Sound pattern matching using Fast Fourier Transform in Windows Phone

This article shows an approach for sound pattern recognition using Fast Fourier Transforms on Windows Phone. It provides an alternative to the native APIs for short and time critical speech pattern matching.

Introduction

Windows Phone 7.5 provides an online solution (web service) for speech recognition, and Windows Phone 8 is expected to feature a Voice Recognition API for third party apps. While these offer great solutions for many use cases, they do have their limitations. The techniques described in this article are not intended to replace the above APIs, but to offer an alternative when their performance is unsatisfactory.

Conceptually the operation of the code is simple: we use the Fast Fourier Transform to convert the microphone's raw data into a function that can be compared with a sample to match. Using the Mean squared error and setting a reasonable error threshold allows us to match a small sound with a pattern. In practice of course the sound to match doesn't necessarily occur at some particular point in time, so we actually compare arrays of samples.

The code was originally used with great success in a "Shooter Assistant" app to detect voice commands (as described in the article How to access and manage the Microphone raw data in WP7), and might be similarly suitable for voice command recognition in video games (for example like "up", "down", "left", "turn" etc.). Another possible application is to modify sounds using filters, to reduce certain frequencies, or for noise reduction.

Warning: In an ideal world, I'd provide a single function taking a sample of recorded audio and a sample to match, and return a Boolean for the pattern match. Unfortunately using FFT is necessarily more complex, and without some understanding of the math principles, you will be unable to use the tools provided correctly. The article is therefore mathematics dense, and definitely not for the "faint hearted".

Fast Fourier transform

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. The **discrete Fourier transform** (DFT) transforms one function into another, which is called the frequency domain representation of the original function. The DFT requires an input function that is discrete. Such inputs are often created by sampling a continuous function, such as a person's voice. The discrete input function must also have a limited (finite) duration, such as one period of a periodic sequence or a windowed segment of a longer sequence.

This is exactly what the Windows Phone Microphone's API does for us providing a raw data buffer at some time interval.

Fast Fourier Transform compute features of spectral-domain of the speech.

The FFT algorithm only applies to signals comprising a number of elements which is equal to 2^n and returns a set of complex numbers , with the exception of the spectral components at f=0 and $f_s/2$ (the Nyquist frequency), which are both real. The number of FFT elements is equal to the size of the time sample.

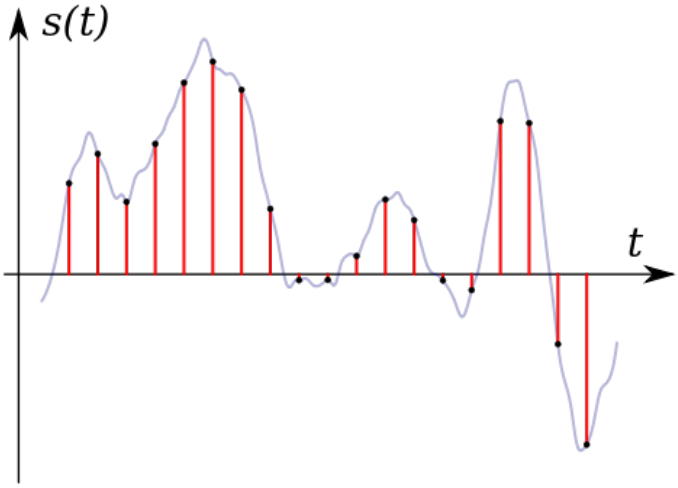
The second half of these complex numbers corresponds to negative frequencies and contains complex conjugates of the first half for the positive frequancies, and does not carry any new information.

Sampling rate

The sampling rate, or sampling frequency defines the number of samples per unit of time (usually seconds) taken from a continuous signal to make a discrete signal. The choice of sampling frequency is arbitrary. A signal contains various frequency components that can be detected.

