Hector Perez

Due Wednesday February 15th

CSC 475 Music Information Retrieval

Assignment 2

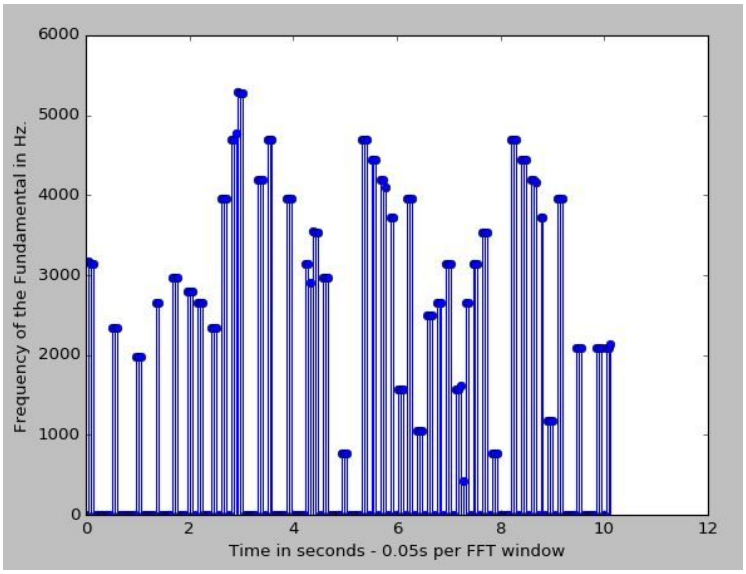
Question 1

Question 1 A . Monophonic pitch estimation through fundamental contour

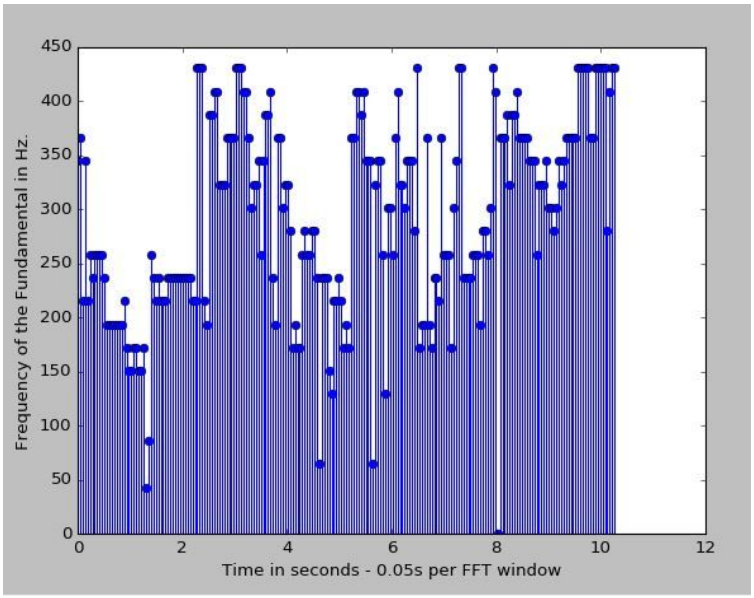
CODE : ***Appendix A (Python Code).*** fundamentalContour.py. And getPeak.py

Extract fundamental frequency contour from: (windowsize = 2048. Fs = 44.1kHz)

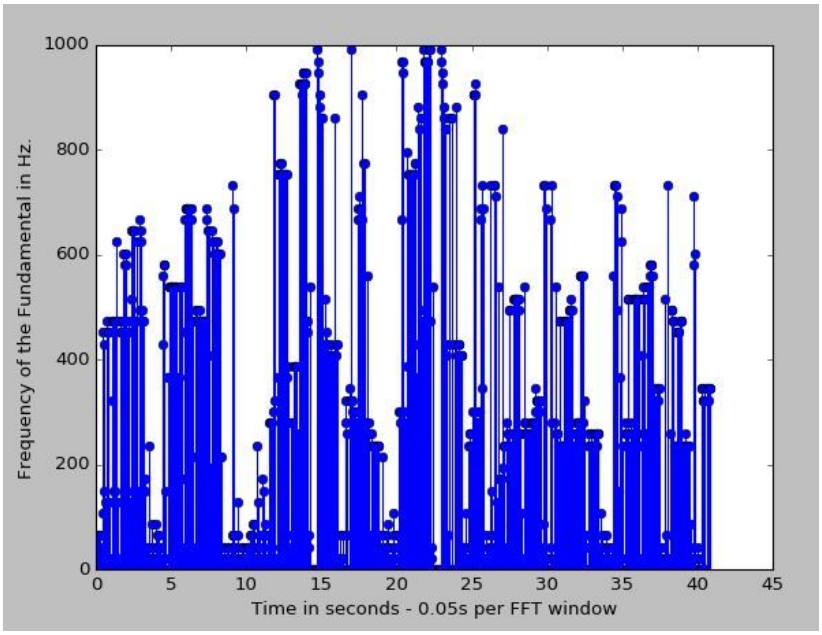
- Melody: (Mario Bros Main Theme).



