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

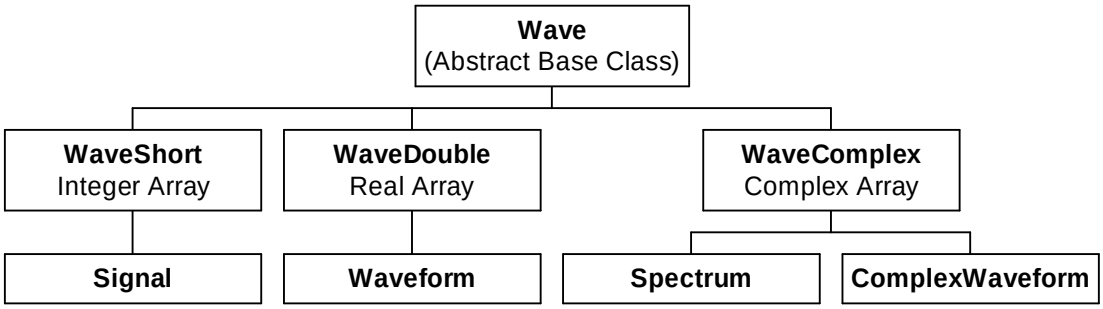
1.2 DSP Classes

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.

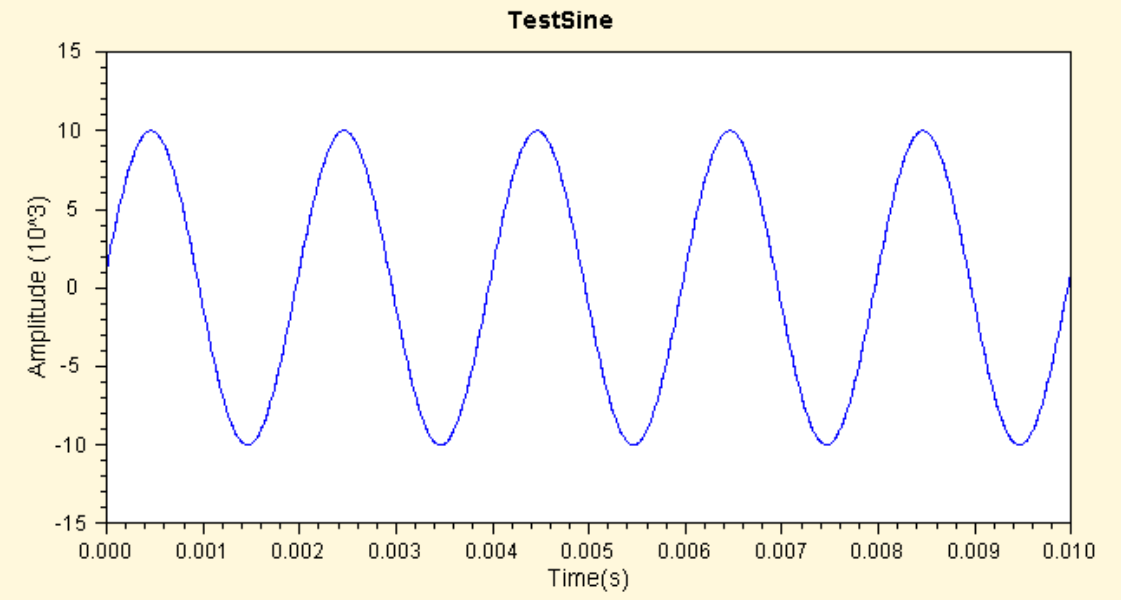
To build a VB application using BasicDSP you will need to download

- 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.