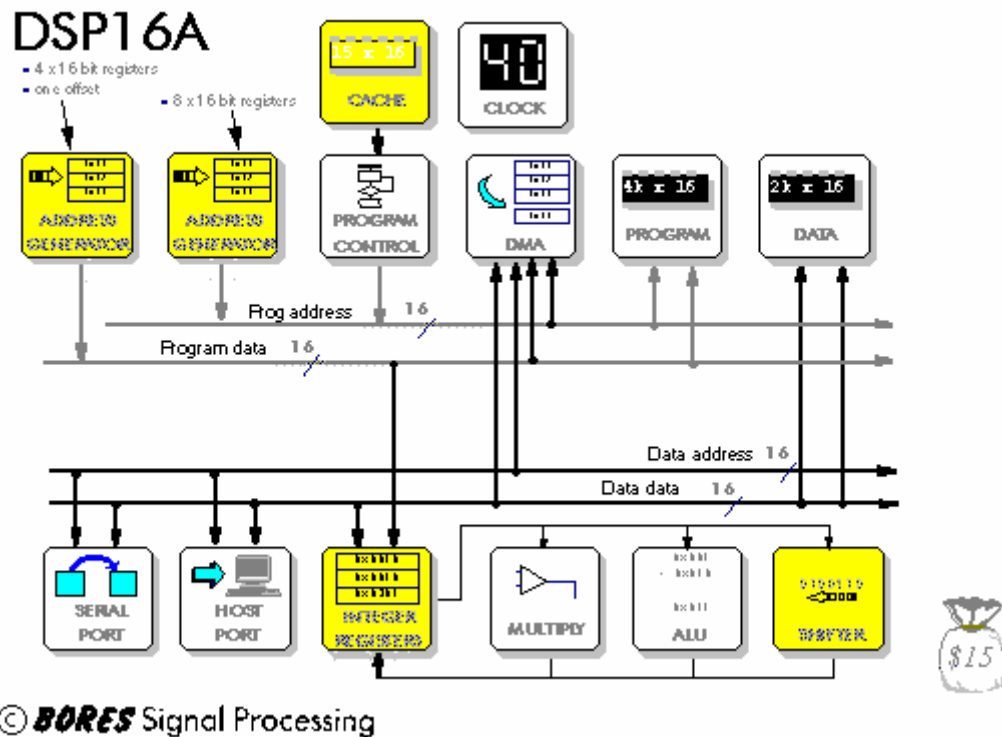
- multiple on-chip memories
- external memory bus
- hardware add and multiply in parallel
- lots of registers
- serial interface
- host interface

The DSP32C is unusual in having a true von Neuman architecture: rather than use multiple buses to allow multiple memory accesses, it handles up to four sequential memory accesses per cycle. The DMA controller handles serial I/O, independently in and out, using cycle stealing which does not disturb the DSP execution thread.



The simple DSP32C design uses the address registers to hold integer data: and there is no hardware integer multiplier: astonishingly, integers have to be converted to floating point format, then back again, for multiplication. We can excuse this lack of fast integer support by recalling that this was one of the first DSP processors, and it was designed specifically for floating point, not fixed point, operation: the address registers are for address calculations, with integer operations being only a bonus.

For a fixed point DSP, the address generation needs to be separated from the integer data registers: this may also be efficient for a floating point DSP if integer calculations are needed very often. Lucent's more modern fixed point DSP16A processor shows the separation of fixed point from address registers:



The DSP16A also shows a more conventional use of multiple internal buses (Harvard plus cache) to access two memory operands (plus an instruction) . A further arithmetic unit (shifter) has been added.

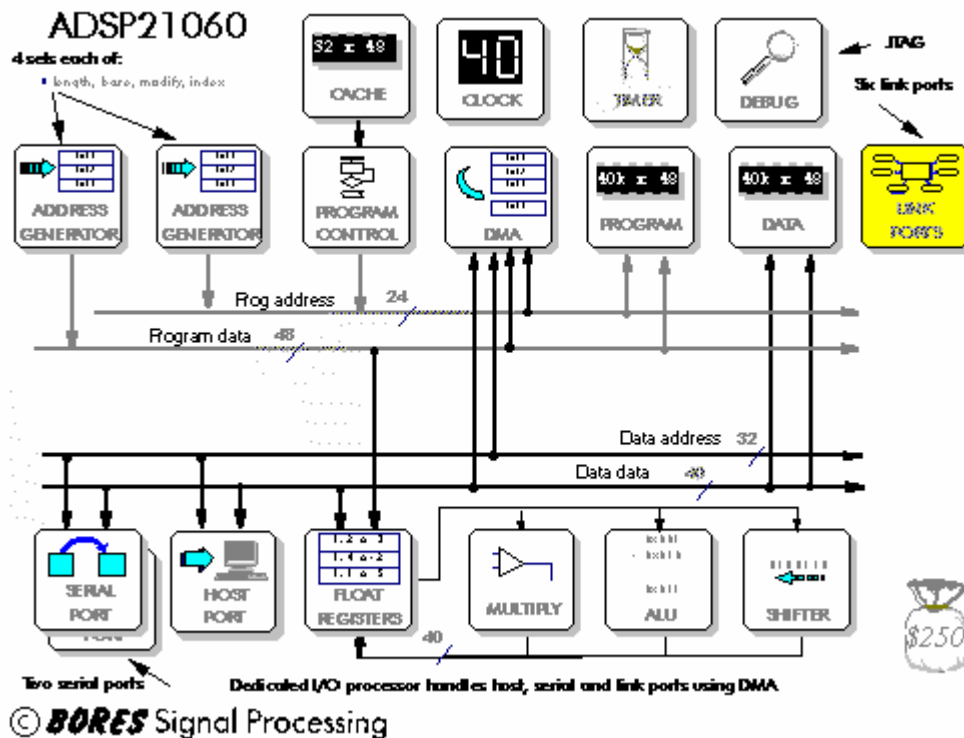
DSP processors: Review of DSP processors

DSP often involves a need to switch rapidly between one task and another: for example, on the occurrence of an interrupt. This would usually require all registers currently in use to be saved, and then restored after servicing the interrupt. The DSP16A and the Analogue Devices ADSP2181 use two sets of address generation registers:

- 4 sets each of:
 - + length, base, modify, index



It is interesting to see how far a manufacturer carries the same basic processor model into their different designs. Texas Instruments, Analog Devices and Motorola all started with fixed point devices, and have carried forward those designs into their floating point processors. AT&T (now Lucent) started with floating point, then brought out fixed point devices later. The Analog Devices ADSP21060 looks like a floating point version of the integer ADSP2181:

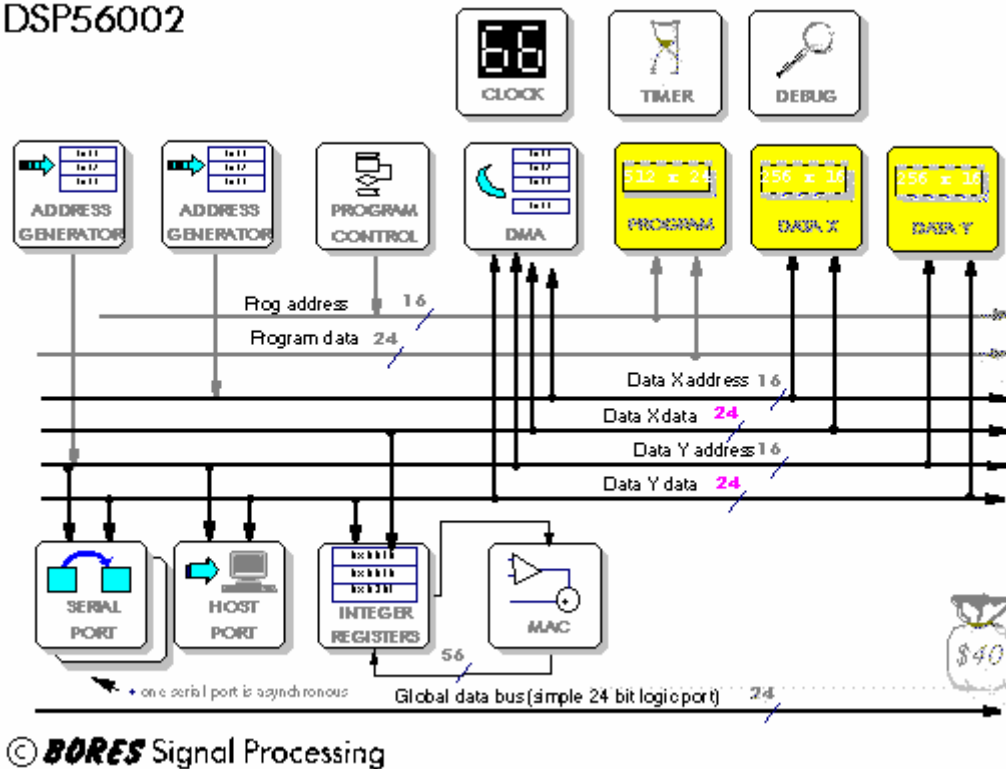


The 21060 also has six high speed link ports which allow it to connect with up to six other processors of the same type. One way to support multiprocessing is to have many fast inter-processor communications ports: another is to have shared memory. The ADSP21060 supports both methods. Shared memory is supported in a very clever way: each processor can directly access a small area of the internal memory of up to four other processors. As with the ADSP2181, the 21060 has lots of internal memory: the idea being, that most applications can work without added external memory: note, though, that the full Harvard architecture is not brought off chip, which means they really need the on-chip memory to be big enough for most applications.

DSP processors: Review of DSP processors

The problem of fixed point processors is quantisation error, caused by the limited fixed point precision. Motorola reduce this problem in the DSP56002 by using a 24 bit integer word length:

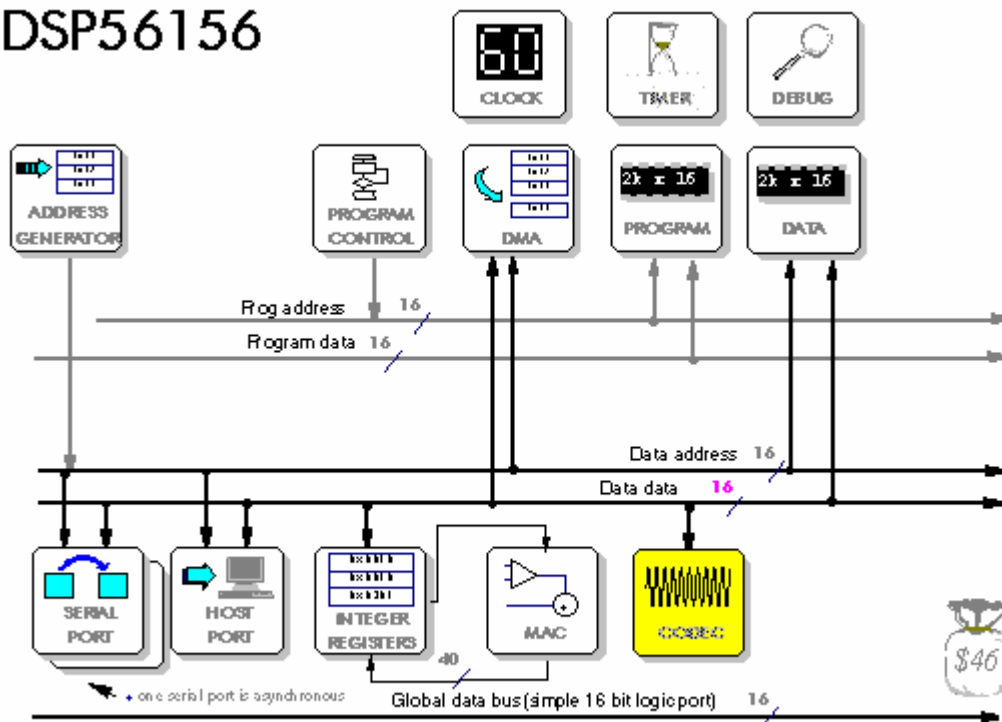
DSP56002



They also use three internal buses - one for program, two for data (two operands). This is an extension of the standard Harvard architecture which goes beyond the usual trick of simply adding a cache, to allow access to two operands and the instruction at the same time.

