Lecture 15, 19th October 2012

Digital Signal Processing:

In the last lecture we looked at how to convert an analogue signal into digital form. While users of CDs, MP3s or online videos may appreciate why this is useful for media contents, we need to think about why one would do this with experimental data *despite the losses incurred*. The answer lies in the ever increasing *number of ways to manipulate digital data* on a computer. Here a few of them will be introduced.

Contents:

- Why do off-line digital data analysis?
- Filters and transforms
- An example non-linear filter: the median filter

Off-line digital data analysis:

You have seen that filters can be built to manipulate analogue signals. So, why bother to first digitize them with a loss of information and only later manipulate them on a computer?

It is often sensible to record data in digital form

- Some data is naturally digital
- Digital data is more robust to noise (only "0"s and "1"s are recognised)
- Renders analysis by computer possible (see below why this may be of advantage)

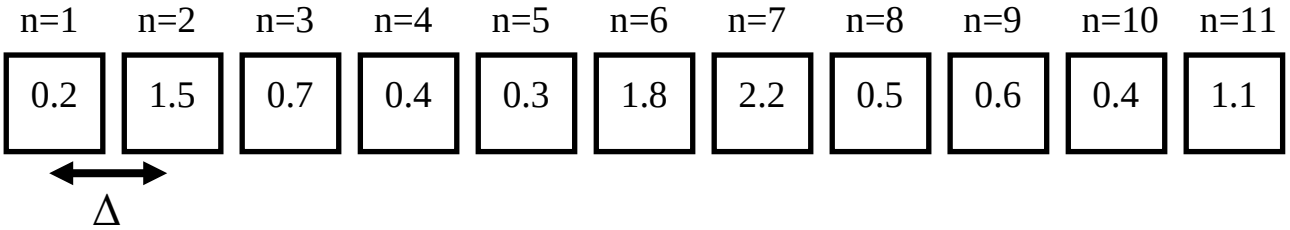
With an on-line ("live") treatment of analogue data you may just get *one shot* at doing what you want. If you make a mistake your data is corrupted, likely beyond recovery. It is better to record your raw data and *separate the analysis* step from the recording.

Digital memory is much *cheaper* and *faster* in recording and playback than analog data storage. It pays to make use of it!

Once the data is digitized and recorded it can be analysed repeatedly. Different approaches can be tried until an appropriate method is found. Later insight, whether theoretical, experimental or methodical, can inspire people to return to old data to re-examine it with new methods or correction procedures.

Digital data:

The process of manipulating digital data is called **digital signal processing** (**DSP**). We now will take a few elements of an example data set and think about what we might want to do with it. Each box below is one data sample rather than one bit. So we ignore the issue of how the data in each box is represented in binary form.



The digital data above is a series of discrete samples

$$y_n = y(n\Delta) \text{ where } \quad n = \cdots, -3, -2, -1, 0, 1, 2, 3, \cdots$$

Δ is the time interval between samples and so $1/\Delta$ is the sampling rate. Remember, the Nyquist frequency is:

$$f_c = \frac{1}{2\Delta}$$

This is the highest frequency that can be represented in a data set sampled at rate $1/\Delta$.
Now we can take a look at something that you might want to do with this data set.

Example: simple smoothing

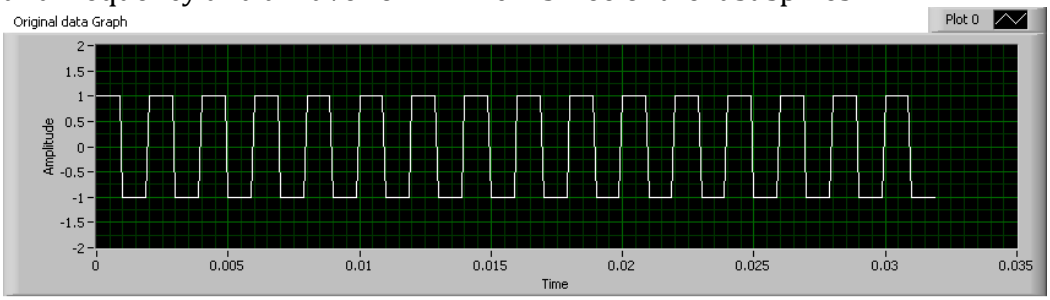
Some sources of extraneous noise may provide much faster changes than the signal you want to measure. In this case it might be of advantage to average neighbouring data elements to smooth out the noise spikes. This is done in the plot below showing a time-variant recording of a voltage measurement. The noisy signal is subjected to an algorithm which averages each pixel with the three pixels taken before and the three pixels taken afterwards. You see that:

- The random noise is reduced in favour of the underlying signal.

