YIN algorithm as a music transcription tool

Abstract

This report presents a method to retrieve pitch, onset and duration of musical notes present in monophonic audio files. The report describes the implementation of the YIN algorithm for pitch estimation and its performance under different case scenarios. The algorithm is tested for string, wind and percussive instruments in order to determine under what circumstances YIN provides a good pitch estimation. The YIN algorithm performs well with non percussive instruments and low to high playing speed.

Introduction

One of the most important analysis performed on a signal is the estimation of its pitch or fundamental frequency. Considering Fourier analysis and in particular the Fourier Series decomposition of a signal, the fundamental frequency is the first frequency of the series (also called first harmonic) and it's usually the frequency that contains more power.[1] The estimation of the pitch is key to systems that perform speech recognition, analysis of musical pieces, musical transcription or musical information retrieval among others.

In order to approximate the fundamental frequency of an audio signal different methods have been developed. Signal analysis can be performed in the time domain or in the frequency domain. Examples from the first group include the YIN algorithm and other autocorrelation methods such as (look for examples). The most important method in the second group is the Fast Fourier Transform which decomposes the signal into an orthogonal family of functions that have sines and cosines as bases of their vectorial space.

As said before, the YIN algorithm makes use of autocorrelation to find parts of the signal that are similar. Therefore, its precision and in general its use is highly influenced by the periodicity of the signal. In an ideal case, the algorithm will give a precision of 100% when the signal is 100% periodic. Unfortunately, common audio signals used and perceived as pleasant by humans tend to have a considerable amount of aperiodicity with relatively high levels of noise and a spectrum that depends on time. Examples of this are speech and musical instruments where the spectrum tends to be very complex. Therefore, one can not expect a precise pitch estimation when using an autocorrelation method.

The achievement of the YIN algorithm is the minimization of the aperiodicity influence though a sequence of steps that when correctly performed provide a highly precise result.

The core set of audio signals used to test the YIN implementation is formed by five different instruments playing a chromatic scale that contains most of the midi notes. In addition, the same scale is played at different speed in

[1] William E. Boyce and Richard C. DiPrima (2005). *Elementary Differential Equations and Boundary Value Problems* (8th ed.).

order to determine the limits of use of the algorithm.

YIN estimates the pitch of an audio signal with an error of 1% for a wide variety of signals and situations[2]. The limitations are drawn by the fact that it is an algorithm that is based in autocorrelation and therefore it needs the signal to be as periodic as possible. Despite this fact, it can be used to estimate the pitch of musical instruments and voice giving a precise result. At the same time, YIN can only be used to estimate pitches of monophonic audio signals.

This report describes the implementation of the algorithm from the original paper and its use on different case scenarios in order to determine when it can be used.

The background section in the report describes historical facts related with pitch and contains a description of the YIN algorithm. The following section contains an analysis of the performance of YIN under different scenarios. The system design section illustrates how the implementation of the algorithm was performed. Finally, the conclusion discuses the achievement of the aims of the project and summarises this report.

Background

The first association between pitch and frequency was made in 1563 when the mathematician Giovanni Battista Benedetti proposed a theory that stated that the perception of pitch was determined by the frequency of vibration of a source of sound.[3] Consider a sine tone with a frequency of 440Hz and another one with a frequency of 220Hz. The first one will be perceived as a higher sound than the second one.

The Fourier theory is a mathematical framework that states that any periodic signal can be expressed as an infinite sum of sinusoids. Note that in general they will have different frequencies, amplitudes and phases. Therefore, sine tones are the fundamental forming blocks of signals and they represent the simplest of the signals. All the other sounds will be called complex tones (not to be confused with real/complex signals) and can be represented as an addition of simple tones via Fourier Theory.

In the case of a sine tone, its Fourier transform is a Dirac delta with value infinite in the fundamental frequency (pitch). However, in the case of complex tones, how can one determine its pitch if it is formed from infinite sinusoids with different frequencies? Which of those frequencies should be considered as responsible for the pitch perception?

In general terms this question has not been answered yet because pitch is a perceptual property of sound. It is a subjective product of the brain's hearing system processing which is not fully understood at the present time. However, for most of the times[4] the pitch perceived can be quantified as the frequency from the first sinusoid

[2] Alain de Cheveigne, Hideki Kawahara (June 2001). "YIN, a fundamental frequency estimator for speech and music". http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf

[3] Psychology of Music, Siu-Lan Tan, Peter Pfordresher and Rom Harré. Psychology Press, 2010.

[4] Schwartz, D.A.; Purves, D. (May 2004). "Pitch is determined by naturally occurring periodic sounds". *Hearing Research* **194**: 31–46. doi:10.1016/j.heares.2004.01.019. Retrieved 4 September 2012.

(also called first harmonic) of the Fourier decomposition of the signal, in general it is also the frequency that has more power in the series. A complementary view of this fact is that the pitch perceived is the rate at which the signal repeats itself. This two different approaches of pitch quantification represent the two classes where most of the pitch estimation methods fall: Time-domain algorithms analyse the signal respect to time, some examples are: zero-crossing and autocorrelation methods. Frequency-domain methods imply analysing the signal in the Fourier transformed space.

From above, there are three obvious methods to determine the pitch of a signal: Zero-crossing, detect the cycles of the signal via autocorrelation and the Fourier transform or Fast Fourier Transform for digital signals.

When the signals are simple, the pitch will be determined by the rate at which it crosses the zero axe, this method is called zero-crossing. However, for complex tones zero-crossing introduces errors in the pitch estimation and it should be avoided. In this later case the Fourier transform could be applied and the first harmonic taken as pitch. However, in the case of digital signals, the maximum precision of the Fast Fourier Transform depends on the signal length and precise results imply a high computational expense. Autocorrelation methods that look for repetitions in the signal are a relatively fast and precise way to determine the pitch of complex sounds. At the same time, they need the signal to be highly periodic, which unfortunately is not the case most of the time.

As seen, the problem of pitch estimation is difficult to solve and all the methods have drawbacks. Despite this situation, the YIN algorithm provides a robust method for pitch estimation. It is based in autocorrelation and five additional steps that minimise the effect of aperiodicity in the signal.

