UNIT 1: PROGRAMMING ENVIRONMENT

BASICDSP

1.1 Introduction

This unit introduces the programming environment for the Basic Digital Signal Processing course. It gives a brief description of the Visual Basic classes available, and reviews the support for signal access, complex arithmetic and graphics that are used in the DSP units.

When you have worked through this unit you should:

- understand the three basic classes used in the course
- know how to use this document for reference information
- know how to manipulate complex numbers in your programs
- know how to create and manipulate signals, waveforms, spectra and complex waveforms
- know how to plot graphs
- know how to play and record signals
- have tried out a simple programming example

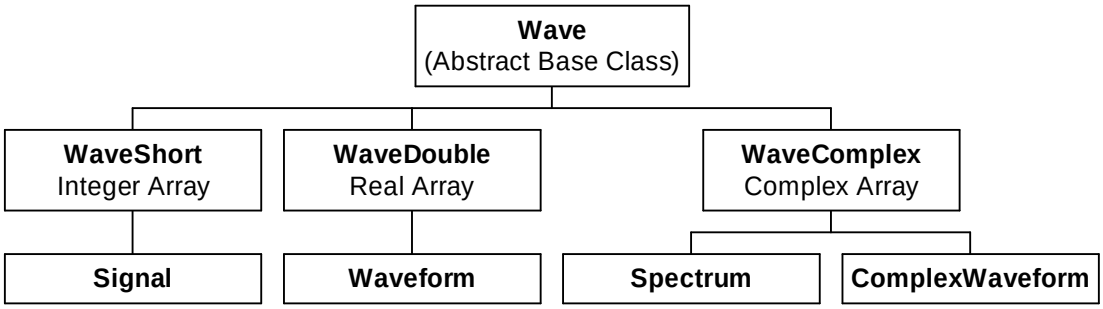
1.2 DSP Classes

The DSP classes used for the course were developed to provide a simple and clean programming environment for the definition and demonstration of signal processing algorithms. To a great extent the complexity of writing applications that create graphs and replay sound is hidden by the classes provided, so that the implementation details of the DSP concepts themselves are not obscured. Through support for file I/O, replay, acquisition and graphics, the classes also make demonstration programs succinct and easy to understand.

There are three main base classes:

- `Complex` support for complex arithmetic
- `Wave` support for waveform vectors (one-dimensional arrays)
- `Graph` support for simple chart-style graphics

The `Complex` and `Graph` classes are straightforward and described in more detail below. The `Wave` class is an abstract base class from which a small hierarchy of classes has been developed:



The implementation classes `WaveShort`, `WaveDouble` and `WaveComplex` provide an implementation of basic construction, assignment, element access, concatenation, and partitioning of arrays of integers, floating-point numbers and complex numbers respectively. The lowest level of classes provide convenient objects for manipulation in DSP algorithms:

`Signal`	For reading, writing, recording, and replaying amplitude-time waveforms
`Waveform`	For manipulation of amplitude-time waveforms
`Spectrum`	For complex amplitude-frequency objects, like a spectrum or a frequency response
`ComplexWaveform`	For complex amplitude-time waveforms

The other classes within BasicDSP provide library functions for a number of signal processing operations. We will meet these later in the course:

`LTISystem` `LTISystemChain`	For manipulating linear systems
`Sample`	For sampling from various signal sources
`Window`	For generating signal windows
`Filter`	For designing filters
`DFT`	For performing Fourier analysis
`LPC`	For performing linear prediction analysis

1.3 Complex Class

Complex numbers consist of two double-precision floating-point numbers, representing the real part and the imaginary part of the complex number (see Unit 2 for a mathematical introduction). Complex numbers may be declared as:

```
Dim c As Complex
Dim d As New Complex(1)
Dim e As New Complex(2,3)
Dim f As Complex=e
```

Here complex number c has undefined values for its real and imaginary parts. Complex number d has its real part set to 1.0 and its imaginary part set to 0.0. Complex number e has its real part set to 2.0, and its imaginary part set to 3.0. Complex number f is set to be the same as e.

Arithmetic support is available for addition, subtraction, multiplication and division of complex numbers through overloading of the standard operators:

```
c = d + e
c = d - e
c = d * e
c = d / e
```

Mixed expressions with integer and double expressions are automatically converted to Complex. The real and imaginary parts of complex numbers can be accessed and changed through these properties, of type Double:

`c.Real`	Real part of complex number c
`c.Imag`	Imaginary part of complex number c

The magnitude and argument of a complex number can be accessed through these read-only properties, of type Double:

`c.Mag`	Magnitude of complex number c
`c.Arg`	Argument of complex number c (in radians)

The following shared functions are also available in the Complex class. Each returns a Complex value:

`Complex.Sqrt(c as Complex)`	Square root of complex number c
`Complex.Exp(c as Complex)`	Exponential of complex number c

Complex numbers can be also displayed or printed using the overloaded `ToString()` method.

1.4 Wave Class Hierarchy

Conceptually, the Wave classes support one-dimensional arrays of numerical values that have an associated temporal or frequency parameter. Thus a waveform is an array of sample values at a particular sample rate. The classes allow the indexing of the elements of the array, concatenation of compatible arrays, and partitioning of arrays. The arrays are created dynamically to a particular size, but they may also be 'grown' in size.

The Wave abstract base class provides the following interface through member functions:

`wv.Count as Integer`	Number of samples in wv
`wv.First as Integer`	Index of first sample in wv
`wv.Last as Integer`	Index of last sample in wv
`wv.Rate as Double`	Sampling rate of wv (samples/sec)
`wv.Period as Double`	Sampling period of wv (sec)

The implementation classes WaveShort, WaveDouble and WaveComplex provide typed arrays with appropriate indexing to access individual elements:

`sample = wv(idx)`	Gets the sample value at index idx
`wv(idx) = sample`	Sets the sample value at index idx

These indexing operators are protected against range errors. Attempts to index elements outside the current size of the arrays is guaranteed to return the value 0. Attempts to set values of elements outside the current size of the array are safely ignored. This characteristic of the Wave classes is used to simplify some of the signal processing algorithm implementations.

Compatible wave objects may be concatenated using the & operator, for example:

```
Dim iwv1 as WaveShort, iwv2 as WaveShort, owv as WaveShort
...
owv = iwv1 & iwv2
```

This code creates an output object owv that is the concatenation of objects iwv1 and iwv2 which must match in type and sampling rate. Wave objects may also be grown one sample at a time using the Add() method:

```
Dim owv as New WaveShort(0,10000)
...
owv.Add(123)
```

Every wave object also supports the selection of a sub-array through the member function

`Cut(start_sample,number_of_samples)`

For example, this cuts out 100 samples starting at index 500:

```
Dim iwv As New WaveShort(1000,10000)
Dim owv as WaveShort = iwv.Cut(500,100)
```

Signal Class

The `Signal` class supports quantised (short integer) time series, a format that is used to import and export waveforms from/to disk or from/to analogue signals.

A typical signal object is constructed as follows:

```
Dim sig As New Signal(1000,20000)
Dim sig As New Signal("sig.wav")
```

The first statement constructs a signal of 1000 samples at a sampling rate of 20,000 samples/second. The samples are indexed from 1 to 1000. The second statement creates a signal from a stored waveform on disk, using the Load() method.

Signals know how to play themselves and how to load/save themselves from/to disk files through these member functions:

`sig.Replay()`	replay signal sig through PC audio
`sig.LoadWaveFile(filename)`	load signal from WAV file on disk into sig
`sig.SaveWaveFile(filename)`	save signal to WAV file on disk from sig

The library uses Microsoft WAV file format for disk files. Only Linear PCM coding is supported.

Alternatively you can record a new signal using the default audio acquisition set up on the PC. You start the recording with the RecordStart() method, then wait for recording to complete with the RecordWait() method, and finally get the recorded samples with the RecordDone() method. Here is an example:

```
Dim wv As New Signal(0, SAMPRATE)
wv.RecordStart(10 * SAMPRATE)  ' record 10 seconds
wv.RecordWait()
wv.RecordDone()
```

A Signal object can be converted into a Waveform object, using its `Float()` method:

```
Dim sig as New Signal("sig.wav")
Dim wav as Waveform=sig.Float()
```

Waveform Class

The Waveform class is used as a generic container for discrete time series used in DSP applications. Waveform objects are indexed from 1 since this is the convention found in many DSP texts. Thus the constructor

```
Dim wav as New Waveform(1000,20000)
```

creates an array of 1000 floating point numbers from `wav(1)` to `wav(1000)` with an associated sample rate of 20,000 samples per second. Knowledge of the sample rate is used to label graph axes correctly.

A Waveform object can be converted to a Signal object using its Quantise() method:

```
Dim wav as New Waveform(1000,20000)
Dim sig1 as Signal = wav.Quantise(0.1)
Dim sig2 as Signal = wav.Quantise()
```

The first conversion uses a specified quantisation value for each quantised signal level. The second conversion performs an automatic scaling to best use the dynamic range available in the Signal object.

A Waveform object can be converted to a ComplexWaveform object using its Complex() method:

```
Dim wv as New Waveform(1000,20000)
Dim cwv as ComplexWaveform = wv.Complex()
```

ComplexWaveform Class

The `ComplexWaveform` class is used to contain complex time series, such as that produced by the inverse Fourier transform. Like Waveform objects, these are also indexed from 1.

```
Dim cwv as New ComplexWaveform(100,10000)
```

ComplexWaveforms can be converted to Waveforms in different ways, based on the Real part, the Imaginary part, the Magnitude or the Argument of the complex signal:

```
Dim wv1 as Waveform=cwv.Real()
Dim wv2 as Waveform=cwv.Imag()
Dim wv3 as Waveform=cwv.Mag()
Dim wv4 as Waveform=cwv.Arg()
```

Spectrum Class

The `Spectrum` class is a container for complex spectral information. Like the `ComplexWaveform` class, it contains an array of complex numbers, but the associated rate parameter specifies the frequency range spanned by the Spectrum object rather than a sampling rate. `Spectrum` objects are also indexed from 0. Thus the definition:

```
Dim spect as New Spectrum(1000,20000)
```

Represents a spectrum of 1000 samples extending from `resp(0)` at 0Hz, to `resp(999)` at 19,980Hz (each sample corresponds to 20000/1000 = 20Hz). To generate a suitable Spectrum object to plot the frequency characteristics of a linear system operating at an associated sample rate of SRATE samples/sec, choose enough samples to make a clear graph (say 1000) but choose an upper limit of half the sampling rate, like this:

```
Dim resp as New Spectrum(1000,(SRATE/2))
```

Since for real signals, the upper half of a spectrum object resulting from Fourier analysis is often discarded, the Spectrum class has a method that returns just the first half of the Spectrum array:

```
Dim sphalf as Spectrum = spect.Half()
```

1.5 Graph Class

The Graph class supports the graphical display of simple X-Y graphs. The class is implemented using the free ZedGraph control from www.zedgraph.org. This supports a wide range of charts and allows both printing of charts and saving of charts to image files.

Graph objects are constructed with a certain number of sub-graphs in the vertical and horizontal dimensions, and with an optional overall title. Individual graphs are then

positioned on this page by their index number (starting as graph number 1 for the top left graph, and incrementing left to right, top to bottom). Each sub graph may have individual titles.

The Graph constructor is usually called in the main Form_Load method. For example:

```
Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 3, "Example Graphs")
```

would create a page with 2 rows and 3 columns of graphs with the overall title 'Example Graphs'. The variable Me.CreateGraphics provides a handle to the underlying graphics engine, while "zgc" here is the name of the ZedGraph control on the form. If the number of rows and columns of graphs is not supplied, then a single graph is produced. The title is disabled if no title string is given. Here are some other possible constructor examples:

```
Dim gp As New Graph(Me.CreateGraphics, zgc, "Example Graphs")
Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 3)
Dim gp As New Graph(Me.CreateGraphics, zgc)
```

The Graph object has member functions for plotting Wave objects according to type. These are the most common. Refer to documentation for complete list.

```
Dim sig as New Signal(100,10000)
Dim wav as New Waveform(100,10000)
Dim cwv as New ComplexWaveform(100,10000)
...
PlotWaveShort(gno,sig,"Title","y-label","x-label")
PlotWaveDouble(gno,wav,"Title","y-label","x-label")
PlotWaveComplexMag(gno,cwv,"Title","y-label","x-label")
PlotWaveComplexArg(gno,cwv,"Title","y-label","x-label")
...
PlotSignal(gno,sig,"Title");
PlotWaveform(gno,wav,"Title")
PlotDbSpectrum(gno,cwv,"Title")
PlotPhaseSpectrum(gno,cwv,"Title")
```

1.6 Using BasicDSP Classes

The BasicDSP classes use the .NET framework and Visual Basic 2008. A free implementation of Visual Basic called Visual Basic 2008 Express Edition can be downloaded from Microsoft.

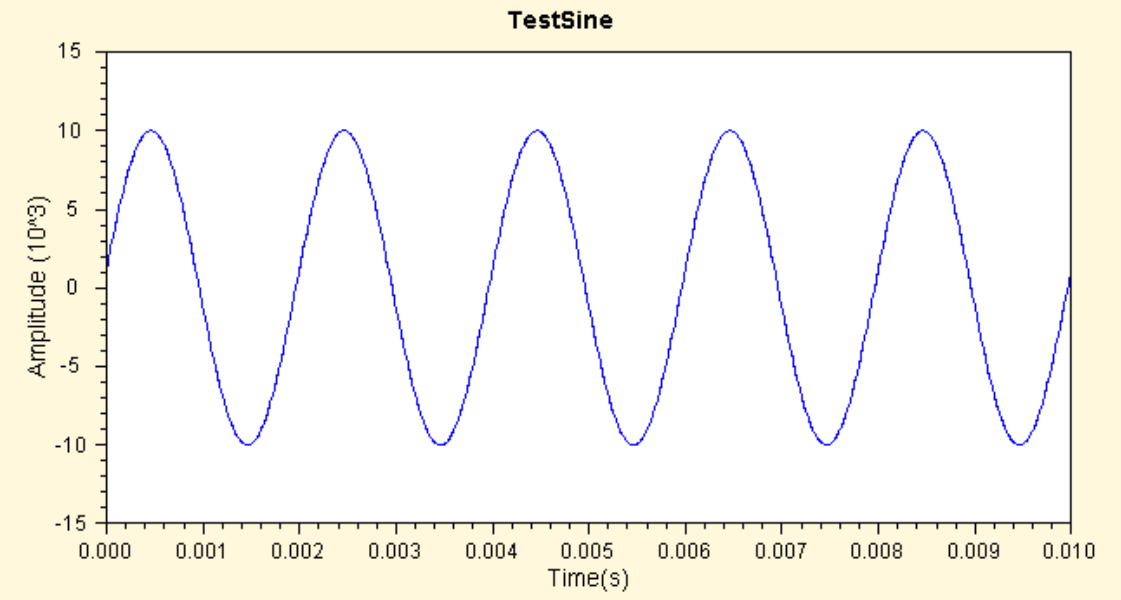
To build a VB application using BasicDSP you will need to download

- BasicDSP.DLL from the UCL Phonetics and Linguistics web site:
  http://www.phon.ucl.ac.uk/home/mark/basicdsp/
- ZedGraph.DLL from the ZedGraph web site at zedgraph.org

In VB, create a new Windows application project and add references to BasicDSP and ZedGraph. Then drag a ZedGraph control onto the form and set its Dock property so that it fills the form. Name the ZedGraph control "zgc". Add handlers for the form Load event and the ZedGraph control Click event.

The VB code for a simple application that calculates, displays and replays a sinewave is then just:

```
Imports BasicDSP
Imports ZedGraph
Public Class TestSine
  '
  Const NUMSAMPLE As Integer = 44100      ' number of samples
  Const SAMPRATE As Double = 44100.0      ' sampling rate
  Const SINEFREQ As Double = 500.0        ' sine at 500Hz
  Const SINEAMP As Double = 10000.0       ' sine amplitude
  '
  Dim wv as Signal
  '
  Private Sub TestSine_Load(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs) Handles MyBase.Load
    '
    wv = New Signal(NUMSAMPLE, SAMPRATE)
    For i As Integer = 1 To NUMSAMPLE
        wv(i) = SINEAMP * Math.Sin(2 * Math.PI * SINEFREQ*i/SAMPRATE)
    Next
    '
    Dim gp As New Graph(Me.CreateGraphics, zgc)
    gp.PlotSignal(1, wv.Cut(1, SAMPRATE / 100), "TestSine")
  End Sub
  '
  Private Sub zgc_Click(ByVal sender As Object, _
                   ByVal e As System.EventArgs) Handles zgc.Click
    wv.Replay()
  End Sub
End Class
```



1.7 Exercises

1.1 Type in, compile and run the example program above.

UNIT 2: MATHEMATICAL ENVIRONMENT

BASICDSP

2.1 Introduction

This unit introduces some basic mathematical concepts and relates them to the notation used in the course.

When you have worked through this unit you should:

- appreciate that a mathematical formalism can be expressed in an algorithmic language like Visual Basic as a procedure or function.
- know how array indexing is formalised in the notes and programs
- know how complex numbers arise and how to perform basic arithmetic on complex numbers
- understand the special characteristics of the multiplication of complex numbers when expressed in polar form
- know how to represent polynomials as a coefficient series and as a set of roots

2.2 Relating algebra to algorithms

The mathematical notation used to describe signal processing involves scalars, complex numbers and vectors; it exploits operations of addition, subtraction, multiplication and division, as well as exponents, logarithms, indexing and summation. Each of these elements of mathematical notation have equivalents in the Visual Basic computer language implementation.

Consider the formal expression for the calculation of the roots of a quadratic (the values of x where the equation is zero):

$$ax^2 + bx + c = 0 = (x - x_1)(x - x_2)$$

$$where\ x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a},\ x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

To map such mathematical notation onto a computer program, we need to identify the inputs and constants, the outputs and intermediary values, the operations and procedure of calculation. In this example the inputs are the constants a b c, the outputs x1 x2, the operations + - * / sqrt, and the procedure one of computing the expressions for the roots and their assignment to the variables x1 and x2. With a procedural wrapper:

```
Sub FindQuadRoot(ByVal a As Double, ByVal b As Double, ByVal c As
Double, ByRef x1 As Double, ByRef x2 As Double)

    x1 = (-b + Math.sqrt(b*b - 4*a*c))/(2*a)
    x2 = (-b – Math.sqrt(b*b - 4*a*c))/(2*a)

EndSub
```

Where the notation ByRef indicates the values that are returned to the calling program.

Note that this version will fail for quadratics where $4ac > b^2$.

To manipulate signals as entities, we need to represent them as *vectors*, that is as an array of simple scalar values. We also need to be able to identify single elements (samples) of the vector by subscripting or *indexing*. Although in conventional mathematical notation we might write a vector symbol as $\underline{\mathbf{x}}$, its expansion as $(x_1, x_2, \ldots, x_N)$ and an element of the vector as x_k, in this course we choose:

vector:	x[]
expansion:	x[1], x[2], ..., x[N]
element:	x[k]

This allows a straight-forward mapping to the Visual Basic notation for arrays and array indexing.

Consider the conventional mathematical expression for convolution. This takes two vectors, and returns a third which is a kind of vector product:

$$\underline{z} = \underline{x} * \underline{y}$$

$$where\ z_j = \sum_{k=-\infty}^{k=+\infty} x_k y_{j-k}$$

In this course such a formula would be expressed in terms of a procedure that takes signals x[] and y[] and calculates each output sample z[j] using:

$$z[j] = \sum_{k=0}^{\infty} x[k+1]y[j-k]$$

Here we not only simplify the notation but incorporate standard assumptions such as that signals are indexed from time 1, and are zero at earlier times.

This formulation leads to a natural Visual Basic implementation:

```
Public Shared Function Convolve(ByRef x As Waveform, _
                                ByRef y As Waveform) As Waveform
   Dim z As New Waveform(x.Count + y.Count - 1, x.Rate)
   Dim v As Double

   ' for each output sample
   For j As Integer = 1 To z.Count

       ' sum of product with reversed signal
       v = 0
       For k As Integer = 0 To h.Count() - 1
           v += x(k + 1) * y(j - k)
       Next
       z(j) = v
   Next
   Return z
End Function
```

The only difficult part with this translation is the selection of the upper limit for k in the summation loop, which should not be allowed to access samples before the start of the

y[] signal. We also assume that x[] is at least as long as y[].