- Melody Hummed:



- Qbhexamples: (It's getting better all the time…).



Question 1 B: Monophonic pitch detection through autocorrelation.
Note: I wrote an autocorrelation function - but it was brute force so it was computationally expensive. (Included anyways, in Appendix B).

Code: ***Appendix B. (Python Code).*** acSumSignalWindowedFFT.py and others.
Note: I did two versions - one using the autocorrelation function given by numpy, and the other one using the FFT computation of the autocorrelation information (e.g. multiply by the conjugate spectrum in the frequency domain equals convolution in the time domain).

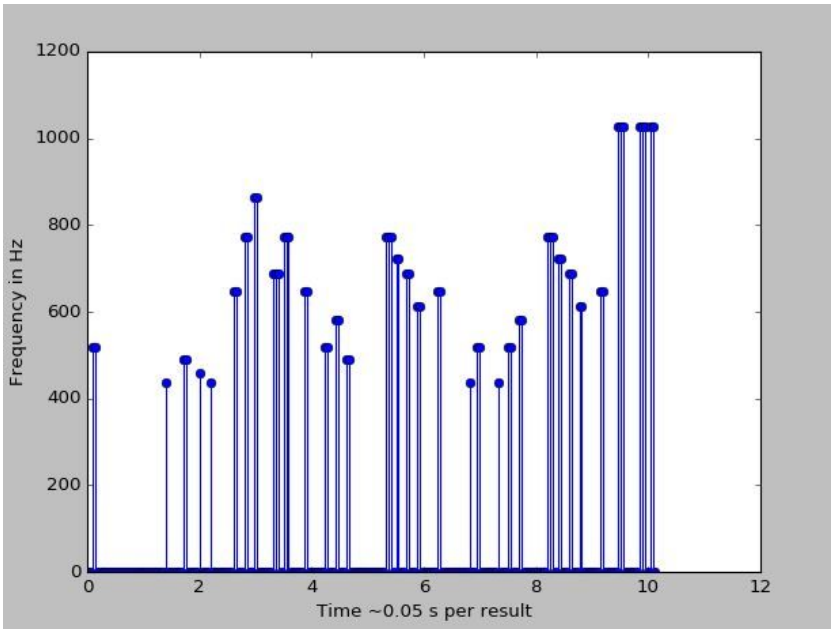
From the autocorrelation function we can estimate the frequency - since there will be a global maximum which is when the delay between the signal and the shifted copy of itself is zero (zero -lag). Then the next 'peak' must be found and based on the period (we know how long it takes to 'sample' again), we can calculate the frequency.

ISSUE: Using numpy.argmax gives back the index of the 'first' max found only -> this means that we must start looking for the next peak a little later in the data, because otherwise it'll return the same index and the frequency appears really high when it's not..

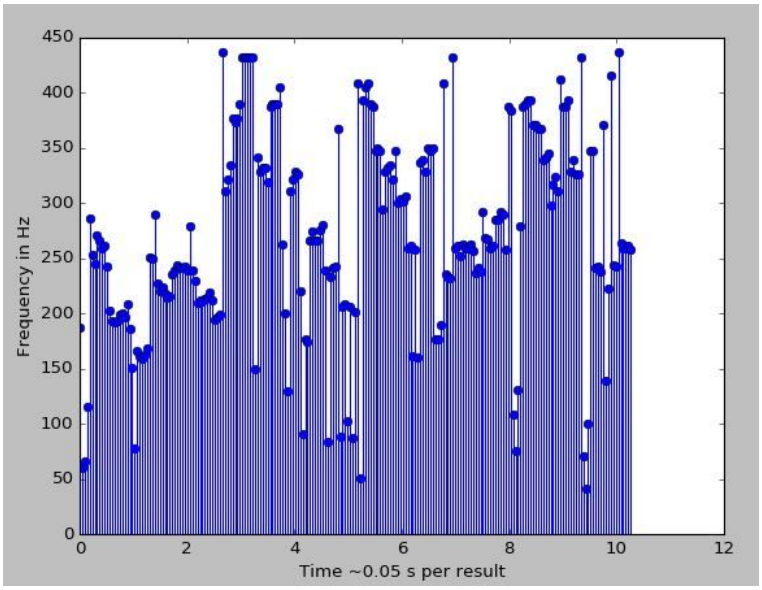
Pitch detection on the same signals through autocorrelation: Fs = 44100 Hz.