Analysis

The YIN algorithm is a method to estimate the fundamental frequency or pitch of an audio signal. It is a time domain method based on autocorrelation and five additional steps that minimise the impact of the aperiodicity present in the signal.

The autocorrelation of a signal is mathematically defined by the Equation 1:

$$r_t(\tau) = \sum_{j=t+1}^{t+W} x_j x_{j+\tau}$$

Equation 1

Where t is an arbitrary time and W is the integration window size.

The Figure 1 shows the autocorrelation values corresponding to an audio signal containing a piano note at 440 Hz.

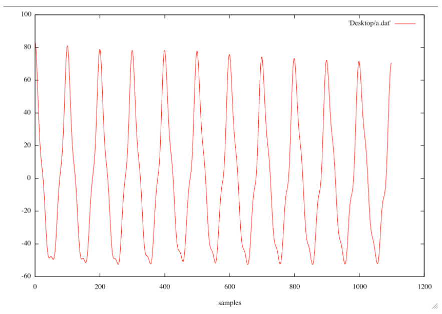

Figure 1

The peaks present in the Figure 1 represent parts of the signal that are similar or ideally the same (the cycles). Because the period of a signal is defined as the time between cycles and each cycle is depicted by an extrema in the autocorrelation function, the frequency of the wave will be described by: $F_0 = \frac{1}{\tau}$ where τ is the time between the extrema of the autocorrelation function.[5]

The next step is to calculate the difference function defined by the equation 2:

$$d_t(\tau) = r_t(0) + r_{t+\tau}(0) - 2r_t(\tau)$$

Equation 2

From the equation 2 we can see that the cycles of the signal that were represented by extrema using the Equation 1 are now represented by minima.

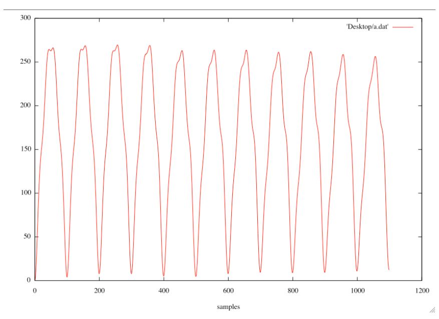

Figure 2

The Figure 2 plots the difference function for a piano playing at 440 Hz. The local minima represent parts of the signal that are the same (period dips). A close observation of the graph manifests that not all the minima have the same value. From the Equation 2 we can deduct that the cycle estimation precision is inverse proportional to the minima value. This fact will have relevance in further steps.

The use of a difference function drops the error rate from 10.0% to 1.95%.[6]

The cumulative mean normalized difference function is defined by the Equation 3:

$$d_t'(\tau) = 1 \text{ , if } \tau = 0$$

$$d_t'(\tau) = d_t(\tau) \Big/ \left[(1/\tau) \sum_{j=1}^{\tau} d_t(j) \right] \text{,}$$

otherwise

Equation 3

We saw that the global minimum of the Equation 2 is the most precise result. At the same time, the Equation 2 is 0 (global minimum) for $\tau = 0$ and the method will fail. Even if a threshold is set, strong resonances might produce secondary dips and those can be deeper than the period deep introducing errors in the estimation. The solution is to use the Equation 3 which normalizes the Equation 2 and it is 1 for $\tau = 0$.

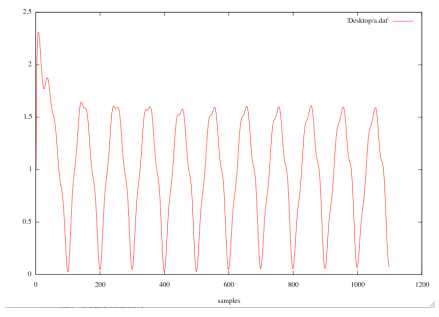

Figure 3

Figure 3 shows the cumulative mean normalized difference function. As it happened with the Equation 2, the precision of the period dip is inversely proportional to the minima value. After

[5] $\tau = s \cdot sampleRate^{-1}$ where s is the number of samples between extrema.

[6] Alain de Cheveigne, Hideki Kawahara (June 2001). "YIN, a fundamental frequency estimator for speech and music". http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf

applying the Equation 3 the error rate falls from 1.95% to 1.69%.[7]

The next step consists of detecting the local minima. In order to do so, a threshold is applied to the results of the Equation 3.

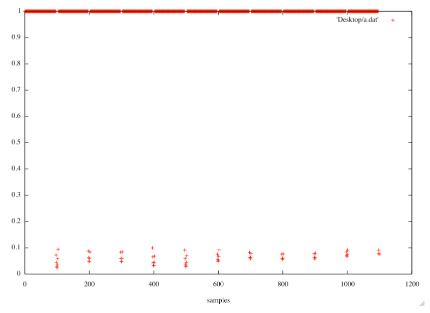

Figure 4

The figure 4 shows the points of the Figure 3 that are below a threshold of 0.1. All the other points are assigned a value of 1.

The previous steps work when the period is a multiple of the sampling period, otherwise the estimate might be incorrect by up to half the sampling period[8]. A solution to this is parabolic interpolation. Each dip is fitted to a parabola of the type $y = ax^2 + bx + c$ and then its minimum value is calculated using the inverse parabolic interpolation.

The final step is the selection of the best estimate. The parabolic interpolation from the last step provided different candidates for the period. From all the candidates in the window, the global minima is chosen as the best estimate. The fundamental frequency of the wave is defined by the Equation 4:

$$F_0 = \frac{x_m}{n} \cdot sampleRate^{-1}$$

Equation 4

where m is the x position of the lowest dip and n is the number of dip (first, second, third, etc...).

Parameters of YIN

The YIN algorithm has only two parameters that have to be defined by the user: The integration window size and the absolute threshold.