2.2 Complex numbers

The fact that there are equations such as

$$x^2 + 3 = 0 \qquad x^2 - 10x + 40 = 0$$

which are not satisfied by any real value for x, leads to the introduction of complex numbers. A complex number is an ordered pair of real numbers, usually written in the convenient form x + *i*y, where x and y are real numbers. Complex numbers are subject to rules of arithmetic as defined below. We can of course refer to a complex number by a single algebraical variable, say z:

$$z = x + iy$$

The symbol *i* is called the *imaginary* unit (in engineering texts, it is sometimes referred to as *j*). The number *x* is called the *real part* of the complex number *z* and *y* is called the *imaginary* part of *z*. Thus the solutions for the equation x^2 - 4x + 13 = 0, may be written:

$$x^2 - 4x + 13 = 0 = (x - z_1)(x - z_2)$$
$$where\ z_1 = 2 + i3,\ z_2 = 2 - i3$$

Complex numbers can be represented as points on a plane where the horizontal or x-axis is called the real axis, and the vertical or y-axis is called the imaginary axis. Complex numbers are then points in a cartesian co-ordinate system on this plane, which is sometimes called the *complex plane*.

Two complex numbers

$$z_1 = x_1 + iy_1 \qquad z_2 = x_2 + iy_2$$

are defined to be *equal* if and only if their real parts are equal and their imaginary parts are equal, that is

$$z_1 = z_2 \ \ if\ and\ only\ if\ x_1 = x_2\ and\ y_1 = y_2$$

Relational expressions between complex numbers, such as $z_1 < z_2$, have no meaning, although the magnitudes of complex numbers may be compared, see below.

Addition. The sum $z_1 + z_2$ is defined as the complex number obtained by adding the real and imaginary parts of z_1 and z_2, that is

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$$

This addition is like the addition of vectors on the Cartesian plane.

Subtraction. The sum $z_1 - z_2$ is just the inverse of addition, that is

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$$

Multiplication. The product z_1z_2 is defined as the complex number

$$z_1 z_2 = (x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1 y_2 + x_2 y_1)$$

which is obtained by applying the normal rules of arithmetic for real numbers, treating the symbol *i* as a number, and replacing $i^2 = ii$ by -1.

Division. This is defined as the inverse operation of multiplication; that is the quotient $z = z_1/z_2$ is the complex number $z = x + iy$ which satisfies

$$z_1 = z\, z_2$$
$$(x_1 + iy_1) = (x + iy)(x_2 + iy_2)$$

For which a solution may be found by equating real and imaginary parts, assuming that x_2 and y_2 are not both zero:

$$x = \frac{x_1x_2 + y_1y_2}{x_2^{\,2} + y_2^{\,2}} \qquad y = \frac{x_2\, y_1 - x_1\, y_2}{x_2^{\,2} + y_2^{\,2}}$$

Expressions. For any complex numbers z_1, z_2, z_3 we have:

$z_1 + z_2 = z_2 + z_1$ and $z_1z_2 = z_2z_1$ — *Commutative laws*

$(z_1 + z_2) + z_3 = z_1 + (z_2 + z_3)$ and $(z_1z_2)z_3 = z_1(z_2z_3)$ — *Associative laws*

$z_1(z_2 + z_3) = z_1z_2 + z_1z_3$ — *Distributive law*

Conjugation. If $z = x + iy$ is any complex number, then $x - iy$ is called the *conjugate* of z, and is denoted by z*. The product of a complex number with its conjugate is a purely real number:

$$zz^* = (x + iy)(x - iy) = x^2 + y^2$$

Any complex number of the form $x + i0$ is just the real number x. Any complex number of the form $0 + iy$ is called a *pure imaginary number*.

Polar form of complex numbers. If we introduce polar co-ordinates r, θ in the complex plane by setting $x = r\cos\theta$, $y = r\sin\theta$, then the complex number $z = x + iy$ may be written

$$z = r\cos\theta + ir\sin\theta = r(\cos\theta + i\sin\theta)$$

This is known as the polar form of the complex number z. The value r is called the *absolute value* or *modulus* of z, denoted by |z|. Thus

$$|z| = r = \sqrt{x^2 + y^2} = \sqrt{zz^*}$$

The directed angle measured from the positive x axis to the direction of the complex number from the origin of the complex plane is called the *argument* of z, denoted by arg z. Angles are always measured counterclockwise and in radians. Note:

$$arg\ z = \theta = \arcsin\frac{y}{r} = \arccos\frac{x}{r} = \arctan\frac{y}{x}$$

By application of standard addition theorems of trigonometry we find the following relations for the polar form of the product of two complex numbers

$$|z_1 z_2| = |z_1||z_2|$$
$$arg(z_1 z_2) = arg\ z_1 + arg\ z_2$$

That is the magnitude of the product is the product of the input magnitudes, and the argument of the product is the sum of the input arguments. From these we can obtain the important result for the polar form of the powers of a complex number

$$z^n = r^n(\cos\theta + i\sin\theta)^n = r^n(\cos n\theta + i\sin n\theta)$$

This reduces to the so-called *De Moivre formula* for unity magnitude *z*:

$$(\cos\theta + i\sin\theta)^n = \cos n\theta + i\sin n\theta$$

From this we can see that the sequence of powers of a complex number of magnitude 1 are simply a sequence of counter-clockwise rotations by θ around the origin on the complex plane.

Complex exponential. The exponential of a real number x, written e^x or exp x, has the series expansion:

$$\exp x = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

The exponential function for complex z = x + *i*y is denoted by e^z and defined in terms of the real functions e^x, cos y and sin y, as follows:

$$e^x = e^{x+iy} = e^x(\cos y + i\sin y)$$

From this we obtain the *Euler formula* for imaginary *z*:

$$e^{iy} = \cos y + i\sin y$$

This in turn leads to the following important identities:

$$|eiy| = \sqrt{\cos^2 y + \sin^2 y} = 1$$
$$e^{2\pi i} = 1$$
$$e^{\pi i} = e^{-\pi i} = -1$$
$$e^{\frac{\pi i}{2}} = i$$
$$e^{\frac{-\pi i}{2}} = -i$$

The complex numbers $e^{i\theta}$, $0<\theta<2\pi$ lie on the unit circle of the complex plane; and the values 1, *i*, -1, -*i* are the points where the unit circle crosses the axes.

2.3 Polynomials

The Z-transform of a signal, which we will meet later, is a way of expressing a time series of samples as a single mathematical object: a polynomial in the variable *z* where the coefficients of the polynomial are simply the sample values. Here we consider some

basic operations on polynomials.

In its simplest form a polynomial may be expressed

$$a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

where the size of the polynomial *n* is called the *order* of the polynomial, and the constants a_0, a_1, etc are called the polynomial coefficients. Note that this expression may also be written

$$\sum_{j=0}^{n} a_j x^j$$

Operations such as addition, subtraction, multiplication and division may be performed on polynomials by elementary arithmetic and the collection of terms of equivalent powers of x. For multiplication, the product of a polynomial *p* of order *m* with a polynomial *q* of order *n* is a polynomial of order *m+n*. The division of a polynomial of order *m* by a polynomial of order *n* (where n < m), leaves a polynomial or order *m-n* and a remainder of degree *n-1*.

Polynomials may be reduced into a product of *factors*; a polynomial *p(x)* of order *n* having *n* factors:

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n = b_0 (x - b_1)(x - b_2)...(x - b_n)$$

For many purposes it is more convenient to work with the coefficients *b* than the polynomial coefficients *a*. This is because it easy to see that the values *b* are also the values of *x* for which the polynomial has the value 0. Thus the *b* coefficients are also called the *roots* of the polynomial p(x).

A polynomial of order *n* must have *n* roots, but these may not all be real. However if the coefficients of p(x) are real, any complex valued roots must occur in complex conjugate pairs. For polynomials of an order > 4, there exists no formula which directly calculates the values of the roots from the *a* coefficients, and so iterative numerical methods need to be used.

Finally, there is a close relationship between the form of polynomials and the form of *power series* approximations to functions. Here we just give some series expansions of some functions that may be of use later in the course:

$$(1 + x)^a = 1 + ax + \frac{a(a-1)}{2!}x^2 + \frac{a(a-1)(a-2)}{3!}x^3 + ... \; where \; |x| < 1$$

$$(1 - x)^{-1} = 1 + x + x^2 + x^3 + ... \; where \; |x| < 1$$

$$(1 - ax)^{-1} = 1 + ax + a^2 x^2 + a^3 x^3 + ... \; where \; |x| < 1, |a| < 1.$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

Exercise

2.1 Use the `Complex` class to implement a function that will solve arbitrary quadratic equations. In particular, write a program to accept the coefficients a, b and c, and which prints out the values of the two roots in complex notation.

UNIT 3:
DIGITAL WAVEFORMS

BASICDSP

3.1 Introduction

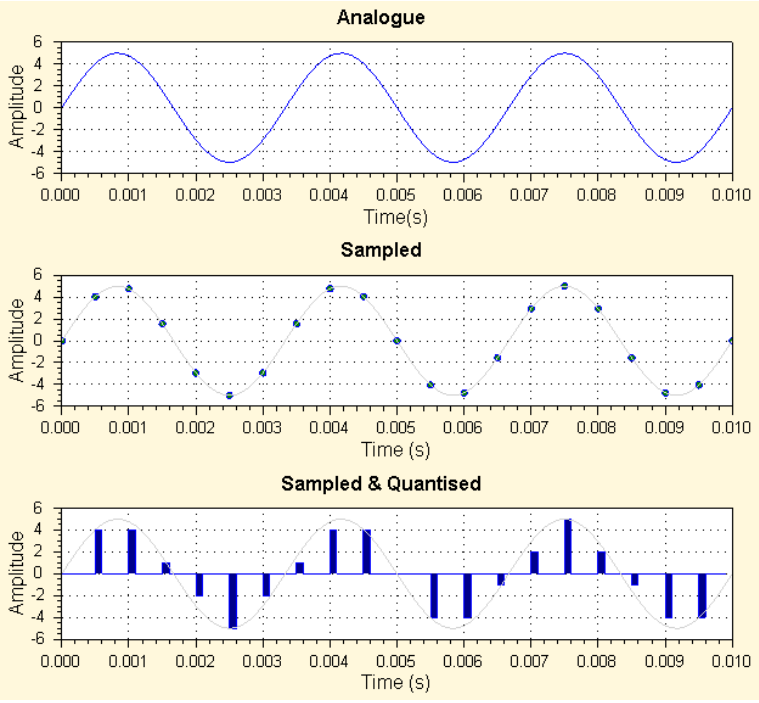
This unit is concerned with the representation and measurement of signals as digital waveforms: specifically with sampling, quantisation, the measurement of energy and the generation of sine and noise signals.

When you have worked through this unit you should:

- understand the concept of sampling
- know the limits to the setting of sampling frequency
- be able to state the sampling theorem
- be able to explain how and why aliasing occurs
- understand the concept of quantisation
- be able to explain the origin of quantisation noise
- understand how sinewaves and noise waveforms can be generated by a program
- have tried to create programs to build and replay waveforms to specification

3.2 Concepts

Sampling is the process whereby an analogue signal (continuous in time and continuous in value) is converted to a series of impulses of a size equal to the amplitude of the signal at regularly spaced instants of time. Related ideas are the *sample period* (T): the time between samples expressed in seconds, and *sampling frequency* (F_s=1/T): the number of samples taken per second of signal, usually expressed in units of samples per second or less accurately and more commonly in hertz (Hz).



Quantisation is the process of converting the sampled analogue signal (discrete-time continuous value) into impulses of discrete values of amplitude (analogous to the conversion from 'floating-point' values to 'integer' values in a program). Related ideas are *quantisation error*: the error in amplitude estimate introduced by quantisation, and *quantisation noise*: the noise introduced into a quantised signal by the approximation of the quantisation to the real analogue values. The most common forms of quantisation are *linear* and produce a binary code conveniently described in terms of numbers of bits. Each bit approximates to 6dB of additional signal/noise improvement. However, non-linear quantisation schemes are also used, primarily in telecommunication systems to make the best use of available capacity. In a logarithmic quantisation scheme, the amplitude levels are further apart at large amplitudes and closer at smaller amplitudes, thereby giving a quantisation error which is proportional to the size of the signal at that instant.

The *sampling theorem* (of Nyquist and Shannon) states that a signal which has a highest frequency component of f Hz must be sampled at a rate of at least 2f Hz if the signal is to be faithfully reconstructed from the digital samples. The consequence of not following the sampling theorem is *aliasing*, whereby the spectral components above f Hz are 'mapped' down to the frequency region 0..f Hz by the sampling process thereby distorting the representation. To prevent aliasing a *pre-sampling* or *anti-aliasing* filter is used to remove spectral components above half the sampling frequency; this is usually implemented as a low-pass filter of high order, with a corner frequency a little less than half the sampling frequency. A similar low-pass *reconstruction* filter is also used in digital-to-analogue conversion, to rebuild the analogue waveform from the digital samples in the frequency region 0..f Hz only.

To demonstrate formally that a sampled signal has many possible aliases as continuous waveforms, consider the sampled sinusoid:

$$x[n] = \sin(n\Omega)$$

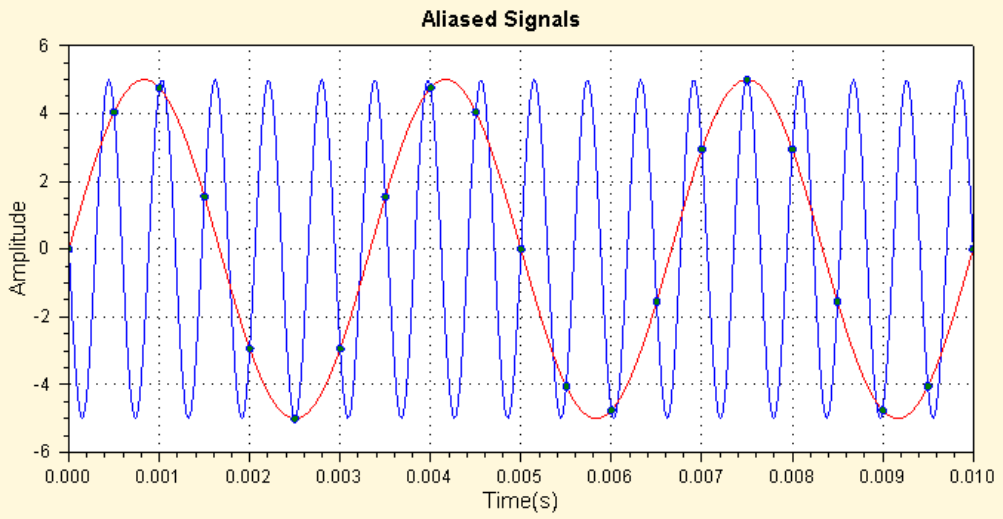
where Ω controls the frequency of the sinusoid (there are $2\pi/\Omega$ samples per period). Consider the sequence $y_1[n]$ obtained by increasing Ω by 2π:

$$\begin{aligned} y_1[n] &= \sin(n(\Omega + 2\pi)) \\ y_1[n] &= \cos(n\Omega)\sin(2\pi n) + \sin(n\Omega)\cos(2\pi n) \\ y_1[n] &= \sin(n\Omega) \end{aligned}$$

In other words, the same sequence as x[n]. Similarly the sequence $y_2[n]$ given by:

$$\begin{aligned} y_2[n] &= -\sin(2\pi n - n\Omega) \\ y_2[n] &= -[\cos(2\pi n)\sin(-n\Omega) + \sin(2\pi n)\cos(-n\Omega)] \\ y_2[n] &= -\sin(-n\Omega) \\ y_2[n] &= \sin(n\Omega) \end{aligned}$$

is also identical with x[n].



The value Ω is just the angular change per sample, expressed in radians. To convert this to a basis in time, we set Ω = ωT, where T is the sample interval, and ω is the *angular frequency* expressed in radians per second. ω is related to conventional frequency f (in hertz) by ω = 2πf. When f = F_s, then Ω = 2π. From this we can see that the aliases of a sinusoid at a frequency f are just mF_s+f and mF_s–f, (*m* is some integer), since:

$$x[n] = \sin(n2\pi fT)$$
$$y_1[n] = \sin(n2\pi(F_s + f)T)$$
$$y_2[n] = -\sin(n2\pi(F_s - f)T)$$

The *total energy* in a digital waveform is simply the sum of the squared amplitude values, the *average energy* is the total energy divided by the number of samples, and the *r.m.s. amplitude* is the square root of the average energy. For a sine wave, the r.m.s. amplitude is just the peak amplitude/√2.

Algorithms

Sinewaves can be constructed from the `Sin()` function provided in the Math library. Noise signals can be generated through the use of the Visual Basic built-in function `Rnd()`.

Algorithm 3.1 Sample.Sine - Sampling a sinusoid

```
Public Shared Function Sine(ByVal freq As Double, _
            ByVal amp As Double, ByVal phase As Double, _
            ByVal time As Double)
    ' angular frequency (radians/sec)
    Dim rfreq As Double = 2.0 * Math.PI * freq
    ' phase in radians
    Dim rphase As Double = 2.0 * Math.PI * phase / 360.0
    ' sample sine function
    Return amp * Math.Sin(rfreq * time - rphase)
End Function
```

Algorithm 3.2 Sample.Noise - Sampling a noise signal

```
Public Shared Function Noise(ByVal amp As Double)
    ' get a random co-ordinate inside the unit circle
    Dim x As Double, y As Double, r As Double
    Do
        x = (2 * Rnd()) - 1.0
        y = (2 * Rnd()) - 1.0
        r = (x * x) + (y * y)
    Loop While (r = 0) Or (r > 1)
    ' transform into a normal distribution (Box-Muller transform)
    Dim rval As Double = x * Math.Sqrt(-2.0 * Math.Log(r) / r)
    ' return scaled sample
    Return amp * rval
End Function
```

Algorithm 3.3 Sample.Quantise - Quantisation of a sample

```
Public Shared Function Quantise(ByVal amp As Double, _
                                ByVal quanta As Double) As Short
    Dim ival As Integer = CInt(amp / quanta)
    If (ival > 32767) Then
        Return 32767
    ElseIf (ival < -32768) Then
        Return -32768
    Else
        Return ival
    End If
End Function
```

Algorithm 3.4 – Waveform.Quantise – Quantisation of a Waveform

```
    Public Function Quantise(ByVal quanta As Double) As Signal
        Dim owv As New Signal(Count, Rate)
        For i As Integer = 1 To Count
            owv(i) = Sample.Quantise(Item(i), quanta)
        Next
        Return owv
    End Function

    Public Function Quantise() As Signal
        Dim owv As New Signal(Count, Rate)
        Dim max As Double = 0
        For i As Integer = 0 To Count - 1
            If (Math.Abs(Item(i)) > max) Then max = Math.Abs(Item(i))
        Next
        For i As Integer = 1 To Count
            owv(i) = Sample.Quantise(Item(i), max / 24000)
        Next
        Return owv
    End Function
```

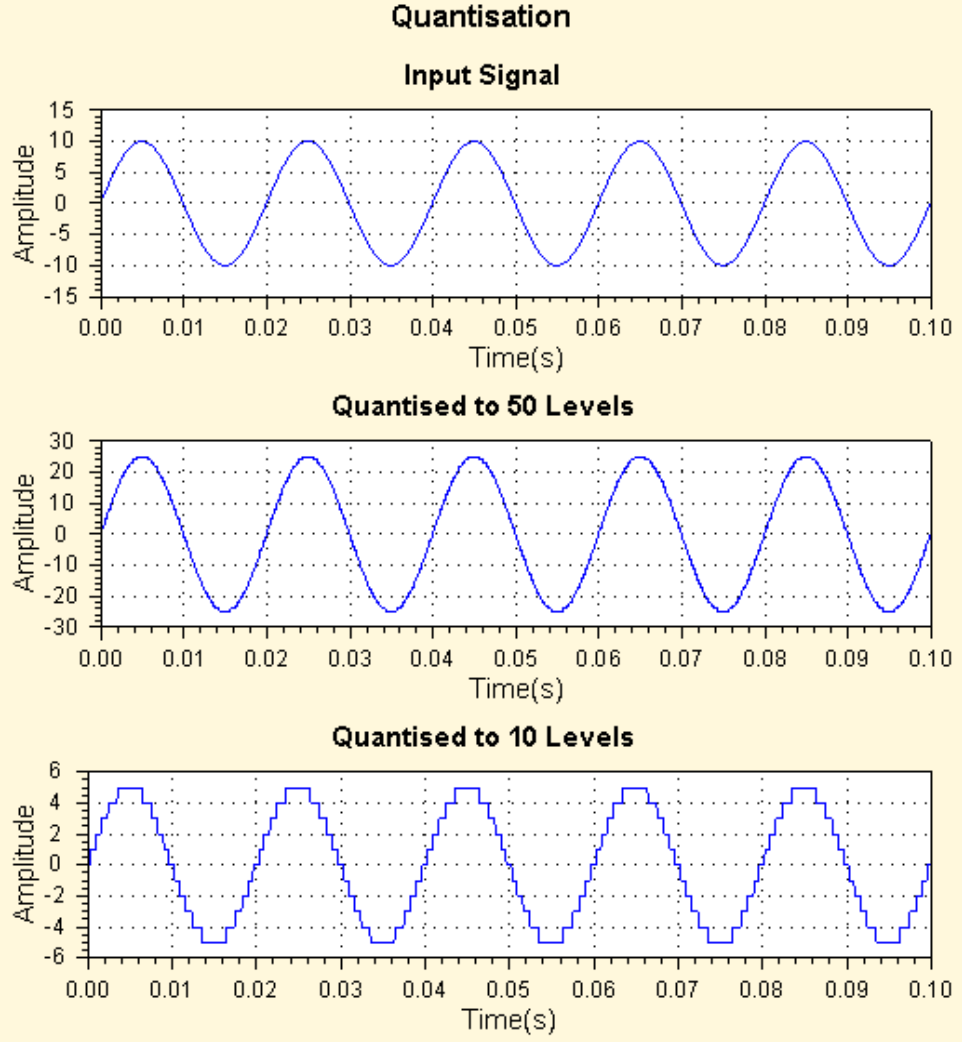
Bibliography

- Rosen & Howell, Signals and Systems for Speech and Hearing, Chapter 14.
- Meade & Dillon, Signals and Systems, Chapter 1.
- Lynn & Fuerst, Introductory Digital Signal Processing, Sections 1.1-1.4.
- Orfanidis, Introduction to Signal Processing, 1.1-1.4.

