Development of a Fourier transform in C#

Christopher Oakley

11/15/2011

Keywords: C#, Fourier transform, complex number

Abstract

This paper documents the development of a Fourier transform algorithm using the C# programming language. This will begin with a brief discussion of complex numbers, followed by the development of a complex number class. These will then be used in the development of both a discrete Fourier transform function, and finally a fast Fourier transform function.

Introduction

Signal analysis is a powerful tool which can reveal a great deal of information about the operation of any given system. Analysis of time varying signals typically requires spectral analysis. Spectrum information of any signal can be obtained with a Fourier transform of a signal, which can be calculated through either the use of the discrete Fourier transform, or more commonly, the fast Fourier transform. While the discrete Fourier transform can be used, it is rather slow. As a result, the fast Fourier transform, or FFT, is often preferred. In C#, an FFT can be used based on existing third-party code libraries, or can be developed with a minimal amount of programming.

Complex Numbers

Most Fourier transforms are based on the use of complex numbers. While any Fourier transform can be written to use strictly real numbers, we can use a complex number with the imaginary portion set equal to zero. Complex numbers are typically represented in a polar form. However, for the purpose of our calculations, we will represent these in a Cartesian form. These are related as follows:

$$Ae^{j\theta} = Acos(\theta) + jAsin(\theta) = a + jb$$
$$A = \sqrt{a^2 + b^2}$$
$$\theta = tan^{-1}\left(\frac{b}{a}\right)$$

Here, A is the magnitude of a complex number, and theta is the phase. While this mathematical representation is simple to use for hand calculations, by representing these in a Cartesian form, we can minimize the number of computations required for operations such as addition, subtraction and multiplication.

Complex number multiplication can be executed using either the polar or the Cartesian form. The expressions for both of these operations are shown here:

$$Ae^{j\theta}Ce^{j\varphi} = (AC)e^{j(\theta+\varphi)} = (ac - bd) + j(ad + bc)$$

Addition and subtraction of complex numbers are simple to execute in Cartesian form. These operations consist only of adding or subtracting the real components, and the imaginary components.

C# Objects, Classes and Namespaces

C# is an object oriented language. This allows for simple data types to be combined into more complex forms, as well as allows for additional functionality to be built in to these more complex data types. The most basic unit is the “object”. An object can be simple such as an integer or string, to a

complex collection of data and functions. This allows for a wide range of functionality, as well as simplicity in development of complex applications.

A class is a collection of data and/or functions. Classes are typically developed to simplify data manipulation and group similar functionality into operational blocks. When used in a program, an object is created using a class as its framework.

Namespaces are collections of classes, functions and objects. Namespaces should each have a unique name when used in a program, and should contain similar information. Each program developed will exist in its own namespace.

Developing a Complex Number Class

A new class to hold and manipulate data related to a complex number needs to be created in order to simplify the Fourier transform calculations. To begin, a namespace and a class name must be defined. As an example, a namespace of "ece480" can be created, with a class name "complex".

```
namespace ece480
{
	class complex
	{
		….
	}
}
```

This class should also contain two data objects to hold the real and imaginary components of the complex number. These are added inside the class definition as shown below.

```
class complex
{
	public float real;
	public float imag;
}
```

Constructors

Constructors should be developed to initialize the data objects upon instantiation of the complex number object. By "overloading" the constructor, a varying number of parameters can be specified when the object is created. This allows for additional flexibility when working with these complex numbers later. Constructor definition, as well as overloading, can be seen in this example:

```
class complex
{
	public float real;
	public float imag;

	public complex()
	{
		real = new float(0);
		imag = new float(0);
	}
```

```
        public complex(float real)
        {
                this.real = new float(real);
                this.imag = new float(0);
        }

        public complex (float real, float imag)
        {
                this.real = new float(real);
                this.imag = new float(imag);
        }
}
```

Methods

A function, or "method", should be included to convert from polar form to Cartesian, and return these results. This method should be included after the constructors for the complex number class.

```
public static complex from_polar(float r, float theta)
{
        complex data = new complex(r * Math.Cos(radians), r * Math.Sin(radians));
         data;
}
```

This method is public so it may be used from any part of our application. This is also declared as a static method, so it may be called without requiring an object to be created. When the method is called, it will convert from polar form to Cartesian, and return this value as a new complex object, which can then be stored and used for other calculations.

Operator Overloading for Mathematical Operation

As was described previously, addition and subtraction of complex numbers can easily be accomplished in the Cartesian form. When a class is developed, mathematical operators need to be defined so standard calculations such as addition and subtraction can be performed as though the complex number was a simple data type, such as an integer. This can be accomplished through the practice of "operator overloading".

