DFT: Discrete Fourier Transform & Linear Signal Processing

2nd Year Electronics Lab

IMPERIAL COLLEGE LONDON

Table of Contents

Equipment

- Lab Computer with **MATLAB**

Aims

This experiment leads you through the theory of **Discrete Fourier Transforms** and **Discrete Time Signal Processing**. These have many similarities to **Continuous Fourier Transforms**, covered in Year 1, but are not identical.

- Familiarization with **MATLAB**
- Able to generate discrete signals in **MATLAB**
- Perform analysis on signals by applying Fourier Transforms and Windows
- Perform analysis on digital filters and their responses

Objectives

- Derive equations for the 3 types of **Discrete Fourier Transforms**.
- Generate sinusoidal signals in **MATLAB** as vectors and investigate the effects of **DFT** and **Windowing**. You will then use these techniques to investigate an unknown signal (provided).
- Filter noise from the unknown signal by removing unwated frequencies in the frequency domain.
- Investigate the effects of passing **Pulse** and **Impulse** signals through a digital filter.
- Investigate simple digital filter and their responses (**FIR** and **IIR**).

Recommended Textbooks

This experiment is designed to support the second year course Signals and Linear Systems (EE2-5). Since the laboratory and the lectures may not be synchronized to each other, some of you may need to learn certain aspects of signal processing ahead of the lectures. While the notes provided in this experiment attempt to explain certain basic concepts, it is far from complete.

You are recommended to refer to the excellent textbook '**An Introduction to the Analysis and Processing of Signals**' by Paul A. Lynn. A more advanced reference is '**Discrete-Time Signal Processing**', by A.V. Oppenheim & R.W. Schafer, as well as the 'Signals and Linear Systems' course notes. Another good recommendation is '**Signals and Systems (2nd Edition)**' by A.V. Oppenheim, Alan S. Willsky with S. Hamid Nawab.

Recommended Timetable

There are 9 exercises in this handout; you should aim to complete all of them in the 4 timetabled lab sessions.

1. Introduction

Apart from teaching you some fundamentals of **Digital Signal Processing (DSP)**, this experiment also introduces you to **MATLAB**, a mathematical tool that integrates numerical analysis, matrix computation and graphics in an easy-to-use environment. **MATLAB** is highly interactive; its interpretative nature allows you to explore mathematical concepts in signal processing without tedious programming. Once grasped, the same tool can be used for other subjects such as circuit analysis, communications, and control engineering.

Exercise 1 – Learning MATLAB

Launch **MATLAB** from your computer.

NOTE: the online help is readily available within **MATLAB** by simply typing **`help`** at the cursor input '**>**'.

Let us create a sampled sine wave at a frequency f_{sig} of 1KHz. We shall be using a sampling frequency of 25.6KHz and generating 128 data points. How many cycles of the sine wave will there be?

To do this, a list of time points are needed to be created (0, $1/f_{samp}$, $2/f_{samp}$, …, $127/f_{samp}$). In **MATLAB**, create a vector with 128 elements by entering the following:

```
% Define constants and the time vector t
fsamp= 25600
fsig = 1000
tsamp = 1/fsamp
t = 0 : tsamp : 127*tsamp;
```

MATLAB tips:

- You do not need to declare variables.
- Anything between `%` and newline is a comment.
- The result of the **MATLAB** statement is echoed unless the statement terminates with a semicolon '**;**'.
- The colon '**:**' has special significance. For example, **`x=1:5`** creates a row vector containing five elements, vector x=[1 2 3 4 5]. In this example it creates a vector **`t`** with the first element 0, the last element 127*t_{samp}, and with elements in between at an increment of t_{samp}.

Now try the command **`whos`** to check which variables are defined at this stage and also how large they are (matrices are reported as row by column).

To create the sine wave **y** and plot it against **t** simply enter:

```
y = sin(2*pi*fsig*t);
plot (y)
```

This plots **y** against the data number (from 1 to 128). Alternatively use **plot(t, y)** for plotting **y** against **t**. To add more thrills, try:

```
grid
title ('Simple Sine Wave')
xlabel ('Time')
ylabel ('Amplitude')
```

Obtain hardcopy of graphics with the **print** command. You may also plot the sine wave as points by using **plot(t, y, '*')** and change the colour of the curve to red with **plot(t, y, 'r')**. The output can be given as a discrete signal using the command **stem(y)**. The plot window can also be divided with the **subplot** command and the **zoom** command allows examination of a particular region of the plot (use help to discover more).