The frequency equal to half of the sampling frequency is the highest detectable frequency. It's called the **Nyquist or folding frequency**. For a signal sampled with the sampling rate *fs*, due to the limitation imposed by the Shannon Theorem, components of the spectrum can only be extracted for frequencies between -fs/2 <= f <= fs/2, or between the negative and positive Nyquist frequency.

 (C) Copyright Nokia 2013. All rights reserved.



FFT result interpretation

Assuming that the number of samples *k*, being a power of 2, was sampled with a frequency *fs*, then

- The frequency resolution Δf = fs/k
- The maximun frequency of the spectrum is the Nyquist frequency = fs/2

It is important to notice that the Δf depends solely on the duration of observation T. The longer the duration the better (the smaller) the resolution. For the same number of samples the resolution can be improved having a slower sampling frequency *fs*.

FFT returns *k* complex numbers with *spectral components* as follows:

- The first item is for f=0. Since it has to be real, its imaginary component is 0.
- The maximun frequency of the spectrum is the Nyquist frequency = fs/2.
- The first k/2 items are for all positive frequencies in increasing order from Δf, 2Δf, 3Δf, ... , (k/2 - 1)Δf (or fs/2)
- the next k/2 - 1 elements contains the negative frequency components and can be ignored.

How to use Fast Fourier Transform

- Download DSP.cs and add it into your project. **DSP.cs** provides a namespace called `DSP` and a class `FourierTransform` containing a set of funtions to compute the FFT.
- refer to How to access and manage the Microphone raw data in WP7 for full instructions on how to manage the microphone on WP7.

```
using DSP;

private Microphone microphone = Microphone.Default;

// Event handler for getting audio data when the buffer is full
microphone.BufferReady += new EventHandler<EventArgs>(microphone_BufferReady);

microphone.BufferDuration = TimeSpan.FromMilliseconds(100);
```

The most important functions are `Compute()` and `Spectrum()`.

`Spectrum()`

`Spectrum()` takes in as input the microphone's raw data (in the Time Domain) and returns the spectrum calculated. The signature is:

```
public const int Raw = 1;
public const int Decibel = 2;

double[] Spectrum(ref double[] data, int method=Raw);
```

- `data` is the raw data buffer to analyze
- `method` take two value `Raw` and `Decibel`. `Raw` return the raw magnitude for each frequency, `Decibel` return the magnitude expressed in Decibel.

```
void microphone_BufferReady(object sender, EventArgs e)
{
            if (buffer.Length <= 0) return;

            // Retrieve audio data
            microphone.GetData(buffer);

            double[] sampleBuffer = new double[ FFT.FourierTransform.NextPowerOfTwo((uint)buffer.Length) ];

            for (int i = 0; i < 2048; i += 2)
            {
                sampleBuffer[index] = Convert.ToDouble(BitConverter.ToInt16((byte[])buffer, i)); index++;
            }

            double[] spectrum = FFT.FourierTransform.Spectrum(sampleBuffer);
}
```

http://www.developer.nokia.com/Community/Wiki/Sound_pattern_matching_using_Fast_Fourier_Transform_in_Windows_Phone (C) Copyright Nokia 2013. All rights reserved.

Compute()

Compute the FFT and is intended for more advanced users.

> **Note:** The `Compute()` function is not particularly useful for sound matching but is useful in other applications that use FFT applications - for a "real world" example of where this function is useful see Audio Noise Reduction in Windows Phone.