Of course, the problem of 24 bit fixed point is its expense: which probably explains why Motorola later produced the cheap, 16 bit DSP56156 - although this looks like a 16 bit variant of the DSP56002:

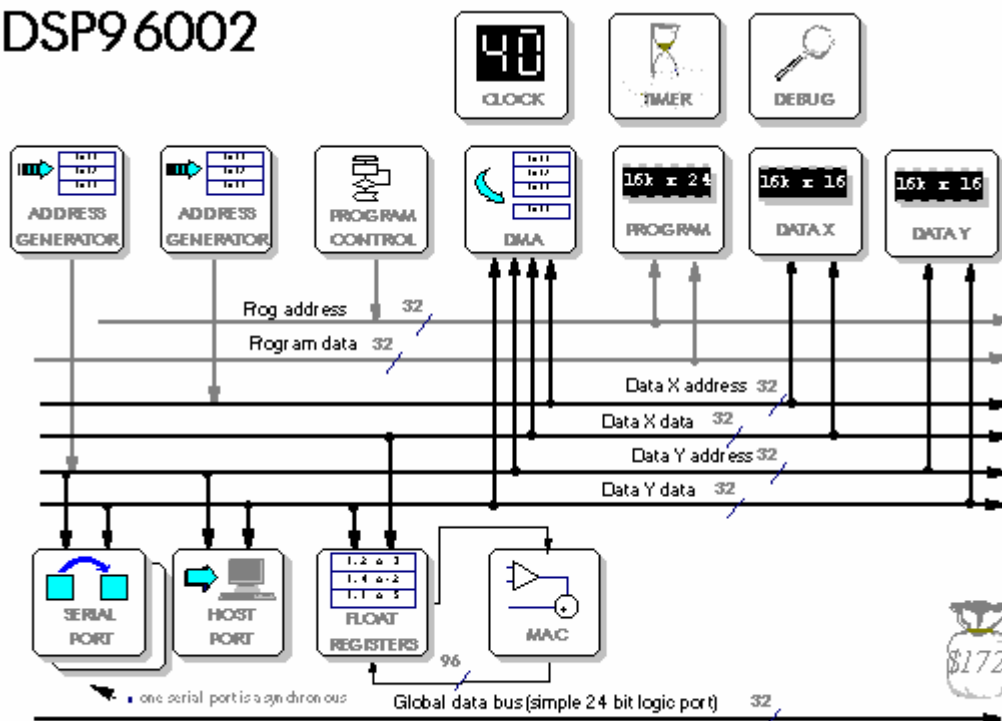
DSP56156



© **BORES** Signal Processing

And of course there has to be a floating point variant - the DSP96002 looks like a floating point version of the DSP56002:

DSP96002

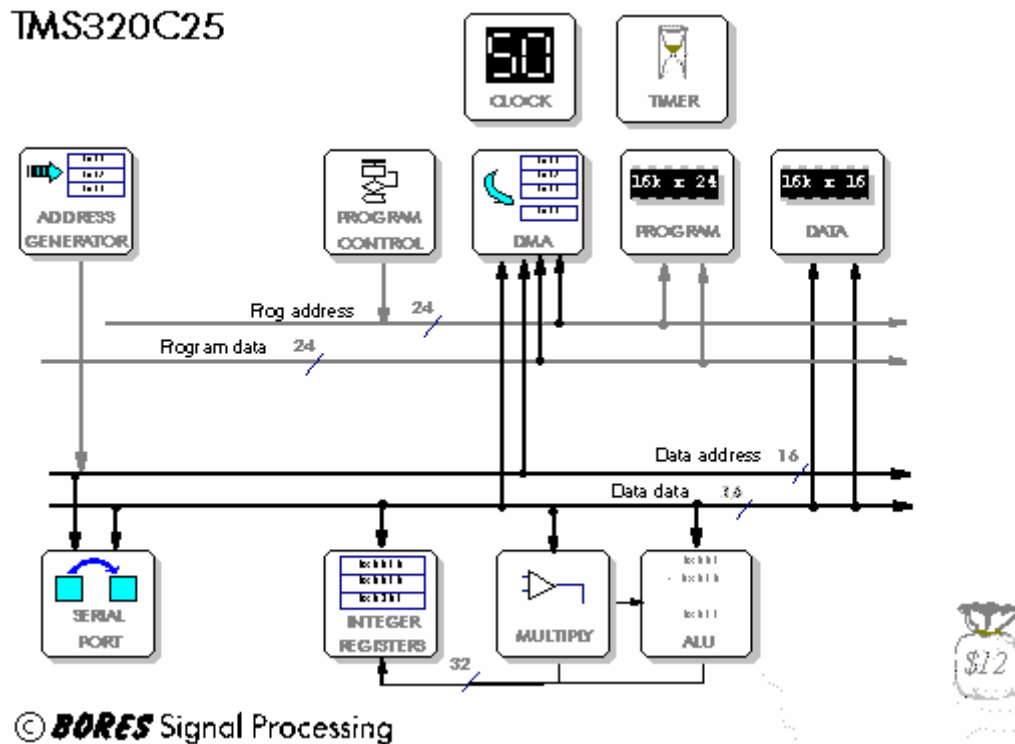


© **BORES** Signal Processing

The DSP96002 supports multiprocessing with an additional 'global bus' which can connect to other DSP96002 processors: it also has a new DMA controller with its own bus

DSP processors: Review of DSP processors

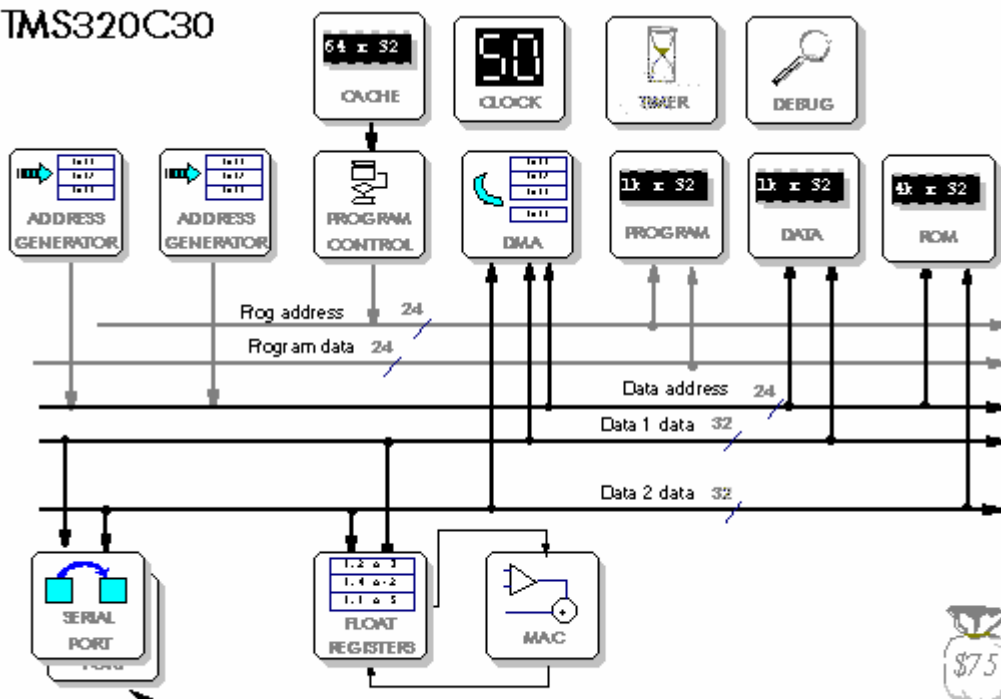
The Texas TMS320C25 is quite an early design. It does not have a parallel multiply/add: the multiply is done in one cycle, the add in the next and the DSP has to address the data for both operations. It has a modified Harvard bus with only one data bus, which sometimes restricts data memory accesses to one per cycle, but it does have a special 'repeat' instruction to repeat an instruction without writing code loops



The Texas TMS320C50 is the C25 brought up to date: the multiply/add can now achieve single cycle execution if it is done in a hardware repeat loop. It also uses shadow registers as a fast way to preserve registers when context switching. It has automatic saturation or rounding (but it needs it, since the accumulator has no guard bits to prevent overflow), and it has parallel bit manipulation which is useful in control applications



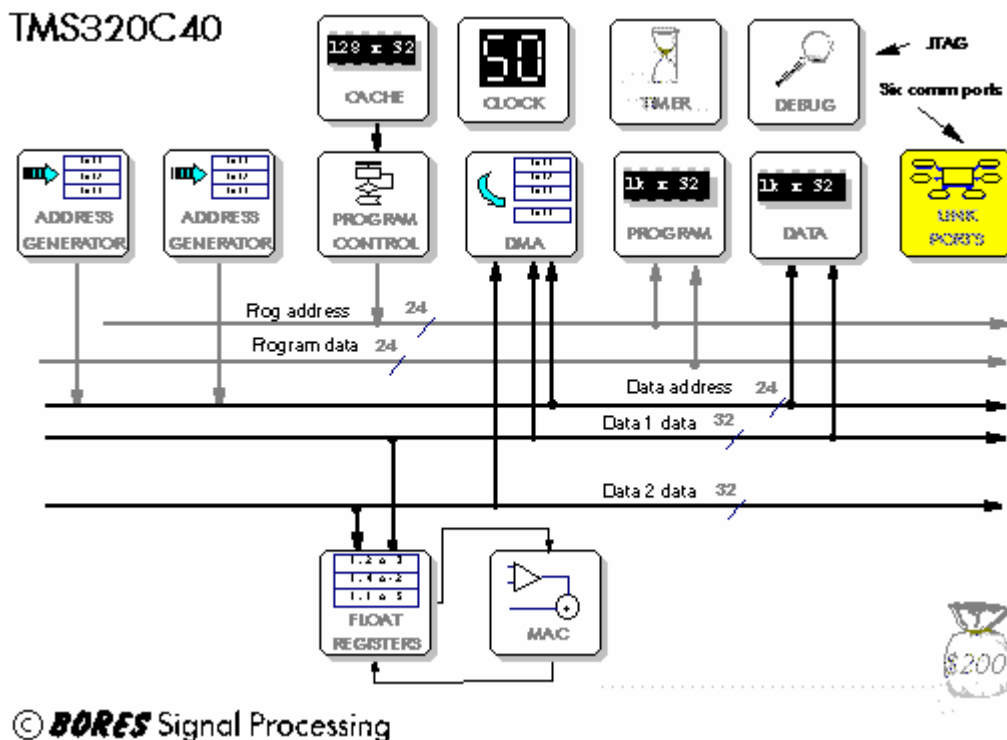
TMS320C30



© **BORES** Signal Processing



The Texas TMS320C40 is similar to the C30, but with high speed communications ports for multiprocessing. It has six high speed parallel comm ports which connect with other C40 processors: these are 8 bits wide, but carry 32 bit data in four successive cycles.