Sine waves are such useful waveforms that it is worthwhile to write a function to generate them for any signal frequency, sampling frequency and any number of samples. In **MATLAB**, functions are individually stored as a file with extension '**.m**'. To define a function **sinegen**, create a file **sinegen.m** using the **MATLAB** editor by selecting **New M-file** from the **File** menu.

```
function y = sinegen(fsamp, fsig, nsamp)
••••• body of the function •••••
end
```

Save the file in your home directory with the **Save As** option from the **File** menu. To test **sinegen**, simply enter:

```
s = sinegen(•••••SUPPLY PARAMETERS•••••);
```

If **MATLAB** gives the error message **undefined function**, use the **cd** command to make your home directory the local directory.

Exercise 2 – Three forms of Fourier Transforms

We have seen how **MATLAB** can manipulate data as a 1x128 vector. Now let us create a matrix **S** containing 4 rows of sine waves such that the first row is frequency f_{sig}, the second row is the second harmonic 2*f_{sig} and so on. **nsamp** should be set to equal 128.

```
S(1,:) = sinegen(fsamp, fsig, nsamp);
S(2,:) = sinegen(fsamp, 2*fsig, nsamp);
S(3,:) = sinegen(fsamp, 3*fsig, nsamp);
S(4,:) = sinegen(fsamp, 4*fsig, nsamp);
```

Note the use of '**:**' as the second index of **S** to represent all columns. Examine **S** and then try plotting it by entering **plot(S)**. Is this plot correct? If not, why? Plot the transpose of **S** by using **plot(S')**.

We can do the same thing to the above 4 statements by using a for-loop. The following statements creates **S** with 10 rows of sine waves:

```
for i = 1:10
S(i,:) = sinegen(fsamp, i*fsig, nsamp);
end
```

Next let us explore what happens when we add all the harmonics together. This can be done by first creating a row vector **p** containing 'ones', and then multiplying this with the matrix **S**:

```
p = ones(1,10);
f = p*S;
plot(f)
```

This is equivalent to calculating:

$$f_k(t) = \sum_{n=1}^{10} \sin(n\omega_s t_k) \qquad \textbf{(Eqn.1)}$$

Where ω_s = fundamental frequency and t_k = [0, tsamp, 2tsamp, ..., 127tsamp]. Explain the result **f**.

Instead of using a unity row vector, we could choose a different weighting **bn** for each harmonic component in the summation:

$$f_k(t) = \sum_{n=1}^{10} b_n \sin(n\omega_s t_k)$$

Try using **bn** = [1 0 1/3 0 1/5 0 1/7 0 1/9 0]. What do you get? What should **bn** be if we want to produce a saw-tooth waveform (use 10 terms only)? You may not be able to derive this during the lab session. Look it up in the library or work it out from first principles at home.

So far we have been using sine waves as our basis functions. Let us now try using cosine signals. Create a 10x128 matrix **C** contain 10 harmonics of cosine waveforms. Use the weighting vector **an** = [1 0 -1/3 0 1/5 0 -1/7 0 1/9 0], compute and examine **g=an*C**.

How does **g** differ from **f** obtained earlier from using sine waves? What general conclusions can you make about the sine and cosine series with relation to the even and odd nature of the signal?

The basis of the Fourier series is that a complex periodic waveform may be analyzed into a number of harmonically related sinusoidal waves, which constitute an orthogonal set. If we have a periodic signal $f(t)$ with a period equal to T, then $f(t)$ may be represented by the series:

$$f(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(n\omega_s t) + \sum_{n=1}^{\infty} b_n \sin(n\omega_s t) \qquad \textbf{(Eqn. 2)}$$

Prove that this equation can be rewritten as:

$$f(t) = a_0 + \sum_{n=1}^{\infty} c_n \cos(n\omega_s t - \phi_n) \qquad \textbf{(Eqn. 3)}$$

Derive the equations for c_n and ϕ_n in terms of a_n and b_n. What is the form of Eqn. 3 with **sine** as the orthogonal set?