Here the signature:

```
void Compute(UInt32 NumSamples, Double[] pRealIn, Double[] pImagIn, Double[] pRealOut, Double[] pImagOut, Boolean bInverseTransform)
```

- `NumSamples` Number of samples (must be power two)
- `pRealIn` Real raw data samples
- `pImagIn` Imaginary (optional, may be null), to be filled when calculating inverse Fourier Transform
- `pRealOut` Real coefficient output
- `pImagOut` Imaginary coefficient output
- `bInverseTransform` bInverseTransform when true, compute Inverse FFT

A basic example of its usage is given below:

```
void microphone_BufferReady(object sender, EventArgs e)
{
            if (buffer.Length <= 0) return;

            // Retrieve audio data
            microphone.GetData(buffer);

            double rms = 0; double volume = 0;

            double cMagnitude = 0;
            double cPhase = 0;

            double[] sampleBuffer = new double[ FFT.FourierTransform.NextPowerOfTwo((uint)buffer.Length) ];
            double[] xre = new double[sampleBuffer.Length]; // Real part
            double[] xim = new double[sampleBuffer.Length]; // Imaginary part
            double[] spectrum = new double[sampleBuffer.Length / sampleSize];

            for (int i = 0; i < 2048; i += 2)
            {
                sampleBuffer[index] = Convert.ToDouble(BitConverter.ToInt16((byte[])buffer, i)); index++;
            }

            FFT.FourierTransform.Compute((uint)sampleBuffer.Length, sampleBuffer, null, xre, xim, false);

            for (int i = 0; i < 512; i++)
            {
                rms += Math.Pow((10.0 * Math.Log10((float)(Math.Sqrt((xre[i] * xre[i]) + (xim[i] * xim[i])))), 2);

                cMagnitude = (float)(Math.Sqrt((xre[i] * xre[i]) + (xim[i] * xim[i]))); // Magnitude
                cPhase     = (float)(Math.Atan(xim[i] / xre[i])); // Phase
                spectrum[i] = cMagnitude;
                cMagnitude = 0; cPhase = 0;
            }

            rms /= (double)512;
            volume = (int)Math.Floor(Math.Sqrt(rms));

}
```

Sound pattern matching

At time of writing in Windows Phone the sample rate is fixed at 16.000 samples per second (as reported by the `SampleRate` property). According to the Nyquist sample theorem, this is suitable for recording sounds up to a frequency of 8KHz. This is fine for voice, but don't expect great results with music.

Each sample is 2 bytes wide and monaural, which means that each second of recorded sound requires 32KB, and each minute requires 1.9MB.

The only option that microphone allows you to specify is the byte size of the buffer passed `GetData()`. The `BufferDuration` property is a `TimeSpan` value, and must be between 100 ms and 1000 ms (one second) in increments of 10 ms.

Assuming `TimeSpan` is 100ms, this means that each time `microphone.BufferReady` is called we receive a buffer of size 3200 bytes/1600 audio samples (remember a sample is two bytes wide).

Because of FFT works with a number of sample that is a power of two we apply the FFT to the first 1024 samples. It means that the frequency resolution Δf = 16000 / 1024 = 15.625.

The magnitude of each FFT resulting element is referred to the frequencies (in Hz) Δf, 2Δf, 3Δf, ... , (k/2 - 1)Δf (or fs/2) or :

- spectrum[0] is the magnitude for 15.625 Hz
- spectrum[1] is the magnitude for 31.25 Hz
- spectrum[2] is the magnitude for 62.5 Hz
  ...
- spectrum[512] is the magnitude for 8000 Hz

With all this information you should be able to draw a *Spectrum graph*.

Note that the FFT does not "care" where in time each frequency exists - it assumes that each frequency exists over all time. Signals whose frequency components exists at all times, are said to be stationary. While this may be great for the mathematical world, the real world is different. Common,

http://www.developer.nokia.com/Community/Wiki/Sound_pattern_matching_using_Fast_Fourier_Transform_in_Windows_Phone (C) Copyright Nokia 2013. All rights reserved.

everyday signals, such as the signals from speech, are rarely stationary - they will almost always have frequency components that exists for only over a short period of time. To solve this problem we break the signal into very short intervals of time and take the Fourier Transform for each of these intervals. This gives the frequencies that a signal contains, and the location in time of those frequencies.