In this example, operators are defined for addition, subtraction and multiplication. While these definitions appear similar in structure to a typical method, their use in a program is rather simple.

```
public static complex operator +(complex a, complex b)
{
        complex data = new complex(a.real + b.real, a.imag + b.imag);
        return data;
}

public static complex operator -(complex a, complex b)
{
        complex data = new complex(a.real - b.real, a.imag - b.imag);
```

```
        return data;
}


public static complex operator *(complex a, complex b)
{
        complex data = new complex((a.real * b.real) - (a.imag * b.imag),
                                        (a.real * b.imag) + (a.imag * b.real));
        return data;
}
```

Data Accessors

C# allows for the definition of special functions, known as "data accessors", to be defined for a class. A data accessor can be used to set/retrieve specific data. In this example, these accessors can be used to return the magnitude and phase information for a complex number. These simply program development as they eliminate the need for additional typing for function calls to calculate each parameter.

```
public float Magnitude
{
        get
        {
                return Math.Sqrt(real*real + imag*imag);
        }
}

public float Phase
{
        get
        {
                if(real != 0)
                {
                        return Math.Atan(imag/real);
                }
                else if(imag > 0)
                {
                        return 90;
                }
                else
                {
                        return -90;
                }
        }
}
```

Fourier Transform

A class for Fourier transforms can be created to simplify calculations which must be carried out repeatedly. By keeping these functions inside a class, it is easy to maintain the program structure and simplify readability. First the discrete Fourier transform will be discussed, followed by the fast Fourier transform, or FFT.

Discrete Fourier Transform

The discrete Fourier transform is the most basic transform of a discrete time-domain signal. However, while simple, it is also quite slow. The discrete Fourier transform is defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2i\pi kn}{N}} \quad K = 0, 1, \dots, N-1$$

In this equation, K represents a frequency for which the Fourier transform is being calculated, N represents the number of samples in the time-domain data set, and n is a time-domain sample. As shown, each time-domain sample is multiplied by a complex number. This calculation can be achieved with a very simple function, as can be seen here:

```
public static complex[] DFT(complex[] x)
{
	int N = x.Length;
	complex[] X = new complex[N];

	for (int k = 0; k < N; k++)
	{
		X[k] = new complex(0, 0);

		for (int n = 0; n < N; n++)
		{
			complex temp = complex.from_polar(1, -2 * Math.PI * n * k / N);
			temp *= x[n];
			X[k] += temp;
		}
	}

	return X;
}
```

This function takes as an argument an array of complex time-domain samples, and returns an array of spectrum data, which the same number of elements as the time-domain data set. Each element of the spectrum data set corresponds to a particular frequency, related by the frequency resolution of the sample set.

Frequency Resolution

While the array returned from the Fourier transform function is useful, it doesn't directly contain all the information that would be required to determine the actual response of a given signal.

One factor to consider during the development of a Fourier transform function is the frequency resolution of such a transform. The frequency resolution is calculated as

$$f_r = \frac{sample\ rate}{\#\ samples}$$

Given the array output from a Fourier transform function, the frequency corresponding to each value in that array can be calculated as

$$f = n * f_r$$

DFT Shortcomings

The DFT, while simple, is very slow. Due to the number of calculations required for each sample in the time-domain data set, as the number of samples increases linearly, the number of calculations required increases quadratically. This may be suitable for small data sets, but with very large sample sets, this becomes prohibitively slow. As a result, the fast Fourier transform is the preferred method for spectral analysis in most applications.

Fast Fourier Transform

The fast Fourier transform can be easily accomplished through the use of the Cooley-Tukey algorithm. This algorithm performs a radix-2 decimation-in-time, reordering the time-domain samples, and using a divide-and-conquer approach to reduce the number of operations required to calculate the spectral information.

The Cooley-Tukey algorithm is defined as:

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{\frac{-2\pi i(2n)k}{\frac{N}{2}}} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{\frac{-2\pi i(2n+1)k}{\frac{N}{2}}} = E_k + e^{\frac{-2\pi ik}{N}} O_k$$

Using this definition, and a recursive function, the fast Fourier transform can be calculated in a short period of time.

To begin, a function must be defined, as well as references to the even and odd samples of the time-domain data. This is accomplished the same way as was used for the DFT.

```
public complex FFT(complex[] x)
{
    int N = x.Length;
    complex[] X = new complex[N];

    complex[] e, E, d, D;

}
```

Because this function will be used in a recursive fashion when rearranging the time-domain

samples, a check must be put in place to verify the size of the array being passed is greater than one. If it is equal to one, only that value should be returned.

```
if(N == 1)
{
        X[0] = x[0];
        return X;
}
```

The next step is to initialize these arrays and populate them with the correct values, then calculate the Fourier transforms of these arrays. This is accomplished with a simple loop.

```
int k = 0;

e = new complex[N/2];
d = new complex[N/2];

for(k = 0; k < N/2; i++) {
        e[k] = x[k*2];
        d[k] = x[k*2 + 1];
}

E = FFT(e);
D = FFT(d);
```