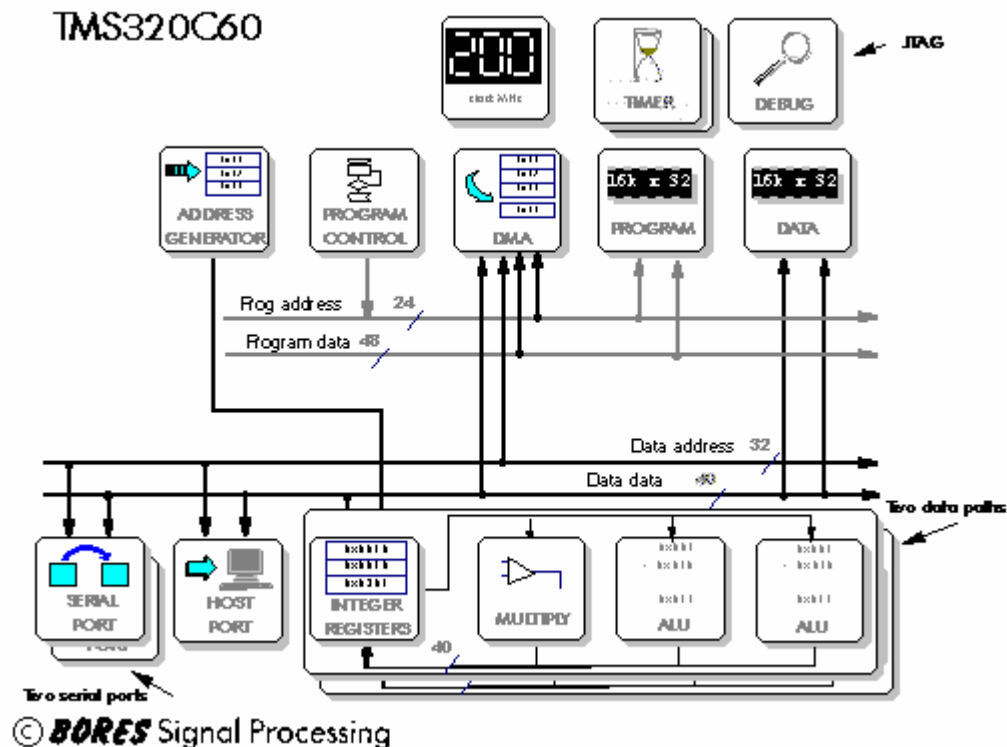


DSP processors: Review of DSP processors

The Texas TMS320C60 is radically different from other DSP processors, in using a Very Long Instruction Word (VLIW) format. It issues a 256 bit instruction, containing up to 8 separate 'mini instructions' to each of 8 functional units.

Because of the radically different concept, it can be hard to compare this processor with other, more traditional, DSP processors. But despite this, the 'C60 can still be viewed in a similar way to other DSP processors, which makes apparent the fact that it basically has two data paths each capable of a multiply/accumulate.

TMS320C60



Note that this diagram is very different from the way Texas Instruments draw it. This is for several reasons:

- Texas Instruments tend to draw their processors as a set of subsystems, each with a separate block diagram
- my diagram does not show some of the more esoteric features such as the bus switching arrangement to address the multiple external memory accesses that are required to load the eight 'mini-instructions'

An important point is raised by the placing of the address generation unit on the diagram. Texas Instruments draw the C60 block diagram as having four arithmetic units in each data path - whereas my diagram shows only three. The fourth unit is in fact the address generation calculation. Following my practice for all other DSP processors, I show address generation as separate from the arithmetic units - address calculation being assumed by the presence of address registers as is the case in all DSP processors. The C60 can in fact choose to use the address generation unit for general purpose calculations if it is not calculating addresses - this is similar, for example, to the Lucent DSP32C: so Texas Instruments' approach is also valid - but for most classic DSP operations address generation would be required and so the unit would not be available for general purpose use.

There is an interesting side effect of this. Texas Instruments rate the C60 as a 1600 MIPS device - on the basis that it runs at 200 MHz, and has two data paths each with four execution units: $200 \text{ MHz} \times 2 \times 4 = 1600 \text{ MIPS}$. But from my diagram, treating the address generation separately, we see only three execution units per data path: $200 \text{ MHz} \times 2 \times 3 = 1200 \text{ MIPS}$. The latter figure is that actually achieved in quoted benchmarks for an FIR filter, and reflects the device's ability to perform arithmetic.

This illustrates a problem in evaluating DSP processors. It is very hard to compare like with like - not least, because all manufacturers present their designs in such a way that they show their best performance. The lesson to draw is that one cannot rely on MIPS, MOPS or Mflops ratings but must carefully try to understand the features of each candidate processor and how they differ from each other - then make a choice based on the best match to the particular application. It is very important to note that a DSP processor's specialised design means it will achieve any quoted MIPS, MOPS or Mflops rating only if programmed to take advantage of all the parallel features it offers.

Programming a DSP processor

This is the seventh module of the BORES Signal Processing DSP course - Introduction to DSP.

This module shows how a DSP processor can be programmed efficiently. It uses a simple filter implementation in C language to show the basic techniques, followed by a transcription of the C code into assembly language. Some of the C language techniques are applicable to any processor. The aim of this module is to show how simple rules can lead to efficient DSP code. It covers the following subjects:

- a simple FIR filter program
- using pointers
- avoiding memory bottlenecks
- assembler programming
- extra requirements for real time filters
- MIPS, MOPS and Mflops

Programming a DSP processor: A simple FIR filter

The simple FIR filter equation:

$$y[n] = \sum c[k] * x[n-k]$$

can be implemented quite directly in C language:

```
y[n] = 0.0;
for (k = 0; k < N; k++)
y[n] = y[n] + c[k] * x[n-k];
```

But this naive code is inefficient:

```

y[n] = 0.0;
for (k = 0; k < N; k++)
    y[n] = y[n] + c[k] * x[n-k];

```

this element is
accessed repeatedly

accessing by array
index is inefficient

arithmetic is needed to
calculate this array index

© **BORES** Signal Processing

The code is inefficient because:

- it uses array indices [i] rather than pointers *ptr
- it needs a lot of address arithmetic
- it repeatedly accesses the output array

Programming a DSP processor: Using pointers

A naive C language program to implement an FIR filter is inefficient because it accesses array elements by array index:

```

y[n] = 0.0;
for (k = 0; k < N; k++)
    y[n] = y[n] + c[k] * x[n-k];

```

To understand why accessing by array index is inefficient, remember that an array is really just a table of numbers in sequential memory locations. The C compiler only know the start address of the array. To actually read any array element the compiler first has to find the address of that particular element. So whenever an array element is accessed by its array index [i] the compiler has to make a calculation:

```

y[n] = 0.0;
for (k = 0; k < N; k++)
    y[n] = y[n] + c[k] * x[n-k];

```

address	value
&x	0.0
&x + 1	1.0
...	...
&x + (N-1)	1.2

this array is really a table in memory

to access this array element we have to:

- load the start address of the table &x
- load the value of n n
- load the value of k k
- calculate the offset n-k n-k
- add the offset to the start address &x + (n-k)
- read the element data from this address

four operations
– three loads and a calculation
before we can get the data

© **BORES** Signal Processing

The diagram shows how the compiler would calculate the address of an array element specified by index as $x[n - k]$. The calculation requires several steps:

- load the start address of the table in memory
- load the value of the index n
- load the value of the index k
- calculate the offset $[n - k]$
- add the offset to the start address of the array

This entails five operations: three reads from memory, and two arithmetic operations. Only after all five operations can the compiler actually read the array element.

C language provides the 'pointer' type precisely to avoid the inefficiencies of accessing array elements by index.

In C, the syntax `*ptr` indicates that `ptr` is a pointer which means:

- the variable `ptr` is to be treated as containing an address
- the '*' means the data is read from that address

Pointers can be modified after the data has been accessed. The syntax `*ptr++` means:

- the variable `ptr` is to be treated as containing an address
- the '*' means the data is read from that address
- the '++' means that, having read the data, the pointer `ptr` is incremented to point to the next sequential data element

Accessing the array elements using pointers is more efficient than by index:

```

float  *y_ptr, *c_ptr, *x_ptr;

y_ptr = &y[0];

for (k = 0; k < N; k++)
    *y_ptr = *y_ptr + *c_ptr++ * *x_ptr--;
  
```

y_ptr is treated as an address (pointer)

we still have to calculate the address once

but then we just increment the pointer

- because we are accessing successive elements

data is read from the address pointed to by the pointer

© **BORES** Signal Processing

Each pointer still has to be initialised: but only once, before the loop; and only to the start address of the array, so not requiring any arithmetic to calculate offsets. Within the loop, the pointers are simply incremented so that they point automatically to the next array element ready for the next pass through the loop.

Using pointers is more efficient than array indices on any processor: but it is especially efficient for DSP processors because DSP processors are excellent at address arithmetic. In fact, address increments often come for free. For example, the Lucent DSP32C processor has several 'free' modes of pointer address generation:

*rP - register indirect : read the data pointed to by the address in register rP

*rP++ - postincrement: having read the data, postincrement the address pointer to point to the next value in the array

*rP++rI - register postincrement: having read the data, postincrement the address pointer *by the amount held in register rI* to point to *rI* values further down the array

*rP++rIr - bit reversed: having read the data, postincrement the address pointer to point to the next value in the array, *as if the address bits were in bit reversed order*

The address increments are performed in the same instruction as the data access to which they refer: and they incur no overhead at all. More than this, as we shall see later, most DSP processors can perform two or three address increments for free in each instruction. So the use of pointers is crucially important for DSP processors.

Some C compilers optimise code. For example, one of the Texas Instruments C compilers would, with full optimisation selected, take the initial naive C code but produce assembler that corresponds closely to the code using pointers. This is very nice but there are three cautions to be observed:

- optimisation can often be used only in restrictive circumstances - for example in the absence of interrupts
- optimisation is compiler dependent: so code that relies on compiler optimisation could become very inefficient when ported to another compiler

One reason to use C is so that the programmer can write code that is very close to the operation of the processor. This is often desirable in DSP, where we want to have a high degree of control over exactly what the processor is doing at all times. Optimisation changes the code you wrote into code the compiler thought was better: in the worst case the code may not actually work when optimised.

Programming a DSP processor: Limiting memory accesses

Memory accesses are bottlenecks.

```
for (k = 0; k < N; k++)  
    *y_ptr = *y_ptr + *c_ptr++ * *x_ptr--;
```

one memory write, plus three memory reads

DSP processors can make multiple memory accesses in a single instruction cycle. But the inner loop of the FIR filter program requires four memory accesses: three reads for each of the operands, and one write of the result to memory. Even without counting the need to load the instruction, this exceeds the capacity of a DSP processor. For instance the Lucent DSP32C can make four memory accesses per instruction cycle: two reads of operands, plus one write of the result, plus the read of one instruction. Even this is not enough for the simple line of C code that forms the inner loop of the FIR filter program.

Fortunately, DSP processors have lots of registers which can be used to hold values inside the processor for later use - thus economising on memory accesses. We can see that the result of the inner loop is used again and again during the loop: it as the code is written, it has to be read from memory and then written back to memory in each pass. Making this a register variable will allow it to be held within the processor, thus saving two memory accesses:

```
register float temp;
temp = 0.0;
for (k = 0; k < N; k++)
temp = temp + *c_ptr++ * *x_ptr--;
```

The C declaration 'register float temp' means that variable temp is to be held in a processor register: in this case, a floating point register. The inner loop now only requires two memory accesses, to read the two operands *c_ptr and *x_ptr (three accesses if you count the instruction load) - this is now within the capabilities of the DSP processor in a single instruction.

A small point to note is that the initialisation of the register variable `temp=0.0` is wasted. It is simple to make use of this necessary initialisation to make the first calculation, thus reducing the number of iterations of the inner loop:

```
register float temp;
temp = *c_ptr++ * *x_ptr--;
for (k = 1; k < N; k++)
temp = temp + *c_ptr++ * *x_ptr--;
```

This leads to a more efficient C program for the FIR filter:

```
float y[N], c[N], x[N];
float *y_ptr, *c_ptr, *x_ptr;
register float temp;
int n, k;
y_ptr = &y[0];
for (n = 0; n < N-1; n++) {
c_ptr = &c[0]; x_ptr = &x[N-1];
temp = *c_ptr++ * *x_ptr--;
for (k = 1; k < N; k++)
temp = temp + *c_ptr++ * *x_ptr--;
*y_ptr++ = temp;
}
}
```

Programming a DSP processor: Assembler

To illustrate transcribing the C program for the FIR filter into DSP assembly language, we will use the assembler syntax of the Lucent DSP32C processor. This processor is excellent for this

purpose, because its assembler syntax is remarkably similar to C language and so makes it easy to see how the C code maps onto the underlying DSP architecture. It is important to note that the illustration remains valid in general for most DSP processors, since their basic design features are so similar: but the other processors have more impenetrable assembler syntax.