The window size will determine the overall performance of the algorithm. A high window size will imply more computational time and vice versa. At the same time, the window size determines the minimum frequency that the algorithm can detect. Therefore, it seems adequate to adjust the window size in order to allow YIN to detect a desired minimum frequency. A window size of 1103 has been chosen in order to allow YIN to detect frequencies until 40 Hz.

$$\min Window = \frac{1}{f_m} \cdot sampleRate$$

where f_m is the minimum frequency to be detected.

The YIN paper discusses different threshold values and analyses the gross error. In any case, the parameters for YIN don't have a dramatic effect in the error rates and therefore, the recommended by the paper are used (a window size of 1103 samples and a threshold of 0.1).

As explained at the beginning of this paper, YIN is a pitch detection

[7] Alain de Cheveigne, Hideki Kawahara (June 2001). "YIN, a fundamental frequency estimator for speech and music". http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf

[8] Alain de Cheveigne, Hideki Kawahara (June 2001). "YIN, a fundamental frequency estimator for speech and music". http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf

algorithm based in autocorrelation that uses additional steps to minimise the influence of the aperiodicity of the signal. Even though the additional steps succeed in decreasing the gross error from 10% to 1% the algorithm is still dependant on the periodicity of the signal. Unfortunately, most of the audio signals that are interesting for humans are not perfectly periodic.

In general, any audible rapid change in the audio will cause YIN to fail or give a gross error in the pitch estimation. Therefore, musical effects such as vibrato and attacks will be difficult to analyse for YIN. Because of the autocorrelation nature of YIN, only monophonic signals can be analysed.

Optimal window size

The minimum frequency is determined by the window size. The YIN paper states that the maximum frequency that the algorithm can detect is about a quarter of the sampling rate. Despite this affirmation, the current implementation of YIN can only detect frequencies up to a fourth of the Nyquist frequency.

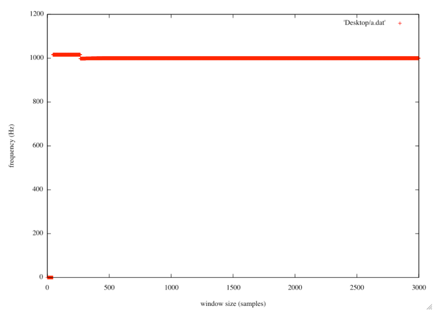

Figure 5

The Figure 5 shows the performance of the algorithm when analysing a sine tone of 1000 Hz with a window size of 0 samples to 3000. For window sizes greater than 500 samples, the estimated pitch is almost constant and very precise. The conclusion is that the window size doesn't have a dramatic effect on the algorithm precision. Therefore, as stated before a window size of 1103 samples is optimum.

Accuracy of YIN under different audio signals

Because different instruments produce signals that have different properties, it is to expect that the precision of Yin will depend on the type of instrument. The instruments analysed are piano, cello, tuba, harpsichord and xylophone. They correspond to the wind, string, hammered strings, plugged strings and tuned percussion family instruments. Even before any test one can foresee that because of the autocorrelation nature of YIN, percussive instruments will perform very badly. At the same time, a high playing speed or high frequencies will also affect the precision of the pitch estimate. Each instrument will be analysed playing a full chromatic scale with crochet, quavers, semiquavers and demisemiquaver at 120 BPM.