From the above discussion, we have shown that almost any periodic signal can be expressed either as a set of sine and cosine waves of appropriate amplitude and frequency, or as a sum of sinusoidal components which are harmonically related and are defined by their *amplitudes* and *relative phases*.

A third form of the Fourier series, the exponential form, can be obtained by substituting the first of the following identities into Eqn. 3:

$$\cos(x) = \frac{1}{2}(e^{jx} + e^{-jx}) \qquad \text{and} \qquad \sin(x) = \frac{1}{2}(e^{jx} - e^{-jx})$$

Which gives:

$$f(t) = \sum_{n=-\infty}^{\infty} A_n e^{jn\omega_s t} \qquad \textbf{(Eqn. 4)}$$

Show the following:

- The coefficients of the exponential series are in general complex.
- The imaginary part of a coefficient A_n is equal but opposite in sign to that of coefficient A_{-n}.

In our discussion so far, we have deliberately omitted two important issues:

- How can the coefficients A_n be determined?
- What happens if the signal is not periodic?

These two questions will be dealt with in the lectures. It is sufficient for this experiment to state that the coefficients A_n are given by:

$$A_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\, e^{-jn\omega_s t}\, dt \qquad \textbf{(Eqn. 5)}$$

We can further generalize Eqn. 4 and Eqn. 5 for a non-periodic signal by letting $T \to \infty$ and $\omega_s \to 0$. But we shall not consider this any further for this experiment.

Exercise 3 – DFT and Spectra of Signals

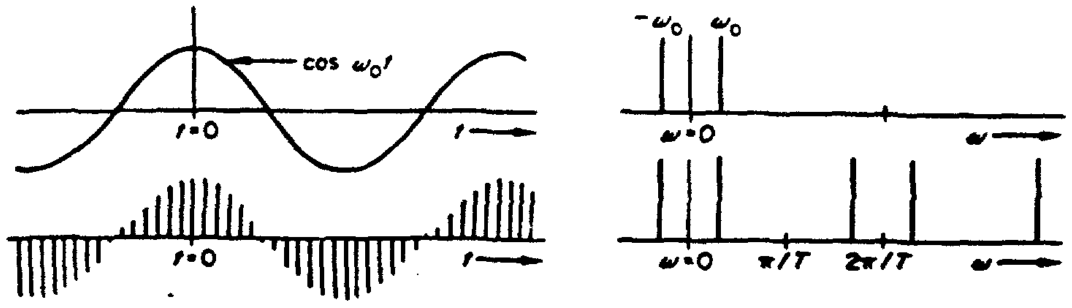

Figure 1 - A cosine signal and its spectrum

We have seen that a signal can be described either in the time-domain (as voltage samples) or in the frequency domain (as complex coefficients). The Discrete Fourier transform (DFT) defines the mapping between the discrete time domain and the discrete frequency domain of the sampled-data signals. The forward transform is from the time to the frequency domain, whereas the inverse transform is from the frequency to the time domain. Figure 1 shows a cosine signal in both continuous and sampled forms and their respective spectra. Note that the sampled-data spectra repeat indefinitely at intervals of $\omega = 2\pi/T$, where T is the sampling period. To calculate the spectrum of the periodic sample-data signal, we can rewrite the integral in Eqn. 5 above as a summation:

$$A_k(\mathrm{j}\omega) = \frac{1}{N}\sum_{n=0}^{N-1} X_n e^{-j(2\pi kn)/N} \qquad \textbf{(Eqn. 6)}$$

Where X_n are the discrete time-domain samples, n is the sample number, k is the harmonic number and N is the total number of samples of the signal.

The number of multiplications needed for an N-point DFT is proportional to N^2. The algorithm is said to be of order $O(N^2)$. Fast algorithms have been developed to calculate the DFT (called Fast Fourier Transforms or FFT), which require only approximately $N log_2 N$ multiplications.