*r3 is equivalent to the C syntax `*c_ptr`

*r3++ is equivalent to the C syntax `*c_ptr++`

a1 is equivalent to the C declaration `float temp`

Some examples of simple DSP32C instructions show the similarity to C further:

`a1=*r3`

fetch a floating point value from memory pointed to by address register r2 and store it in the float register a1

`a1=*r3++`

fetch a floating point value from memory pointed to by address register r3 and store it in the float register a1: having done so, increment address register r3 to point to the next floating point value in memory

The general DSP32C instruction syntax shows the typical DSP processor's ability to perform a multiplication and addition in a single instruction:

`a = b + c * d`

Each term in the instruction can be any of the four floating point registers, or up to three of the terms can access data through address registers used as pointers:

`a0=a1 + a2 * a3` - using only registers

`a0=a1 + *r2 * *r3` - using pointers for two memory reads

`a1=a1 + *r2++ * *r3++` - using pointers for memory reads and incrementing those pointers

Armed with the above rudimentary knowledge of this DSP processor's assembler syntax, we can substitute assembler variables for the C variables:

`temp: - a1 (floating point register)`

`y_ptr: - r2 (address register to be used as a pointer)`

`c_ptr: - r3 (address register to be used as a pointer)`

`x_ptr: - r4 (address register to be used as a pointer)`

The appropriate assembler can now be written underneath the C code, exploiting the great similarity of the assembler to C in this case:

```
temp = *c_ptr++) * *x_ptr--);
a1    = *r3++ * *r4--
for (k = 1; k < N-1; k++)
```

```

do 0,r1
temp = temp +      *c_ptr++ * *x_ptr--
a1    = a1 +      *r3++ *   *r4--
*y_ptr++ = temp
*r2++    = a1

```

Note that for this processor, one line of C compiles down to one assembler instruction.

The 'do 0,r1' instruction is an efficient and concise way to replace the loop control: it means, "do the next (0+1) instructions (r1+1) times. This is an example of a 'zero overhead do loop': the processor supports this special instruction with no overhead at all for the actual execution of the loop control.

Programming a DSP processor: Real time

Both the naive FIR filter program and its more efficient version assume we can access the whole array of past input values repeatedly:

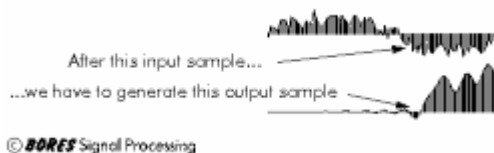
```

y[n] = 0.0;
for (k = 0; k < N; k++)
    y[n] = y[n] + c[k] * x[n-k];

```

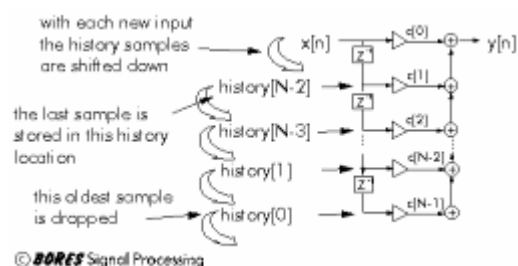
this assumes we can access
the input data repeatedly

But this is not the case in real time. Real time systems face a continuing stream of input data: often, they have to operate on one input sample at a time and generate one output sample for each input sample:



A similar restriction is likely if the filter program is implemented as a subroutine or function call. Only the current input and output are available to the filter so the filter function itself has to maintain some history of the data and update this history with each new input sample. Management of the history takes up some processing time.

The filter needs to know the most recent [N] input samples. So the real time filter has to maintain a history array, which is updated with each new input sample by shifting all the history data one location toward 0:



The necessary updating of the history array involves simply adding two extra lines to the C program, to implement the array shifting:

history array instead of data array
↓

```

output = (*hist_ptr++) * (*coef_ptr++);

for (k = 2; k < N; k++) {
    output = output + (*hist_ptr) * (*coef_ptr++);
    *hist1_ptr++ = *hist_ptr++;
}
output = output + input * (*coef_ptr);
*hist1_ptr = input;
  
```

shift the history array

include the last input

put the last input in the history array

© **BORES** Signal Processing

The pointer to previous input samples, `*x_ptr`, is replaced by a pointer to the history array, `*hist_ptr`. A new pointer, `*hist1_ptr`, is initialised to point one further down the history array and is used in the shifting of data down the array.

The two extra lines of C code represent extra computation: actually, the filter now takes two lines of C code instead of one for the inner loop.

Programming a DSP processor: MIPS, MOPS and Mflops

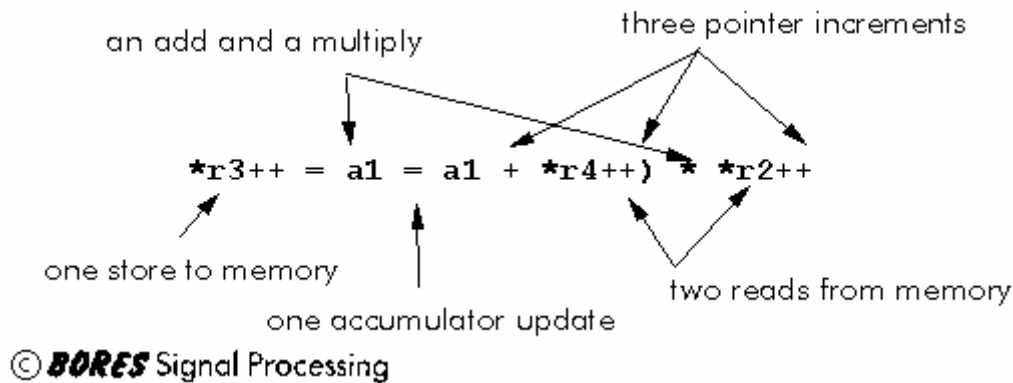
The development of efficient assembly language code shows how efficient a DSP processor can be: each assembler instruction is performing several useful operations. But it also shows how difficult it can be to program such a specialised processor efficiently.

```

temp = *c_ptr++ * *x_ptr--;
a1 = *r3++ * *r4--
for (k = 1; k < N-1; k++)
do 0,r1
temp = temp + *c_ptr++ * *x_ptr--
a1 = a1 + *r3++ * *r4--
*y_ptr++ = temp
*r2++ = a1
  
```

Bear in mind that we use DSP processors to do specialised jobs fast. If cost is no object, then it may be permissible to throw away processor power by inefficient coding: but in that case we would perhaps be better advised to choose an easier processor to program in the first place. A sensible reason to use a DSP processor is to perform DSP either at lowest cost, or at highest speed. In either case, wasting processor power leads to a need for more hardware which makes a more expensive system which leads to a more expensive final product which, in a sane world, would lead to loss of sales to a competitive product that was better designed.

One example shows how essential it is to make sure a DSP processor is programmed efficiently:



The diagram shows a single assembler instruction from the Lucent DSP32C processor. This instruction does a lot of things at once:

- two arithmetic operations (an add and a multiply)
- three memory accesses (two reads and a write)
- one floating point register update
- three address pointer increments

All of these operations can be done in one instruction. This is how the processor can be made fast. But if we don't use any of these operations, we are throwing away the potential of the processor and may be slowing it down drastically. Consider how this instruction can be translated into MIPS or Mflops.

The processor runs with an 80 MHz clock. But, to achieve four memory accesses per instruction it uses a modified von Neuman memory architecture which requires it to divide the system clock by four, resulting in an instruction rate of 20 MIPS. If we go into manic marketing mode, we can have fun working out ever higher MIPS or MOPS ratings as follows:

80 MHz clock

20 MIPS = 20 MOPS

but 2 floating point operators per cycle = 40 MOPS

and four memory accesses per instruction = 80 MOPS

plus three pointer increments per instruction = 60 MOPS

plus one floating point register update = 20 MOPS

making a grand total MOPS rating of 200 MOPS

Which exercise serves to illustrate three things:

- MIPS, MOPS and Mflops are misleading measures of DSP power
- marketing men can squeeze astonishing figures out of nothing

Of course, we omitted to include in the MOPS rating (as some manufacturers do) the possibility of DMA on serial port and parallel port, and all those associated increments of DMA address pointers, and if we had multiple comm ports, each with DMA, we could go really wild...

Apart from a cheap laugh at the expense of marketing, there is a very serious lesson to be drawn from this exercise. Suppose we only did adds with this processor? Then the Mflops rating falls from a respectable 40 Mflops to a pitiful 20 Mflops. And if we don't use the memory accesses, or the pointer increments, then we can cut the MOPS rating from 200 MOPS to 20 MOPS.

It is very easy indeed to write very inefficient DSP code. Luckily it is also quite easy, with a little care, to write very efficient DSP code.

Advanced DSP

Matched filters - index

This is a module of the BORES Signal Processing advanced DSP course - Matched filters.

To follow this course module properly, you should be familiar with the basic ideas of DSP which are introduced in the earlier course:

- Introduction to DSP

and specifically with the section on filters:

- Introduction to digital filters

This course module covers the following subjects:

- filtering to extract signals from noise
- filtering and signal to noise
- matched filters

The course is intended for self study over the Internet only. All material is copyright, and you are not permitted to make copies, either for personal use or for teaching purposes.

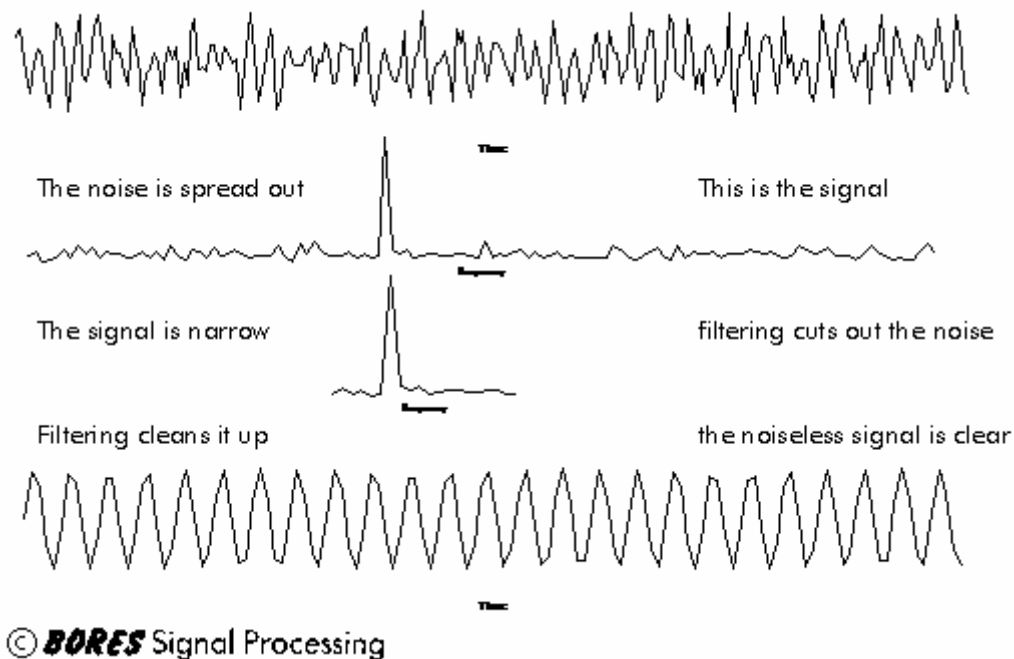
The complete course -Introduction to DSP - can be presented as an 'on site' one day intensive course by Dr Chris Bore, who is able to explain the more difficult concepts of DSP in a very clear and understandable way with many anecdotes and every day examples which help to demystify the subject.

Advanced DSP - matched filters

Filtering to extract signals from noise

A common use for digital filtering is to reduce unwanted noise.

Filtering is a frequency selective operation: we seek to reduce noise by suppressing noise frequency components but passing signal frequency components:



The diagram shows a simple case where the desired signal has a narrow frequency spectrum but the noise is spread out: here a narrow band pass filter is used to pass the narrow band signal but suppress most of the broad band noise.

The example works well for a narrow band signal: but if the signal's frequency spectrum were more complicated, things would not be quite so simple. In fact it is easy to imagine that if we know the frequency spectrum of the desired signal, then we might be able to design a more intelligent filter that would pass the maximum signal power while suppressing as much of the noise power as possible.