Piano crochet

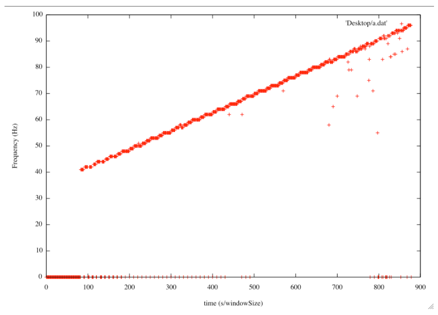

Figure 6

Piano quavers

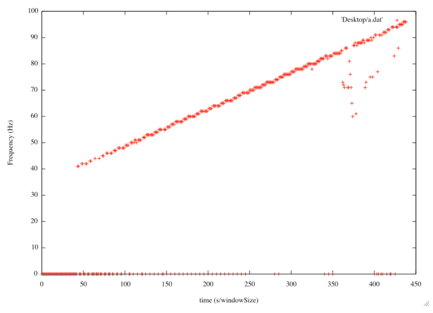

Figure 7

Cello quavers

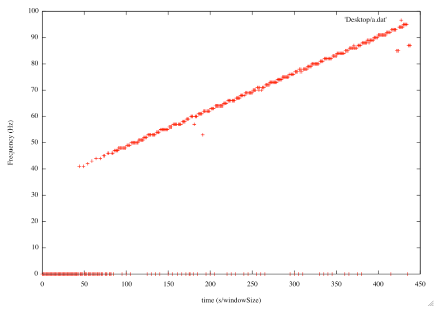

Figure 11

Piano semiquaver

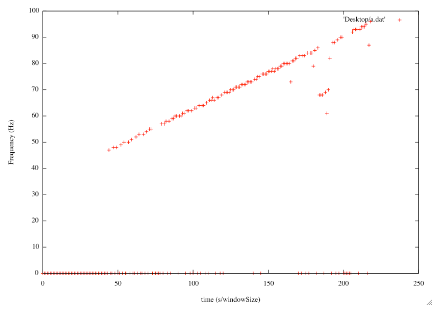

Figure 8

Cello semiquavers

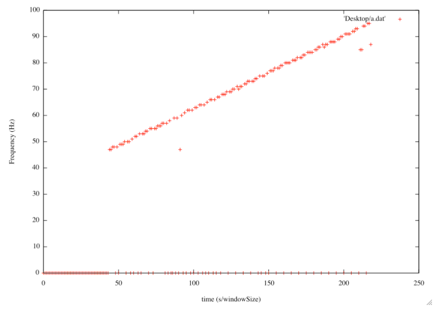

Figure 12

Piano demisemiquaver

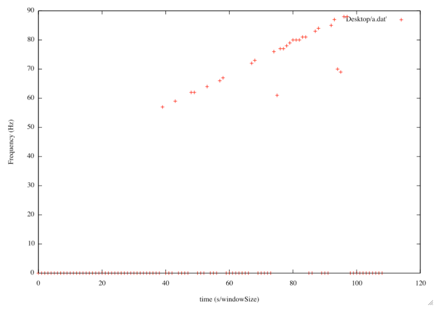

Figure 9

Cello demisemiquaver

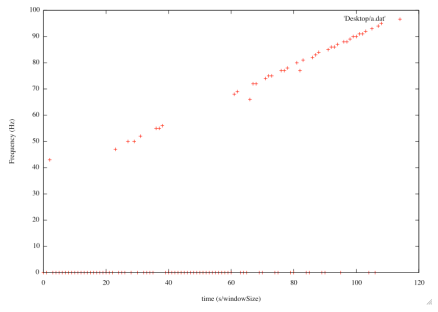

Figure 13

Cello crochet

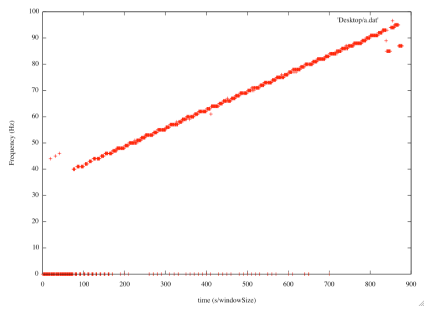

Figure 10

Tuba crochet

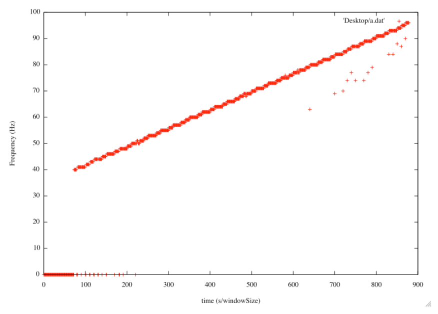

Figure 14

Tuba quavers

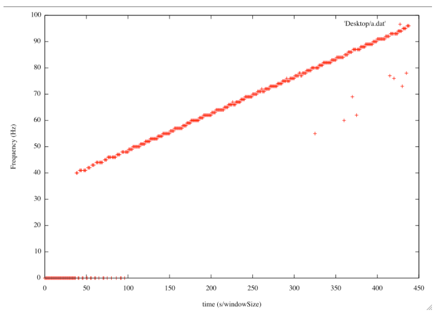

Figure 15

Tuba semiquavers

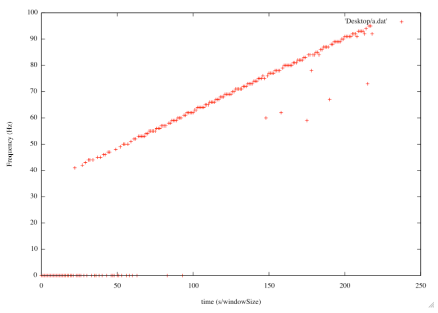

Figure 16

Tuba demisemiquaver

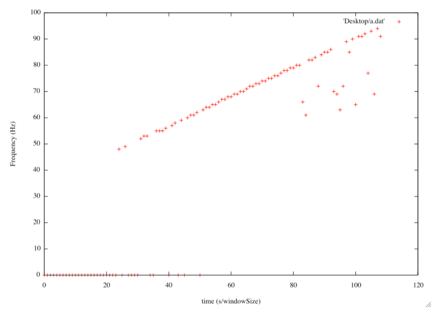

Figure 17

Harpsichord crochet

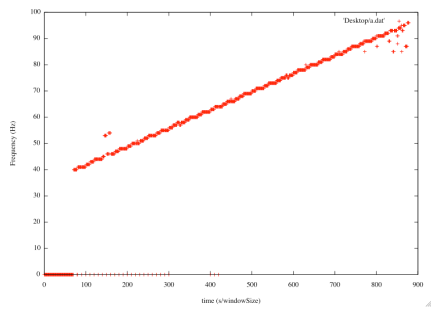

Figure 18

Harpsichord quavers

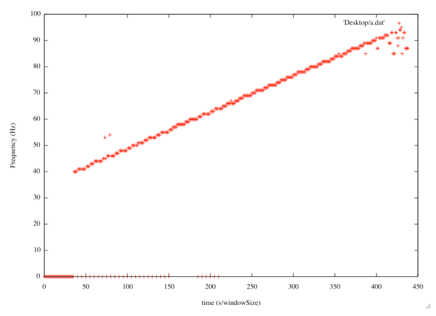

Figure 19

Harpsichord semiquavers

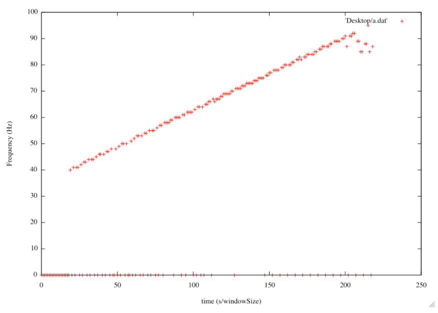

Figure 20

Harpsichord demisemiquaver

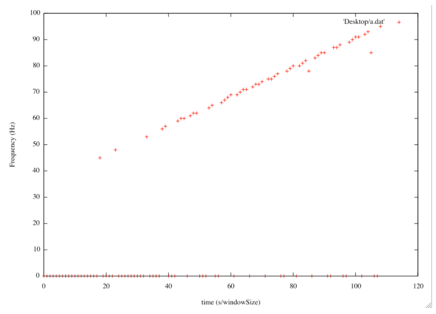

Figure 21

Xylophone crochet

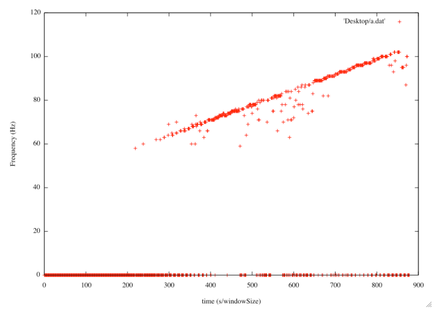

Figure 22

Xylophone Quavers

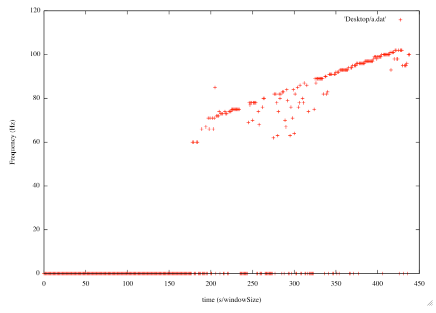

Figure 23

Xylophone semiquavers

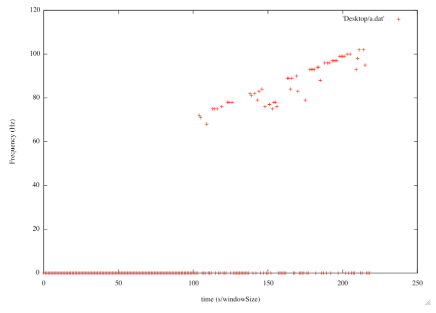

Figure 24

Xylophone demisemiquaver

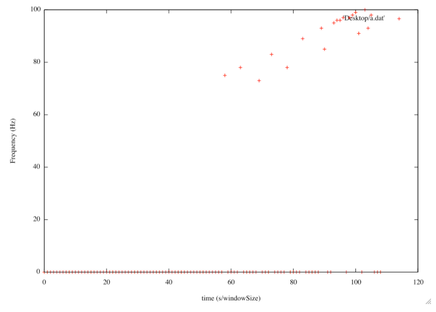

Figure 25

The figures above show the pitch estimation for a chromatic scale with different note durations at 120 BPM. A perfect measure would be represented by a stepped line with positive gradient starting at 40 midi number. Measures of 0 or out of the line represent errors in the estimation. For all of the instruments, the pitch estimation is more precise when the notes are long and low in pitch. For notes that are faster than demisemiquavers at 120 BPM, YIN performs badly. Also, for instruments that are highly percussive such as xylophone, YIN is not a good method. For the rest of instruments YIN performs very well providing precise pitch estimation in most of the cases. This is results are remarkable and a great achievement for YIN because any musical instrument creates audio signals that are high in aperiodicity.

System design

The implementation of the YIN algorithm follows the system proposed in the original paper.[9] The different steps are performed in order as each of the steps uses the results calculated in the previous steps.

The first step is the calculation of the autocorrelation of the signal. The Equation 1 in pseudocode:

```
for each k sample in the window
for each j sample in the window
add the multiplication of the sample k*(k+j)
```

The second step consists in the calculation of the difference function. The Equation 2 in pseudocode:

```
for each k sample in the window
autoCorrelationTau=0
        for each sample j in the window
        add the multiplication of j*j
```

[9] Alain de Cheveigne, Hideki Kawahara (June 2001). "YIN, a fundamental frequency estimator for speech and music". http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf

```
differenceFunction=autoCorrelation[0]*(1-autoCorrelation[k]*autoCorrelation[k]/autoCorrelation[0]*autoCorrelationTau
```