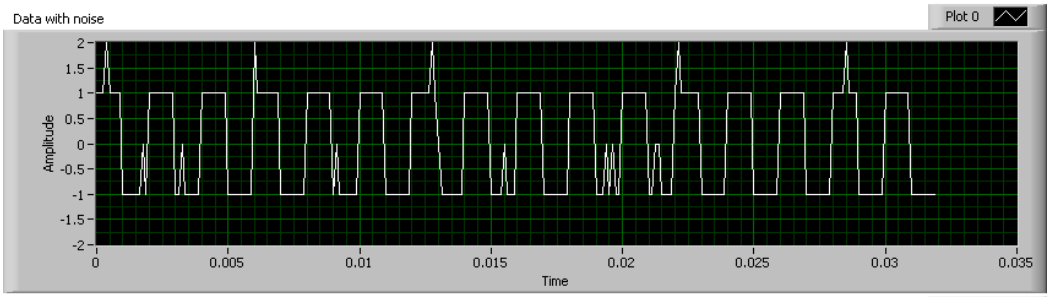
But this comes at a significant price:

- The highest spatial frequency that can be represented is also reduced. The algorithm works like a crude form of low-pass filtering and the waveform of the signal is significantly distorted. Still one correctly gets the characteristic parameters of amplitude and frequency and a wave-form which is free of the fast spikes.

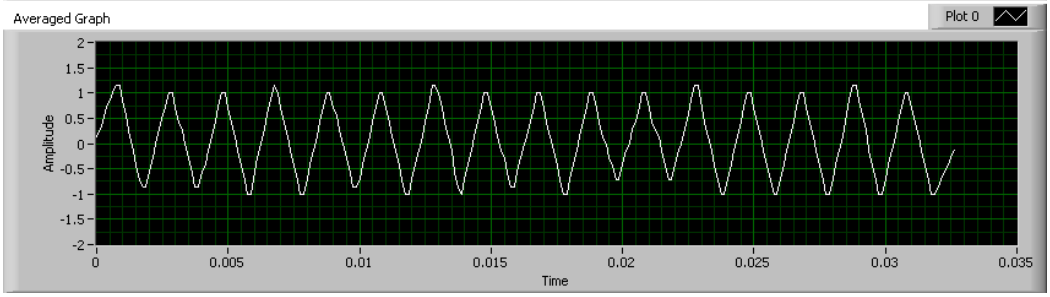
Clean signal



Signal + Noise



Average of 7 samples



It is helpful to understand this in more detail. We are thinking about this filter exclusively in the *time domain*. The raw data is recorded in the array (y) with index n. Each y_n is a data sample where n decodes the time stamp of the recording. In the averaging algorithm above 7 successive y_i around a centre y_n are processed, data sample for data sample, and the result is stored in a new array (z) as element z_n. What happens mathematically is the following:

$$z_n = \alpha y_{n+3} + \alpha y_{n+2} + \alpha y_{n+1} + \alpha y_n + \alpha y_{n-1} + \alpha y_{n-2} + \alpha y_{n-3} \quad \text{with} \quad \alpha = \frac{1}{7}$$

Or generally for *N*-element smoothing:

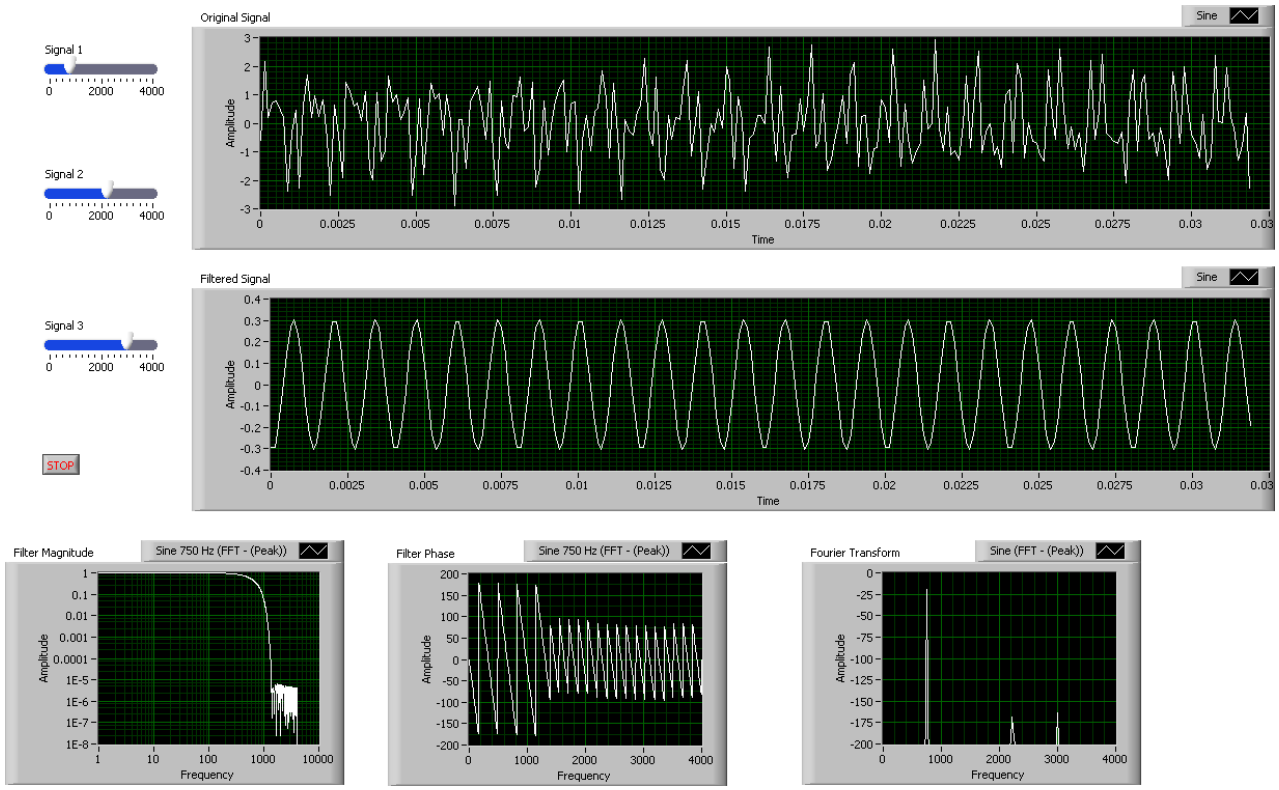
$$z_n = \frac{1}{N} \sum_{k=n-{}^{(N-1)}/_2}^{n+{}^{(N-1)}/_2} y_k$$

This is a very simple filter and improvements immediately come to mind. You could imagine that element z_n should be more strongly based on element y_n than on elements further away. Ideally, rather than weighting every element by *(1/N)* the weightings would be a function of the index *k*. When *k=n* the weighting is quite close to 1. For *k* very different from *n* a sensible smoothing filter would have a low weighting.

Still this is a rather simplistic approach. In general filters can be arbitrarily complicated algorithms.

Example: sophisticated filtering

We are looking at a filter which can be best understood in the *frequency domain*. In the lecture on op-amp based filters Bode Plots were introduced and the idea was discussed that sometimes you may want to *remove some frequency ranges* from a signal. In the example below the input signal (top trace) is composed from three sine-waves with *different frequencies*. In the Fourier Transform image graph (bottom right plot) one can see the three spikes corresponding to the three frequencies of the input signal (750Hz, 2250Hz and 3000Hz).