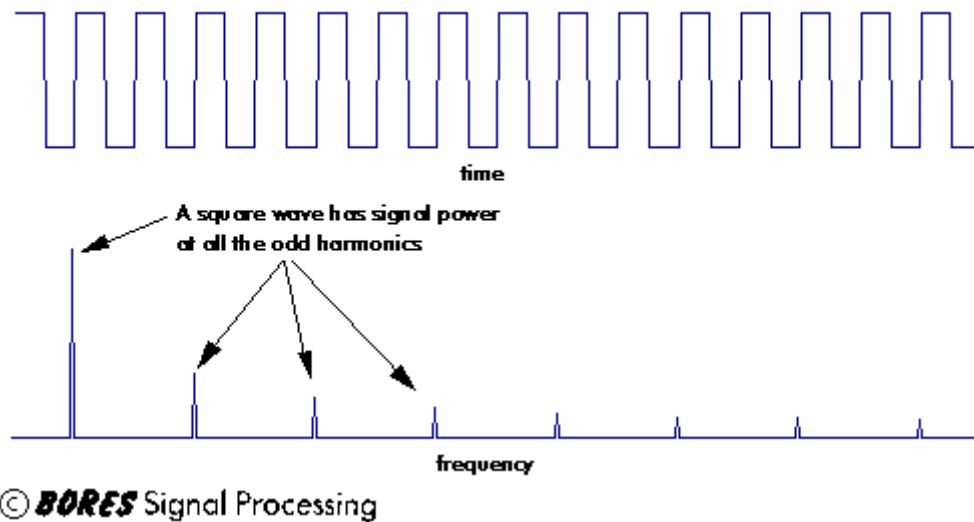
This leads to the idea of a matched filter - a filter which increases the signal to noise ratio by as much as possible.

Advanced DSP - matched filters

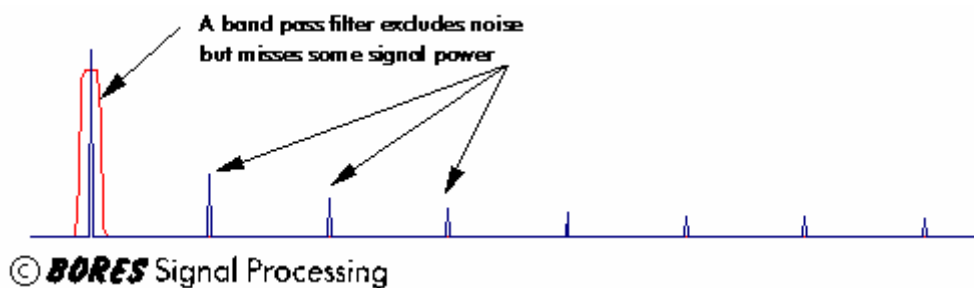
Filtering and signal to noise ratio

A common use for digital filtering is to reduce unwanted noise.

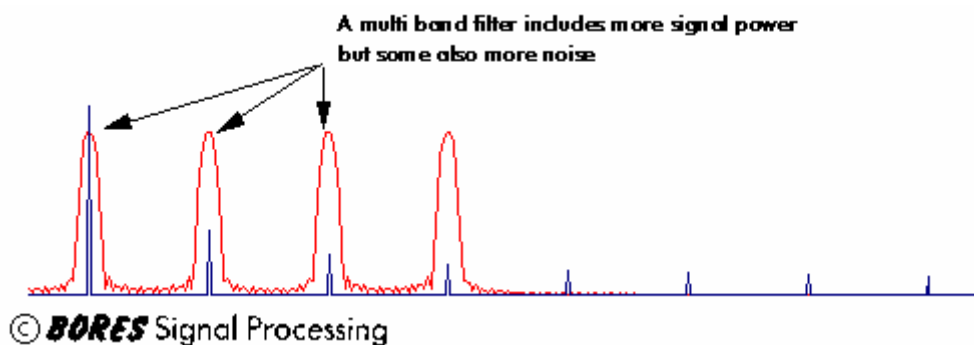
Take the example of a simple square wave. The frequency spectrum of a square wave consists of a fundamental at the square wave frequency, plus the odd harmonics decreasing in size:



If we applied a band pass filter centred around the fundamental, the signal to noise ratio would be improved because all the noise power outside the band would be suppressed:



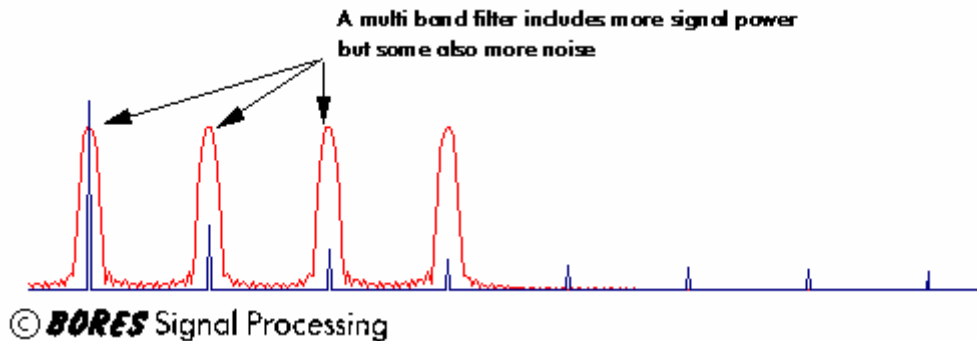
But although the noise is suppressed, so too is significant signal power coming from the other signal frequency components. This signal power could have contributed to improving the signal to noise ratio. In fact, to make the best of all available signal power we want to include all frequency components where the signal is strong, and eliminate all frequency components where there is only noise:



The diagram shows a filter which has several bands - each centred on one of the signal frequency components. This filter will pass the maximum amount of signal power, while excluding all noise that does not coincide with a signal frequency component.

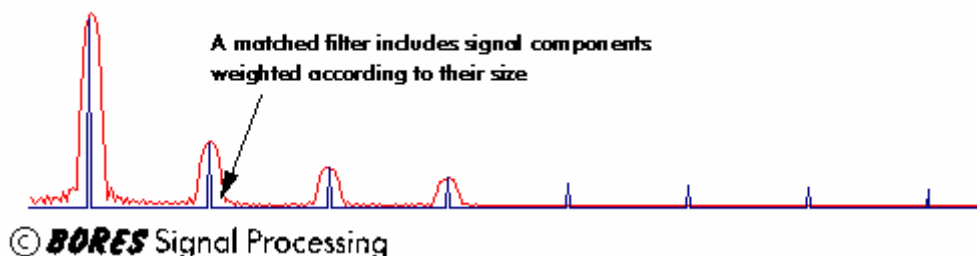
Matched filters

A filter which passes all the signal frequency components while suppressing any frequency components where there is only noise allows to pass the maximum amount of signal power:



Unfortunately, there is still significant noise power in each of the filter bands. So this frequency components also allows more noise to pass through. To make matters worse, some signal frequency components are rather small - so they contribute relatively little extra signal power, while still allowing through all the noise from that filter band.

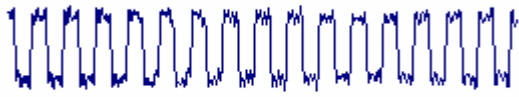
To get the maximum signal to noise ratio, we want to allow through all the signal frequency components - but to take more notice of signal frequency components that are large and so contribute more to improving the overall signal to noise ratio. We can do this by weighting the contributions from each filter band proportionally to the signal power:



The diagram shows a filter whose frequency response is designed to exactly match the frequency spectrum of the signal. Such a filter gives the best possible improvement in signal to noise ratio, and is called a matched filter.



a noisy square wave...



... is improved by filtering
with a single band pass filter...



... but the best signal to noise
comes from a matched filter

© **BORES** Signal Processing

The diagram shows a noisy square wave. Passing this through a single band passfilter improves the signal to noise ratio: but passing it through a matched filter gives the maximum possible improvement in signal to noise ratio.

A matched filter is in fact the same as correlating a signal with a copy of itself.

Advanced DSP - FFT windows

Index

This is a module of the BORES Signal Processing advanced DSP course - FFT windows.

To follow this course module properly, you should be familiar with the basic ideas of DSP which are introduced in the earlier course:

- Introduction to DSP

and specifically with the section on windowing

- Introduction to FFT window functions

This course module covers the following subjects:

- Window function kernels
- The FFT as a series of filters
- Coherent Power Gain
- Equivalent Noise Bandwidth
- Processing Loss
- Spectral leakage
- Resolution
- Figures of merit

The course is intended for self study over the Internet only. All material is copyright, and you are not permitted to make copies, either for personal use or for teaching purposes.

The complete course - Introduction to DSP - is presented regularly as a one day, 'hands on' workshop where delegates use DSP hardware and software to complete exercises intended to help in understanding the concepts which are introduced.

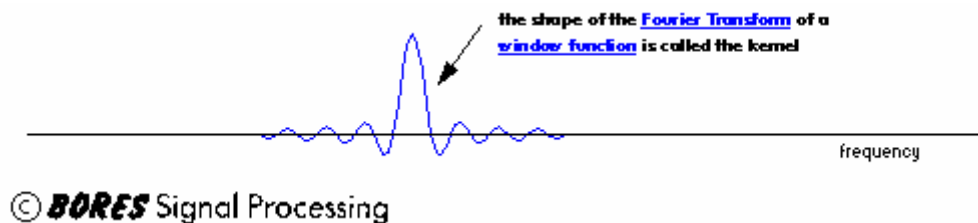
Advanced DSP - FFT windows

Windows kernels

When a window function is applied to a signal before the Fourier Transform, this changes the measured frequency spectrum. Specifically, the window function alters the spectral leakage.

One way to visualise spectral leakage is as spreading of the frequency components. Each frequency component of the signal should contribute only to one single frequency of the Fourier Transform (called an FFT 'bin'): but spectral leakage causes the energy to be spread. The window function controls the spreading. The contribution from any real frequency component to a given FFT bin is weighted by the amplitude of the window function's frequency spectrum centred at the FFT bin.

For example, in the special case of a rectangular window (that is, no window at all except for the inevitable truncation) the frequency spectrum of the window function is the 'sinc' function shown below. The shape of the Fourier Transform of a window function is called the kernel.

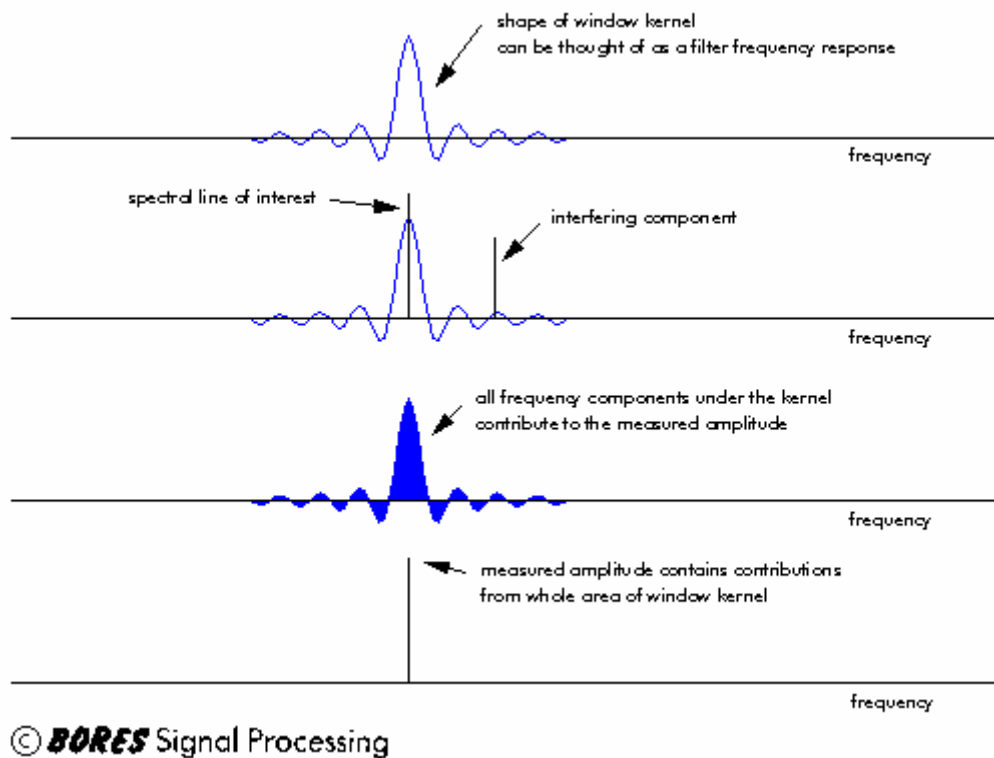


The kernel of a rectangular window function is formally called the 'Dirichlet kernel'.