This is why we used a very small `TimeSample` of 100ms! If we want to match the sound for the speech "up" (i.e for Video Games) each 100ms spectrum[i] will provide information of all frequencies involved and their magnitude.

The simplest form of sound pattern matching we might use would be to compare the two spectrum magnitudes, calculating the MSE, and if lower than a threshold than assume the sounds are equal. Unfortunately this simple approach may not be particularly effective - saying the same phrase using more or less volume, when angry or happy, or in different different environmental conditions can affect the frequency components enough to fail the comparison.

This article instead uses a more advanced approach called MFCC (Mel-frequency cepstral coefficients), a method that extracts and compares just the sound's "key" features. This approach is more efficient than the simple spectrum comparison - and is less influenced by environmental noise, or the volume or tone of the phrase being compared.

Mel-Frequency Cepstrum & Cepstral Coefficients

In sound processing, the mel-frequency cepstrum (MFC) is a representation of the short-term power spectrum of a sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency.

Mel-frequency cepstral coefficients (MFCCs) are coefficients that collectively make up an MFC. They are derived from a type of cepstral representation of the audio clip (a nonlinear "spectrum-of-a-spectrum"). The difference between the cepstrum and the mel-frequency cepstrum is that in the MFC, the frequency bands are equally spaced on the mel scale, which approximates the human auditory system's response more closely than the linearly-spaced frequency bands used in the normal cepstrum. This frequency warping can allow for better representation of sound, for example, in audio compression.

The MFCCs are the amplitudes of the resulting spectrum.

MFCC consists of six computational steps. Each step has its function and mathematical approaches as discussed briefly in the following:

Step 1: Pre–emphasis
This step processes the passing of signal through a filter which emphasizes higher frequencies. This process will increase the energy of signal at higher frequency.

```
Y [n] =  X [n] -0 . 95 x[n - 1]
```

Lets consider a = 0.95, which make 95% of any one sample is presumed to originate from previous sample.

Step 2: Framing
The process of segmenting the speech samples obtained from analog to digital conversion (ADC) into a small frame with the length within the range of 20 to 40 msec. The voice signal is divided into frames of N samples. Adjacent frames are being eparated by M (M<N).

Typical values used are M = 100 and N= 256.

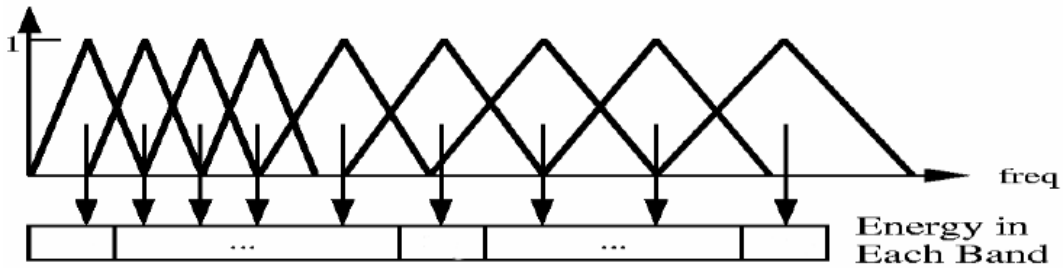
Step 3: Hamming windowing
Hamming window is used as window shape by considering the next block in feature extraction processing chain and integrates all the closest frequency lines.
The Hamming window equation is given as:

```
Y (n) = X(n) x W (n) for 0 ≤ n ≤ N-1
```

where:

- N = number of samples in each frame
- Y[n] = Output signal
- X (n) = input signal
- W (n) = 0.54 - 0.46 cos(2πn / N -1), the Hamming window

Step 4: Take the Fourier transform of the signal
Step 5: Mel Filter Bank Processing
Map the powers of the spectrum obtained above onto the mel scale, using triangular overlapping windows.
The frequencies range in FFT spectrum is very wide and voice signal does not follow a linear scale. The bank of filters according to Mel scale is then performed.