Example Programs

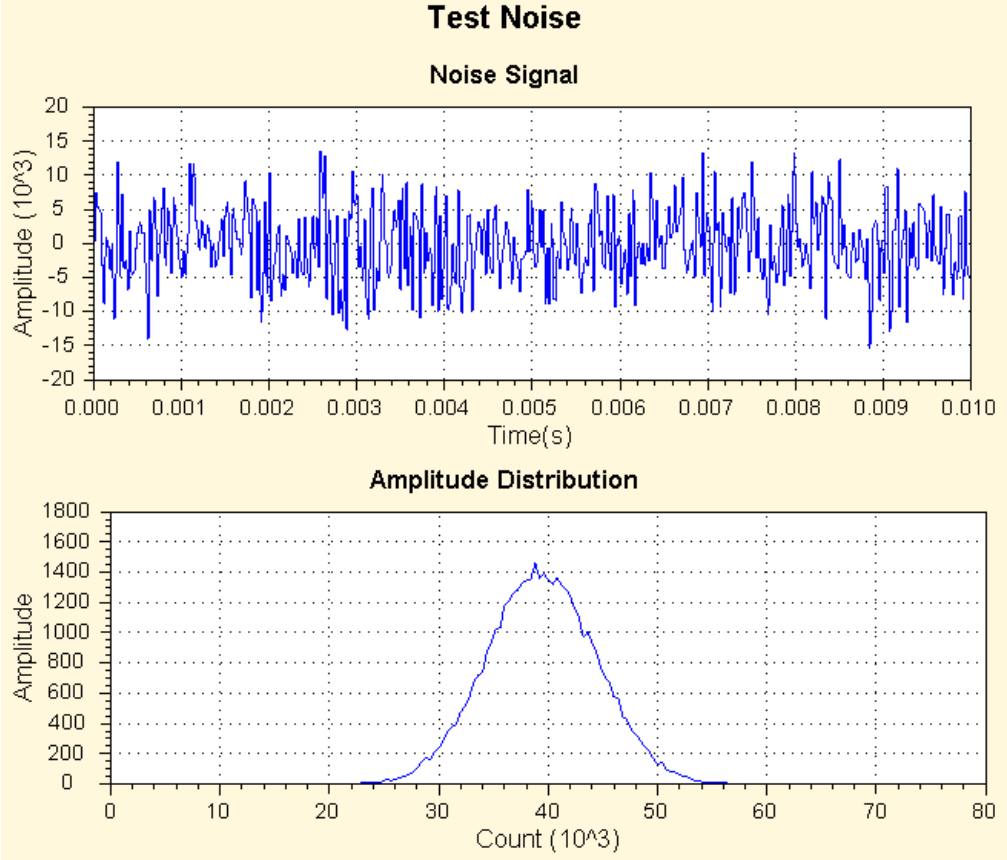
Example 3.1 TestQuantise - Demonstrate Sampling and Quantisation

```
Imports BasicDSP
Imports ZedGraph
Public Class TestQuantise
    Const NUMSAMPLE As Integer = 1000      ' number of samples
    Const SAMPRATE As Double = 10000.0      ' sampling rate
    Const SINEFREQ As Double = 50.0        ' sine at 500Hz
    Const SINEAMP As Double = 10.0       ' sine amplitude
    Dim rwv As Waveform
    Dim qwv As Signal

    Private Sub TestSine_Load(ByVal sender As System.Object, _
                  ByVal e As System.EventArgs) Handles MyBase.Load
        ' set up graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 3, 1, _
                                                "Quantisation")
        ' calculate and display sine
        rwv = New Waveform(NUMSAMPLE, SAMPRATE)
        For i As Integer = 1 To NUMSAMPLE
            rwv(i) = Sample.Sine(SINEFREQ, SINEAMP, 0, i / SAMPRATE)
        Next
        gp.PlotWaveform(1, rwv, "Input Signal")

        ' quantise to 50 levels
        qwv = rwv.Quantise(0.4)
        gp.PlotSignal(2, qwv, "Quantised to 50 Levels")

        ' quantise to 10 levels
        qwv = rwv.Quantise(2.0)
        gp.PlotSignal(3, qwv, "Quantised to 10 Levels")

    End Sub

End Class
```


Quantisation
Input Signal
Amplitude
Time(s)
Quantised to 50 Levels
Amplitude
Time(s)
Quantised to 10 Levels
Amplitude
Time(s)

Example 3.2 TestNoise - Demonstrate noise signal generation and replay

```
Imports BasicDSP
Imports ZedGraph
Public Class TestNoise
    Const NUMSAMPLE As Integer = 44100      ' number of samples
    Const SAMPRATE As Double = 44100.0      ' sampling rate
    Const NOISEAMP As Double = 5000.0      ' amplitude
    Dim wv As Signal
    Private Sub TestNoise_Load(ByVal sender As System.Object, _
                  ByVal e As System.EventArgs) Handles MyBase.Load

        ' create graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 1, "Test Noise")

        ' build noise waveform
        Randomize()
        wv = New Signal(NUMSAMPLE, SAMPRATE)
        For i As Integer = 1 To NUMSAMPLE
            wv(i) = Sample.Noise(NOISEAMP)
        Next
        gp.PlotSignal(1, wv.Cut(1, SAMPRATE / 100), "Noise Signal")

        ' calculate and plot amplitude histogram
        Dim hist As New WaveDouble(201, 0.0025)
        Dim idx As Integer
        For i As Integer = 1 To NUMSAMPLE
            ' map -40000..40000 to 0..200
            idx = (wv(i) + 40000) \ 400
            ' keep count
            hist(idx) = hist(idx) + 1
        Next
        gp.PlotWaveDouble(2, hist, "Amplitude Distribution", _
                                                "Amplitude", "Count")

    End Sub
    Private Sub zgc_MouseClick(ByVal sender As Object, _
                  ByVal e As System.Windows.Forms.MouseEventArgs) _
                  Handles zgc.MouseClick
        If e.Button = Windows.Forms.MouseButtons.Left Then
            wv.Replay()
        End If
    End Sub
End Class
```



Exercises

3.1 Use example program 3.1 to construct a program to display 1s of a sinewave signal of 25Hz sampled at 1,000 samples per second.

3.2 Adapt your program from exercise 3.1 to display a sine signal that increases linearly in frequency from 10Hz to 50Hz over an interval of 1 second.

3.3 Adapt your program from exercise 3.1 to construct and display 1s of a 10Hz square wave made up from the sum of 16 odd harmonics. The relative amplitude of the *n*th harmonic is given by 1/*n*, i.e. $H_1 + 0.33H_3 + 0.2H_5 + \ldots$, or

$$y[m] = \sum_{n=0}^{15} \frac{\sin((2n+1)20\pi mT)}{2n+1}$$

UNIT 4: DIGITAL SYSTEM MODELS

BASICDSP

4.1 Introduction

This unit is concerned with the description of digital systems, it introduces the concepts of a linear time-invariant system, convolution, the general system difference equation and its implementation, and the frequency response characteristics of systems.

When you have worked through this unit you should:

- be able to define the characteristics of a linear, time-invariant system
- understand what is represented by an impulse response and a frequency response of a system
- be able to explain how and why systems may be simulated with discrete convolution
- understand how a system can be described with the general difference equation
- be able to describe one direct form method for the implementation of the general difference equation in an algorithm
- be able to state the z-transform of a sampled sequence
- appreciate that the z-transform may be used to obtain a polynomial statement of the general difference equation.
- understand how the response of a system may be stated in terms of poles and zeros
- understand that the response can be evaluated for any particular frequency to get the frequency response graph of a system
- know how to use an algorithm for implementing an LTI system and calculating its response
- be able to relate the parameters of a simple resonator to its implementation

4.2 Concepts

We are concerned with systems that transform one digital signal into another, in particular with systems that perform a *linear* processing that does not change with time (is *time-invariant*). We seek to describe the behaviour of such systems and to find a general algorithmic description suitable for digital implementation.

Generally we describe systems that modify signals in terms of their *Frequency Response* (a graph of system response against frequency) and their *Phase Response* (a graph of phase shift against frequency). A linear time-invariant (LTI) system may also be described in the time-domain by means of its *Impulse response*, that is: the graph of the output of the system after an impulse is presented on its input. This is possible because we can describe any digital signal as a sequence of impulses of a height equal to the signal amplitude occurring at the sampling instants. The output of a LTI system to any signal can therefore be predicted by a suitable linear combination of impulse responses starting at each sample instant (this is a consequence of the principle of *superposition*).

Mathematically, this process can be described as taking a signal x[] and an impulse response h[] and calculating each output sample y[n] with the formula:

$$y[n] = \sum_{k=0}^{\infty} x[n-k]\,h[k+1]$$

Assuming h[] is zero at negative times, and samples are counted from 1. The operation is analogous to laying the impulse response backwards alongside the input signal prior to *n* and then cross multiplying and adding. This process is known as *convolution*. y[n] is said to be generated by the convolution of x[] and h[].

Convolution with the impulse response is the time-domain equivalent of multiplying by the frequency response. You can see this by considering convolution of h[] with a sinusoid signal given by:

$$x[n] = \exp(i2\pi f n T_s)$$

so that the output of the system is given by:

$$y[n] = \sum_{k=0}^{\infty} \exp(i2\pi f (n-k) T_S)\, h[k+1]$$

which can be rewritten as:

$$y[n] = \exp(i2\pi f n T_S) \sum_{k=0}^{\infty} \exp(-i2\pi f k T_S)\, h[k+1] = x[n] H(f)$$

That is the output is also a sinusoid, at the same frequency as the input, but with an amplitude scaling and a phase change given by H(f).

The impulse response of a simple resonator is simply an exponentially decaying sinewave of a frequency equal to the natural frequency and of a rate of decay related to the bandwidth.

Systems that have some kind of 'memory' can have impulse responses of infinite duration (called Infinite-Impulse-Response or *IIR* systems), others have an impulse response of finite duration (called Finite-Impulse-Response or *FIR* systems). Our simple resonator theoretically continues vibrating for ever, so is an IIR system.

To describe an LTI system algorithmically, we need a formulation of its behaviour that allows both finite and infinite impulse responses. We do this by noting that the characteristics of an LTI system must be expressed in a constant linear relationship between previous output samples and previous input samples. That is, given input samples x[] and output samples y[], some linear combination of output samples is related to some linear combination of input samples:

$$y[n] + b_1\, y[n-1] + b_2\, y[n-2] + \ldots + b_q\, y[n-q] =$$
$$a_0\, x[n] + a_1\, x[n-1] + a_2\, x[n-2] + \ldots + a_p\, x[n-p].$$

This is the general difference equation of an LTI system. It defines the behaviour of an LTI system in terms of a set of coefficients a[0..p] that operate on current and previous input samples, and a set of coefficients b[1..q] that operate on previous output samples. b[0] is always taken as 1. This is perhaps easier to see with this reformulation:

$$y[n] = \sum_{i=0}^{p} a_i x[n-i] - \sum_{j=1}^{q} b_j y[n-j]$$

This can be readily implemented as a computer algorithm.

The general difference equation leads to a simple classification of LTI systems:

(i) systems with the coefficients b[1..q] are zero, in which case output samples are generated by a linear combination of previous input samples. These are called *non-recursive* or *moving average* systems.

(ii) systems in which a[0] is 1, and a[1..p] are zero, in which case the output is formed from a single input and a linear combination of past output samples. These are called *simply recursive* or *autoregressive* systems.

(iii) systems with arbitrary a[] and b[] coefficients are sometimes called *autoregressive/moving average* or *ARMA* systems. They are of course also recursive.

The *direct form* implementation of the general difference equation requires two memory arrays: one for past inputs x[] and one for past outputs y[]. It can be shown that the same result can be achieved with only a single memory of an 'internal' waveform s[]. This *direct form II* is the basis for the `LTISystem()` implementation below.

From the description of an LTI system in terms of a[] and b[] coefficients we can also determine its frequency response characteristics (note that we leave to Unit 7 the reverse problem of the determination of a[] and b[] from the frequency response). This is normally performed using a mathematical transformation of the a[] and b[] coefficients akin to the discrete Fourier transform, called the *Z transform*.

Put simply, the Z transform converts a sequence of digital samples at successive sampling instants to a polynomial of some operator z^{-1}, in which the amplitudes of the samples become the coefficients of the polynomial. Thus a finite signal x[1..n] has a Z transform:

$$X(z) = x[1]z^{-1} + x[2]z^{-2} + x[3]z^{-3} + \ldots + x[n]z^{-n}$$

that is

$$X(z) = \sum_{n} x[n]z^{-n}$$

One way of thinking of z^{-1} is that it represents a unit delay. In other words multiplying by z^{-n} delays a z-transformed signal by *n* samples. This analogy allows to view a z-transformed signal as scaled impulses delayed appropriately and added together.

The conversion of a sequence to a polynomial is useful because it allows us to write the general difference equation as two polynomials:

$$Y(z)\,(1 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_q z^{-q}) =$$
$$X(z)\,(a_0 z^{-0} + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_p z^{-p}).$$

Where X(z) is the z-transform of the input x[], and Y(z) the z-transform of the output y[]. The cross multiplication and collection of terms in z^{-n} is equivalent to convolution on the original signals. This polynomial form of the difference equation allows us to form the response of the system:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{p} a_i z^{-i}}{1 + \sum_{j=1}^{q} b_j z^{-j}}$$

that is, as a ratio of two polynomials in z. We can then see that this response will have peaks at the roots[1] of the denominator polynomial - these are called the *poles* of the system, and dips at the roots of the numerator polynomial - these are called the *zeros* of the system. Our LTI system can be described completely (with the addition of an overall gain factor) by the location of the poles and zeros of the system. These will in general be complex numbers and can be plotted on the complex plane on a diagram called a *z-plane diagram*.

To obtain the frequency response of our system we need to put into the system sinewaves at different frequencies but of unit amplitude and plot the output amplitude. We do this by substituting z^{-1} in our polynomial for the complex sine $e^{i\Omega}$ where Ω is the angular frequency ($= 2\pi f/F_s$) at which we require the response. We can now obtain a numerical value for the response for a range of frequencies between 0 and the sampling rate (2π). What we are actually calculating is the product of the distances between each system zero and a point on the *unit circle* specified by $e^{i\Omega}$ divided by the product of the distances between each pole and that point. Thus the nearer the poles are to the unit circle the higher the peaks in the response and the closer the zeros are to the unit circle the lower the dips. The magnitude of this ratio at a given frequency is the magnitude of the frequency response, while the argument of this ratio is the phase response.

[1]The *roots* of a polynomial in x are those values of x for which the polynomial is zero. If the polynomial is *factored* into a form (x-a)(x-b)(x-c)..(x-p), then the roots are the values a,b,c,..,p.

Algorithms

Algorithm 4.1 Convolution

```
''' Convolves two waveforms
Public Shared Function Convolve(ByRef x As Waveform, ByRef h As Waveform) As
Waveform
    Dim y As New Waveform(x.Count + h.Count - 1, x.Rate)
    Dim v As Double

    ' for each output sample
    For n As Integer = 1 To y.Count

        ' sum of product with reverse IR
        v = 0
        For k As Integer = 0 To h.Count() - 1
            v += x(n - k) * h(k + 1)
            ' exploits bad index capability of Waveform class
        Next
        y(n) = v
    Next
    Return y
End Function
```

Algorithm 4.2 Linear time-invariant system model

```
Public Class LTISystem
    ' Numerator coefficients
    Public a() As Double
    ' Denominator coefficients
    Public b() As Double
    ' Internal state memory
    Public s() As Double

    ' Create new linear system
    Public Sub New(ByVal na As Integer, ByVal nb As Integer)
        ReDim a(na)
        ReDim b(nb)
        If (na > nb) Then ReDim s(na) Else ReDim s(nb)
        Clear()
    End Sub

    ' Create a copy of a linear system
    Public Sub New(ByVal ltis As LTISystem)
        a = ltis.a
        b = ltis.b
        s = ltis.s
    End Sub

    ' Clear the state memory
    Public Sub Clear()
        System.Array.Clear(s, 0, s.Length)
    End Sub
```

```
    ' Pass a single sample value through the linear system
    Default Public ReadOnly Property Item(ByVal ival As Double) As Double
        Get
            Dim i As Integer
            ' shift state memory
            For i = s.Length - 1 To 1 Step -1
                s(i) = s(i - 1)
            Next

            ' compute s[0] from y[] coefficients
            s(0) = ival
            For i = 1 To b.Length - 1
                s(0) -= b(i) * s(i)
            Next

            ' compute output from x[] coefficients
            Dim out As Double
            For i = 0 To a.Length - 1
                out += a(i) * s(i)
            Next

            Return out
        End Get
    End Property

    ' Pass a waveform through the filter
    Public Function Filter(ByRef iwv As Waveform) As Waveform
        Dim owv As New Waveform(iwv.Count, iwv.Rate)
        Clear()
        For i As Integer = iwv.First To iwv.Last
            owv(i) = Item(iwv(i))
        Next
        Return owv
    End Function

    ' Pass a signal through the filter
    Public Function Filter(ByRef iwv As Signal) As Signal
        Dim owv As New Signal(iwv.Count, iwv.Rate)
        Clear()
        For i As Integer = iwv.First To iwv.Last
            owv(i) = Item(iwv(i))
        Next
        Return owv
    End Function

    ' Calculate frequency response of linear system at a single frequency
    Public Function Response(ByVal freq As Double) As Complex
        Dim i As Integer
        Dim omega(s.Length) As Complex
        Dim z As Complex = Complex.Exp(New Complex(0, 2 * Math.PI * freq))

        ' initialise polynomial of complex frequency
        omega(0) = New Complex(1.0)
        omega(1) = z
        For i = 2 To s.Length - 1
            omega(i) = omega(i - 1) * z
        Next

        ' compute response of numerator
        Dim rnum As New Complex(0)
        For i = 0 To a.Length - 1
            rnum = rnum + a(i) * omega(i)
        Next

        ' compute response of denominator
        Dim rden As New Complex(1.0)
        For i = 1 To b.Length - 1
            rden = rden + b(i) * omega(i)
        Next
```

```
        ' compute ratio
        If (rden.Mag = 0) Then
            Return New Complex(100000.0)        ' i.e. infinity
        Else
            Return rnum / rden
        End If

    End Function
End Class
```

Algorithm 4.3 Digital resonator

```
' Builds a linear system for a resonator
Public Shared Function Resonator(ByVal freq As Double, ByVal bwidth As Double)
As LTISystem
    Dim ltis As New LTISystem(1, 3)

    ' get parameters as angles
    Dim wf As Double = 2 * Math.PI * freq
    Dim wb As Double = 2 * Math.PI * bwidth

    ' estimate pole radius from bandwidth (rule of thumb)
    Dim r As Double = 1.0 - wb / 2

    ' set up numerator
    ltis.a(0) = 1.0

    ' set up denominator: quadratic formula
    ltis.b(1) = -2.0 * r * Math.Cos(wf)
    ltis.b(2) = r * r

    ' adjust numerator for unity gain at DC
    ltis.a(0) /= ltis.Response(0).Mag

    Return ltis
End Function
```

Bibliography

Meade & Dillon, Signals and Systems, Chapter 5.