Let us try the built-in FFT function to explore the spectra of various signals. The two functions **`fft(x)`** and **`ifft(x)`** implement the forward and inverse transforms respectively according to the following equations:

$$\text{Forward:} \quad A_{k+1} = \sum_{n=0}^{N-1} X_{n+1} e^{-j(2\pi kn)/N} \qquad \textbf{(Eqn. 7)}$$

$$\text{Inverse:} \quad X_{n+1} = \frac{1}{N} \sum_{k=0}^{N-1} A_{k+1} e^{-j(2\pi kn)/N} \qquad \textbf{(Eqn. 8)}$$

Where k is the harmonic or frequency bin number, n is the sample number and N is the number of samples in x. Note that the series is written in an unorthodox way, running over $n + 1$ and $k + 1$ instead of the usual n and k because vectors in **MATLAB** are indexed from 1 to N instead of from 0 to $N - 1$. Note also that Eqn. 6 and Eqn. 7 differ by a factor of N. This is due to the fact that the transform can be defined mathematically in different ways. Eqn. 7 is a more popular formulation, but Eqn. 6 is easier to relate to the Fourier series.

Now let us find the spectrum of the sine wave produced with **`sinegen`**:

```
x = sinegen(8000,1000,8);
A = fft(x)
```

Explain the numerical result in **`A`**. Make sure that you know the significant of each number you get back. If in doubt, ask a demonstrator. You might also like to evaluate the FFT of a cosine signal, is it what you would expect?

Next, let us try to find the spectrum for an impulse with 16 samples:

```
x(1:16) = zeros(1,16);
x(1) = 1;
```

Examine the spectrum of **`x`** in terms of real and imaginary parts and magnitude and phase. Then, delay this impulse signal by 1 sample and find its spectrum again. Examine the real and imaginary spectra. Why are they different for the 2 signals? What do you expect to get if you delay the impulse by 2 samples instead of 1?

Instead of working with real and imaginary numbers, it is often more convenient to work with the magnitude and phase of the components, as defined by:

```
amp = sqrt ( A .* conj (A));
phase = atan2( imag(A), real(A));
```

If you precede the `*` or `/` operators with a period (`.`), **MATLAB** will perform an element-by-element multiplication or division of the two vectors instead of the normal matrix operation. **`conj(A)`** returns the conjugate of each elements in **`A`**. Investigate how the phase spectrum changes as you gradually increase the delay of the impulse; you might try the **`unwrap`** function.

Write a function **`plotspec(x)`** which plots the magnitude and phase spectra of **`x`** (the magnitude spectrum should be plotted as a bar graph). This function will be useful later.

Exercise 4 – The effects of using Windows

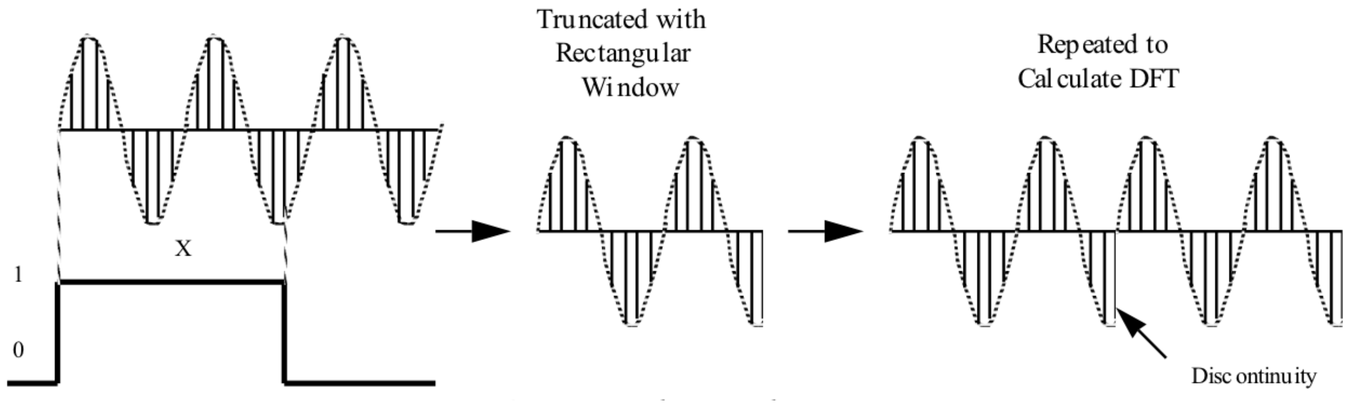

Figure 2 - Rectangular Window

So far we have only dealt with sine waves with an integer number of cycles. Generate a sine wave with an incomplete last cycle using:

```
x = sinegen(20000,1000,128);
```