Q1B. a - autocorrelation through correlate function by numpy. (For windows of 2048 samples).

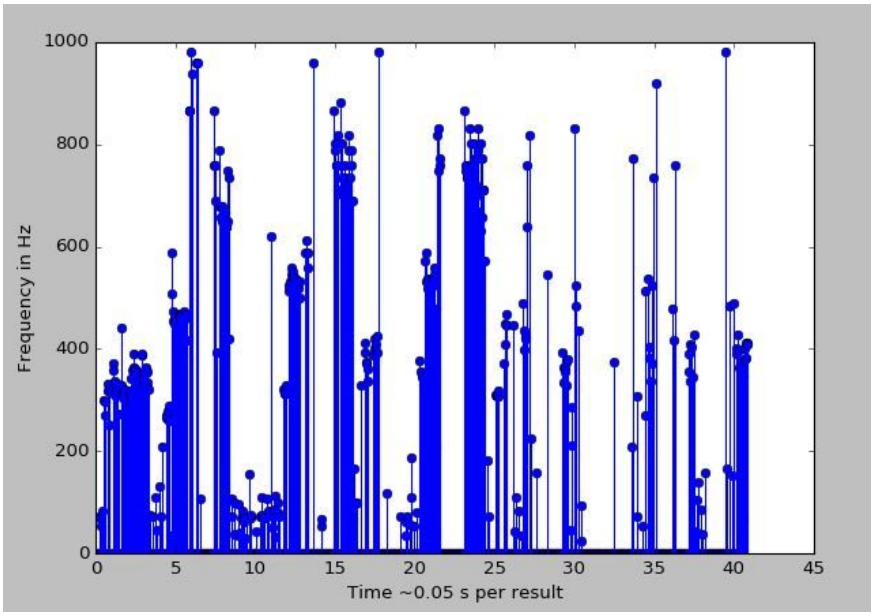
- Melody:



- Melody sung:

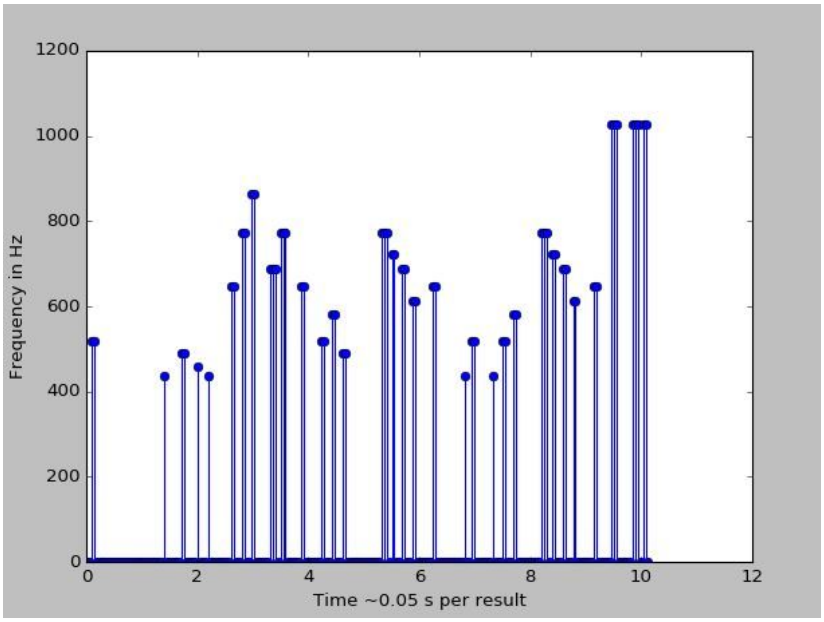


- Qbhexamples:

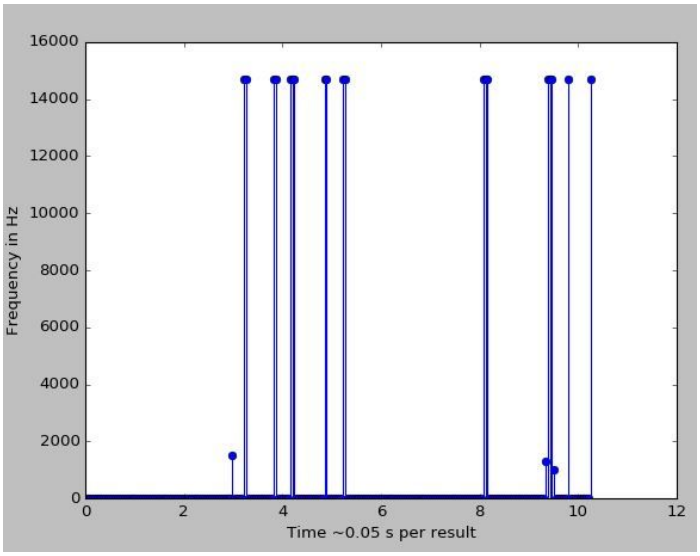


Q1B. b. - autocorrelation through FFT calculation. (For windows of 2048 samples)

- Melody:

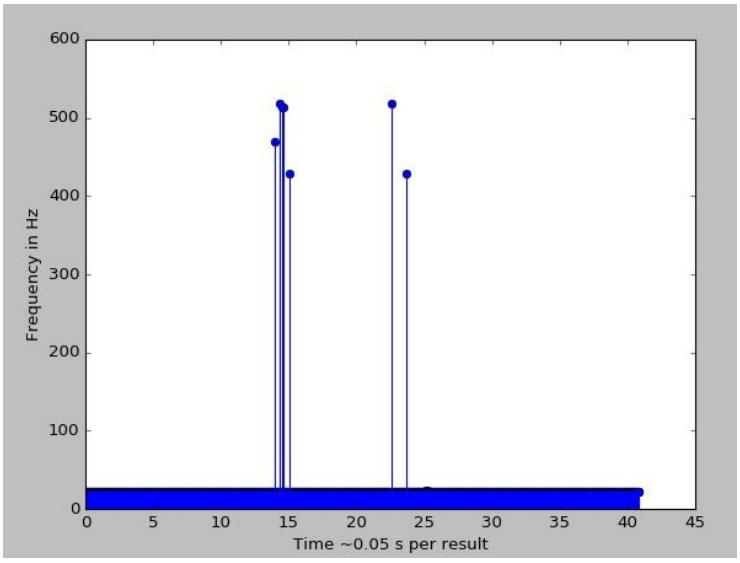


- Melody sung:



Note: Recording very noisy - FFT might've picked it as high frequencies. Recommendable to window.

- Qbhexamples:



Even after multiple different parameter attempts, the plot for the FFT-calculated-autocorrelation pitch detection is not so great for this specific example of audio.

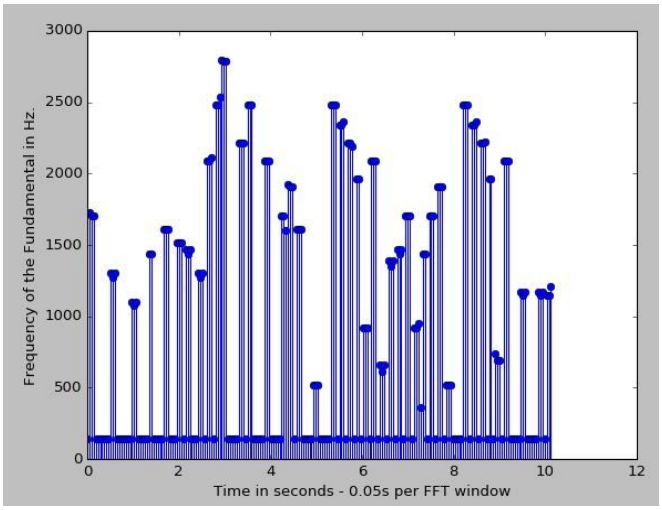
Note: Similar in shape but the autocorrelation with np.correlate worked much better.
(That is on previous part - Q1B- a.).