By calling the FFT function recursively, the amount of code required to develop this algorithm can be minimized. The next step is to multiply each element of the odd Fourier transformed array with its corresponding complex number. This is accomplished with a simple loop.

```
for (k = 0; k < N / 2; k++)
{
        complex temp = complex.from_polar(1, -2 * Math.PI * k / N);
        D[k] *= temp;
}
```

The final step in this process is to carry out the addition and subtraction of each of these values. In this step, for values of N between zero and N/2 -1, each even value is added to each odd value. For values between N/2 and N – 1, each odd value is subtracted from each even value. The final loop to accomplish these calculations, and return the final array containing the Fourier transform, is shown here:

```
for (k = 0; k < N / 2; k++)
{
        X[k] = E[k] + D[k];
        X[k + N / 2] = E[k] - D[k];
}

return X;
```

This resulting array, X, holds the spectrum information of the given signal. This information is subject to the same frequency resolution as in the DFT.

Advantages and Disadvantages of FFT

The FFT does have a vast improvement in speed over the DFT. However, this does come with some restrictions. The DFT requires (N^2) calculations to return the spectral information of a signal. The FFT only requires N*log(n) calculations to return the same data.

However, the FFT requires the number of samples N to be some power of 2. This can be accomplished by initializing an array with a sufficient number of data points to satisfy this requirement, copying the existing data points into their corresponding locations in this array, and setting the remaining points to zero. This only requires a few additional operations to complete.

Conclusion

Spectral analysis of a signal can reveal a significant amount of information. While there are many methods to accomplish this analysis, the fast Fourier transform using the Cooley-Tukey algorithm is the simplest to implement. Though the development of a suitable complex number class in C#, as well as a collection of functions to compute an FFT, development can be a quick procedure. This allows for the use of a simple Fourier transform in a variety of applications.

Complete Program

complex.cs

```
namespace ece480
{
   class complex
   {
      public float real = 0.0;
      public float imag = 0.0;

      //Empty constructor
      public complex()
      {
      }

      public complex( float real, float im)
      {
         this. real = real;
         this. imag = imag;
      }

      public string ToString()
      {
         string data = real.ToString() + " " + imag.ToString() + "i";
         return data;
      }
```

```
    //Convert from polar to rectangular
    public static complex from_polar(double r, double radians)
    {
       complex data = new complex(r * Math.Cos(radians), r * Math.Sin(radians));
       return data;
    }

    //Override addition operator
    public static complex operator +(complex a, complex b)
    {
       complex data = new complex(a. real + b.real , a. imag + b.imag );
       return data;
    }

    //Override subtraction operator
    public static complex operator -(complex a, complex b)
    {
       complex data = new complex(a. real - b.real , a. imag - b.imag );
       return data;
    }

    //Override multiplication operator
    public static complex operator *(complex a, complex b)
    {
       complex data = new complex((a. real * b.real ) - (a.imag * b.imag ),
                                  (a. real * b.imag + (a.imag * b.real ));
       return data;
    }

    //Return magnitude of complex number
    public float magnitude
    {
       get
       {
          return Math.Sqrt(Math.Pow(real, 2) + Math.Pow(imag, 2));
       }
    }

    public float phase
    {
       get
       {
          return Math.Atan( imag / real);
       }
    }
  }
}
```

fourier.cs

```
namespace _480_Project_Testing
{
    class fourier
    {
        public static complex[] DFT(complex[] x)
        {
            int N = x.Length;
            complex[] X = new complex[N];

            for (int k = 0; k < N; k++)
            {
                X[k] = new complex(0, 0);

                for (int n = 0; n < N; n++)
                {
                    complex temp = complex.from_polar(1, -2 * Math.PI * n * k / N);
                    temp *= x[n];
                    X[k] += temp;
                }
            }

            return X;
        }

        public static complex[] FFT(complex[] x)
        {
            int N = x.Length;
            complex[] X = new complex[N];

            complex[] d, D, e, E;

            if (N == 1)
            {
                X[0] = x[0];
                return X;
            }

            int k;

            e = new complex[N / 2];
            d = new complex[N / 2];

            for (k = 0; k < N / 2; k++)
            {
```

```
            e[k] = x[2 * k];
            d[k] = x[2 * k + 1];
        }

        D = FFT(d);
        E = FFT(e);

        for (k = 0; k < N / 2; k++)
        {
            complex temp = complex.from_polar(1, -2 * Math.PI * k / N);
            D[k] *= temp;
        }

        for (k = 0; k < N / 2; k++)
        {
            X[k] = E[k] + D[k];
            X[k + N / 2] = E[k] - D[k];
        }

        return X;
    }

  }
}
```