Obtain its magnitude spectrum. Instead of obtaining just two spikes for the positive and negative frequencies, you will see many more components around the main signal component. Since we are not using an integer number of cycles, we introduce discontinuity after the last sample as shown in Figure 2. This is equivalent to multiplying a continuous sine wave with a rectangular window as shown in Figure 2. It is this discontinuity that generates the extra frequency components. To reduce this effect, we need to reduce or remove the discontinuities. Choosing a window function that gradually goes to zero at both ends can do this.

Two common window functions are defined mathematically as:

$$\text{Hanning Window:}\quad w[n] = 0.5\left(1-\cos\left(2\pi\frac{n}{N-1}\right)\right)$$

$$\text{Blackman Window:}\quad w[n] = 0.42 - 0.5\cos\left(2\pi\frac{n-1}{N-1}\right) + 0.08\cos\left(4\pi\frac{n-1}{N-1}\right)$$

Where $n = 1, 2, \dots, N$ for both windows.

MATLAB provides a number of window functions. Compare the magnitude spectrum of the sine wave without windowing to those weighted by the **Hanning** and **Blackman** window functions. Comment on the results.

Exercise 5 – The analysis of an unknown signal

To test how much you have understood so far, you are given an unknown signal stored in a disk file. The file contains over 1000 data samples. It is also known that the signal was sampled at 1 kHz and contains one or more significant sine waves. Take at least two 256-sample segments from this data and compare their spectra. Are they essentially the same? Make sure that you can interpret the frequency axis (ask a demonstrator if you are not sure). Find out the frequency and magnitude of its constituent sine waves. You can load the data into a vector using the **MATLAB** command:

```
load unknown;
```

2. What is a Digital Filter?

The purpose of a digital filter, as for any other filter, is to enhance the wanted aspects of a signal, while suppressing the unwanted aspects. In this experiment, we will restrict ourselves to filters that are linear and time-invariant. A filter is linear if its response to the sum of the two inputs is merely the sum of its responses of the two signals separately, while a time-invariant filter is one whose characteristics do not alter with time. We make these two restrictions because they enormously simplify the mathematics involved. Linear time-invariant filters may also be made from analogue components and it is often cheaper to do so. The advantages of digital filters lie in their repeatability with time, i.e. they do not suffer from ageing or temperature effects, their ability to handle very low frequencies and the ease with which their parameters may be altered. In addition, non-linear or time varying filters are very much easier to implement as digital filters than as analogue ones.

Exercise 6 – Filtering in the Frequency Domain

An obvious way to filter a signal is to remove the unwanted spectral components in the frequency domain. The steps are:

1. Take the FFT of the signal to convert it into its frequency domain equivalent.
2. Set unwanted components in the spectrum to zero.
3. Take the inverse FFT of the resulting spectrum.

Filter the noise-corrupted signal by removing all components above 300 Hz. Compare the filtered waveform with the original signal. What are the disadvantages of filtering in the frequency domain?