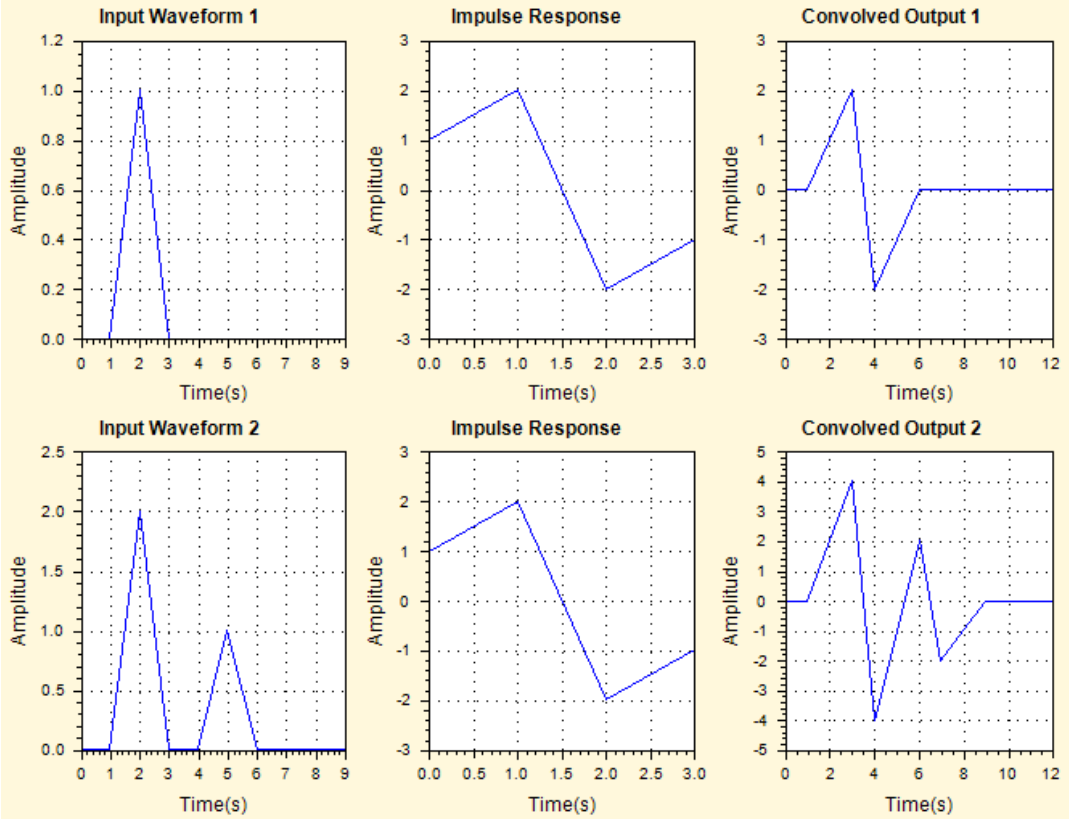
Lynn & Fuerst, Introductory Digital Signal Processing, Chapters 2 & 4.

Orfanidis, Introduction to Signal Processing, Sections 5.1, 5.4, 6.1-6.3.

Example Programs

Example 4.1 Demonstration of convolution

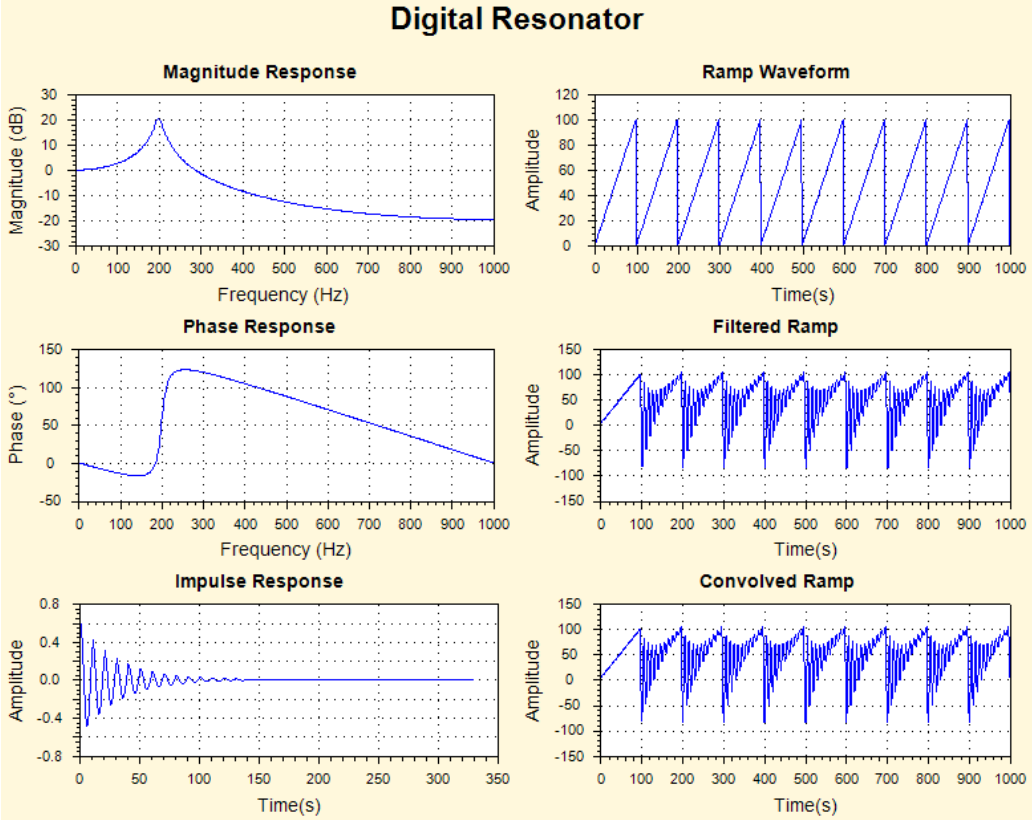
```
Imports BasicDSP
Imports ZedGraph
Public Class TestConvolve

    Private Sub TestConvolve_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        ' set up graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 3)

        ' create input signal and display
        Dim x As New Waveform(10, 1)
        For i As Integer = 1 To 10
            x(i) = 0
        Next
        x(3) = 1
        gp.PlotWaveform(1, x, "Input Waveform 1")

        ' create impulse response
        Dim h As New Waveform(4, 1)
        h(1) = 1
        h(2) = 2
        h(3) = -2
        h(4) = -1
        gp.PlotWaveform(2, h, "Impulse Response")

        ' do convolution
        Dim y As Waveform = Filter.Convolve(x, h)
        gp.PlotWaveform(3, y, "Convolved Output 1")

        ' try a different input
        For i As Integer = 1 To 10
            x(i) = 0
        Next
        x(3) = 2
        x(6) = 1
        gp.PlotWaveform(4, x, "Input Waveform 2")

        ' same impulse response
        gp.PlotWaveform(5, h, "Impulse Response")

        ' do convolution
        y = Filter.Convolve(x, h)
        gp.PlotWaveform(6, y, "Convolved Output 2")

    End Sub
End Class
```



Example 4.2 Equivalence of filtering and convolution

```
Imports BasicDSP
Imports BasicDSP.Filter
Imports ZedGraph
Public Class TestResonator
    Const ILIMIT As Double = 0.00001    ' length limit for impulse response

    Private Sub TestResonator_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        ' set up graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 3, 2, "Digital Resonator")

        ' create resonator
        Dim ltis As LTISystem = Resonator(0.1, 0.01)

        ' calculate frequency response
        Dim fresp As New Spectrum(500, 1000)
        For i As Integer = fresp.First To fresp.Last
            fresp(i) = ltis.Response(i / 1000.0)
        Next

        gp.PlotDbSpectrum(1, fresp, "Magnitude Response")
        gp.PlotPhaseSpectrum(3, fresp, "Phase Response")

        ' calculate impulse response
        Dim iresp As New Waveform(0, 1)
        Dim lval As Double              ' last output
        Dim oval As Double = ltis(1.0)  ' put in unit pulse

        While (Math.Abs(oval) > ILIMIT) Or (Math.Abs(oval - lval) > ILIMIT)
            iresp.Add(oval)         ' append sample
            lval = oval             ' remember sample
            oval = ltis(0.0)        ' get next sample
        End While
```

```
        gp.PlotWaveform(5, iresp, "Impulse Response")

        ' create a ramp waveform
        Dim ramp As New Waveform(1000, 1)
        For i As Integer = 1 To 1000
            ramp(i) = i Mod 100
        Next
        gp.PlotWaveform(2, ramp, "Ramp Waveform")

        ' filter the ramp waveform
        Dim framp As New Waveform(1000, 1)
        ltis.Clear()
        For i As Integer = 1 To 1000
            framp(i) = ltis(ramp(i))
        Next
        gp.PlotWaveform(4, framp, "Filtered Ramp")

        ' convolve ramp with impulse response
        Dim cramp As Waveform = Filter.Convolve(ramp, iresp)
        gp.PlotWaveform(6, cramp.Cut(1,1000), "Convolved Ramp")

    End Sub
End Class
```



Exercises

4.1 Explain the first three values of 'Convolved Output 2' (they are: 1, 4 & 5).

4.2 Modify example program 4.1 so that the impulse response is a square wave.

4.3 Modify example program 4.1 so that the impulse response matches the input waveform in each case.

4.4 Adapt example program 4.2 to send a stream of pulses through the resonator rather than the ramp wave, displaying the input and output signals.

4.5 Adapt example program 4.2 to generate the output from a resonator with a natural frequency of 25Hz and a bandwidth of 7.5Hz when fed with a pulse train at 10Hz lasting 1 second and sampled at 1,000Hz. Display the result. (Hint: create a Signal of 1000 samples at 1000 samples/sec, create a Resonator of centre frequency (25/1000) and bandwidth (7.5/1000), then run the resonator into the signal, putting in pulses every 100 samples)

UNIT 5: DISCRETE FOURIER TRANSFORM

BASICDSP

5.1 Introduction

This unit introduces the Discrete Fourier Transform as a means for obtaining a frequency based representation of a digital signal. The special characteristics of the Fast Fourier Transform implementation are described.

When you have worked through this unit you should:

- be able to explain what information is represented on a magnitude spectrum and on a phase spectrum of a discrete signal
- be able to state the mathematical expression for the Discrete-time discrete-frequency Fourier Transform (DFT)
- understand how the direct implementation of the DFT operates
- appreciate that the direct implementation of the DFT is very inefficient, and that the Fast discrete-time discrete-frequency Fourier Transform (FFT) provides a more efficient means of its calculation
- have a qualitative understanding of the operation of the decimation-in-time form of the FFT
- be able to predict how simple sine and cosine signals appear on the DFT in the region from 0 to F_s.

5.2 Concepts

A digital signal of finite duration x[1..N] can be specified in the time domain as a sequence of N scaled impulses occurring at regular sampling instants: each impulse taking on the amplitude of the signal at that instant. The same signal may also be described as a combination of N complex sinusoidal components X[0..N-1], each of a given frequency and phase, and each being a harmonic of the sampling rate/N. This representation, called a frequency-domain representation, may be obtained from the time-domain form through the use of the *Discrete Fourier Transform* or *DFT*. The time domain form and the frequency domain form are simply different ways of representing the same digital signal, one is chosen over the other simply in terms of utility for a given purpose.

The Discrete Fourier Transform X(f) of a signal x[1..N] is defined as:

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n+1] \; e^{-i\frac{2\pi kn}{N}}$$

where X[k] is the amplitude of the k*th* harmonic, where k varies from 0 to N-1 and where k/N represents a fractional frequency value of the sampling rate. In general X[] and x[] can hold complex values, and X[] will be complex even if x[] is purely real.

The graph of |X(f)| against frequency is known as the *magnitude spectrum*. The graph of arg X(f) against frequency is known as the *phase spectrum*.

By copying a real digital signal into a complex sequence x[1..N], the DFT has a straightforward algorithmic implementation (shown in `ComplexDFT()`), using complex multiplication with the complex exponential defined as:

$$e^{a+ib} = a\cos(b) + ia\sin(b)$$

and hence:

$$e^{-i\frac{2\pi kn}{N}} = \cos(\frac{2\pi kn}{N}) + i\sin(\frac{2\pi kn}{N}).$$

We can also define the inverse transform, from the frequency representation X[] back to the time representation x[]:

$$x[n] = \sum_{k=0}^{N-1} X[k]\ e^{i\frac{2\pi(n-1)k}{N}}$$

where n varies from 1 to N. We have chosen to place the scaling factor 1/N in the forward transform, but many authorities place it in the inverse transform. We choose to place it here because it leads to harmonic amplitudes which are normalised to the sequence length, and hence independent of the amount of signal analysed. This leads naturally to an equivalence between the energy in the signal and the energy in the spectrum:

$$Average\ Energy = \frac{1}{N}\sum_{n=1}^{N} x[n]^2 = \sum_{k=0}^{N-1} |X[k]|^2$$

The frequency spectrum is often displayed in log magnitude terms in units of *decibels (dB)*, and may be converted using:

$$A[k](dB) = 10\log_{10}(|X[k]|^2) = 20\log_{10}(|X[k]|)$$

From a real signal at a sampling rate F_s, the DFT provides N harmonic amplitudes at frequencies from 0 to $\left(\frac{N-1}{N}\right)F_S$. However the frequencies from 0 to $F_s/2$ are aliased to the region $F_s/2$ to F_s, so only the lower N/2 amplitudes are important.

The phase angles may be converted to degrees using:

$$P[k] = \frac{360}{2\pi}arg(X[k])$$

The *Fast Fourier Transform* or *FFT* is simply a particular implementation of the DFT, that gives identical results, but which takes considerably less calculation. It does this by eliminating a great deal of redundant computation in the DFT in the circumstances when the sequence length N is a power of 2 (i.e. 4, 8, 16, 32, 64, etc).

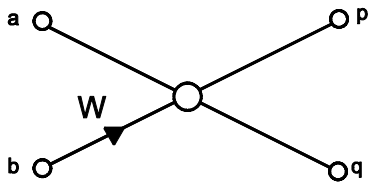
In a normal DFT, each harmonic amplitude is the result of N complex multiplies with N different complex exponentials - giving a total of N^2 multiplies for all N harmonics. When N is a power of 2, many of these multiplies concern identical numerical multiplicands and many of the complex exponentials are zero or 1. When redundant

computation is removed, the number of multiplies is $N\log_2(N)$ rather than N^2 and this represents a very large saving when N is large (e.g. for 1024 samples there is 100 times less calculation).

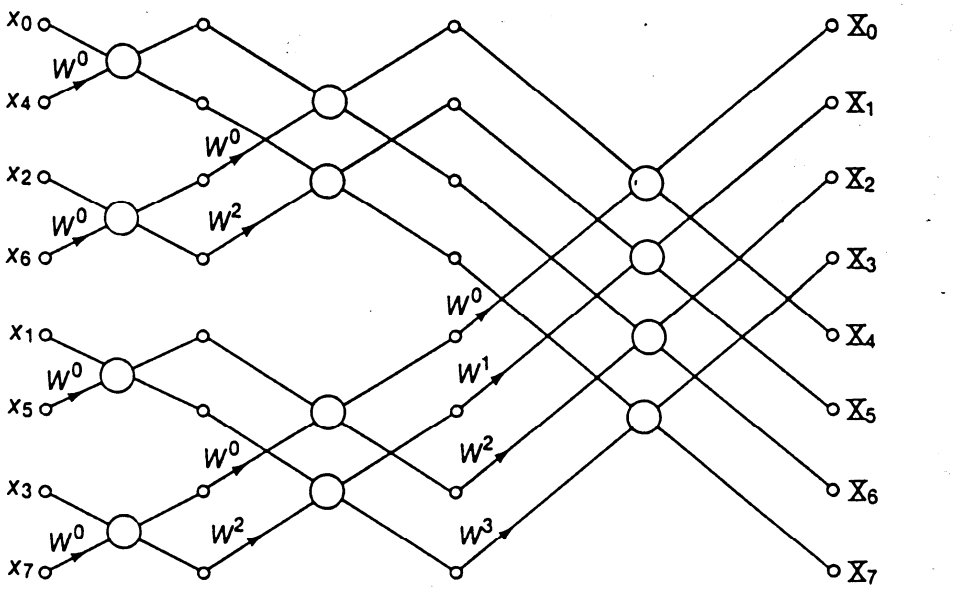
At the heart of the FFT is a simple algebraic manipulation that takes two input values, performs a multiply operation with a complex exponential and produces two output values. This basic unit is called a 'butterfly', and for two complex inputs a and b, a frequency W (=2πkn/N), and outputs p and q, the butterfly calculation is

$$p = a + be^{-iW}. \quad q = a - be^{-iW}.$$

We shall diagram this basic operation as:



This actually represents the DFT of a 2 sample waveform. Longer waveforms can be processed by combining these butterfly operations with variations on the value of W. Thus a DFT of an 8 sample waveform x[0] to x[7] can be graphed as:



Where W^j is one of the frequencies in the DFT calculation (=2πj/N). This **signal flow graph** is the basis of the `ComplexFFT()` implementation. To operate the graph, the input signal is shuffled into the spectrum array in an order known as **bit-reversed addressing**. Then each column of butterfly operations is performed, so that the signal 'moves' left-to-right through the graph, turning into the DFT spectrum.

Implementation Note

The FFT routines have not been written for maximum efficiency. Note that the calculation of the bit-reversed addresses and the values of the complex exponentials need only be performed once for a given size of transform. Since typical use of the FFT is the repeated use of the transform on constant lengths of waveform, these tables may be pre-calculated and stored.

Bibliography

Meade & Dillon, Signals and Systems, Chapter 7, pp130-144.

Lynn & Fuerst, Introductory Digital Signal Processing, Chapter 7.

Orfanidis, Introductory Signal Processing, Chapter 9.