The third step is the calculation of the cumulative mean difference function, in pseudocode:

```
sum=0
for each k sample in the window
if k=0 then cumulativeDifference = 1
else sum+=differenceFunction[k]
cumulativeDifference=differenceFunction[k]/(sum/k)
```

The next step is to apply the absolute threshold to the cumulative mean difference function, in pseudocode:

```
for each k sample in the window
if cumulativeDifference[k] > threshold then absoluteThreshold[k]=cumulativeDifference[k]
else cumulativeDifference[k]=1
```

The fourth step consists of finding the minima of the absolute threshold array. When a dip has been found, the dip, its previous and its next point are passed to a function that fits a parabola of type $y = ax^2 + bx + c$ to them and calculates the "x" position of the vertex using inverse parabolic interpolation[10].

The final step is to choose the minima that will give the best estimation. This point will be the global minima of the parabola vertices calculated in the fourth step. In pseudocode:

```
for all the candidates
        the minima is the first point in the list
        if the next point < than the minima then minima=next point
```

The fifth step gives the most precise pitch estimation. However, the result has units of samples. The final step is to convert the units of the best estimate from samples to Hz. In pseudocode:

```
frequency=1/(bestEstimate/index in the list)/(sampleRate)
```

The present implementation contains an additional step which is the conversion from Hz to MIDI number.

Because usually one would like to find what the different notes in an audio file are, the algorithm is applied in the next way:

An audio .wav file is loaded in an array. Additionally a buffer is created containing only a section of the audio file of size equal to the window size. The YIN algorithm is performed over this window and the results printed. The buffer is filled with the consecutive data from the audio file and the process is repeated until the end of the audio file. The data printed will contain not only pitch values but also onset information. This way the algorithm can detect notes with pitch and duration.

Conclusion

The implementation of the YIN algorithm presented in this report estimates the pitch of monophonic audio signals in audio files with an average error of 1%. The program can be used for the transcription of audio as it provides pitch, onset and duration

10 James F. Epperson, An introduction to numerical methods and analysis, pages 182-185, Wiley-Interscience, 2007.

of each of the notes present in the audio file. The project tests the YIN algorithm with different audio data in order to define its behaviour under different audio signals. Performing an exhaustive test would be impossible as the variety of audio signals to be analysed is infinite. However, the algorithm was tested against a generic range of instruments and the result orientates a potential user on how YIN estimates pitch under different circumstances.

The project fulfils its initial aims. At the same time the project can be developed further by implementing a FFT calculation of the autocorrelation function that would speed up the performance and allow real time analysis.

Additionally, the automatic conversion of an audio file to a midi file could be implemented.