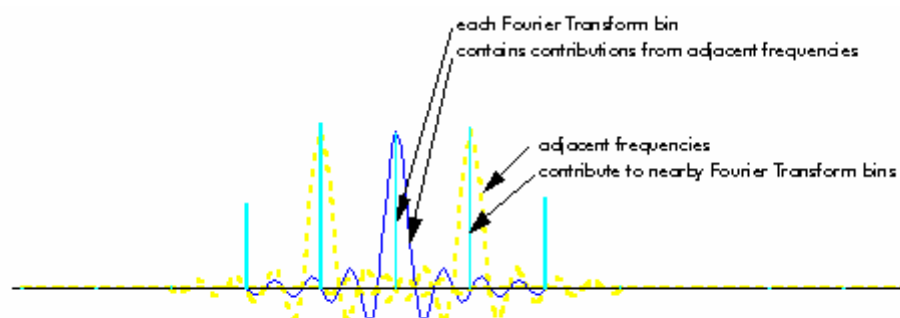
Confusingly, the Fourier Transform of a window function is often also called the window function (instead of the kernel) - we have to judge from the context whether we are talking about the frequency spectrum or the time domain function.

The FFT as a series of filters

A productive way to visualise spectral leakage is to view the Fourier Transform as equivalent to a series of filters, one centred on each FFT bin. The filter's frequency response is the shape of the window function's kernel. Each FFT bin contains contributions from all other frequency components within the bandwidth of the filter, weighted by the filter's frequency response (the kernel).



The diagram shows the kernel of a rectangular window function. The frequency component we think we are measuring is the FFT bin at the middle of the kernel. But an interfering frequency component some distance away will also contribute to the measured value of this frequency component.



In fact, the measured value of the FFT bin will include contributions from all frequency component in the bandwidth of the kernel, weighted by the kernel's value at those frequencies. This will include contributions from broad band noise as well as from narrow band signals at other frequencies.

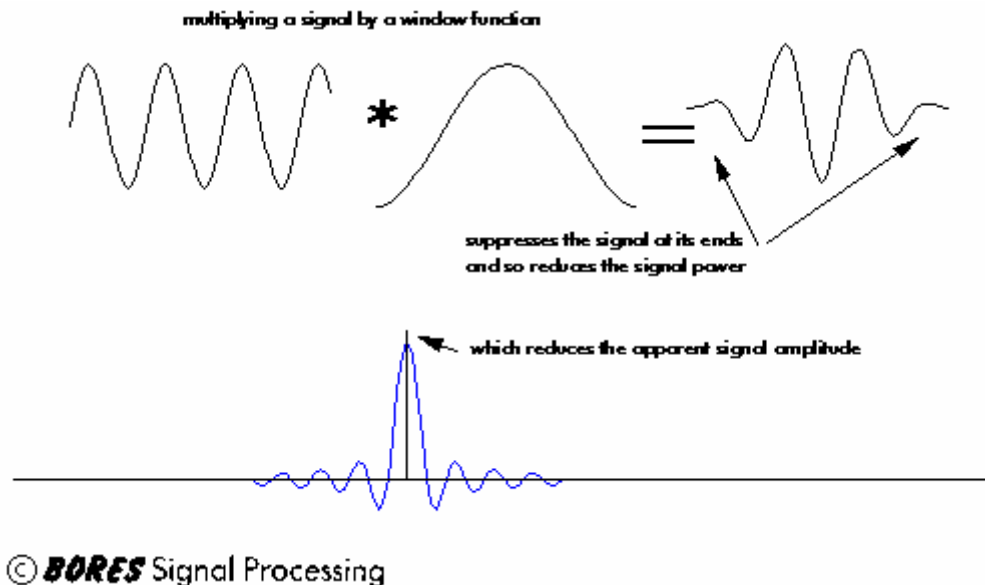
There are two common situations with which we have to deal:

- detection of a frequency component in the presence of broadband noise
- distinguishing between close narrow band frequency components

The choice of window function may be different in the two cases. To help in choosing a suitable window function we need some quantitative measures of the quality of a window function.

Coherent Power Gain

Applying a window function makes the signal smoothly approach zero at both ends. This affects the total signal power:



Because the window function attenuates the signal at both ends, it reduces the overall signal power. This reduction in signal power is called the Coherent Power Gain. Its result is that the amplitude you measure at the FFT bin is not the same as the 'real' amplitude of the signal's frequency component at that frequency. The contribution from the signal's frequency component at the FFT bin is reduced by the Coherent Power Gain. This is one reason why amplitudes measured from a Fourier Transform never quite seem to be as expected.

Don't blame me for calling a reduction in signal power a Coherent Power Gain - I don't make up these terms, I just have to live with them.

For an ideal, single discrete line frequency component, the 'noiseless' signal contribution to the FFT bin is proportional to the signal amplitude. The proportionality factor is the area under the window function's kernel - or in a sampled system, the sum of the amplitudes of the window function. The Coherent Power Gain is the square of this, or in other words the Coherent Power Gain is the square of the sum of the amplitudes of the window function's kernel.

You may notice that the Coherent Power Gain is just the DC gain of the window function.

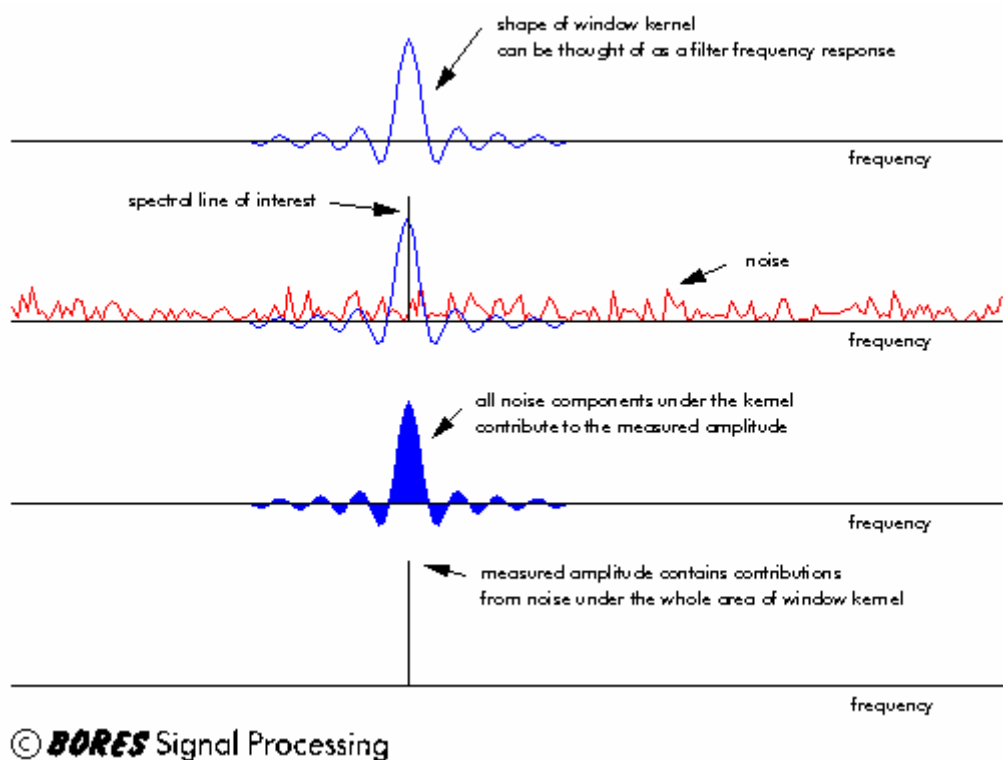
For a rectangular window, where every amplitude is 1, the DC gain is N - the number of terms - and the Coherent Power Gain is N^2 : but for any other window function the DC gain will be reduced because the window function goes smoothly to zero at its ends and so reduces the signal power.

Coherent Power Gain is important because it represents a definite scaling of the amplitudes of the measured frequency spectrum which requires correction for any absolute measurements to be correct.

Coherent Power Gain is usually normalised by dividing by the number of terms N , so that the Coherent Power Gain of a rectangular window function would be normalised to 1.

Equivalent Noise Bandwidth

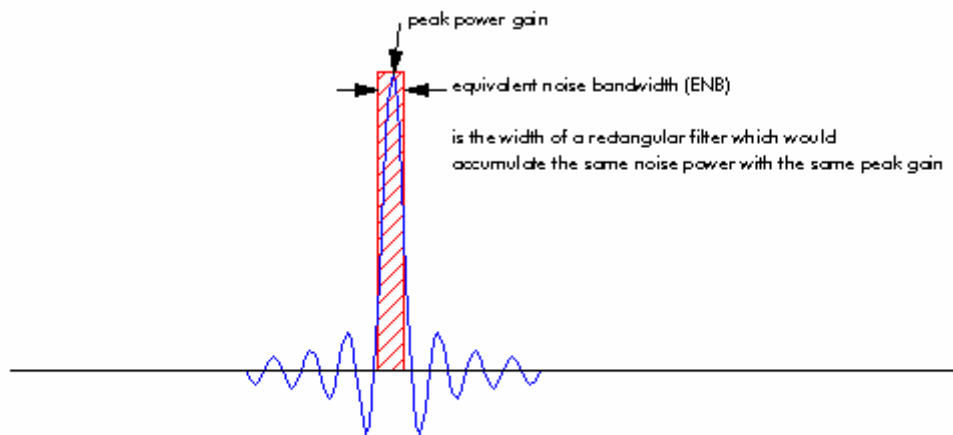
A given FFT bin contains contributions from other frequency component including broadband noise:



The shape of the window function's kernel can be thought of as a filter's frequency response. Noise will contribute to the measured value at the FFT bin proportionally to the filter's frequency response.

To detect a narrow band signal in the presence of broadband noise, we want to narrow the bandwidth from which noise contributions are significant. This can be done by narrowing the window function's kernel.

Equivalent Noise Bandwidth measures the noise performance of the window function:



© **BORES** Signal Processing

Equivalent Noise Bandwidth is the width of an ideal rectangular filter which would accumulate the same noise power from white noise as the window function's kernel with the same peak power gain. This is a fruitful concept, and quite easy to visualise.