Algorithms

Algorithm 5.1 Complex to Complex Discrete Fourier Transform

```
' Computes complex discrete Fourier transform
Public Shared Function ComplexDFT(ByVal x As ComplexWaveform) As Spectrum

    Dim s As New Spectrum(x.Count, x.Rate)

    ' for each output harmonic
    For i As Integer = s.First To s.Last
        ' get frequency
        Dim f As Double = (2 * Math.PI * i) / s.Count
        ' compute complex sum
        Dim sum As New Complex(0)
        For j As Integer = 0 To x.Count
            sum += x(j + 1) * Complex.Exp(New Complex(0.0, -f * j))
        Next
        ' scale
        s(i) = sum / x.Count
    Next
    Return s
End Function

' Computes complex inverse discrete Fourier transform
Public Shared Function ComplexIDFT(ByVal s As Spectrum) As ComplexWaveform

    Dim x As New ComplexWaveform(s.Count, s.Rate)

    ' for each output sample
    For i As Integer = 0 To x.Count
        ' get frequency
        Dim f As Double = (2 * Math.PI * i) / x.Count()
        ' compute complex sum
        Dim sum As New Complex(0)
        For j As Integer = 0 To x.Count
            sum += s(j) * Complex.Exp(New Complex(0.0, f * j))
        Next
        x(i + 1) = sum
    Next
    Return x
End Function
```

Algorithm 5.2 Complex Fast Fourier Transform

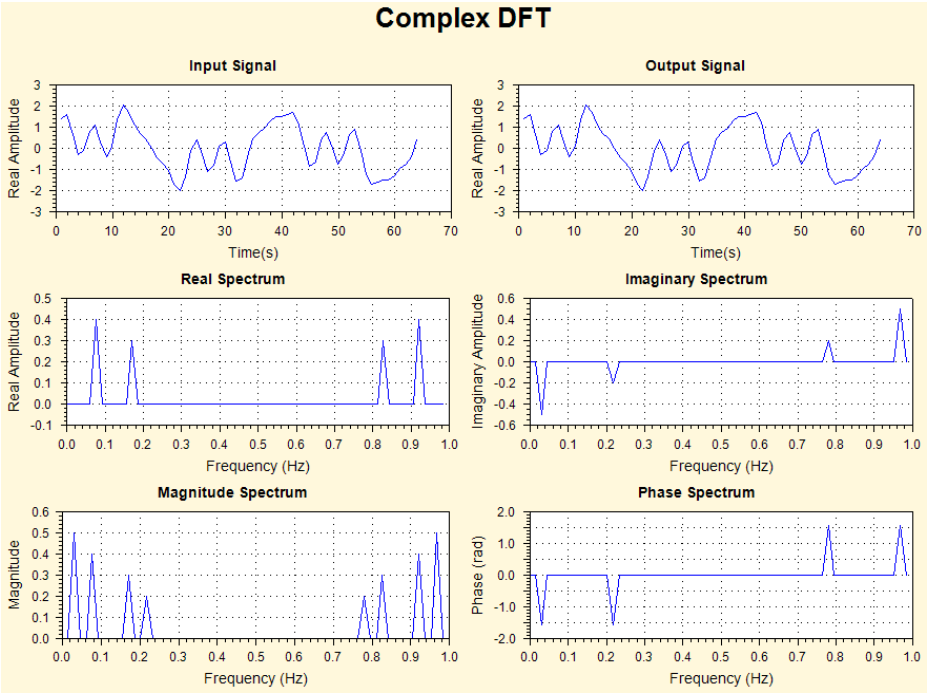
```
' integer logarithm base 2
Private Shared Function ILog2(ByVal x As Integer) As Integer
    Dim p As Integer = 0
    Dim y As Integer = 1
    While (y < x)
        p += 1
        y = 2 * y
    End While
    Return p
End Function

' integer power base 2
Private Shared Function IPow2(ByVal p As Integer) As Integer
    Dim x As Integer = 1
    While (p > 0)
        p -= 1
        x = 2 * x
    End While
    Return x
End Function

' The FFTBitReverseTable function returns a
' table of indexes for FFT in-place sample shuffling
Private Shared Sub FFTBitReverseTable(ByVal size As Integer, ByRef idx() As
Integer)

    ' find # bits involved
    Dim nbit As Integer = ILog2(size)

    ' for each table entry
    For i As Integer = 0 To size - 1
        ' store bit reversed index
        Dim a1 As Integer = i
        Dim a2 As Integer = 0
        For j As Integer = 1 To nbit
            a2 *= 2
            If (a1 And 1) Then a2 = a2 Or 1
            a1 \= 2
        Next
        idx(i) = a2

    Next
End Sub

' The FFTButterfly function performs the key FFT cross-multiply
Private Shared Sub FFTButterfly(ByRef upper As Complex, ByRef lower As Complex,
ByVal w As Complex)
    Dim temp As Complex = lower * w
    lower = upper - temp
    upper = upper + temp
End Sub

' The ComplexFFT function implements a fast complex to
' complex discrete fourier transform
Public Shared Function ComplexFFT(ByRef x As ComplexWaveform) As Spectrum

    Dim size As Integer = IPow2(ILog2(x.Count))
    Dim s As New Spectrum(size, x.Rate)

    ' get bit reverse table
    Dim amap(size) As Integer
    FFTBitReverseTable(size, amap)

    ' shuffle input data into spectrum
    For i As Integer = 0 To size - 1
        s(amap(i)) = x(i + 1)       ' uses bad index capability of x[] to pad
    Next

    ' do multiple butterfy passes over data
```

```
    ' with steps of 1,2,4,..,N
    Dim d As Integer = 1
    While (d < size)
        ' for each start position
        For j As Integer = 0 To d - 1

            Dim w As Complex = Complex.Exp(New Complex(0, -(Math.PI * j) / d))
            ' for each step
            Dim i As Integer = j
            While (i < size)
                FFTButterfly(s(i), s(i + d), w)
                i += 2 * d
            End While
        Next
        d *= 2
    End While

    ' normalise
    For i As Integer = 0 To size - 1
        s(i) /= x.Count
    Next
    Return s
End Function

' The ComplexIFFT function implements a fast complex to
' complex inverse discrete fourier transform
Public Shared Function ComplexIFFT(ByRef s As Spectrum) As ComplexWaveform

    Dim x As New ComplexWaveform(s.Count, s.Rate)

    ' get bit reverse table
    Dim amap(s.Count) As Integer
    FFTBitReverseTable(s.Count, amap)

    ' shuffle input data into waveform
    For i As Integer = 0 To s.Count - 1
        x(i + 1) = s(amap(i))
    Next

    ' do multiple butterfy passes over data
    ' with steps of 1,2,4,..,N
    Dim d As Integer = 1
    While (d < s.Count)

        ' for each start position
        For j As Integer = 0 To d - 1

            Dim w As Complex = Complex.Exp(New Complex(0, Math.PI * j / d))
            ' for each step
            Dim i As Integer = j + 1
            While (i <= s.Count)
                FFTButterfly(x(i), x(i + d), w)
                i += 2 * d
            End While
        Next
        d *= 2
    End While

    Return x
End Function
```

Example Programs

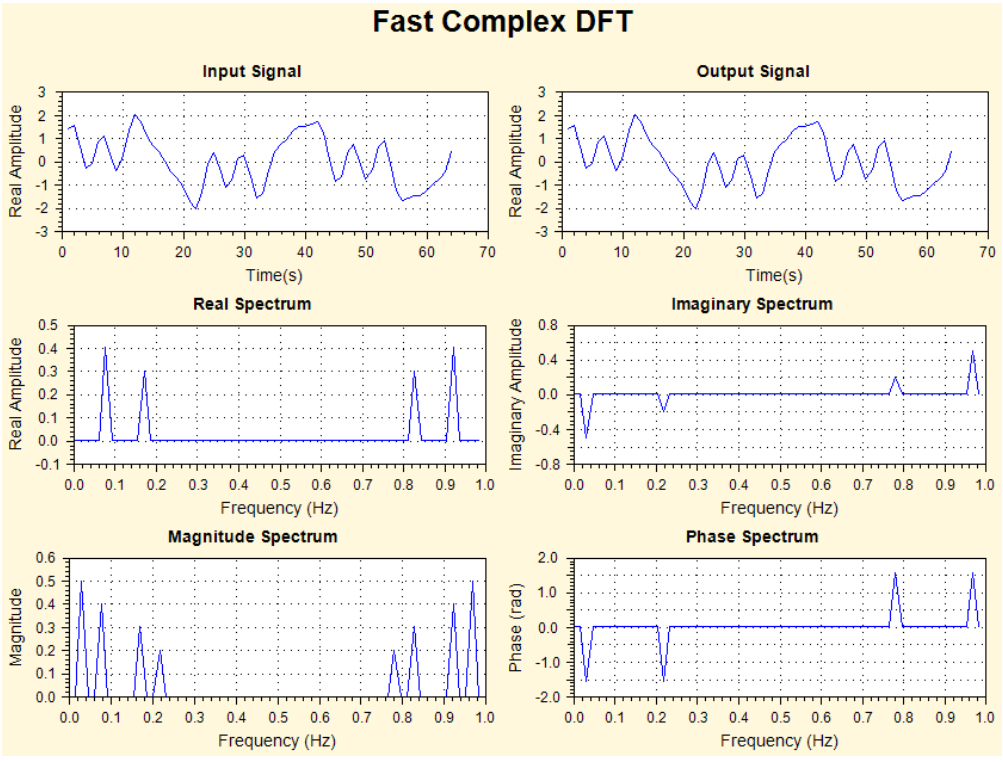
Example 5.1 Complex Discrete Fourier Transform

```
Imports BasicDSP
Imports ZedGraph
Public Class TestCDFT
    Const DFTSIZE As Integer = 64

    Private Sub TestCDFT_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ' create a test signal
        Dim ip As New ComplexWaveform(DFTSIZE, 1)
        For i As Integer = 1 To DFTSIZE
            Dim f As Double = 2 * Math.PI * (i - 1)
            ip(i) = New Complex(1.0 * Math.Sin(2 * f / DFTSIZE) + _
                0.8 * Math.Cos(5 * f / DFTSIZE) + _
                0.6 * Math.Cos(11 * f / DFTSIZE) + _
                0.4 * Math.Sin(14 * f / DFTSIZE), 0)
        Next

        ' generate spectrum
        Dim s As Spectrum = DFT.ComplexDFT(ip)

        ' generate signal again
        Dim op As ComplexWaveform = DFT.ComplexIDFT(s)

        ' plot as graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 3, 2, "Complex DFT")

        gp.PlotComplexWaveform(1, ip, "Input Signal")
        gp.PlotComplexWaveform(2, op, "Output Signal")
        gp.PlotSpectrumReal(3, s, "Real Spectrum")
        gp.PlotSpectrumImag(4, s, "Imaginary Spectrum")
        gp.PlotSpectrumMag(5, s, "Magnitude Spectrum")
        gp.PlotSpectrumArg(6, s, "Phase Spectrum")

    End Sub
End Class
```



Example 5.2 Complex Fast Fourier Transform

```
Imports BasicDSP
Imports ZedGraph
Public Class TestCDFT
    Const DFTSIZE As Integer = 64

    Private Sub TestCDFT_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ' create a test signal
        Dim ip As New ComplexWaveform(DFTSIZE, 1)
        For i As Integer = 1 To DFTSIZE
            Dim f As Double = 2 * Math.PI * (i - 1)
            ip(i) = New Complex(1.0 * Math.Sin(2 * f / DFTSIZE) + _
                0.8 * Math.Cos(5 * f / DFTSIZE) + _
                0.6 * Math.Cos(11 * f / DFTSIZE) + _
                0.4 * Math.Sin(14 * f / DFTSIZE), 0)
        Next

        ' generate spectrum
        Dim s As Spectrum = DFT.ComplexFFT(ip)

        ' generate signal again
        Dim op As ComplexWaveform = DFT.ComplexIFFT(s)

        ' plot as graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 3, 2, "Fast Complex DFT")

        gp.PlotComplexWaveform(1, ip, "Input Signal")
        gp.PlotComplexWaveform(2, op, "Output Signal")
        gp.PlotSpectrumReal(3, s, "Real Spectrum")
        gp.PlotSpectrumImag(4, s, "Imaginary Spectrum")
        gp.PlotSpectrumMag(5, s, "Magnitude Spectrum")
        gp.PlotSpectrumArg(6, s, "Phase Spectrum")

    End Sub
End Class
```



Exercises

5.1 The inverse DFT provides a simpler method to synthesize a square wave: set up the spectrum of a square wave and call `ComplexIDFT()`. Set up a spectrum as follows:

```
Spectrum(1000,0.05);      // 0..19,980Hz in 1000 steps
```

Then put in the odd harmonics of 100Hz with appropriate amplitudes (amplitude of *n*th harmonic is just 1/n). That is, put in amplitudes of 1.0 at spectrum position 5, 0.33 at position 15, 0.2 at position 25, etc. Don't forget to put in the mirror images at positions 995, 985, 975, etc. Plot your spectrum and the result of the inverse DFT.

How would you change your solution to use an FFT? Why might you want to?

5.2 Modify Examples 5.1 and 5.2 to explore the differences between an exact DFT of a 40 point test waveform and the FFT of the same waveform suffixed with 24 zeros. Plot a graph showing the magnitude spectrum and the phase spectrum of the same signal analysed in these two ways.

What differences are there between an FFT of a 64 point waveform and an FFT of the same waveform appended with 64 zeros?

UNIT 6: WINDOWING

BASICDSP

6.1 Introduction

This unit considers the consequences of analysing sections or 'windows' of a continuous signal, and how the use of smoothing functions reduces the spectral artifacts introduced.

When you have worked through this unit you should:

- be able to state why and when spectral leakage occurs
- understand how a smooth window reduces the amount of spectral leakage
- be able to describe and apply a Hamming window in your own work

6.2 Concepts

Where the frequency components of signals we analyse with the DFT are exact harmonics of the DFT analysis, the resulting spectrum shows clean spectral lines. Since mostly the signals we want to analyse will not contain just these harmonics, we find that the energy at one input frequency is spread to a number of nearby harmonics in the DFT. This is called *spectral leakage*.

A similar problem arises for signals which are changing with time, where we usually want to analyse a 'snapshot' of the signal, short enough in which we could consider the generating system to be 'stationary'. To do this we must take a sample or a *window* of the signal over some defined interval. If we simply cut out a section of the signal, however, we introduce artefacts into the waveform - namely the sudden onset and offset - which will be manifested as distortions in our spectral analysis. What is happening is that our spectrum results from the convolution of the signal spectrum with the spectrum of a *rectangular* window. The spectrum of a rectangular window has the familiar sin(x)/x shape or *sinc* function, and so each signal component is broadened by this shape.

We can reduce these distortions by ensuring that the section has no sudden onset or offset, which we can do by multiplying our section with a smoothing function which reduces the size of the signal at the edges. We need a shape which has a spectrum with a narrow central lobe and small sidelobes. A window based on a *raised-cosine* shape called the *Hamming* window is a common compromise:

$$w[n] = 0.54 - 0.46\cos(\frac{2\pi n}{N})$$

Where N is the number of samples in the section. See `Hamming()`.

Algorithms

```
' Generate a Hamming Window
Public Shared Function Hamming(ByVal len As Integer) As Waveform
    Dim owv As New Waveform(len, 1)

    Dim omega As Double = 2.0 * Math.PI / (len - 1)

    For i As Integer = 1 To owv.Count
        owv(i) = (0.54 - 0.46 * Math.Cos(omega * (i - 1)))
    Next

    Return owv
End Function

' Applies a Hamming window to the given waveform segment
Public Shared Function Hamming(ByVal iwv As Waveform) As Waveform
    Dim owv As New Waveform(iwv.Count, iwv.Rate)

    Dim omega As Double = 2.0 * Math.PI / (iwv.Count() - 1)

    For i As Integer = 1 To owv.Count
        owv(i) = (0.54 - 0.46 * Math.Cos(omega * (i - 1))) * iwv(i)
    Next
    Return owv
End Function

' Applies a Hamming window to the given complex waveform segment
Public Shared Function Hamming(ByVal iwv As ComplexWaveform) As ComplexWaveform
    Dim owv As New ComplexWaveform(iwv.Count, iwv.Rate)

    Dim omega As Double = 2.0 * Math.PI / (iwv.Count() - 1)

    For i As Integer = 1 To owv.Count
        owv(i) = (0.54 - 0.46 * Math.Cos(omega * (i - 1))) * iwv(i)
    Next

    Return owv
End Function
```

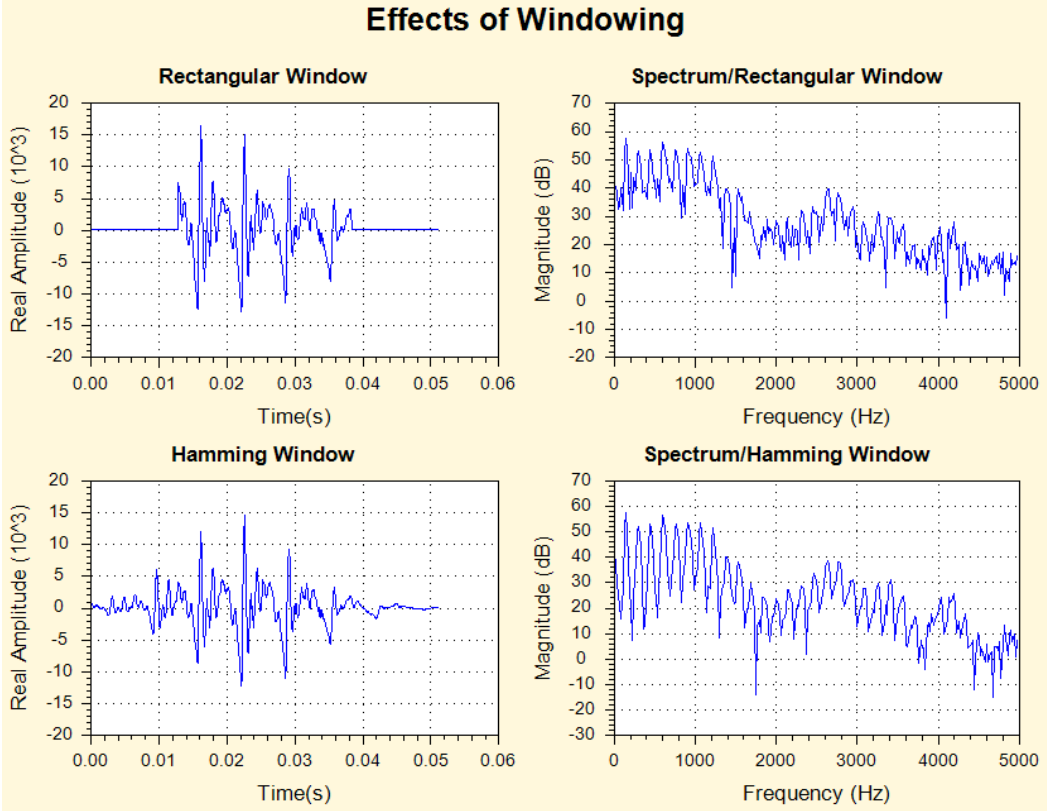
Bibliography

Lynne & Fuerst Introductory Digital Signal Processing, Section 8.2.2

Example Program 6.1

```
Imports BasicDSP
Imports ZedGraph
Public Class TestWindow

    Private Sub TestWindow_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ' get sample waveform
        Dim sig As New Signal("c:/sfs/demo/six.wav")
        Dim wv As Waveform = sig.Cut(14537, 1024).Float
        Dim cwv1 As ComplexWaveform = wv.Complex

        ' create the graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 2, "Effects of
Windowing")

        ' apply rectangular window and plot
        For i As Integer = 1 To cwv1.Count \ 4
            cwv1(i) = 0
        Next
        For i As Integer = 3 * cwv1.Count \ 4 To cwv1.Count
```

```
            cwv1(i) = 0
        Next
        gp.PlotComplexWaveform(1, cwv1, "Rectangular Window")

        ' FFT without windowing and plot
        Dim sp1 As Spectrum = DFT.ComplexFFT(cwv1)
        gp.PlotDbSpectrum(2, sp1.Half.Half, "Spectrum/Rectangular Window")

        ' apply Hamming window and plot
        Dim cwv2 As ComplexWaveform = Window.Hamming(wv.Complex)
        gp.PlotComplexWaveform(3, cwv2, "Hamming Window")

        ' FFT with Hamming window and plot
        Dim sp2 As Spectrum = DFT.ComplexFFT(cwv2)
        gp.PlotDbSpectrum(4, sp2.Half.Half, "Spectrum/Hamming Window")

    End Sub
End Class
```



Exercise

6.1 Adapt Example 6.1 to plot the waveforms and FFT spectra of sine waves at $F_s/4$ (exact) and $F_s/6$ (non-exact) harmonic frequencies with rectangular and Hamming windows. Use a short FFT of about 128 samples. Interpret the results.