Question 1 C:

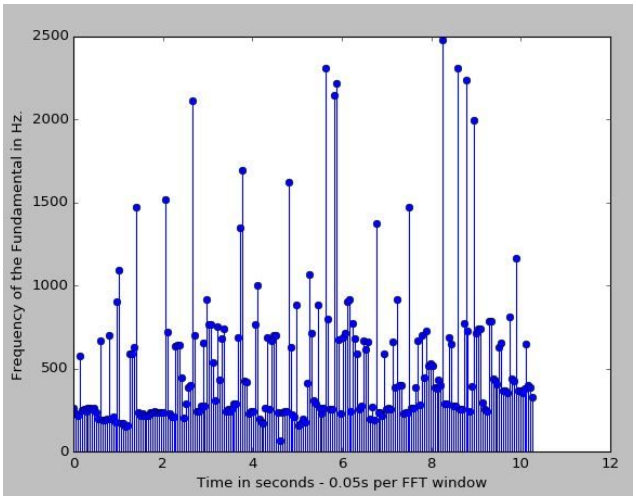
Code is in **Appendix C -** averageF0.py Compare the contours: *Is one consistently better than the other ?* (FFT or Autocorrelation).
Autocorrelation worked better, as is seen in the plots. It was more difficult to get a consistent plot for FFT without a threshold for the frequencies calculated - there were peaks that were very high and took away from the detail of the plot.

Types of errors observed:
Plot the 'sum of both F0 estimates' (FFT and Autocorr). I didn't understand why the sum, so I averaged the two values.

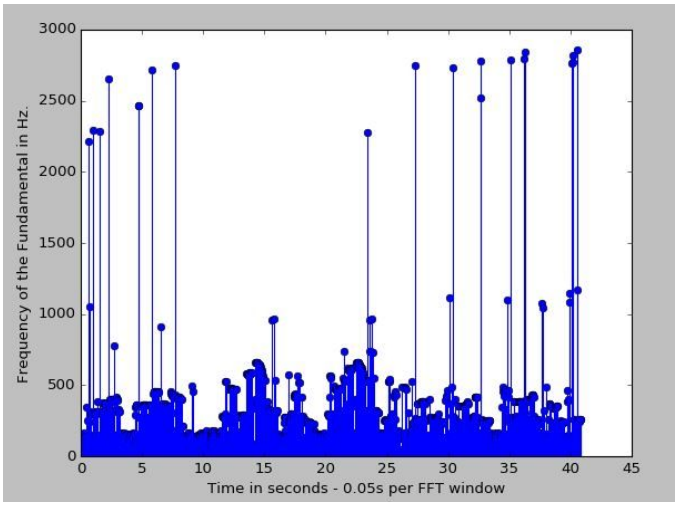
- Melody: (average of FFT peak detection and Autocorrelation pitch detection).



- Melody sung:



- Qbh Examples:



Question 2 Centroid Sonication

Question 2 A - Sonication (For Window Size 2048)

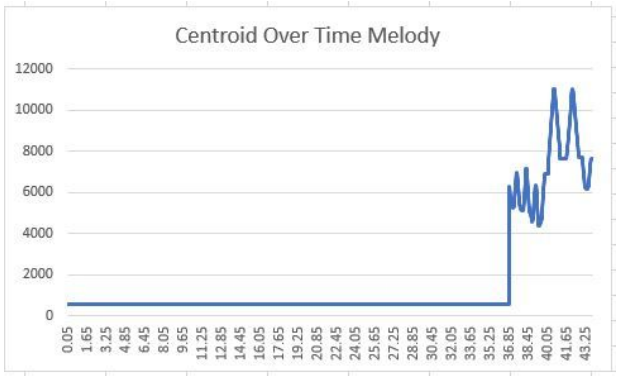
$$\text{Centroid} = \frac{\sum_{n=0}^{N-1} f(n)x(n)}{\sum_{n=0}^{N-1} x(n)}$$

The centroid is the 'center of gravity' of the audio signal. There would be equal energy on either side of the centroid in the spectrum (the centroid is a frequency). f(n) is the frequency of the bin. x(n) is the magnitude for that frequency bin. It's a good predictor of 'brightness of a sound/timbre'.

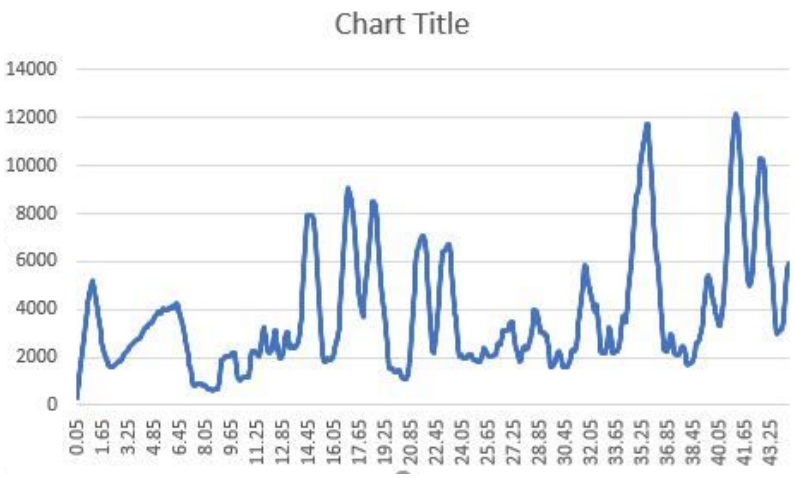
Marsyas code: **Appendix B -** baseFile.mrs

Plot the centroid over time for the three audio files. (Used Excel for this).