3. Impulse Response and Convolution

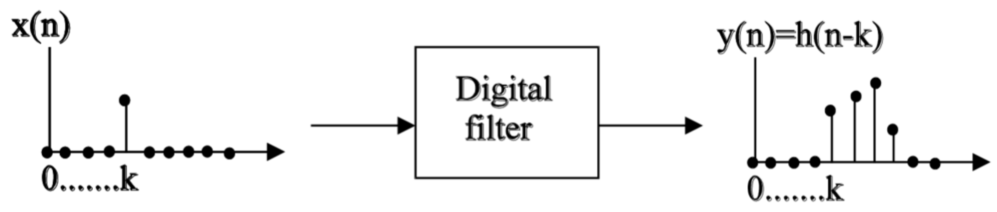

Figure 3 - Impulse response of a Filter

Let us represent the input sequence applied to a digital filter by **x(n)** and the corresponding output sequence by **y(n)**. Suppose we apply an impulse to the filter at a particular time, **n = k** as shown in Figure 3. The output sequence **y(n)** will be zero for **n < k** and may have some value for **n > k**. The sequence of numbers **h(n)** where **n = 0, 1, 2, ..., ∞** is called the impulse response of the system. In Figure 3 the impulse response is shifted to start at **k**, i.e. **h(n-k)**, due to the shift in the input. Once the impulse response is known, the output sequence of a filter can be evaluated for any input signal. We can illustrate this by regarding the input sequence as the sum of many impulses occurring at different sampling intervals, the height of the sample at position **k** is **x(k)**. Since the filter is linear, each impulse causes an output **h(n-k)*x(k)** starting at position **k**. Therefore the total output **y(n)** is just the sum of the contributions from all the impulse responses from **x(n)**, **x(n-1)**, **x(n-2)**, ... **x(0)**, as depicted in Figure 4. Thus:

$$\mathrm{y}[\mathrm{n}] = \sum_{k=0}^{n} h[n-k] * x[k] \qquad \textbf{(Eqn. 9)}$$

If we substitute r for n-k, this becomes:

$$\mathrm{y}[\mathrm{n}] = \sum_{r=0}^{n} h[r] * x[n-r] \qquad \textbf{(Eqn. 10)}$$

Thus for any linear time-invariant filter, the output values consist of a weighted sum of the past input values, with the weights being the elements of the impulse response. The operation described by Eqn. 10 is known as discrete convolution. Here, the input signal **x(n)** is said to be convolved with the impulse response of the filter **h(n)**.