UNIT 7: DIGITAL FILTER DESIGN

BASICDSP

7.1 Introduction

This unit is concerned primarily with the design of digital systems having frequency response characteristics appropriate to low-pass, high-pass and band-pass filters. It is concerned with both non-recursive and recursive designs: the former based on the Fourier transform method, the second through the so-called 'bilinear' transformation of analogue designs.

When you have worked through this unit you should:

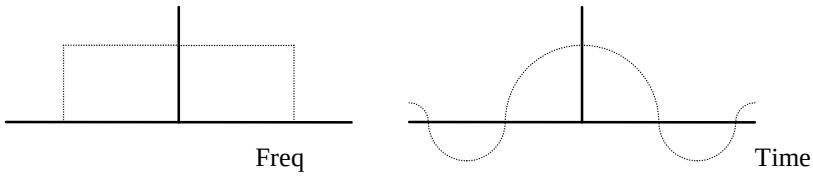
- be able to explain how simple non-recursive filters may be designed by the DFT method
- understand how low-pass non-recursive filters may be transformed into high-pass and band-pass designs
- understand how the Kaiser window provides a means for building a simple non-recursive filter from specifications of the stop-band ripple and the width of the transition region.
- be able to explain how simple recursive filters may be designed through the use of the Butterworth formulae
- understand how the use of cascaded second-order sections provides a means for implementing Butterworth filters from the pole-zero diagram
- understand how recursive low-pass filters may be transformed into high-pass and band-pass designs.
- be able to discuss the relative merits of recursive and non-recursive designs for different applications.

7.2 Concepts

Non-recursive Filter Design

We seek to find a set of a[] (numerator) coefficients for a linear time-invariant system that would implement a low-pass filter: that is the impulse response of a low-pass non-recursive filter. We then consider how to *transform* these coefficients into high-pass or band-pass forms.

We first note that the Fourier transform of a rectangular window has a characteristic shape known as a *sin(x)/x* or *sinc(x)* function. Thus for an ideal low-pass filter response, which has a rectangular shape, the related impulse response must follow this sin(x)/x shape. Since the response of a digital system repeats periodically with frequency (once per sampling frequency), we can choose to consider a rectangular filter within a response that extends from -sample-rate/2 to +sample-rate/2, and this will give us an impulse response that is also symmetric about t=0:



For a cut-off angular frequency of w_c the formula for this impulse response is just:

$$h[n] = \frac{1}{n\pi}\sin(nw_c) = \frac{w_c}{\pi}\mathrm{sinc}(nw_c)$$

where $w_c = 2\pi f_c$, where f_c = fraction of sampling rate, and where n extends over all positive and negative times.

To generate an impulse response of a certain maximum size, it is best to window the resulting coefficients, rather than simply truncate them. This can be easily performed with a Hamming window. We may then shift the filter coefficients in time to get a causal filter. See `NRLowPass()`.

To obtain the impulse response of a high-pass or band-pass filter, we note that all we need do is shift the frequency response curve along the frequency axis - so the rectangular response is now either symmetric about the half-sample-rate frequency - in which case we get a high-pass filter, or is now over a block of frequencies - in which case we get a band-pass filter. The process of shifting the response in the frequency domain is a process of convolution with another response which consists solely of an impulse at the new centre frequency of the block, that is the spectrum of a signal having a single frequency component at the new centre frequency, that is the spectrum of a sinusoidal signal. This convolution in the frequency domain manifests itself in the time domain by a multiplication of the impulse response by a cosine wave of the appropriate frequency. So to shift the impulse response of a low-pass filter up to a new centre angular frequency w, we generate:

$$h'[n] = h[n]\cos(nw)$$

For the special case that the new rate is half the sample rate (i.e. when $w=\pi$) the modulation consists of multiplying h[] by +1, -1, +1, -1, ... See the implementations of `NRHighPass()` and `NRBandPass()` to exemplify this.

While the Hamming window is a simple and useful method for truncating the impulse response, it is rather inflexible. The Hamming window gives us a flat pass band and fairly good attenuation in the stop band, but at the expense of a fairly broad transition band. Sometimes we wouldn't mind more ripple in the pass band if that allowed us to have a narrower transition region. We can achieve this by designing a windowing function with the appropriate properties and use that to truncate the sinc function impulse response. A simple way to achieve this is with the parametric **Kaiser** window.

The Kaiser window is defined as:

$$w[n] = \frac{I_0\left(\alpha\sqrt{1-\left(\frac{n}{M}\right)^2}\right)}{I_0(\alpha)}, \quad -M \le n \le M$$

Where I_0 is the zeroth order modified Bessel function of the first kind, which has a complex mathematical description, but a straightforward algorithmic description shown in Algorithm 7.2. The parameter α controls the tradeoff between transition width and ripple. We can give a heuristic for the selection of α and *M* from a specification of (i) the allowed stop band ripple *A* expressed as decibels below the passband, and (ii) the transition width Δ expressed as a fraction of the sample rate:

if A >= 50,	then $\alpha = 0.1102(A - 8.7)$
if 21 < A < 50,	then $\alpha = 0.5842(A - 21)^{0.4} + 0.07886(A - 21)$
if A <= 21,	then $\alpha = 0$

Where the ripple level is less than 21dB from the pass band, then we end up with a rectangular window. *M*, the half-size of the window is then given by:

$$M \ge \frac{A - 7.95}{28.72\Delta}$$

Algorithm 7.2, Kaiser designs a window according to these formulae, which could be used to substitute for the Hamming window used in `NRLowPass`, etc.

Recursive Filter Design

The design of recursive filters is considerably more complex than the design of non-recursive filters. The design task is to place poles and zeros on the complex plane, such that the frequency response has the desired characteristics. This can either be performed through experience or with an interactive design tool. We choose to look at the design formulae developed for analogue filters: specifically for Butterworth designs. These can then be transformed into the digital domain through the use of an algebraic transformation that maps the analogue frequencies 0..∞ into the periodically repeating angular frequencies 0..2π. The mechanism used is called the *bilinear transformation*.

A digital Butterworth low-pass filter of order *n* has n poles arranged on a circular arc on the complex plane and n zeros located at z = (-1,0). If *n* is even, the poles form *n*/2 conjugate pairs. If *n* is odd, then a pole on the real axis is added. If we keep to even *n* for convenience the location of the poles are given by the formulae:

$$polepair[m] = \frac{RE[m]}{d} \pm i\frac{IM[m]}{d}, \; m = \frac{n}{2},..,n-1$$

where the real and imaginary parts are given by:

$$RE[m] = 1 - \tan^2\left(\frac{w_c}{2}\right)$$

$$IM[m] = 2\tan\left(\frac{w_c}{2}\right)\sin\left(\frac{(2m+1)\pi}{2n}\right)$$

and where

$$d = 1 - 2\tan(\frac{w_c}{2})\cos(\frac{(2m+1)\pi}{2n}) + \tan^2(\frac{w_c}{2})$$

For simplicity of implementation, the routine `ButterworthPoles()` calculates the positions of n/2 poles on the upper half of the complex plane, and then each pole is combined with its complex conjugate to implement a separate linear system (a *second-order section*) in the routine `ButterworthLowPass()`. The resulting chain of linear systems is returned for filtering operation.

The transformation of the low-pass design into high-pass is straightforward: it simply involves reflecting the poles and zeros around the imaginary axis. See `ButterworthHighPass()`.

The transformation of the low-pass design into band-pass is more complex and involves a doubling of the filter order and a rotation of the poles around the unit circle. For a band-pass filter extending from angular frequencies w_l to w_h, we first calculate the poles and zeros for a low pass filter with a cut-off frequency $w_c = (w_h - w_l)/2$. Then each pole or zero at z is moved to a new location by the formula:

$$z' = 0.5A(1+z) \pm \sqrt{0.25A^2(1+z)^2 - z}$$

where A is given by:

$$A = \frac{\cos(\frac{w_h + w_l}{2})}{\cos(\frac{w_h - w_l}{2})}$$

See `ButterworthBandPass()`.

A Butterworth design gives us a filter with a maximally-flat passband and good stop-band attenuation, so that it makes a good general purpose filter. As before, however, other designs can give us a narrower transition region at the cost of greater ripples in the pass- and/or stop-bands. Many DSP packages give design programs for Chebyshev and Elliptic filter designs, but their essential operation is the same as the Butterworth design.

Algorithms

Algorithm 7.1 Non-recursive filter design

```
' sinc(x) = sin(x) / x
Public Function sinc(ByVal x As Double) As Double
    If (x = 0) Then
        Return 1
    Else
        Return Math.Sin(x) / x
    End If
End Function

' Design non-recursive low-pass filter.
Public Shared Function NRLowPass(ByVal freq As Double, ByVal ncoeff As Integer)
As LTISystem

    ' convert frequency
    Dim omega As Double = 2 * Math.PI * freq

    ' build half-sized window from sinc function
    Dim nhalf As Integer = ncoeff \ 2
    Dim hwv As New Waveform(nhalf, 1.0)
    For i As Integer = 1 To nhalf
        hwv(i) = omega * sinc(i * omega) / Math.PI
    Next

    ' window with (half-)hamming window
    For i As Integer = 1 To nhalf
        hwv(i) *= 0.54 + 0.46 * Math.Cos(i * Math.PI / nhalf)
    Next

    ' create new LTI System
    Dim lpfilt As New LTISystem(2 * nhalf, 0)

    ' copy impulse response into system (indexed -nhalf .. 0 .. nhalf)
    lpfilt.a(nhalf) = omega / Math.PI
    For i As Integer = 1 To nhalf
        lpfilt.a(nhalf - i) = hwv(i)
        lpfilt.a(nhalf + i) = hwv(i)
    Next

    Return lpfilt

End Function

' Transform non-recursive low-pass filter to high-pass
Public Shared Function NRHighPassTransform(ByVal lpfilt As LTISystem) As
LTISystem
    Dim hpfilt As New LTISystem(lpfilt)

    ' now modulate with cos(n*PI) = +1,-1,+1,-1,...
    Dim nhalf As Integer = lpfilt.a.Length \ 2
    hpfilt.a(nhalf) = lpfilt.a(nhalf)
    For i As Integer = 1 To nhalf Step 2
        hpfilt.a(nhalf - i) = -lpfilt.a(nhalf - i)
        hpfilt.a(nhalf + i) = -lpfilt.a(nhalf + i)
    Next
    Return hpfilt
End Function

' Transform non-recursive low-pass filter to band-pass
Public Shared Function NRBandPassTransform(ByVal lpfilt As LTISystem, ByVal
freq As Double) As LTISystem
    Dim bpfilt As New LTISystem(lpfilt)

    ' now modulate with cos(n*centrefreq)
    Dim nhalf As Integer = lpfilt.a.Length \ 2
    Dim cf As Double = 2 * Math.PI * freq
    bpfilt.a(nhalf) = 2 * lpfilt.a(nhalf)
```

```
    For i As Integer = 1 To nhalf
        bpfilt.a(nhalf - i) = 2 * lpfilt.a(nhalf - i) * Math.Cos(i * cf)
        bpfilt.a(nhalf + i) = 2 * bpfilt.a(nhalf + i) * Math.Cos(i * cf)
    Next

    Return bpfilt
End Function

' Design non-recursive high-pass filter
Public Shared Function NRHighPass(ByVal freq As Double, ByVal ncoeff As
Integer) As LTISystem
    Dim hpfilt As LTISystem = NRHighPassTransform(NRLowPass(0.5 - freq,
ncoeff))
    Return hpfilt
End Function

' Design non-recursive band-pass filter
Public Shared Function NRBandPass(ByVal lofreq As Double, ByVal hifreq As
Double, ByVal ncoeff As Integer) As LTISystem
    Dim bpfilt As LTISystem = NRBandPassTransform(NRLowPass((hifreq - lofreq) /
2, ncoeff), (lofreq + hifreq) / 2)
    Return bpfilt
End Function
```

Algorithm 7.2 - Kaiser Window

```
' zeroth order modified Bessel function of the first kind
Private Const ITERLIMIT As Integer = 15
Private Const CONVERGE As Double = 0.00000001

Private Shared Function besselI0(ByVal p As Double) As Double
    ' initialise iterative loop
    p = p / 2
    Dim n As Double = 1
    Dim t As Double = 1
    Dim d As Double = 1

    ' iteration
    Dim k As Integer = 1
    Dim v As Double
    Do
        n = n * p
        d = d * k
        v = n / d
        t = t + v * v
        k += 1
    Loop While ((k < ITERLIMIT) And (v > CONVERGE))

    Return t
End Function

' Calculates Kaiser window to specification
Public Shared Function Kaiser(ByVal ripple As Double, ByVal twidth As Double,
ByVal kmaxlen As Integer) As Waveform
    Dim alpha As Double     ' window coefficient
    Dim hlen As Integer     ' half window length
    Dim v As Double, d As Double

    ' set Kaiser window coefficient (design rule)
    If (ripple <= 21) Then
        alpha = 0
    ElseIf (ripple < 50) Then
        alpha = 0.5842 * Math.Exp(0.4 * Math.Log(ripple - 21)) + 0.07886 *
(ripple - 21)
    Else
        alpha = 0.1102 * (ripple - 8.7)
    End If

    ' set Kaiser window size (design rule)
    If (ripple < 7.95) Then
        hlen = 0
    Else
        hlen = 1 + ((ripple - 7.95) / (28.72 * twidth))
    End If
    If ((hlen + 1) > kmaxlen) Then hlen = kmaxlen - 1

    ' build output window
    Dim kwin As New Waveform(hlen + 1, 1.0)

    ' calculate window 1..hlen+1
    d = besselI0(alpha)
    kwin(1) = 1.0
    For n As Integer = 1 To hlen
        v = n / hlen
        kwin(n + 1) = besselI0(alpha * Math.Sqrt(1 - v * v)) / d
    Next

    ' return windows
    Return kwin

End Function
```

Algorithm 7.3 Butterworth Recursive Filter Design - Chain of Linear Systems

```
' Supports chain of linear systems
Public Class LTISystemChain
    Private _nsection As Integer = 0
    Private _section() As LTISystem

    ''' Creates chain of linear systems
    Public Sub New(ByVal n As Integer)
        _nsection = n
        ReDim _section(_nsection - 1)
    End Sub

    ''' Creates copy of chain of linear systems.
    Public Sub New(ByVal lch As LTISystemChain)
        _nsection = lch.NSection
        ReDim _section(_nsection - 1)
        For i As Integer = 0 To lch.NSection - 1
            _section(i) = lch.Section(i)
        Next
    End Sub

    ''' Gets number of linear systems in chain.
    Public ReadOnly Property NSection() As Integer
        Get
            Return _nsection
        End Get
    End Property

    ''' Gets/sets the linear systems in the chain
    Public Property Section(ByVal idx As Integer) As LTISystem
        Get
            Return _section(idx)
        End Get
        Set(ByVal value As LTISystem)
            _section(idx) = value
        End Set
    End Property

    ''' Clears state memory in all linear system chain
    Public Sub Clear()
        For i As Integer = 0 To _nsection - 1
            _section(i).Clear()
        Next
    End Sub

    ''' Processes a single sample value through the chain
    Default Public ReadOnly Property Item(ByVal ival As Double) As Double
        Get
            For i As Integer = 0 To _nsection - 1
                ival = _section(i)(ival)
            Next
            Return ival
        End Get
    End Property

    ''' Pass a waveform through the filter
    Public Function Filter(ByRef iwv As Waveform) As Waveform
        Dim owv As New Waveform(iwv.Count, iwv.Rate)
        Clear()
        For i As Integer = iwv.First To iwv.Last
            owv(i) = Item(iwv(i))
        Next
        Return owv
    End Function
```

```
    ''' Pass a signal through the filter
    Public Function Filter(ByRef iwv As Signal) As Signal
        Dim owv As New Signal(iwv.Count, iwv.Rate)
        Clear()
        For i As Integer = iwv.First To iwv.Last
            owv(i) = Item(iwv(i))
        Next
        Return owv
    End Function

    ''' Gets response of linear system chain at given frequency
    Public ReadOnly Property Response(ByVal freq As Double) As Complex
        Get
            If (_nsection = 0) Then Return New Complex(0, 0)
            Dim resp As Complex = _section(0).Response(freq)
            For i As Integer = 1 To _nsection - 1
                resp = resp * _section(i).Response(freq)
            Next
            Return resp
        End Get
    End Property
End Class
```

Butterworth filter design:

```
' Calculates pole positions for Butterworth design low-pass filter
Public Shared Function ButterworthPoles(ByVal freq As Double, ByVal nsection As
Integer) As Complex()

    ' get array of complex values
    Dim poles(nsection - 1) As Complex

    ' calculate angles
    Dim w As Double = Math.PI * freq
    Dim tanw As Double = Math.Sin(w) / Math.Cos(w)

    ' calculate +im pole position for each section
    For m As Integer = nsection To 2 * nsection - 1
        ' Butterworth formula adapted to z-plane
        Dim ang As Double = (2 * m + 1) * Math.PI / (4 * nsection)
        Dim d As Double = 1 - 2 * tanw * Math.Cos(ang) + tanw * tanw
        poles(m - nsection) = New Complex((1 - tanw * tanw) / d, 2 * tanw *
Math.Sin(ang) / d)
    Next
    Return poles
End Function

' Design Butterworth low-pass recursive filter
Public Shared Function ButterworthLowPass(ByVal freq As Double, ByVal nsection
As Integer) As LTISystemChain
    ' create empty system chain
    Dim lpfilt As New LTISystemChain(nsection)

    ' get pole positions
    Dim pol() As Complex = ButterworthPoles(freq, nsection)

    ' convert each conjugate pole pair to difference equation
    For i As Integer = 0 To nsection - 1
        lpfilt.Section(i) = New LTISystem(2, 2)
        ' put in conjugate pole pair
        lpfilt.Section(i).b(1) = -2.0 * pol(i).Real
        lpfilt.Section(i).b(2) = pol(i).Real * pol(i).Real + pol(i).Imag *
pol(i).Imag
        ' put 2 zeros at (-1,0)
        Dim tot As Double = 4.0 / (1 + lpfilt.Section(i).b(1) +
lpfilt.Section(i).b(2))
        lpfilt.Section(i).a(0) = 1.0 / tot
        lpfilt.Section(i).a(1) = 2.0 / tot
```

```
        lpfilt.Section(i).a(2) = 1.0 / tot
    Next

    Return lpfilt

End Function

' Design Butterworth high-pass recursive filter
Public Shared Function ButterworthHighPass(ByVal freq As Double, ByVal nsection
As Integer) As LTISystemChain
    ' create empty system chain
    Dim hpfilt As New LTISystemChain(nsection)

    ' get pole positions for LP prototype
    Dim pol() As Complex = ButterworthPoles(0.5 - freq, nsection)

    ' flip all the poles over to get high pass
    For i As Integer = 0 To nsection - 1
        pol(i) = New Complex(-pol(i).Real(), pol(i).Imag())
    Next

    ' convert each conjugate pole pair to difference equation
    For i As Integer = 0 To nsection - 1
        hpfilt.Section(i) = New LTISystem(2, 2)
        ' put in conjugate pole pair
        hpfilt.Section(i).b(1) = -2.0 * pol(i).Real
        hpfilt.Section(i).b(2) = pol(i).Real * pol(i).Real + pol(i).Imag *
pol(i).Imag
        ' put 2 zeros at (1,0)
        hpfilt.Section(i).a(0) = 1.0
        hpfilt.Section(i).a(1) = -2.0
        hpfilt.Section(i).a(2) = 1.0
        ' normalise to unity gain at Fs/2
        Dim gain As Double = hpfilt.Section(i).Response(0.5).Mag
        hpfilt.Section(i).a(0) = hpfilt.Section(i).a(0) / gain
        hpfilt.Section(i).a(1) = hpfilt.Section(i).a(1) / gain
        hpfilt.Section(i).a(2) = hpfilt.Section(i).a(2) / gain
    Next

    Return hpfilt

End Function

' Design Butterworth band-pass recursive filter
Public Shared Function ButterworthBandPass(ByVal lofreq As Double, ByVal hifreq
As Double, ByVal nsection As Integer) As LTISystemChain
    ' create empty system chain
    If (nsection Mod 2) = 1 Then nsection += 1
    Dim bpfilt As New LTISystemChain(nsection)

    ' get pole positions for LP prototype
    Dim pol() As Complex = ButterworthPoles(hifreq - lofreq, nsection / 2)

    ' translate the poles to band-pass position
    Dim bpol(nsection) As Complex
    Dim wlo As Double = 2 * Math.PI * lofreq
    Dim whi As Double = 2 * Math.PI * hifreq
    Dim ang As Double = Math.Cos((whi + wlo) / 2) / Math.Cos((whi - wlo) / 2)
    For i As Integer = 0 To nsection / 2 - 1
        Dim p1 As New Complex(pol(i).Real() + 1, pol(i).Imag())
        Dim tmp As Complex = Complex.Sqrt(p1 * p1 * ang * ang * 0.25 - pol(i))
        bpol(2 * i) = (p1 * ang * 0.5) + tmp
        bpol(2 * i + 1) = (p1 * ang * 0.5) - tmp
    Next

    ' convert each conjugate pole pair to difference equation
    For i As Integer = 0 To nsection - 1
        bpfilt.Section(i) = New LTISystem(2, 2)
        ' put in conjugate pole pair
        bpfilt.Section(i).b(1) = -2.0 * bpol(i).Real
        bpfilt.Section(i).b(2) = bpol(i).Real * bpol(i).Real + bpol(i).Imag *
bpol(i).Imag
```

```
        ' put zeros at (-1,0) and (1,0)
        bpfilt.Section(i).a(0) = 1.0
        bpfilt.Section(i).a(1) = 0.0
        bpfilt.Section(i).a(2) = -1.0
        ' normalise to unity gain at centre of filter
        Dim gain As Double = bpfilt.Section(i).Response((hifreq + lofreq) /
2).Mag
        bpfilt.Section(i).a(0) = bpfilt.Section(i).a(0) / gain
        bpfilt.Section(i).a(1) = bpfilt.Section(i).a(1) / gain
        bpfilt.Section(i).a(2) = bpfilt.Section(i).a(2) / gain
    Next

    Return bpfilt

End Function
```