References

- William E. Boyce and Richard C. DiPrima (2005). *Elementary Differential Equations and Boundary Value Problems* (8th ed.).
- Alain de Cheveigne, Hideki Kawahara (June 2001). *YIN, a fundamental frequency estimator for speech and music.* http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf
- Siu-Lan Tan, Peter Pfordresher and Rom Harré (2005). *Psychology of Music.*
- Schwartz, D.A.; Purves, D. (May 2004). *Pitch is determined by naturally occurring periodic sounds.* Hearing Research **194**: 31–46.
- James F. Epperson (2007). *An introduction to numerical methods and analysis.*

Appendix

Please find the .wav files used for testing the YIN algorithm on this address:
http://www.doc.gold.ac.uk/~mu101ag/

JAVA CODE

Example class:

```
/*

Summary
-------

Example class runs a program that detects the pitch, onset and
duration of notes in audio files.


*/

package miniproject;

import java.util.ArrayList;

import javax.sound.sampled.LineUnavailableException;

public class Example {

	/**
	 * @param args
	 * @throws LineUnavailableException
	 */

	public static void main(String[] args) throws
LineUnavailableException {

		ArrayList<double[]> data = new ArrayList<double[]>();
		/* YIN reading a wav file */
		// Store an audio file as samples.

		double[] audioSamplesFromFile;
		audioSamplesFromFile =
StdAudio.read("/Users/Andreugrimalt/Desktop/pap/pianoCrochet120.wav")
;
		int yinWindow=1103;
		// Create a buffer that will contain just 1 chunk of the
audio file. The size has to be double of the Yin window size.
		double[] audioBuffer = new double[2*yinWindow];
		// Load chunks in the buffer and analyze each one of
them. Careful with (i+j) and arrayIndexOutOfBounds.
		for(int j=0; j<audioSamplesFromFile.length-
(2*yinWindow); j+=(2*yinWindow)){
                        for(int i=0; i<audioBuffer.length; i++){
                              audioBuffer[i]=audioSamplesFromFile[i+j];
                        }
                        Yin yin =new Yin(yinWindow);

                        yin.autocorrelation(audioBuffer);
                        yin.differenceFunction(audioBuffer);
                        yin.cumulativeMeanDifferenceFunction();
                        yin.absoluteThreshold();
                        yin.findDips();
                        yin.bestLocalEstimate();
                        data.add(yin.printData(j));
                  }


                  //YIN for signals that have amplitude variations
                  /*
                  Signal sine = new Signal(0.3,440);
                  for(int i=1104; i<1105; i++){

                        // To check how a windowed signal behaves.
                        /*
                        double[] s = new double[2*i];
                        for(int j=0; j<i; j++){
                              s[j] = sine.getSamples()[j] * 0.5*(1-
Math.cos(2*Math.PI*j/(s.length/2-1)));
                        }
                        for(int j=i; j<2*i; j++){
                              s[j] = s[j-i];
                        }
                        for(int j=0; j<s.length; j++){
                              //System.out.println(j+", "+s[j]);
                        }*/
                  /*
                        // Exponential decay
                        double[] s= new double[2*i];
                        for(int j=0; j<2*i; j++){
                              s[j] = sine.getSamples()[j]*Math.exp(-
j/100.0);

                              //System.out.println(j+", "+s[j]);
                        }

                        Yin yin = new Yin(i);
                        // Calculates autocorrelation.
                        yin.autocorrelation(s);
                        yin.differenceFunctionAmplitudeVariation(s);
                        //yin.differenceFunction(s);
                        yin.cumulativeMeanDifferenceFunction();
                        yin.absoluteThreshold();
                        yin.findDips();
                        // Why we need to divide by 2?
```

```
				System.out.println(i+", "+yin.calcFreq()/2.0);
			}*/


			//YIN for sine tones
			/*
			Yin yin = new Yin(1104);
			Signal sine = new Signal(0.3,4234);
			yin.autocorrelation(sine.getSamples());
			yin.differenceFunction(sine.getSamples());
			yin.cumulativeMeanDifferenceFunction();
			yin.absoluteThreshold();
			yin.findDips();
			yin.calcFreq(); // To check that the best estimate is
really the best.
			System.out.println("Best Estimation=
"+yin.bestLocalEstimate());
			*/

			// YIN different window sizes.
			/*
			Signal sine = new Signal(0.3,1000);
			for(int i=1; i<3000; i++){
			Yin yin = new Yin(i);

			// Calculates autocorrelation.

			yin.autocorrelation(sine.getSamples());
			yin.differenceFunction(sine.getSamples());
			yin.cumulativeMeanDifferenceFunction();
			yin.absoluteThreshold();
			yin.findDips();
			yin.bestLocalEstimate();
			System.out.println(i+", "+yin.printData(0)[0]);
			}*/
	}
}
```