Processing loss and scalloping loss

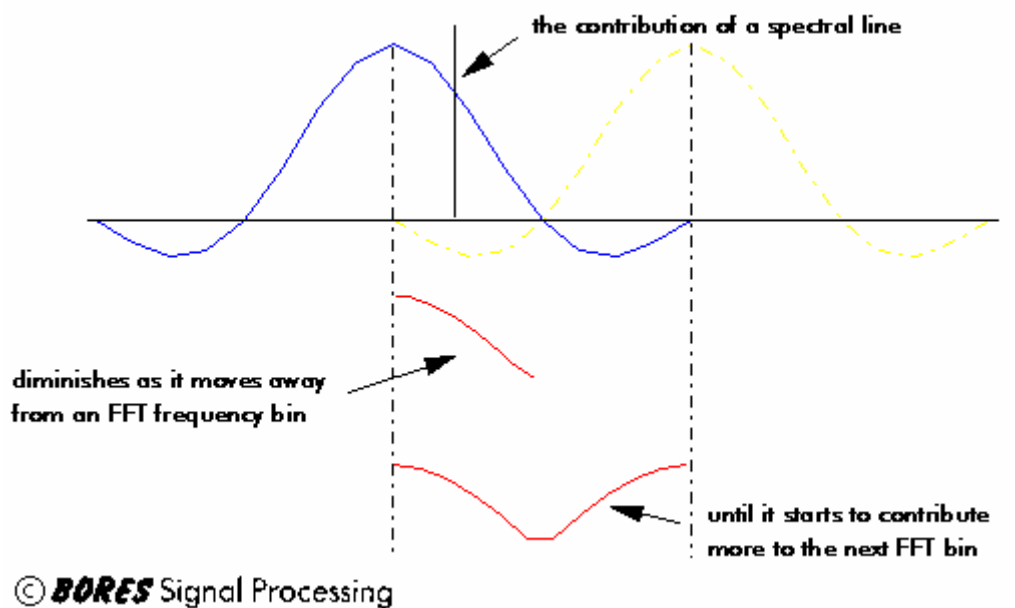
In detecting a signal from noise, it is the signal to noise ratio that is important. The window function affects the signal to noise ratio because the measured value at the FFT bin includes noise from the whole bandwidth of the window function's kernel. We need to compare the amount by which the window function attenuates the signal, with the amount of noise the window function collects:

- attenuation of the signal by the window function is measured by the Coherent Power Gain
- the amount of noise collected by the window function is measured by the Equivalent Noise Bandwidth

Processing Loss measures the degradation in signal to noise ratio due to the window function. It is the ratio of Coherent Power Gain to Equivalent Noise Bandwidth.

For a signal made up of one ideal discrete single line frequency component the Coherent Power Gain is 1 and so the Processing Loss is (not surprisingly) just the reciprocal of the Equivalent Noise Bandwidth.

Processing Loss only relates to signal frequency components that happen to fall exactly on an FFT bin. But the model of the the Fourier Transform as a series of filters centred on each FFT bin suggests that frequencies that do not happen to fall exactly on an FFT bin will not be measured at 100% of their full value as the window function's kernel response falls off away from its centre frequency. In fact one can imagine that the measured value will fall off as the actual signal frequency component's frequency moves away from the FFT bin frequency:



The effect is to make the measured value dip as the actual frequency moves between FFT bins: this effect is sometimes called the 'picket fence effect' or (because it looks a bit like the edge of a scallop shell) 'scalloping'. It is a reasonable assumption that the worst case occurs when the actual signal frequency falls exactly half way between FFT bins.

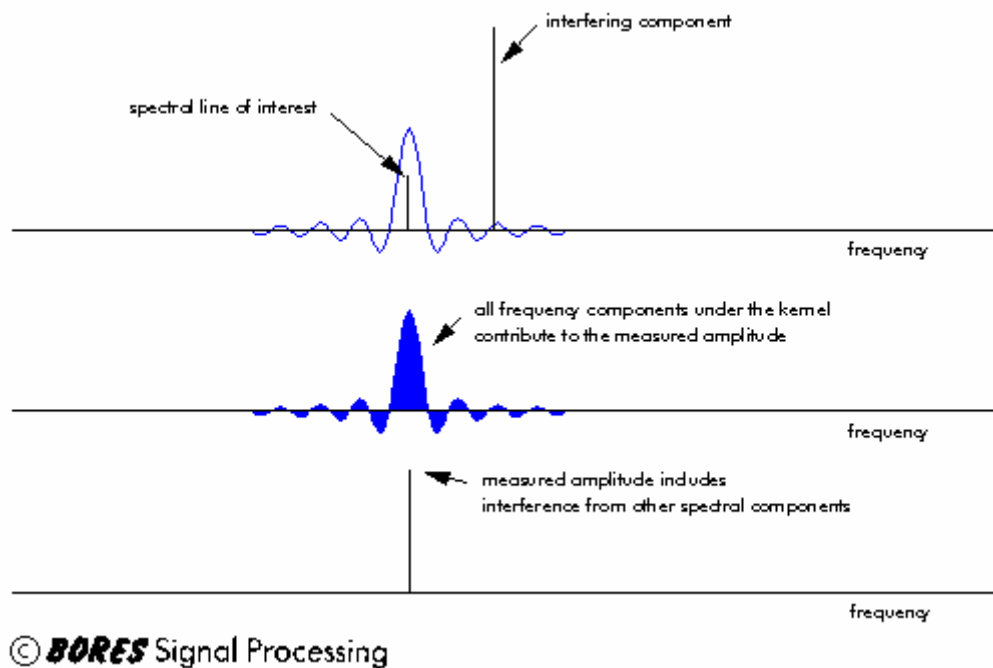
Scalloping Loss is the apparent attenuation of the measured value for a frequency component that falls exactly half way between FFT bins. It is defined as the ratio of the power gain for a signal frequency component located half way between FFT bins, to the Coherent Power Gain for a signal frequency component located exactly on the FFT bin.

Scalloping Loss can be calculated by taking the ratio of the value of the window function's kernel one half a a frequency sample off centre, to its value at the centre.

Worst Case Processing Loss is the sum of Processing Loss and Scalloping Loss. This is a measure of the worst case reduction of signal to noise ratio which results from the combination of the window function and the worst case frequency location. It is related to the minim amplitude for a signal frequency component to be detected in broadband noise.

Spectral leakage

The measured frequency spectrum is not only affected by broadband noise, but also by narrow band frequency components either of noise or of the signal:



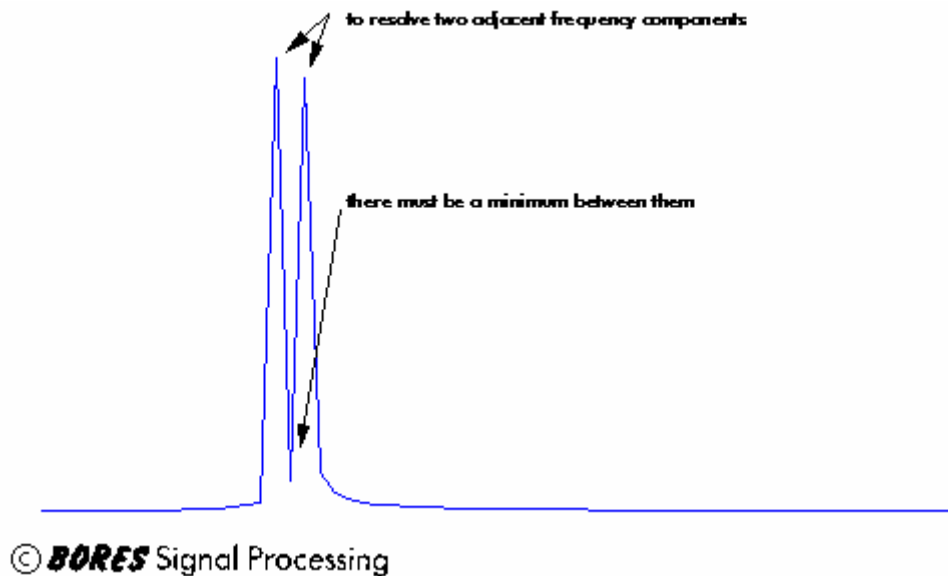
A frequency component of noise or signal at one frequency will contribute to the measured value at an FFT bin. The amount of the contribution is proportional to the amplitude of the window function's kernel, centred at the FFT bin, measured at the interfering frequency. This amount is what is meant when we talk about spectral leakage.

Spectral leakage changes both the amplitude and position of a frequency measurement. The effects of spectral leakage are worst when detecting small signals in the presence of nearby large signals.

To minimise the effect of spectral leakage, a window function's kernel should have low amplitude sidelobes away from the centre: and the rate of fall off to the low sidelobes should be rapid. The peak sidelobe level is a useful indicator of how well a window function suppresses spectral leakage: so is the rate of fall off to the sidelobes.

Frequency resolution

An interesting measure of a window function's quality is the minimum separation needed between two frequency components of equal amplitude, that still allows them to be resolved. By 'resolved', we mean that there is a local minimum between the two peaks:



The usual 'rule of thumb' for being able to resolve two adjacent signal peaks is the width of the peak at the half power point (the '3 dB bandwidth'). This is because two frequency components of equal amplitude will show only a single peak if separated by less than their 3 dB bandwidth, and so cannot be resolved. But the calculation on which this 'rule of thumb' is based assumes that the signals add incoherently. This is an important problem with the Fourier Transform because the addition which is involved in the Fourier Transform is coherent, not incoherent.

The Fourier Transform is the addition of frequency components, weighted by the window function's kernel at each frequency. Because of the coherence, the 6 dB bandwidth determines resolution rather than the 3 dB bandwidth.

Figures of merit

The measures of the quality of an FFT window function are:

- Sidelobe level- the attenuation to the top of the highest sidelobe
- Fall off - the rate of fall off to the side lobe
- 6 dB bandwidth - the bandwidth in which the window function's kernel falls by 6 dB
- Coherent Power Gain - the normalised DC gain
- Equivalent Noise Bandwidth- the bandwidth of an ideal rectangular filter which would accumulate the same noise power from white noise as the window function's kernel with the same peak power gain
- Worst Case Processing Loss - the ratio of input signal-to-noise to output signal-to-noise, including scalloping loss for the worst case

Sidelobe level, fall off and 6 dB bandwidth measure a window function's quality in terms of resolving close frequency components.

Coherent Power Gain measures the amount by which a frequency component's measured value is attenuated by the window function

Equivalent Noise Bandwidth and Worst Case Processing Loss measure a window function's quality in terms of detecting a signal in noise.

Window function	Sidelobe (dB)	Fall off (dB/octave)	Coherent Power Gain	Equivalent Noise Bandwidth (FFT bins)	6 dB bandwidth (FFT bins)	Worst Case Processing Loss (dB)
Rectangular	- 13	- 6	1.00	1.00	1.21	3.92
Triangular	- 27	- 12	0.50	1.33	1.78	3.07
Hanning	-32	-18	0.50	1.50	2.00	3.18
Hamming	- 43	- 6	0.54	1.36	1.81	3.10
Poisson (3.0)	- 24	- 6	0.32	1.65	2.08	3.64
Poisson (4.0)	- 31	- 6	0.25	2.08	2.58	4.21
Cauchy (4.0)	- 35	- 6	0.33	1.76	2.20	3.83
Cauchy (5.0)	- 30	- 6	0.28	2.06	2.53	4.28
Gaussian (3.0)	- 55	- 6	0.43	1.64	2.18	3.40
Kaiser-Bessel (3.0)	- 69	- 6	0.40	1.80	2.39	3.56
Kaiser-Bessel (3.5)	- 82	- 6	0.37	1.93	2.57	3.74

You can download a copy of this technical note in Adobe Acrobat format:

- [FFT windows technical note](#)

FFT window functions

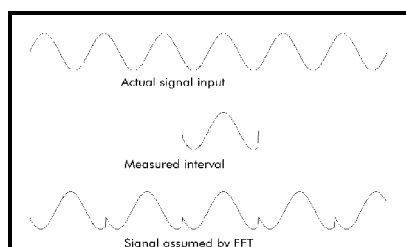
Limits on FFT analysis

When using FFT analysis to study the frequency spectrum of signals, there are limits on resolution between different frequencies, and on detectability of a small signal in the presence of a large one.

There are two basic problems: the fact that we can only measure the signal for a limited time; and the fact that the FFT only calculates results for certain discrete frequency values (the 'FFT bins'). The limit on measurement time is fundamental to any frequency analysis technique: the frequency sampling is peculiar to numerical methods like the FFT.

Limited measurement time

The first problem arises because the signal can only be measured for a limited time. Nothing can be known about the signal's behaviour outside the measured interval. We have to assume something about the signal outside the measured interval, and the Fourier Transform makes an implicit assumption that the signal is repetitive: that is, the signal within the measured time repeats for all time.

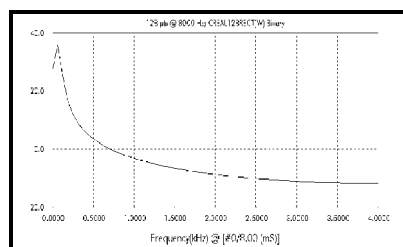


Most real signals will have discontinuities at the ends of the measured time, and when the FFT assumes the signal repeats it will assume discontinuities that are not really there.