We use a set of triangular filters that are used to compute a weighted sum of filter spectral components so that the output of process approximates to a Mel scale. Each filter's magnitude frequency response is triangular in shape and equal to unity at the centre frequency and decrease linearly to zero at

http://www.developer.nokia.com/Community/Wiki/Sound_pattern_matching_using_Fast_Fourier_Transform_in_Windows_Phone (C) Copyright Nokia 2013. All rights reserved.

centre frequency of two adjacent filters. Then, each filter output is the sum of its filtered spectral components. After that the following equation is used to compute the Mel for given frequency f in HZ:

```
F(Mel) = [ 2095 * Log10(1 + f/700)
```

Then take the logs of the powers at each of the mel frequencies. A typical dimension of MFCC array for a 100ms time sampling is 12 items

Step 6: Discrete Cosine Transform
Take the discrete cosine transform of the list of mel log powers, as if it were a signal.

Comparing

Let's assume the spectrum information for the preloaded pattern array for voice command "up" is :

```
[2083] [318,9636] [105,3883] [54,90217] [50,55225] [35,10337] [11,41992] [36,78038] [66,0419] [74,63419] [74,20988] [46,58485]
```

and the current microphone's data stored in spectrum array are:

```
[1835] [231,1179] [38,28688] [78,53581] [99,16002] [29,44954] [30,88765] [49,32907] [42,68513] [22,55693] [15,65068] [28,82779]
```

These samples can be compared using mean squared error method to estabilish if a sound matches or not. Here the formula:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left(x_i - a\right)^2$$

Let's assume `command` is a pre-calculated and pre-loaded array data of a speech command we want to compare into our application (`IsolatedStorage` might be used to store this information).

```
double mse = 0;

for( int i = 0 ; i < spectrum.Length ; i++ ) {
     mse += Math.Pow( (spectrum[i] - command[i]), 2);
}
mse /= spectrum.Length;
```

How to use MFCC

```
double[] melToCompare;

void microphone_BufferReady(object sender, EventArgs e)
{
            if (buffer.Length <= 0) return;

            // Retrieve audio data
            microphone.GetData(buffer);

            double[] sampleBuffer = new double[ DSP.FourierTransform.NextPowerOfTwo((uint)buffer.Length) ];

            for (int i = 0; i < 2048; i += 2)
            {
                sampleBuffer[index] = Convert.ToDouble(BitConverter.ToInt16((byte[])buffer, i)); index++;
            }

            double[] spectrum = DSP.FourierTransform.Spectrum(sampleBuffer);
            double[] mel = DSP.MFCC.compute(ref spectrum);

            double mse = DSP.Utilities.MSE(ref melToCompare, ref mel, melToCompare.Length);

            if( mse < threshold ) {
                // There is a match !! Do something...
            }
}
```

If MSE' is lower than the threshold it is reasonable to assert that the sound matches the voice command.

Results

This code was used in my Shooter Assistant project to catch the voice command that starts timing shooter performance. On the Nokia Lumia 800 there were no measurable delays due to the calculation - reaction time calculations were very accurate to within milliseconds.

This is particularly impressive given the large floating point operations involved in calculating the FFT.

Summary

This article has shown an alternative way to perform local sound pattern matching that is suitable for processing simple time-critical commands.

http://www.developer.nokia.com/Community/Wiki/Sound_pattern_matching_using_Fast_Fourier_Transform_in_Windows_Phone (C) Copyright Nokia 2013. All rights reserved.

http://www.developer.nokia.com/Community/Wiki/Sound_pattern_matching_using_Fast_Fourier_Transform_in_Windows_Phone (C) Copyright Nokia 2013. All rights reserved.

http://www.developer.nokia.com/Community/Wiki/Sound_pattern_matching_using_Fast_Fourier_Transform_in_Windows_Phone (C) Copyright Nokia 2013. All rights reserved.