YIN class:

```
/*

Summary
-------

YIN class implements the YIN algorithm for pitch estimation on audio
signals.

Description
-----------
```

```
1. Calculate autocorrelation of the signal.
2. Calculate difference function.
3. Calculate cumulative mean difference function.
4. Apply absolute threshold.
5. Look for minima and perform parabolic interpolation
6. Select the best estimate.
7. Convert the best estimate units from samples to Hz

*/

package miniproject;

import java.util.ArrayList;

public class Yin {

	private double[] autoCorrelation;
	private double[] differenceFunction;

	private double[] cumulativeMeanDifference;
	private double[] absoluteThreshold;
	ArrayList<Double> tau = new ArrayList<Double>();
	ArrayList<Double> tauY = new ArrayList<Double>();
	ArrayList<Double> frequencies = new ArrayList<Double>();
	private int maxLag;
	double frequencyEstimate;

	public Yin(int theMaxLag){
		maxLag=theMaxLag;
		autoCorrelation = new double[maxLag];
		differenceFunction = new double[maxLag];
		cumulativeMeanDifference = new double[maxLag];
		absoluteThreshold = new double[maxLag];

	}

	public double[] getAutoCorrelation(){
		return autoCorrelation;
	}

	public void autocorrelation(double[] theSignal){

		for(int i=0; i<maxLag; i++){
			for(int j=0; j<maxLag; j++){
				autoCorrelation[i] += 
theSignal[j]*theSignal[(j+i)];
			}
			//System.out.println(i+", "+autoCorrelation[i]);

		}
	}
	/*public void modifiedAutocorrelation(double[] theSignal){

		for(int i=0; i<maxLag; i++){
			for(int j=0; j<maxLag-i; j++){
				autoCorrelation[i] +=
theSignal[j]*theSignal[(j+i)];
			}
			//System.out.println(autoCorrelation[i]+", "+i);
		}
	}*/

	public void differenceFunction(double[] theSignal){

		for(int i=0; i<maxLag; i++){
			double autoCorrelationTau=0;
			// Value of r(0) for each tau.
			for(int j=i; j<i+maxLag; j++){
				autoCorrelationTau+=
theSignal[j]*theSignal[j];
			}
			//System.out.println(autoCorrelationTau);

	differenceFunction[i]=autoCorrelation[0]+autoCorrelationTau-
2*autoCorrelation[i];
			//System.out.println(i+", "+differenceFunction[i]);
		}
		//System.out.println(autoCorrelation[0]);
	}

	public void differenceFunctionAmplitudeVariation(double[]
theSignal){

		for(int i=0; i<maxLag; i++){
			double autoCorrelationTau=0;
			// Value of r(0) for each tau.
			for(int j=i; j<i+maxLag; j++){

	autoCorrelationTau+=theSignal[j]*theSignal[j];
			}
			differenceFunction[i]=autoCorrelation[0]*(1-
((autoCorrelation[i]*autoCorrelation[i])/(autoCorrelation[0]*autoCorr
elationTau)));
			//System.out.println(i+", "+differenceFunction[i]);
		}
		//System.out.println(autoCorrelation[0]);
	}

	public void cumulativeMeanDifferenceFunction(){
		double sum=0;

		for(int i=0; i<maxLag; i++){
			if(i==0){
				cumulativeMeanDifference[i] = 1;
			}else{
				sum+=differenceFunction[i];
				
	cumulativeMeanDifference[i]=differenceFunction[i]/((1.0/i)*sum)
;
			}
			//System.out.println(i+", 
"+cumulativeMeanDifference[i]);
		}
	}

	public void absoluteThreshold(){
		double threshold = 0.1;
		for(int i=0; i<maxLag; i++){
			if(cumulativeMeanDifference[i]<=threshold){
				absoluteThreshold[i] = 
cumulativeMeanDifference[i];
			}else{
				absoluteThreshold[i] = 1;
			}
			//System.out.println(i+", "+absoluteThreshold[i]);
		}
	}

	public void findDips(){
		int j=1;
		ArrayList<Integer> dip = new ArrayList<Integer>();

		while(j<maxLag-1){
			if(absoluteThreshold[j]!=1.0&&absoluteThreshold[j-
1]>absoluteThreshold[j]&&absoluteThreshold[j]<absoluteThreshold[j+1])
{
				dip.add(j);
			}
			j++;
		}
		int k=0;
		while(k<dip.size()){

			k++;
		}
		for(int i=0; i<dip.size();i++){

			//System.out.println("-------------------");
			//System.out.println("dip"+i+": "+dip.get(i)+", 
"+absoluteThreshold[dip.get(i)]);
			//System.out.println("parabolic dips x= 
"+(dip.get(i)-1)+", "+dip.get(i)+", "+(dip.get(i)+1));
			//System.out.println("parabolic dips y= 
"+absoluteThreshold[(dip.get(i)-1)]+", 
"+absoluteThreshold[dip.get(i)]+", 
"+absoluteThreshold[(dip.get(i)+1)]);
                  parabollicInterpolation(dip.get(i)-
1,dip.get(i),dip.get(i)+1);
            }
      }

      // Check all this.
      public void parabollicInterpolation(double a, double b,
double c){
            // Fit to parabola of the form y=a*(x-x2)*(x-x2)+b*(x-
x2)+y2
            double alpha=0;
            double betta=0;
            double gamma=0;

            double x1=a;
            double x2=b;
            double x3=c;
            double y1=absoluteThreshold[(int)a];
            double y2=absoluteThreshold[(int)b];
            double y3=absoluteThreshold[(int)c];
            alpha=((y3-y2)/(x3-x2)-(y2-y1)/(x2-x1))/(x3-x1);
            betta=((y3-y2)/(x3-x2)*(x2-x1)+(y2-y1)/(x2-x1)*(x3-
x2))/(x3-x1);
            gamma=y2;

            //System.out.println("Eq: "+alpha+", "+betta+", "+gamma);


            // Inverse parabolic interpolation finds the vertex.
            double x=0;

            //System.out.println(a+","+b+","+c);
            for(int i=0; i<10; i++){
                  // Iteration
                  x = b-(1.0/2.0)*(Math.pow((b-a),2)/(b-
a))*(absoluteThreshold[Math.round((float)b)]-
absoluteThreshold[Math.round((float)c)])-Math.pow((b-
c),2)*(absoluteThreshold[Math.round((float)b)]-
absoluteThreshold[Math.round((float)a)])/((absoluteThreshold[Math.ro
und((float)b)]-absoluteThreshold[Math.round((float)c)])-(b-
c)*(absoluteThreshold[Math.round((float)b)]-
absoluteThreshold[Math.round((float)a)]));


      if(x>b&&absoluteThreshold[(int)x]<absoluteThreshold[(int)b]){
                        a=b;
                        b=x;
                        //c=c;
                  }
```

```
        if(x>b&&absoluteThreshold[(int)x]>absoluteThreshold[(int)b]){
                        //a=a;
                          //b=b;
                          c=x;
                    }


        if(x<b&&absoluteThreshold[(int)x]<absoluteThreshold[(int)b]){
                          //a=a;
                          b=x;
                          //c=c;
                    }


        if(x<b&&absoluteThreshold[(int)x]>absoluteThreshold[(int)b]){
                          a=x;
                          //b=b;
                          //c=c;
                    }
              }
              // Check the ordinate to determine best estimate.
              double tauOrdinate = alpha*(x-x2)*(x-x2) + betta*(x-x2)
+ gamma;
              //System.out.println("Parabolic interpolation x= "+x+ "
y= "+tauOrdinate);
              tauY.add(tauOrdinate);
              tau.add(x);

        }

        public double bestLocalEstimate(){
              /* Check for the minimum of d'(tau) in the window*/
              double bestFrequency=0;
              if(tau.size()>0){
                    double tempMin=tauY.get(0);
                    double index = 0;
                    for(int i=0; i<tauY.size();i++){
                          if(tauY.get(i)<tempMin){
                                tempMin=tauY.get(i);
                                index=i;
                          }
                    }
                    bestFrequency=calcFreq(index);
                    //System.out.println("On dip= "+(index+1));
                    //System.out.println("BBBest Local is: "+tempMin+",
"+(index+1));
              }
              if(tau.size()==0){
                    bestFrequency=0;
              }
              frequencyEstimate=bestFrequency;
              return bestFrequency;
```

```
            //double
freq=1/((tau.get((int)index)/(index+1))/Signal.SAMPLE_RATE);
            //System.out.println("best freq= "+freq);

      }

      public double calcFreq(double index){
            double freq=0;
            //System.out.println("index is= "+index);
            if(index<tau.size()){

      freq=1/((tau.get((int)index)/(index+1))/Signal.SAMPLE_RATE);
            }else{
                  freq=0;
            }
            return freq;
      }

      public int getMidiNumber(double frequency){
            int midiNumber =
(int)(69+12*Math.log(Math.round(frequency)/440.0));
            return midiNumber;
      }

      public double[] printData(int timeFrame){

                  double[] dataArray = new double[3];
                  dataArray[0]=frequencyEstimate;
                  dataArray[1]=getMidiNumber(frequencyEstimate);
                  // Approximate to 0
                  if(dataArray[1]<0){
                        dataArray[1]=0;
                  }
                  dataArray[2]=timeFrame/44100.0;
                  //System.out.println(dataArray[0]);
                  //System.out.println("Best Estimation (Hz) =
"+dataArray[0]+"        Midi Note = "+dataArray[1]+"        TimeFrame
(s) = "+dataArray[2]);
                  return dataArray;
      }
}
```