Bibliography

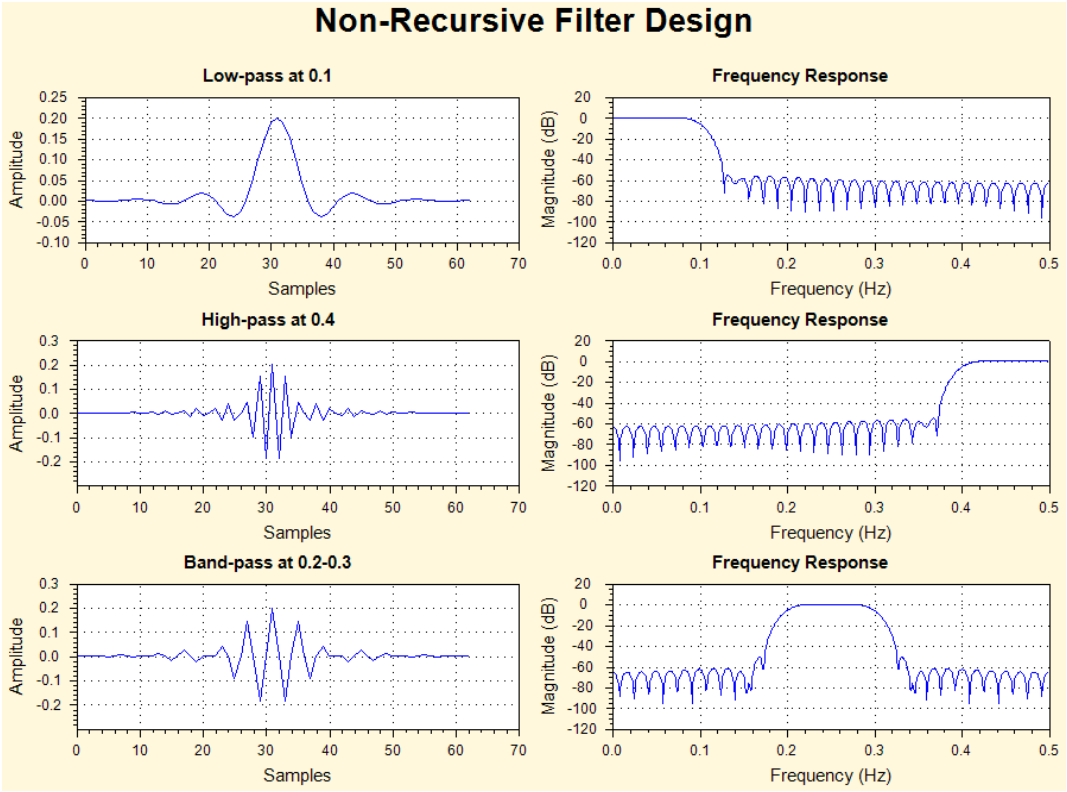
Lynn & Fuerst, Introductory Digital Signal Processing, Chapter 5.

Orfanidis, Introduction to Signal Processing, Chapters 10 & 11.

Example Programs

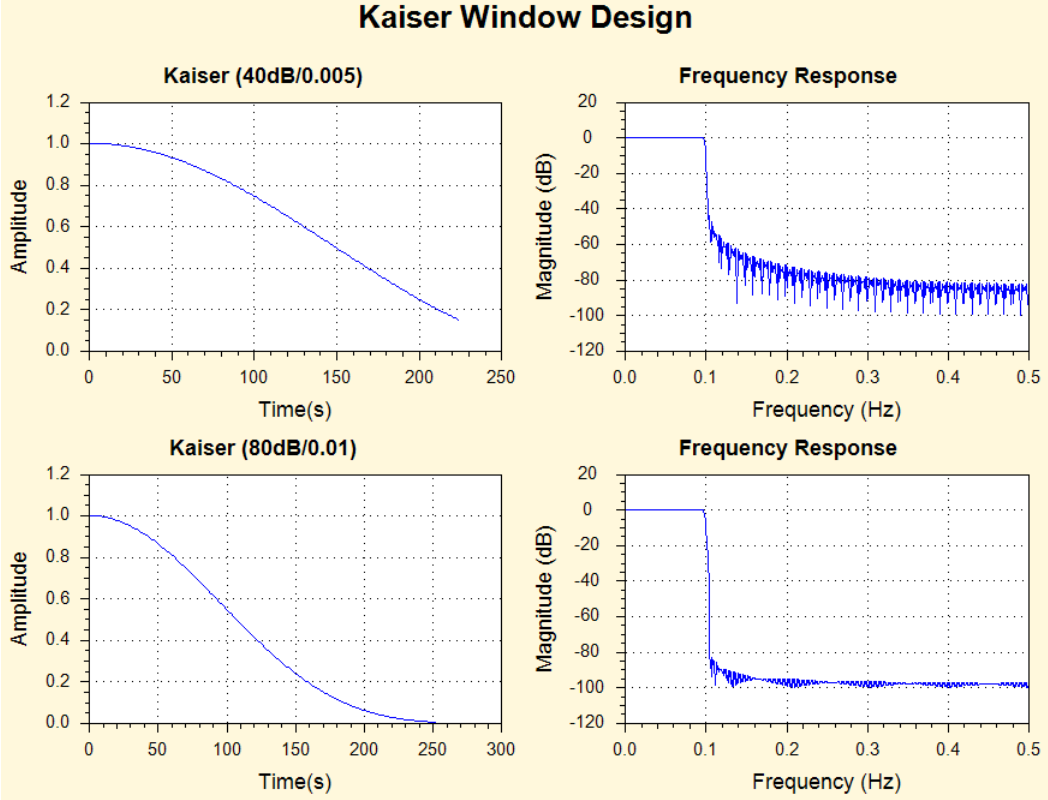
Example 7.1 Non-recursive filter design

```
Imports BasicDSP
Imports ZedGraph
Public Class TestNonRecFilter

    Private Sub TestNonRecFilter_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ' create graphs
        Dim gp As New Graph(Me.CreateGraphics, zgc, 3, 2, "Non-Recursive Filter
Design")

        ' calculate low-pass filter and plot
        Dim lp As LTISystem = Filter.NRLowPass(0.1, 63)
        gp.PlotCoeff(1, lp.a, "Low-pass at 0.1", "Amplitude", "Samples")

        ' calculate frequency response and plot
        Dim lpf As New Spectrum(500, 0.5)
        For i As Integer = 0 To 499
            lpf(i) = lp.Response(i / 1000.0)
        Next
        gp.PlotDbSpectrum(2, lpf, "Frequency Response")

        ' calculate high-pass filter and plot
        Dim hp As LTISystem = Filter.NRHighPass(0.4, 63)
        gp.PlotCoeff(3, hp.a, "High-pass at 0.4", "Amplitude", "Samples")

        ' calculate frequency response and plot
        Dim hpf As New Spectrum(500, 0.5)
        For i As Integer = 0 To 499
            hpf(i) = hp.Response(i / 1000.0)
        Next
        gp.PlotDbSpectrum(4, hpf, "Frequency Response")

        ' calculate band-pass filter and plot
        Dim bp As LTISystem = Filter.NRBandPass(0.2, 0.3, 63)
        gp.PlotCoeff(5, bp.a, "Band-pass at 0.2-0.3", "Amplitude", "Samples")

        ' calculate frequency response and plot
        Dim bpf As New Spectrum(500, 0.5)
        For i As Integer = 0 To 499
            bpf(i) = bp.Response(i / 1000.0)
        Next
        gp.PlotDbSpectrum(6, bpf, "Frequency Response")

    End Sub
End Class
```


Non-Recursive Filter Design
Low-pass at 0.1
Amplitude
Samples
Frequency Response
Magnitude (dB)
Frequency (Hz)
High-pass at 0.4
Amplitude
Samples
Frequency Response
Magnitude (dB)
Frequency (Hz)
Band-pass at 0.2-0.3
Amplitude
Samples
Frequency Response
Magnitude (dB)
Frequency (Hz)

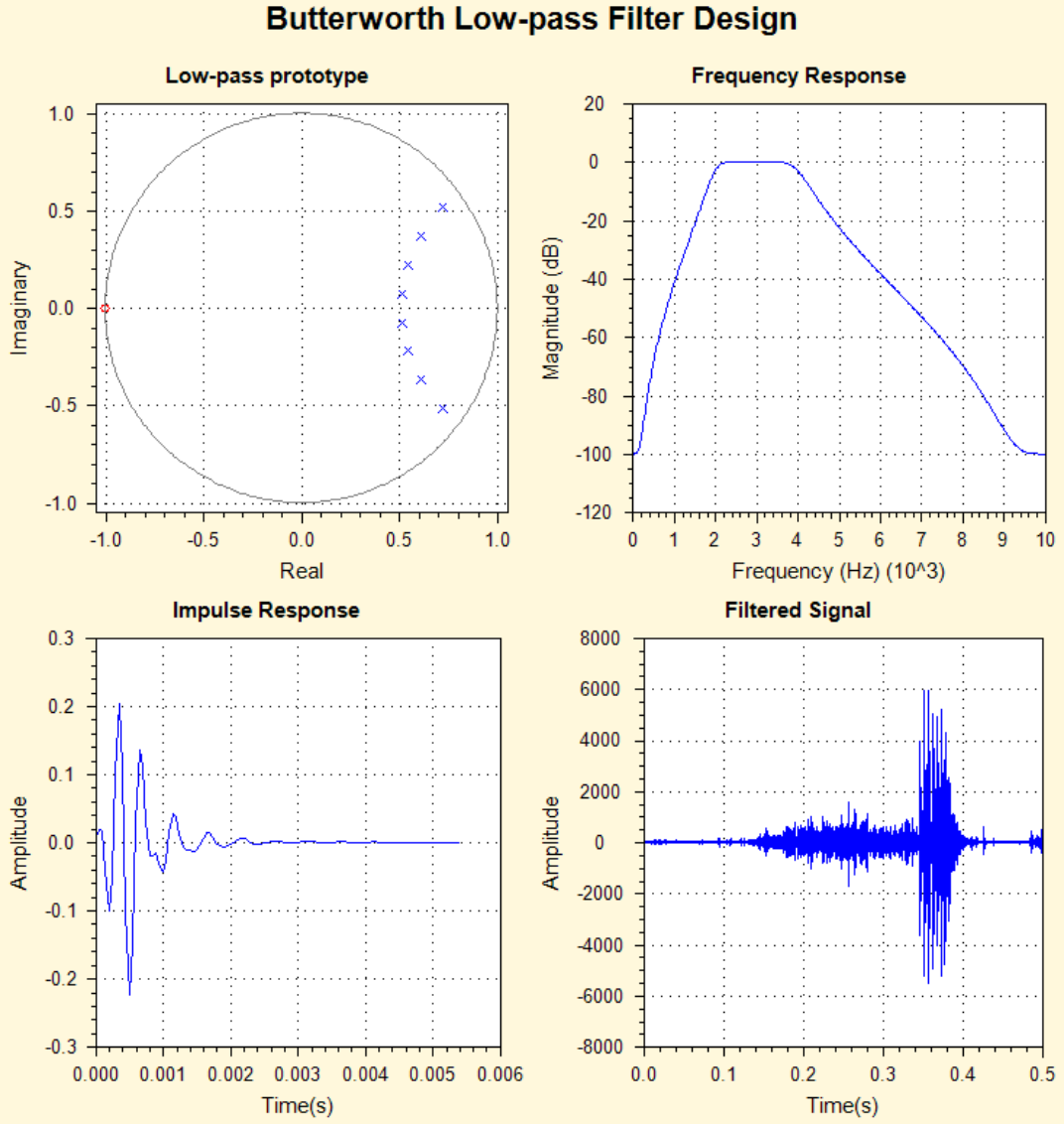
Example 7.2 - Demonstrate Kaiser Window

```
Imports BasicDSP
Imports ZedGraph
Public Class TestKaiser
    Const MAXKAISERWIN As Integer = 256
    ' Create non-recursive low-pass filter using Kaiser window
    ' freq      corner freq  (fraction of sampling rate)
    ' ripple    allowed ripple (dB)
    ' twidth    transition width  (fraction of sampling rate)
    Public Function KaiserLowPass(ByVal freq As Double, ByVal ripple As Double,
ByVal twidth As Double) As LTISystem

        ' get (half-)Kaiser window
        Dim kwin As Waveform = Window.Kaiser(ripple, twidth, MAXKAISERWIN)
        Dim nhalf As Integer = kwin.Count() - 1

        ' generate one half of coefficients from windowed sinc() function
        Dim omega As Double = 2 * Math.PI * freq
        For i As Integer = 0 To nhalf
            kwin(i + 1) *= omega * sinc(i * omega) / Math.PI
        Next

        ' copy into LTI System
        Dim lpfilt As New LTISystem(2 * nhalf, 0)
        lpfilt.a(nhalf) = kwin(1)
        For i As Integer = 1 To nhalf
            lpfilt.a(nhalf - i) = kwin(i + 1)
            lpfilt.a(nhalf + i) = kwin(i + 1)
        Next

        Return lpfilt
    End Function

    Private Sub TestKaiser_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ' initialise graphics
        Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 2, "Kaiser Window
Design")

        ' plot Kaiser window 1
        Dim kwv1 As Waveform = Window.Kaiser(40.0, 0.005, MAXKAISERWIN)
        gp.PlotWaveform(1, kwv1, "Kaiser (40dB/0.005)")

        ' calculate and plot frequency response
        Dim lp1 As LTISystem = KaiserLowPass(0.1, 40.0, 0.005)
        Dim lpf1 As New Spectrum(500, 0.5)
        For i As Integer = 0 To 499
            lpf1(i) = lp1.Response(i / 1000.0)
        Next
        gp.PlotDbSpectrum(2, lpf1, "Frequency Response")

        ' plot Kaiser window 2
        Dim kwv2 As Waveform = Window.Kaiser(80.0, 0.01, MAXKAISERWIN)
        gp.PlotWaveform(3, kwv2, "Kaiser (80dB/0.01)")

        ' calculate and plot frequency response
        Dim lp2 As LTISystem = KaiserLowPass(0.1, 80.0, 0.01)
        Dim lpf2 As New Spectrum(500, 0.5)
        For i As Integer = 0 To 499
            lpf2(i) = lp2.Response(i / 1000.0)
        Next
        gp.PlotDbSpectrum(4, lpf2, "Frequency Response")

    End Sub
End Class
```


Kaiser Window Design
Kaiser (40dB/0.005)
Amplitude
Time(s)
Frequency Response
Magnitude (dB)
Frequency (Hz)
Kaiser (80dB/0.01)
Amplitude
Time(s)
Frequency Response
Magnitude (dB)
Frequency (Hz)

E**xample 7.3** Recursive filter design

```
Imports BasicDSP
Imports ZedGraph
Public Class TestButter
    Const ILIMIT As Double = 0.00001    ' length limit for impulse response
    Const FILENAME As String = "c:/sfs/demo/six.wav"
    Const SAMPLE As Double = 0.5        ' how much waveform to display
    Const CUTFREQ As Double = 2000      ' cut-off frequency in Hertz
    Const NSECTION As Integer = 4       ' number of filter sections

    Private Sub TestButter_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ' initialise graphics
        Dim gp As New Graph(Me.CreateGraphics, zgc, 2, 2, "Butterworth Low-pass
Filter Design")

        ' get waveform to set sample rate for graphs
        Dim isig As New Signal(FILENAME)

        ' get poles and plot them
        Dim pol() As Complex = Filter.ButterworthPoles(CUTFREQ / isig.Rate,
NSECTION)
        Dim poles(2 * pol.Length - 1) As Complex
        Dim zeros(0) As Complex
        For i As Integer = 0 To pol.Length - 1
            poles(i) = New Complex(pol(i))
            poles(pol.Length + i) = New Complex(pol(i).Real, -pol(i).Imag)
        Next
        zeros(0) = New Complex(-1, 0)
        gp.PlotZPlane(1, poles, zeros, "Low-pass prototype")

        ' build chain of second order sections
        Dim lpfilt As LTISystemChain = Filter.ButterworthBandPass(CUTFREQ /
isig.Rate, 4000 / isig.Rate, NSECTION)

        ' calculate frequency response
        Dim lpfr As New Spectrum(500, isig.Rate / 2)
        For i As Integer = 0 To 499
            lpfr(i) = lpfilt.Response(i / 1000.0)
        Next
        gp.PlotDbSpectrum(2, lpfr, "Frequency Response")

        ' measure and plot impulse response
        Dim lpir As New Waveform(0, isig.Rate)
        Dim lval As Double = 0          ' last output
        Dim oval As Double = lpfilt(1.0)    ' put in unit pulse
        While ((Math.Abs(oval) > ILIMIT) Or (Math.Abs(oval - lval) > ILIMIT) Or
(lpir.Count < 100))
            lpir.Add(oval)          ' append sample
            lval = oval             ' remember sample
            oval = lpfilt(0.0)      ' get next sample
        End While
        gp.PlotWaveform(3, lpir, "Impulse Response")

        ' plot some filtered speech
        isig = isig.Cut(1, SAMPLE * isig.Rate)
        Dim fwv As New Waveform(isig.Count, isig.Rate)
        lpfilt.Clear()
        For i As Integer = 1 To isig.Count
            fwv(i) = lpfilt(isig(i))
        Next
        gp.PlotWaveform(4, fwv, "Filtered Signal")

    End Sub
End Class
```



Exercises

7.1. Implement a band-pass filter that emulates the telephone system, using a non-recursive filter with 32 coefficients with a response extending from 300Hz to 3500Hz. Use it to filter and display a speech signal along with some representative spectra and a frequency response curve.

7.2. Adapt the program from exercise 7.1 to use a recursive filter with 2 second-order sections. Find methods for timing the execution speed of programs and compare the efficiency of this program to the original (you may need to comment out the display part for timing).

UNIT 8:
LINEAR PREDICTION

BASICDSP

8.1 Summary

Linear prediction is a method for signal source modelling dominant in speech signal processing and having wide application in other areas. Starting with a demonstration of the relationship between linear prediction and the general difference equation for linear systems, the unit shows how the linear prediction equations are formulated and solved. The unit then discusses the use of linear prediction for modelling the source of a signal and the signal spectrum.