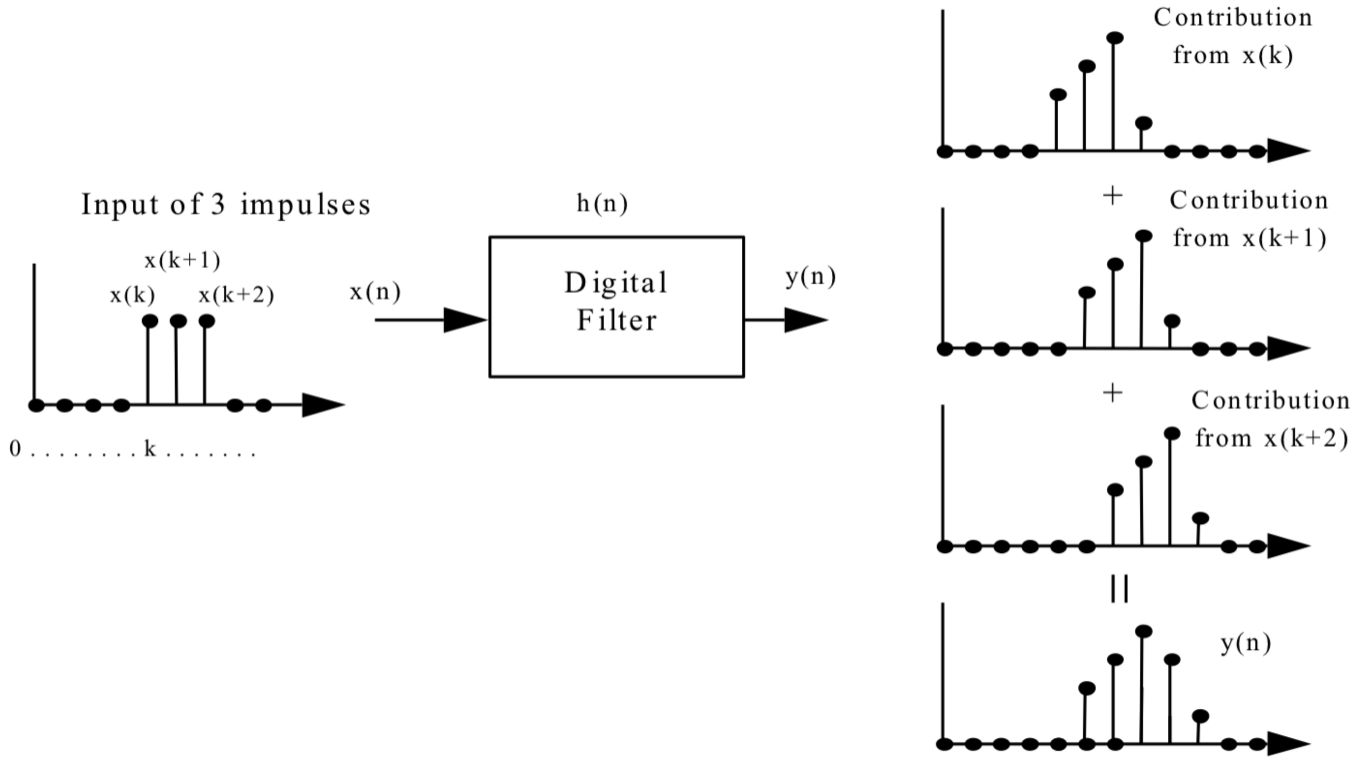

Figure 4 - Discrete Convolution

Exercise 7 – Spectra Pulse and Impulse

Create a pulse signal containing 8 samples of ones and 8 samples of zeros. Obtain its amplitude spectrum and check that it is as you expected. Gradually reduce the width of the pulse until it becomes an impulse, i.e. contains only a single one and 15 zeros. Note how the spectrum changes. Remember the effect of adding a larger and larger number of sine waves together in exercise 2?

4. Frequency Response of a Filter

In the previous exercise, we have seen that an impulse signal contains equal power in all frequency components, i.e. its magnitude spectrum is flat. Therefore the frequency response of a filter is simply the discrete Fourier transform of the filter's impulse response.

Exercise 8 – A simple Finite Impulse Response (FIR) Filter

We are now ready to try out a simple digital filter. Let us assume that the impulse response of the filter **h(n)** is a sequence of 4 ones. Find out the frequency response of this filter. This filter keeps the low frequency components intact and reduces the high frequency components. It is therefore known as a low-pass filter. Suppose the input signal to the filter is **x(n)**, show that this filter produces the output **y(n)** as:

$$y[n] = x[n] + x[n-1] + x[n-2] + x[n-3] \qquad \textbf{(Eqn. 11)}$$

Let us apply this filter to the noise-corrupted signal used in exercise 5. This can be done using the `conv(x)` function in **MATLAB** to perform a convolution between the signal and **h(n)**. Compare the waveform before and after filtering.

Another way of looking at this filter is that the output is produced by averaging 4 consecutive samples. This averaging action tends to smooth out fast varying frequency components, but has little effects on the low frequency components. Such a filter is often termed as a moving average.

Exercise 9 – Infinite Impulse Response (IIR) Filter

The filter described in the last exercise belongs to a class known as **Finite Impulse Response** (**FIR**) filters. It has the property that all output samples are dependent on input samples alone. For an impulse response that contains many non-zero terms, the amount of computation required to implement the filter may become prohibitively large if the output sequence (**y(n)**) is computed directly as a weighted sum of the past input values. Instead, **y(n)** is often computed as a sum of both past input values and past output values. That is:

$$\mathrm{y[n]} = \sum_{r=0}^{nb} b[r] * x[n-r] + \sum_{r=1}^{na} a[r] * y[n-r] \qquad \textbf{(Eqn. 12)}$$

This type of filter has an impulse response of infinite length, therefore it is known as an **Infinite Impulse Response** (**IIR**) filter. It is also called a recursive filter because the outputs are being fed back to calculate the new output.

Derive the impulse response of the simple filter described by the following equation:

$$y[n] \; = \; 0.5x[n] + 0.5y[n-1] \qquad \textbf{(Eqn. 13)}$$

MATLAB provides a function called **`filter`** to perform general **FIR** and **IIR** filtering. It essentially implements Eqn. 12 with the syntax:

```
y = filter(b, a, x)
```

Where **b** = [b_0 b_1 ... b_n] and **a** = [1 a_1 a_2 ... a_n].

Filter the noise corrupted signal with an **IIR** filter having the following filter coefficients:

b = [0.0039 0.0193 0.0386 0.0386 0.0193 0.0039]
a = [1.0000 -2.3745 2.6360 -1.5664 0.4929 -0.0646]

Find the impulse response of this filter and hence its frequency response. Hence explain the effect of filtering the noise-corrupted signal with this filter.

This filter (known as **a Butterworth Filter**) was designed using the **MATLAB** functions **`buttord`** and **`butter`**. If you have time, you might explore and design your own filters.