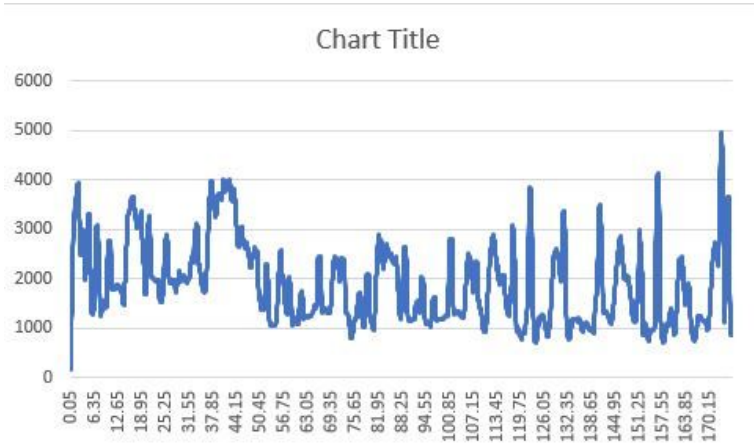
Melody Centroid (Y axis is Frequency in Hz, X axis is time in seconds). (t = 2048/44100 sec)



Melody Hummed Centroid



QbhExamples Centroid:



Audio files created: Melody.wav
(Zip included) MelodyHummed.wav
QbhExamples.wav

Insight into what is happening: The centroid would tell us what part of the spectrum is predominant, because if it is higher - the energy of more bins had to be added in order to balance its energy with the bins after it. If it is lower, we know for example that low frequencies are predominant in the audio, because the sum of not many bins' energies matches the sum of all the other ones after the centroid.

Question 2 B. Experiment with music genres (same .mrs file used)

Comparisons:

Classical Centroid vs Classical Centroid
The centroids are not too high, and they don't vary very often.

Classical Centroid vs Metal Centroid
The centroid for the metal audio was quite high compared to the classical one.

Metal Centroid vs Metal Centroid
These two vary faster and are high.

Commentary:
It is easy to distinguish, we can almost set a threshold and then if the centroid is above or below, it will be classified as metal or as classical music. It seems the centroid is related to the timbre of the sound, so classical music instruments timbre is mostly low frequencies, and lacks high frequencies that metal music from electric guitars and drum cymbals, etc.

Appendix

Appendix A (Python Code). fundamentalContour.py. And getPeak.py

```
Must import mir,getPeak, numpy as np, wave,os,struct,matplotlib.pyplot as plt
	class fundamentalContour():
		def main():
			signal1 = mir.Signal();
			signal1.wav_read("../q1audioSignals/melodyHectorHigh.wav");
			sizeOfFFT = 2048;
			windowSize = 2048;
			non_Aliasing_Points = sizeOfFFT/2;
			sizeOfBin = 44100.0/sizeOfFFT;
			numberOfWindows = int(np.floor(len(signal1.data)/windowSize));
			print numberOfWindows;
			t = windowSize/44100.0;
			fundamentals = np.zeros(numberOfWindows,);
			for i in range (0,numberOfWindows):
				hanning = np.hanning(2048);
				inputSamples = signal1.data[int(i*windowSize):int((i+1)*windowSize)];
				inputSamples = np.multiply(hanning,inputSamples);
				peakBin = getPeak.getPeakFFT(inputSamples, windowSize);
				peakFrequency = peakBin*sizeOfBin;
				fundamentals[i] = peakFrequency;
				print '-----------------------------------------------------';
			xAxis = np.linspace(0,numberOfWindows*t,numberOfWindows);
			plt.stem(xAxis,fundamentals);
			plt.xlabel('Time in seconds - 0.05s per FFT window');
			plt.ylabel('Frequency of the Fundamental in Hz.');
			plt.show();
			return 0;
		if __name__ == "__main__": main()


	getPeak.py

Must import mir,numpy as np,wave,os,struct,matplotlib.pyplot as plt

def getPeakFFT(inputSamples, windowSize):
	fft_data = np.fft.fft(inputSamples);
	magnitude_spectrum = np.absolute(fft_data);
	magnitude_spectrum = magnitude_spectrum[:len(inputSamples)/2];
	peakFrequencyBin = np.argmax(magnitude_spectrum[:len(magnitude_spectrum)/4]);
	#i had to limit to see only the first few bins, because I was getting
	#high frequency peaks that distorted the plots.
	return peakFrequencyBin;
```