Since sharp discontinuities have broad frequency spectra, these will cause the signal's frequency spectrum to be spread out.

Spectral leakage

It is easy to gain an insight by thinking about the special case of a pure sine wave. This has a frequency spectrum which is a single spectral line: but the frequency spectrum calculated by the FFT will show a spread out line. Each spectral line will be spread out in the same way.



The spreading means that signal energy which should be concentrated only at one frequency instead leaks into all the other frequencies. This spreading of energy is called 'spectral leakage'.

Since spectral leakage is related to discontinuities at the ends of the measurement time, it will be worse for signals that happen to fall such that there are large discontinuities.

Some signals may, by coincidence or by design, fall in such a way that there happens to be no discontinuity at the ends of the measurement time: for these signals the effect of spectral leakage may be lessened.

For example a pure sine wave sampled for an exact number of cycles would match up quite correctly when made repetitive: the repetitive signal would be exactly the same as the 'real' signal and so no spectral leakage would occur.

Another example would be a signal which fell smoothly to zero at each end of the measurement interval: such a signal would have no discontinuities when made repetitive, and so would not suffer from spectral leakage.

Such special cases are infrequent,

but they can be arranged: for example frequency analyses are often made by tuning a stimulus frequency precisely so that its frequency exactly fits the measurement interval.

Spectral leakage is not an artifact of the FFT, but is due to the fact that the signal was measured only for a finite time. For a sine wave to have a single line spectrum it must exist for all time. Any practical method of calculating the frequency spectrum of a signal suffers from spectral leakage due to the finite measurement time.

Spectral leakage is not related in any way to the fact of having sampled the signal, but only to the finite measurement time.

Spectral leakage causes at least two distinct problems.

First, any given spectral component will contain not just the signal energy, but also noise from the whole of the rest of the spectrum. This will degrade the signal to noise ratio.

Second, the spectral leakage from a large signal component may be severe enough to mask other smaller signals at different frequencies.

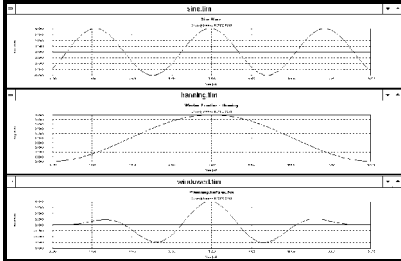
Windowing

In effect, the process of measuring a signal for a finite time is equivalent to multiplying the signal by a rectangular function of unit amplitude: the rectangular function lasting for the duration of the measurement time.

The signal is measured during a finite measurement time or 'window'. This idea leads to the rectangular function being called a 'rectangular window'.

The effects of spectral leakage can be reduced by reducing the discontinuities at the ends of the signal measurement time.

This leads to the idea of multiplying the signal within the measurement time by some function that smoothly reduces the signal to zero at the end points: hence avoiding discontinuities altogether.



The process of multiplying the signal data by a function that smoothly approaches zero at both ends, is called 'windowing': and the multiplying function is called a 'window' function.

It is easy to analyse the effect of a window function: the frequency spectrum of the signal is convolved with the frequency spectrum of the window function.

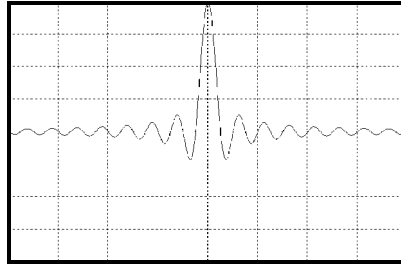
Spectral leakage

One way to visualise spectral leakage is as spreading of the frequency components.

Each frequency should contribute only to one FFT bin but spectral leakage causes the energy to be spread by the window function so it contributes to all other FFT bins.

The contribution is weighted by the window function centred at the frequency component and evaluated at the FFT bin.

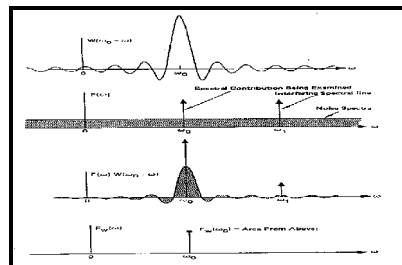
In the special case of the rectangular window (that is, no window at all), the window function is 1 in the interval and 0 outside the interval. Its Fourier transform is known as the 'sinc' function, or more formally the 'Dirichlet kernel'.



The shape of the Fourier transform of a window function is sometimes called the 'kernel'.

(Confusingly, the Fourier Transform of a window function is also often called the 'window function': we have to judge from context whether we are talking about the frequency or the time domain function.)

The FFT as a series of filters



Another productive way to visualise spectral leakage is to regard the FFT as equivalent to a series of filters, centred on each spectral sample.

The filter's frequency response is the shape of the window function.

This is the inverse of the spectral leakage model. Each FFT bin includes contributions from all other frequencies in the bandwidth of the filter, weighted by the window function. This will include contributions from broad band noise as well as narrow band signals at other frequencies.

Detection or resolution?

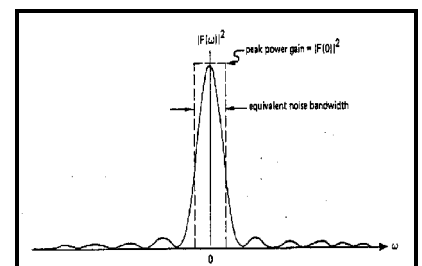
There are two common situations: detection of a spectral component in the presence of broadband noise; or distinguishing between narrow band spectral components. The choice of window function may be different in the two cases of detection or

resolution.

To help in choosing a suitable window function some quantitative measures are needed.

Equivalent noise bandwidth

A given FFT bin includes contributions from other frequencies including accumulated broadband noise. To detect a narrow band signal in the presence of noise, we want to minimise the noise. This can be done by using a narrow bandwidth window function.



The Equivalent Noise Bandwidth (ENB) of the window measures the noise performance of the window. It is the width of a rectangle filter which would accumulate the same noise power with the same peak power gain. This is a fruitful concept, and easy to visualise.

Coherent power gain

'Coherent Power Gain' measures the reduction in signal power due to the window function suppressing a coherent signal at the ends of the measurement interval.

Again, this measurement relates to the model of the FFT as a bank of filters matched to each frequency sample.

For an ideal discrete line frequency component, the 'noiseless' signal contribution to the FFT bin is proportional to the signal amplitude. The proportionality factor is the sum of the window terms, which is just the DC gain of the window. The 'Coherent Gain' is the square of this, or in other words the

coherent power gain is the square of the sum of the window terms.

For a rectangular window the DC gain is N , the number of terms in the window: but for any other window the DC gain will be reduced because the window goes smoothly to zero at the ends of the measurement time.

The reduction in DC gain is important because it represents a definite scaling of the amplitudes of the frequency spectrum which requires correction for any absolute measurements to be correct.

Coherent gain is usually normalised by dividing by N , so that the normalised coherent gain of a rectangular window is 1.

Processing Loss

'Processing Loss' is the ratio of input signal to noise to output signal to noise, which is the Coherent Power Gain divided by the noise power.

For a signal made up of an ideal discrete line frequency component polluted by white noise, the Processing Loss is, not unexpectedly, the reciprocal of the Equivalent Noise Bandwidth (ENB).

Scalloping loss

The analysis of Equivalent Noise Bandwidth and Processing Loss so far related only to signal components whose frequencies exactly match the FFT bins.

The model of the FFT as a series of filters centred on the FFT bins suggests that frequencies that do not coincide with the FFT bins will be attenuated as the filter's response falls off away from the centre frequency.

This effect will vary as the signal frequency ranges between the adjacent FFT bins, and is called 'the picket fence effect' or 'scallop loss'.

It is a reasonable assumption that the worst case occurs when the

signal frequency lies exactly half way between FFT bins.

'Scalloping Loss' is defined as the ratio of coherent gain for a signal frequency component located half way between FFT bins, to the coherent gain for a signal frequency component located exactly at an FFT bin.

This is just the ratio of the window function's value one half a frequency sample off centre, to its value at the centre frequency.

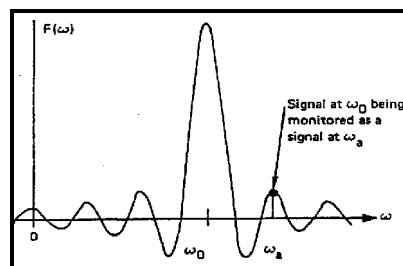
Worst case processing loss

Worst case processing loss is defined as the sum of Scalloping Loss and Processing Loss.

This is a measure of the reduction of output signal to noise ratio resulting from the combination of the window function and the worst case frequency location. It is of course related to the minimum tone that could be detected in broadband noise.

Spectral leakage

The calculated spectrum is not only affected by broadband noise, but also by the narrow band frequency components either of noise or of the signal.



A frequency component at frequency ω_0 will contribute to the calculation at another frequency ω_a proportionally to the gain of the window's Fourier transform, centred at ω_0 and measured at ω_a . This is usually referred to as 'spectral leakage'.

Spectral leakage can change the calculated amplitude and position of

a spectral estimate.

The effects of spectral leakage are worst when detecting small signals in the presence of nearby large signals.

To minimise the effects of spectral leakage, a window function's FFT should have low amplitude sidelobes away from the centre, and the fall off to the low sidelobes should be rapid.

The peak sidelobe level is a useful indicator of how well a window function suppresses spectral leakage: so is the rate of fall off to the sidelobes.

Minimum resolution bandwidth

An interesting measure is the minimum separation needed between two frequency components of equal amplitude, so that they can be resolved.

By resolved, is meant that there should be a local minimum between the two peaks.

The 'rule of thumb' for resolvability is the width of the window at the half power points (the 3 dB bandwidth): because two frequency components of equal strength show a single peak if separated by less than their 3 dB bandwidth, and so cannot be separated.

But this assumes incoherent addition. There is an important problem with this criterion when applied to the FFT because the addition which is involved with the FFT is coherent, not incoherent

The FFT output is the coherent addition of frequency components, weighted by the window function at each frequency. Because of the coherence, the 6 dB bandwidth defines resolution rather than the 3 dB bandwidth.

If two frequency components are involved the sum of the window functions at the crossover point (halfway between the peaks) must

be smaller than the individual peaks if the two peaks are to be resolved. So the gain from each window function must be less than 0.5 (or 6 dB).

Some window functions

The table lists, for various window functions, the following parameters:

Sidelobe level	The attenuation to the top of the highest side lobe
Fall off	The asymptotic rate of fall off to the side lobe
Coherent gain	The normalised DC gain
Equivalent noise bandwidth	The bandwidth of a rectangular filter which would let pass the same amount of broadband noise
6 dB bandwidth	The bandwidth in which the window function falls by 6 dB
Worst case processing loss	The ratio of input signal to noise to output signal to noise, including scalloping loss for the worst case frequency

Window function	Sidelobe level (dB)	Fall off (dB per octave)	Coherent gain	Equivalent noise bandwidth (bins)	6 dB bandwidth (bins)	Worst case processing loss (dB)
Rectangular	-13	-6	1.00	1.00	1.21	3.92
Triangular	-27	-12	0.50	1.33	1.78	3.07
Hanning	-32	-18	0.50	1.50	2.00	3.18
Hamming	-43	-6	0.54	1.36	1.81	3.10
Poisson (3.0)	-24	-6	0.32	1.65	2.08	3.64
Poisson (4.0)	-31	-6	0.25	2.08	2.58	4.21
Cauchy (4.0)	-35	-6	0.33	1.76	2.20	3.83
Cauchy (5.0)	-30	-6	0.28	2.06	2.53	4.28
Gaussian (3.0)	-55	-6	0.43	1.64	2.18	3.40
Kaiser-Bessel (3.0)	-69	-6	0.40	1.80	2.39	3.56
Kaiser-Bessel (3.5)	-82	-6	0.37	1.93	2.57	3.74