When you have worked through this unit you should:

- understand how linear prediction can provide a model of a signal source
- be able to state the operational limitations of the modelling
- understand in qualitative terms how the equations are solved on a computer system
- know how and when to use linear prediction as an alternative to spectral analysis by the Fourier transform

8.2 Concepts

Nature of linear prediction

The object of linear prediction is to form a model of a linear time-invariant digital system through observation of input and output sequences. That is: to estimate a set of coefficients which will describe the behaviour of an LTI system when its design is not available to us and we cannot choose what input to present.

Specifically, linear prediction calculates a set of coefficients which provide an estimate - or *a prediction* - for a forthcoming output sample y'[n] given knowledge of previous input (x[]) and/or output (y[]) samples:

$$y'[n] = \sum_{k=0}^{p} a_k x[n-k] - \sum_{k=1}^{q} b_k y[n-k]$$

Where *a* and *b* are called *predictor coefficients*. It is immediately clear that this estimate of the next output sample given the predictor coefficients and previous input and output samples is directly related to the general system difference equation we met in Unit 4. Even though we might not know the order of some LTI system, such an estimator is an inherently sensible way to model its properties.

The commonest form of linear prediction used in signal processing is one in which the *a* coefficients are zero, so that the output estimate is made entirely on the basis of previous output samples:

$$y'[n] = -\sum_{k=1}^{q} b_k y[n-k]$$

This is called an *all-pole* model, which may be seen by analogy with the frequency response equations of LTI systems. The use of an all pole model is motivated by a number of reasons:

(i) we often do not have access to the input sequence,
(ii) many simple signal sources are faithfully modelled by an all-pole model,
(iii) the all-pole model gives a system of equations which may be efficiently solved.

The relation of the linear predictor equation with the LTI system equation may be more readily understood by introducing a *predictor error* signal e[n], defined as the difference between the real output and the prediction:

$$e[n] = y[n] - y'[n]$$

Hence we may write:

$$y[n] = e[n] - \sum_{k=1}^{q} b_k y[n-k]$$

giving us a formulation in which the error signal takes the role of the excitation signal, and the predictor coefficients define the filter.

Solution of linear prediction equations

To estimate the predictor coefficients b_k we first observe the behaviour of the system over N samples, and set the order q of the predictor required (usually N is much greater than q). We calculate the predictor coefficients by finding values which minimise the energy in the error signal over the N samples[1]. This is called a *least-squares* minimisation, and leads to a system of q equations in q unknowns which may be solved to find the best fitting predictor coefficients:

$$\sum_{k=0}^{q}\left(b_k \sum_{n=0}^{N-1} y[n-k]y[n-j]\right) = 0, \quad j = 1,2,\ldots,q$$

Where b_0 is taken to be 1.

There are a number of methods for finding the solution to this system of linear equations. The formulation above is sometimes known as the *covariance* formulation and is appropriate when estimating coefficients from a sample of a non-stationary signal. If instead we assume that the signal is zero outside the N sample region, we can find a simpler formulation:

$$\sum_{k=0}^{q}\left(b_k \sum_{n=j}^{N-1} y[n]y[n-k+j]\right) = 0, \quad j = 1,2,\ldots,q$$

This is called the *autocorrelation* formulation, since it contains components equivalent to the autocorrelation coefficient r_i defined as:

[1] This is equivalent to flattening the spectrum of the error signal.

$$r_i = \sum_{n=-\infty}^{\infty} y[n]y[n-i]$$

The autocorrelation formulation of the least squares fit of the predictor coefficients produces a system of linear equations which can be represented in matrix form and solved using standard methods such as Gaussian elimination. See `LPCSolve()`.

However the autocorrelation system of equations can be solved much more efficiently since the autocorrelation coefficients in the matrix form of the equations have a very simple and symmetric structure. This allows for a recursive solution procedure in which each predictor coefficient may be derived in turn and used to calculate the next coefficient. This is the dominant method for solving the linear prediction equations, and is usually credited to Levinson and Durbin. See `LPCAutocorrelation()`.

Spectrum Modelling

The determination of the *b* coefficients from some section of signal provides us with an all-pole filter that characterises the source on the assumption that the input signal is white (i.e. has a flat spectrum). Thus the frequency response of the derived filter gives a smooth model of the signal spectrum under the predictor assumptions. See Example program 8.2.

Root Solving

It is also possible to determine the location of the individual poles that make up the linear prediction derived filter - this is useful as the poles can often be considered as being caused by resonances in the driving system. To find the poles, it is necessary to factorise the predictor polynomial, this can be performed numerically through a root-solving procedure, see `FindRoots()`. The location of the poles can then be interpreted in terms of the frequencies and bandwidths of the system resonances.

Bibliography

T. Parsons, Voice and Speech Processing, McGraw-Hill 1987, Ch 6.
Markel & Grey, Linear Prediction of Speech, Springer-Verlag, 1976.

Algorithms

Algorithm 8.1 Linear Prediction Solution - Direct Method

```
Const EPSILON As Double = 0.0000000001

' Gauss-Jordan elimination on set of simultaneous equations
Private Shared Function Eliminate(ByRef mat(,) As Double, ByVal n As Integer)
As Double

    Dim pos As Integer          ' pivot row
    Dim max As Double           ' maximum value in column
    Dim det As Double = 1.0      ' determinant

    ' for each row
    For i As Integer = 1 To n

        ' find pivot row from max column entry
        max = 0
        pos = 1
        For j As Integer = i To n
            If (Math.Abs(mat(j, i)) > max) Then
                max = Math.Abs(mat(j, i))
                pos = j
            End If
        Next

        ' check for column of zeros
        If (max < EPSILON) Then Return (0.0)

        ' transpose current row with pivot row
        ' and normalise diagonal element to unity
        max = mat(pos, i)
        For j As Integer = 1 To n + 1
            Dim temp As Double = mat(pos, j)
            mat(pos, j) = mat(i, j)
            mat(i, j) = temp / max
        Next

        ' keep record of determinant
        If (i <> pos) Then
            det = det * -max
        Else
            det = det * max
        End If

        ' reduce matrix by dividing through by row
        For k As Integer = 1 To n
            If (k <> i) Then
                Dim val As Double = mat(k, i)
                For j As Integer = i To n + 1
                    mat(k, j) = mat(k, j) - val * mat(i, j)
                Next
            End If
        Next
    Next

    ' return determinant
    Return det

End Function
```

```
' set up and solve LPC equations
Public Shared Function Solve(ByVal x As Waveform, ByVal order As Integer) As
LTISystem

    ' compute autocorrelations
    Dim r(order) As Double
    Dim ltis As New LTISystem(0, order)
    Dim sum As Double

    For i As Integer = 0 To order
        sum = 0
        For k As Integer = 1 To x.Count - i
            sum += x(k) * x(k + i)
        Next
        r(i) = sum
    Next

    ' build set of linear equations
    Dim mat(order, order + 1) As Double
    For i As Integer = 1 To order
        For j As Integer = 1 To order
           mat(i, j) = r(Math.Abs(i - j))
        Next
        mat(i, order + 1) = -r(i)
    Next

    ' solve them
    If (Eliminate(mat, order) = 0) Then
        ltis.a(0) = 0.0
        Return ltis
    End If

    ' copy coefficients into LTI System
    ltis.a(0) = 1.0
    ltis.b(0) = 1.0
    For i As Integer = 1 To order
        ltis.b(i) = mat(i, order + 1)
    Next

    ' OK
    Return ltis

End Function
```

Algorithm 8.2 Linear Prediction Solution - Autocorrelation Method

```
' Find predictor coefficients from waveform by Levinson-Durbin recursion method
Public Shared Function Auto(ByVal x As Waveform, ByVal order As Integer, ByRef
pe As Double) As LTISystem
    Dim r(order) As Double
    Dim ltis As New LTISystem(0, order)
    Dim sum As Double

    ' compute autocorrelations
    For i As Integer = 0 To order
        sum = 0
        For k As Integer = 1 To x.Count - i
            sum += x(k) * x(k + i)
        Next
        r(i) = sum
    Next

    ' check power in signal
    If (r(0) = 0) Then
        ' no signal !!
        pe = 0
        Return ltis
    End If
```

```
    ' compute predictor coefficients
    Dim pc(order) As Double             ' temporary array
    pe = r(0)                           ' initialise error to total power
    pc(0) = 1.0                         ' first coefficient (b[0]) must = 1

    ' for each coefficient in turn
    For k As Integer = 1 To order

        ' find next coeff from pc[] and r[]
        sum = 0
        For i As Integer = 1 To k
            sum -= pc(k - i) * r(i)
        Next
        pc(k) = sum / pe

        ' perform recursion on pc[]
        For i As Integer = 1 To k \ 2
            Dim pci As Double = pc(i) + pc(k) * pc(k - i)
            Dim pcki As Double = pc(k - i) + pc(k) * pc(i)
            pc(i) = pci
            pc(k - i) = pcki
        Next

        ' calculate residual error
        pe = pe * (1.0 - pc(k) * pc(k))
        If (pe <= 0) Then
            ' no power left in signal!
            Return ltis
        End If
    Next

    ' copy coefficients into LTI System
    ltis.a(0) = 1.0
    ltis.b(0) = 1.0
    For i As Integer = 1 To order
        ltis.b(i) = pc(i)
    Next

    ' return OK
    Return ltis

End Function
```

Algorithm 8.4 Root Solving the Predictor Polynomial

```
Public Class Roots

    Private Const ROUND_ERROR As Double = 0.00000006
    Private Const IM_RANGE As Double = 0.000001

    ' Find single root using Laguerre's method
    Private Shared Function Lagmethod(ByVal p() As Complex, ByVal order As
Integer, ByVal x0 As Complex)

        ' iteratively improve initial estimate x0
        For iter As Integer = 1 To 100
            ' evaluate polynomial and its first and second derivatives at x0
            Dim x As Complex = x0
            Dim xp As Complex = 1
            Dim p0 As Complex = p(0) + p(1) * x
            Dim p1 As Complex = p(1)
            Dim p2 As Complex = 0
            For i As Integer = 2 To order
                p0 += p(i) * xp * x * x
                p1 += i * p(i) * xp * x
                p2 += i * (i - 1) * p(i) * xp
                xp = xp * x
            Next
            ' use Laguerre's iteration to improve estimate
```

```
            Dim G As Complex = p1 / p0
            Dim H As Complex = G * G - p2 / p0
            Dim a1 As Complex = order / (G + Complex.Sqrt((order - 1) * (order
* H - G * G)))
            Dim a2 As Complex = order / (G - Complex.Sqrt((order - 1) * (order
* H - G * G)))
            If (a1.Mag < a2.Mag) Then
                x0 = x - a1
                If (a1.Mag < x0.Mag * ROUND_ERROR) Then Return (x0)
            Else
                x0 = x - a2
                If (a2.Mag < x0.Mag * ROUND_ERROR) Then Return (x0)
            End If
        Next
        Return (x0)
    End Function

    ''' Solve a real polynomial of order N into N complex roots
    Public Shared Function FindRoots(ByVal ap() As Double, ByVal m As Integer)
As Complex()
        Dim p(m) As Complex
        Dim rroots(m) As Complex

        ' copy polynomial coefficients into complex array
        For j As Integer = 0 To m
            p(j) = ap(j)
        Next

        ' find each root in turn
        For j As Integer = m To 1 Step -1
            ' find a root, using origin as initial estimate
            Dim x As Complex = Lagmethod(p, j, New Complex(0, 0))
            ' eliminate very small imaginary parts
            If Math.Abs(x.Imag) <= (IM_RANGE * Math.Abs(x.Real)) Then x.Imag =
0.0
            rroots(j) = x
            ' divide polynomial through by found root
            Dim bp As Complex = p(j)
            For k As Integer = j - 1 To 0 Step -1
                Dim c As Complex = p(k)
                p(k) = bp
                bp = c + x * bp
            Next
        Next

        ' sort roots by increasing Arg
        For j As Integer = 2 To m
            Dim x As Complex = rroots(j)
            Dim i As Integer = j
            While (i > 1) AndAlso (x.Arg < rroots(i - 1).Arg)
                rroots(i) = rroots(i - 1)
                i = i - 1
            End While
            rroots(i) = x
        Next

        Return rroots
    End Function
End Class
```

Example Programs

Example 8.1 Demonstration of Linear Prediction

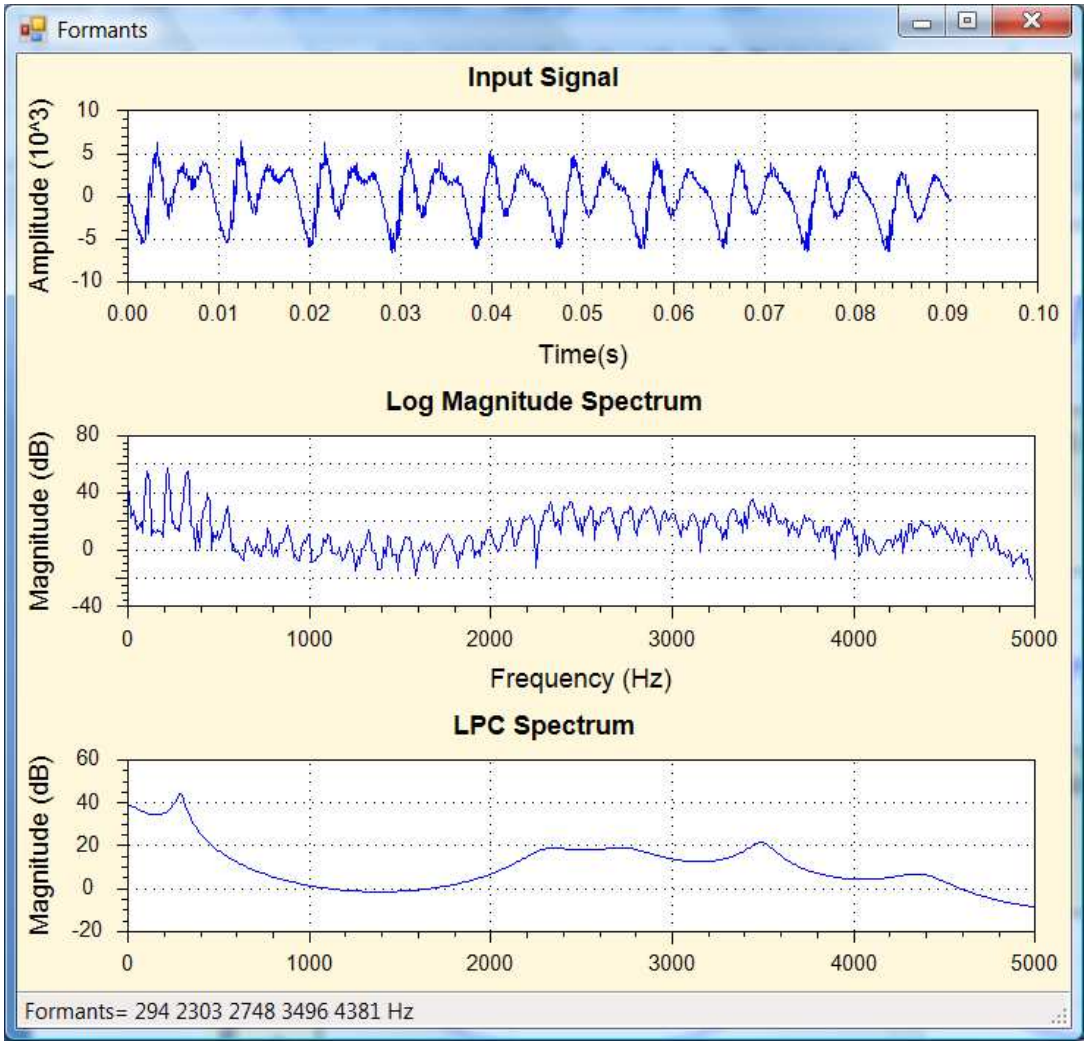
```
Imports BasicDSP
Module Module1
    Const WINDOWSIZE As Integer = 32
    Const NCOEFF As Integer = 4
    Const C1 As Double = 0.16, C2 As Double = -0.12, C3 As Double = 0.08, C4 As
Double = -0.04

    Sub Main()
        Dim iwin As New Waveform(WINDOWSIZE, 1)

        ' make a test signal from recursive filter
        iwin(1) = 0.75          ' put in pulse of power sqr(0.75)
        For i As Integer = 2 To WINDOWSIZE
            ' exploits bad index capability
            iwin(i) = -C1 * iwin(i - 1) - C2 * iwin(i - 2) - C3 * iwin(i - 3) -
C4 * iwin(i - 4)
        Next

        ' do LPC analysis
        Dim osys As LTISystem = LPC.Solve(iwin, NCOEFF)

        If (osys.a(0) = 0) Then
            Console.WriteLine("Analysis failed")
        Else
            Console.WriteLine("Predictor Coefficients:")
            For i As Integer = 1 To NCOEFF
                Console.Write("Coeff{0}={1:F2} ", i, osys.b(i))
            Next
            Console.WriteLine()
        End If

        ' do LPC analysis
        Dim pe As Double
        osys = LPC.Auto(iwin, NCOEFF, pe)

        If (osys.a(0) = 0) Then
            Console.WriteLine("Analysis failed")
        Else
            Console.WriteLine("Predictor Coefficients:")
            For i As Integer = 1 To NCOEFF
                Console.Write("Coeff{0}={1:F2} ", i, osys.b(i))
            Next
            Console.WriteLine(" residual={0:F4}", pe)
        End If

        Console.ReadLine()

    End Sub

End Module
```

Console Output:

```
Predictor Coefficients:
Coeff1=0.16 Coeff2=-0.12 Coeff3=0.08 Coeff4=-0.04
Predictor Coefficients:
Coeff1=0.16 Coeff2=-0.12 Coeff3=0.08 Coeff4=-0.04  residual=0.5625
```

Example 8.2 Linear Prediction Spectrum and Root Solving to find formants

```
Imports BasicDSP
Public Class Formants

    Private Sub Formants_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        ' load a piece of test signal
        Dim isig As New Signal("c:/sfs/demo/six.wav")
        Dim x As Waveform = isig.Cut(24120, 1810).Float

        ' plot the input waveform
        Dim gr As New Graph(Me.CreateGraphics, zgc, 3, 1)
        gr.PlotWaveform(1, x, "Input Signal")

        ' calculate a spectrum of the windowed signal
        Dim y As Spectrum = DFT.ComplexDFT(Window.Hamming(x.Complex))

        ' plot the lower 5000Hz of the spectrum
        Dim hz5000 As Integer = 5000 * y.Count / y.Rate
        gr.PlotDbSpectrum(2, y.Cut(0, hz5000), "Log Magnitude Spectrum")

        ' calculate an LP model
        Dim pe As Double
        Dim np As Integer = x.Rate / 1000
        Dim pc As LTISystem = LPC.Auto(x, np, pe)

        ' calculate frequency response
        Dim r As New Spectrum(500, 5000)
        For i As Integer = 0 To 499
            r(i) = pc.Response((5000 * i / 500) / x.Rate)
        Next

        ' plot frequency response
        gr.PlotDbSpectrum(3, r, "LPC Spectrum")

        ' get roots of filter polynomial
        Dim rroots() As Complex = Roots.FindRoots(pc.b, np)

        ' display frequencies
        Dim msg As String = "Formants="
        For i As Integer = 1 To np
            Dim f As Integer = Math.Atan2(rroots(i).Imag, rroots(i).Real) *
x.Rate / (2 * Math.PI)
            If (f > 0) And (f < 5000) Then
                msg = msg & " " & f
            End If
        Next
        txtStatus.Text = msg & " Hz"
    End Sub
End Class
```



Exercise

8.1 Generate a whispered vowel by passing noise through 3 resonators at say 500, 1500 and 2500Hz (with 100Hz bandwidths) and then use LPC to estimate the frequencies of the resonators from the signal. Display the signal, the FFT spectrum and the LPC spectrum as in Example 8.2. How might you find the exact locations of the peaks in the LPC spectrum?