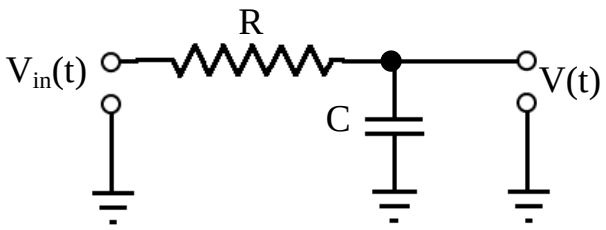
By choosing an appropriate filter (see its gain displayed in the bottom left plot) it is possible to *select a single frequency* (750Hz) and attenuate the other two almost completely (lower trace). This filter employs a cut-off frequency of 750Hz and works extremely well to suppress the two higher frequencies (note the amplitudes in the Fourier Transform plot). The price you pay is that it makes a complete mess of the phase of the signal (bottom middle plot).

Example: Finite difference filters

One way to *design* a digital filter is to begin with an electrical circuit that you might use to filter *analogue data*. Think about how you could implement that circuit in the form of an expression for manipulating digital data. Because we can use either complex numbers or

differentials to describe these filters, the connection between the time domain and the frequency domain might become a little more obvious for this example.

Low-pass filters: The most basic low-pass filter we have looked at (and you have built in the lab) involves just a resistor and a capacitor. Components of the input signal with $\omega > {}^{1}/_{RC}$ are strongly attenuated. We are going to render the behaviour of this circuit as an equation that could easily be applied to a *series of discrete samples*. This involves writing derivatives as differences. The problem solving technique whereby differential equations are *rewritten as differences* is called the **finite difference approach**.



In complex description you would analyse this circuit as (see lecture 7):

$$V(t) = \frac{V_{in}(t)Z_2}{Z_1 + Z_2} = V_{in}(t)\frac{1 - i\omega RC}{1 + \omega^2 R^2 C^2}$$

At high frequencies the capacitor strongly attenuates *V(t)*.

However this circuit can also be dealt with in terms of time. For a finite difference treatment the following equivalent description is the more useful approach. The voltage across *R* is *(V_{in} – V)* so:

$$I = C\frac{dV}{dt} = \frac{V_{in} - V}{R}$$
$$RC\frac{dV}{dt} + V = V_{in}$$

Using the finite difference approach with the following assignments:

- $V_{in} = y$ (the input signal)
- $V = z$ (the filtered signal)
- $RC = T$ (the time constant of the filter)
- $dt = \Delta$ (the infinitesimal sample spacing)
- n = time step index

the above becomes:

$$\frac{T}{\Delta}(z_n - z_{n-1}) + z_n = y_n$$
$$z_n\left(1 + \frac{T}{\Delta}\right) = y_n + \frac{T}{\Delta}z_{n-1}$$
$$z_n = \frac{1}{1 + {}^{T}/_{\Delta}}y_n + \frac{{}^{T}/_{\Delta}}{1 + {}^{T}/_{\Delta}}z_{n-1}$$

leading to:

$$z_n = (1 - \alpha)y_n + \alpha z_{n-1} \qquad \text{with} \quad \alpha = \frac{{}^{T}/_{\Delta}}{1 + {}^{T}/_{\Delta}}$$

The transformed sample z_n only depends of the raw sample y_n and the previous transformed sample z_{n-1}. Apart from starting condition for element z_0 the implementation of this equation is straight forward and a single pass over the data array (y) would be sufficient to fill the output array (z).

High-pass filters: Here the positions of capacity and resistor are swapped. The filter can be transformed into the time domain using the finite difference technique in an analogous way. The result is only marginally more complicated:

$$z_n = \frac{1}{2}(1+\alpha)y_n - \frac{1}{2}(1+\alpha)y_{n-1} + \alpha z_{n-1} \qquad \text{with} \quad \alpha = \frac{T/\Delta}{1+T/\Delta}$$

Fourier transforms and convolutions:

The first example given above was a simple smoothing filter implemented in the time domain. It involved taking the average of seven elements. This is an example of a **convolution**. In general the *convolution* of two time-dependent functions *h(t)* and *g(t)* is defined as:

$$k(t) = g(t) * h(t) = \int_{-\infty}^{+\infty} g(\tau)h(t-\tau)d\tau$$

where the integral of the product of the two functions is calculated with one signal inverted in time and phase shifted through the other. This is a complicated but very powerful and often needed operation.