Appendix A (Python Code). autoCorrWindow.py, acSumSignalWindowedLibrary.py, autoCorrWindowFFT.py, acFFTWindowed.py. (First pair does not use FFT, second pair does).

```
    autoCorrWindow.py

Must import numpy as np,wave,os,struct,matplotlib.pyplot as plt

#my implementation of autocorrelation calculation - acc wikipedia
#frequency calculated based on zero lag and next peak.
def computeAutoCorrelationFrequency(fs,currWindow,windowSize):
        numberOfIterations = windowSize;
        sizeOfWindowCorrelation = 2*windowSize-1;
        autocorrelation = np.zeros(sizeOfWindowCorrelation,);
        autocorrelationIndex = 0;
        aux = 0;
        indexOfLastElement = len(currWindow)-1;
        for i in range (0,numberOfIterations):
                for k in range(0,numberOfIterations):
                        if aux > windowSize-1: #like 'mod'
                                aux = 0;
                        multiplier = currWindow[indexOfLastElement]; #take last element
                        result = multiplier * currWindow[k];
                        autocorrelation[autocorrelationIndex+aux] += result;
                        aux += 1;
                indexOfLastElement -= 1;
                autocorrelationIndex +=1;
        zero_lag = np.argmax(autocorrelation);
        new_autocorr = autocorrelation[zero_lag+1:];
        next_peak = np.argmax(new_autocorr)+zero_lag+1;
        sampling_T = float(1.0/fs);
        period_T_in_Signal = float(next_peak - zero_lag + 1);
        period_inSeconds = period_T_in_Signal*sampling_T;
        fs_in_signal = float(1.0/period_inSeconds);
        return fs_in_signal;

def computeAutoCorrelationFrequencyLibrary(fs,currWindow,windowSize):
        autocorrelation = np.correlate(currWindow,currWindow,mode = 'full');
        zero_lag = np.argmax(autocorrelation);
        new_autocorr = autocorrelation[zero_lag+1:];
        next_peak = np.argmax(new_autocorr)+zero_lag+1;
        sampling_T = float(1.0/fs);
        period_T_in_Signal = float(next_peak - zero_lag + 1);
        period_inSeconds = period_T_in_Signal*sampling_T;
        fs_in_signal = float(1.0/period_inSeconds);
        return fs_in_signal;
```

acSumSignalWindowedLibrary.py

```
import mir,autocorrWindow,numpy as np,wave,os,struct,matplotlib.pyplot as plt

class acSumSignalWindowedLibrary():
	def main():
		signal1 = mir.Signal();
		signal1.wav_read("../q1audioSignals/outMelody.wav");
		fs = 44100;
		windowSize = 2048; #at a 44100 Hz Fs.
		numberOfWindows = int(np.floor(len(signal1.data)/windowSize));
		t = windowSize/44100.0;
		frequencies = np.zeros(numberOfWindows,);
		for  i in range (0,numberOfWindows): #-1?
			currWindow = signal1.data[i*windowSize:(i+1)*windowSize];
			frequencies[i] =
autocorrWindow.computeAutoCorrelationFrequencyLibrary(fs,currWindow,windowSize);
			if frequencies[i] >= 44100.0/2 : frequencies[i] = 0;
		xAxis = np.linspace(0,numberOfWindows*t,numberOfWindows);
		plt.stem(xAxis,frequencies);
		plt.xlabel('Time ~0.05 s per result');
		plt.ylabel('Frequency in Hz');
		plt.show();
		return 0;
	if __name__ == "__main__": main()
```

---

autoCorrWindowFFT.py

```
Must import numpy as np,wave,os,struct,matplotlib.pyplot as plt

def computeAutoCorrelationFrequencyFFT(fs,currWindow,windowSize):
	re_in = currWindow;
	im_in = np.zeros(len(re_in),);

	spectrum = np.fft.rfft(re_in);
	mult_freq_domain = spectrum * np.conj(spectrum);
	autocorrelation = np.fft.irfft(mult_freq_domain);
	maxValue = np.max(autocorrelation);
	zero_lag = np.argmax(autocorrelation);
	next_peak = np.argmax(autocorrelation[zero_lag+1:])+zero_lag+1;
	sampling_T = float(1.0/fs);
	period_T_in_Signal = float(next_peak - zero_lag + 1);
	period_inSeconds = period_T_in_Signal*sampling_T;
	fs_in_signal = float(1.0/period_inSeconds);
	return fs_in_signal;
```

```
    acFFTWindowed.py.
Must import mir,autocorrWindowFFT,numpy as np,wave,os,struct,matplotlib.pyplot as plt
class acFFTWindowed():
       def main():
              signal1 = mir.Signal();
              signal1.wav_read("../q1audioSignals/outMelody.wav");
              fs = 44100;
              windowSize = 2048;
              numberOfWindows = int(np.floor(len(signal1.data)/windowSize));
              t = windowSize/44100.0;
              frequencies = np.zeros(numberOfWindows,);
              for  i in range (0,numberOfWindows): #-1?
                     currWindow = signal1.data[i*windowSize:(i+1)*windowSize];
                     frequencies[i] =
autocorrWindowFFT.computeAutoCorrelationFrequencyFFT(fs,currWindow,windowSize);
                     if frequencies[i] >= 44100.0/2 : frequencies[i] = 0;
              xAxis = np.linspace(0,numberOfWindows*t,numberOfWindows);
              plt.stem(xAxis,frequencies);
              plt.xlabel('Time ~0.05 s per result');
              plt.ylabel('Frequency in Hz');
              plt.show();
              return 0;
       if __name__ == "__main__": main()
```

Appendix B - baseFile.mrs

```
Series {
  -> input: SoundFileSource {filename="wav_files/qbhexamplesB.wav" inSamples = 1024}
  -> MixToMono
  -> Fanout {
      -> Series {
              -> shift: ShiftInput {winSize = (2 * inSamples)}
              -> Windowing {type = "Hanning"}
        -> Spectrum
        -> PowerSpectrum {spectrumType = "magnitude"}
              -> Centroid
              -> Memory {memSize = 20}
              -> Mean
              -> CsvFile { filename = "dataOut.csv"}
              -> freq: FlowToControl
      }
      -> Series {
        -> Fanout {
                     -> Series {
                       -> SineSource { frequency = ((input/israte * 0.5) * freq/value)  }
                       -> Gain { gain = 0.1 }
                     }

                     -> Gain { gain = 0.2 }
        }
        -> AudioSink
              -> SoundFileSink { filename = "qbhexamples_centroid.wav" }
      }
  }
  + done = (input/hasData == false)
}
```

Appendix C: Question 1 C - Average of the two F0 calculation methods.

```
Must import mir,getPeak,autocorrWindow,numpy as np,wave,os,struct,matplotlib.pyplot as plt
class averageF0():
	def main():
		signal1 = mir.Signal();
		signal1.wav_read("../q1audioSignals/qbhexamplesB.wav");
		sizeOfFFT = 2048;
		windowSize = 2048;
		non_Aliasing_Points = sizeOfFFT/2;
		sizeOfBin = 44100.0/sizeOfFFT;
		numberOfWindows = int(np.floor(len(signal1.data)/windowSize));
		fs = 44100.0;
		t = windowSize/fs;
		fundamentals = np.zeros(numberOfWindows,);
		for i in range (0,numberOfWindows):
			hanning = np.hanning(2048);
			inputSamples = signal1.data[int(i*windowSize):int((i+1)*windowSize)];
			inputSamples = np.multiply(hanning,inputSamples);
			peakBin = getPeak.getPeakFFT(inputSamples, windowSize);
			peakFrequencyFFT = peakBin*sizeOfBin;
			peakFrequencyAutoCorr =
autocorrWindow.computeAutoCorrelationFrequencyLibrary(fs,inputSamples,windowSize);
			fundamentals[i] = (peakFrequencyFFT+peakFrequencyAutoCorr)/2.0;
			if fundamentals[i] >= (fs/2): fundamentals[i] = 0;
		xAxis = np.linspace(0,numberOfWindows*t,numberOfWindows);
		plt.stem(xAxis,fundamentals);
		plt.xlabel('Time in seconds - 0.05s per FFT window');
		plt.ylabel('Frequency of the Fundamental in Hz.');
		plt.show();
		return 0;
	if __name__ == "__main__": main()
```