The convolution, expressed above as an *integral*, can be re-expressed as a *summation*. This may make the *relationship to the smoothing filter* more obvious. Say the signal s(t) you have measured is represented by its N sampled values s_j which were taken at equal time intervals. The filter you want to apply to the data is described by a discrete set of M+1 numbers r_k (this is sometimes called the **kernel** of your filter). The discrete convolution of a data sample s_j with a kernel of length M+1 is the sum:

$$(r * s)_j = \sum_{k=-M/2}^{M/2} s_{j-k} r_k$$

That means that for the simple smoothing example plotted above the kernel has a size of M=6 and values of r_k = *1/7* for *k= -3, -2, -1, 0, 1, 2, 3* and zero everywhere else.

Convolutions carried out this way are *computationally very expensive*. There exists a much more efficient way to carry out these convolutions. Let's say that H(f), G(f) and K(f) are the **Fourier transforms** of the time-dependent functions h(t), g(t) and k(t) above, i.e. the corresponding functions in the frequency domain:

$$X(f) = \int_{-\infty}^{+\infty} x(t)e^{-2\pi i f t}dt$$

Or for N discrete samples:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}$$

It turns out that the simple product of the Fourier transforms of g(t) and h(t) equals the Fourier transform of their convolution k(t). This is known as the **Convolution theorem** of the Fourier transformation:

$$K(f) = G(f)H(f)$$

i.e. a *convolution in the time domain* is the equivalent of a *multiplication in the frequency domain*.

As there exists *a very efficient algorithm* to compute the Fourier transform on a finite number of data samples, the **fast Fourier transform** (**FFT**), it is much more efficient to transform two signals sampled in the *time domain* into the *frequency domain* using the FFT, multiply them in

the frequency domain and FFT back into the time domain than to actually compute their convolution. The FFT algorithm amounts to *reordering* the elements of one array (reversing the address bit sequence) and then *summing* up the elements with *weighting factors*.

To summarise:

- Applying filter equations to data is a *convolution*. This is computationally an *expensive* operation.
- As described by the convolution theorem: a *convolution in the time domain* is equivalent to a *multiplication in the frequency domain*. This is much easier to implement.
- The fast Fourier transform algorithm eases very much the transformation between time and frequency domain. It outperforms the direct calculation of a convolution that much that, via the convolution theorem, it becomes the preferred tool to perform convolutions.

Non-linear filters: e.g. median filter:

Above filters have been introduced in both the time domain and the frequency domain. And it has been shown how they can be implemented in digital form. All of these filters share a common behaviour: they are all **linear filters**. Linear filters are underpinned by *Fourier theory* and the *convolution theorem*. There is another class: that of **non-linear filters**. These are underpinned by *set theory* and *cannot be implemented via a convolution*. They can be very useful for minimizing the effect of some kinds of noise.

- Linear filters are ideal for situations when the noise and the important part of the signal have very different frequencies.
- For some transducers, noise is **impulsive**. That means *sharp spikes* corrupt the data. In this case the noise *includes all frequencies*.
- Impulsive noise is *best removed by a non-linear filter*. These remove data at a particular *spatial location* rather than spatial frequency.
- Here we take the example of a **median filter**. This replaces each data element by the *median of it and its closest neighbours*. This means that elements with *extreme values are ignored altogether*.

To study its effect we employ again a noisy square wave like in the beginning of this lecture (see the waveforms below) and observe how it transforms the data in a very different way than the linear averaging algorithm.

The periodic square signal is plotted in the time domain again and is visibly affected by noise spikes. Again the raw data are recorded in the array (y_n) with indexing n. The result of the median filter is stored in the array (z_n) indexed in the same way. A *median filter* then can be implemented as:

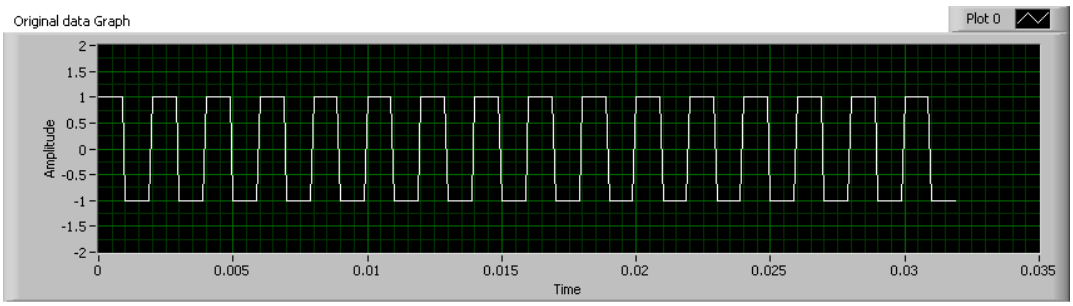
$$z_n = \text{median}(y_{n+1}, y_n, y_{n-1})$$

Or more generally for a window of N samples as:

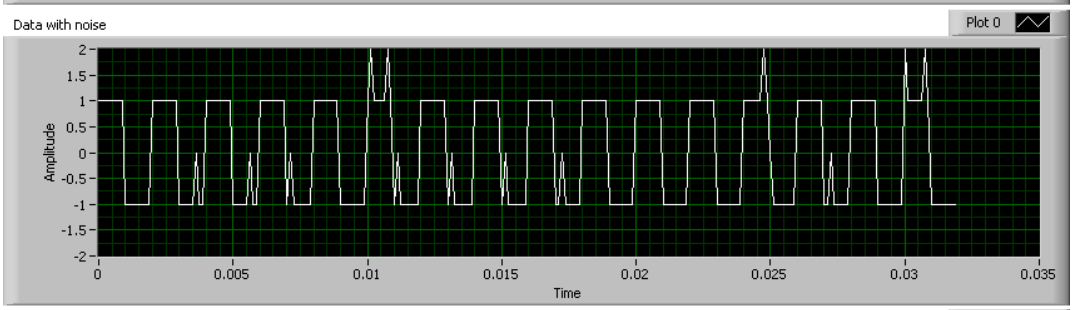
$$z_n = \text{median}(y_{n+(N-1)/2}, \ldots, y_n, \ldots, y_{n-(N-1)/2})$$

The result (third trace in the plot below) is impressive. The noise is almost gone and not only the amplitude and frequency can be reconstructed but also the *shape of the waveform* is maintained. This filter is much better suited to remove this spiky kind of noise from the signal than the simple averaging method in the beginning of the lecture.

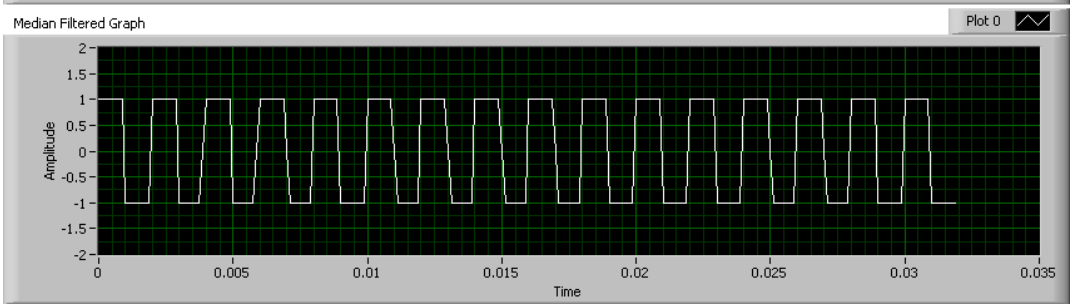
Clean signal



Signal + Noise



Median of 7 samples



This lecture course has covered quite a breadth of topics: from digital logic, via analogue linear electronics, first for DC then extended to AC circuits, moving to non-linear components and their use in operational amplifiers and finally closing the circle with analog-to-digital conversion and a glimpse on digital signal processing. Naturally this course had to cut short in the depth of the topics. But hopefully you will have found it educative, giving you a good start in this field, whether you continue to use it in the future or not. Even if you do not deepen further your knowledge in electronics, at least this course should have given you some understanding what your colleagues are talking about when the discussion comes to the implementation of your data acquisition project into a real-life system. And you should have got some insight in the internal processes taking place in modern consumer electronics.