Signal class:

```
/*

Summary
-------

Signal class generates a sine tone of a specified frequency.
```

```
*/

package miniproject;

public class Signal {

	public static final int SAMPLE_RATE=44100;
	public static int numSamples;
	private double[] samples;

	public Signal(double seconds, double frequency){
		// Number of samples.
		numSamples = (int)(SAMPLE_RATE*seconds);
		// Initialise samples array.
		samples = new double[numSamples];
		// Phase.
		double phase = 2*Math.PI*frequency/SAMPLE_RATE;
		// Fill array with a sine tone with frequency=f.
		for(int i=0; i<numSamples; i++){
			samples[i]=Math.sin(phase*i);
			//System.out.println(samples[i]+","+i);
		}
	}

	public double[] getSamples(){
		return samples;
	}
}
```

StdAudio class:

```
package miniproject;

/*************************************************************************
*****
 *  Compilation:  javac StdAudio.java
 *  Execution:    java StdAudio
 *
 *  Simple library for reading, writing, and manipulating .wav files.

 *
 *  Limitations
 *  -----------
 *    - Does not seem to work properly when reading .wav files from a
.jar file.
 *    - Assumes the audio is monaural, with sampling rate of 44,100.
 *

*************************************************************************
****/
```

```
import java.applet.*;
import java.io.*;
import java.net.*;
import javax.sound.sampled.*;

/**
 *  <i>Standard audio</i>. This class provides a basic capability for
 *  creating, reading, and saving audio.
 *  <p>
 *  The audio format uses a sampling rate of 44,100 (CD quality
audio), 16-bit, monaural.
 *
 *  <p>
 *  For additional documentation, see <a
href="http://introcs.cs.princeton.edu/15inout">Section 1.5</a> of
 *  <i>Introduction to Programming in Java: An Interdisciplinary
Approach</i> by Robert Sedgewick and Kevin Wayne.
 */
public final class StdAudio {

    /**
     *  The sample rate - 44,100 Hz for CD quality audio.
     */
    public static final int SAMPLE_RATE = 44100;

    private static final int BYTES_PER_SAMPLE = 2;
// 16-bit audio
    private static final int BITS_PER_SAMPLE = 16;
// 16-bit audio
    private static final double MAX_16_BIT = Short.MAX_VALUE;
// 32,767
    private static final int SAMPLE_BUFFER_SIZE = 4096;


    private static SourceDataLine line;   // to play the sound
    private static byte[] buffer;          // our internal buffer
    private static int bufferSize = 0;    // number of samples
currently in internal buffer

    // do not instantiate
    private StdAudio() { }


    // static initializer
    static { init(); }

    // open up an audio stream
    private static void init() {
        try {
            // 44,100 samples per second, 16-bit audio, mono, signed
PCM, little Endian
```

```
            AudioFormat format = new AudioFormat((float)
SAMPLE_RATE, BITS_PER_SAMPLE, 1, true, false);
            DataLine.Info info = new
DataLine.Info(SourceDataLine.class, format);

            line = (SourceDataLine) AudioSystem.getLine(info);
            line.open(format, SAMPLE_BUFFER_SIZE * BYTES_PER_SAMPLE);

            // the internal buffer is a fraction of the actual buffer
size, this choice is arbitrary
            // it gets divided because we can't expect the buffered
data to line up exactly with when
            // the sound card decides to push out its samples.
            buffer = new byte[SAMPLE_BUFFER_SIZE *
BYTES_PER_SAMPLE/3];
        } catch (Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }

        // no sound gets made before this call
        line.start();
    }


    /**
     * Close standard audio.
     */
    public static void close() {
        line.drain();
        line.stop();
    }

    /**
     * Write one sample (between -1.0 and +1.0) to standard audio. If
the sample
     * is outside the range, it will be clipped.
     */
    public static void play(double in) {

        // clip if outside [-1, +1]
        if (in < -1.0) in = -1.0;
        if (in > +1.0) in = +1.0;

        // convert to bytes
        short s = (short) (MAX_16_BIT * in);
        buffer[bufferSize++] = (byte) s;
        buffer[bufferSize++] = (byte) (s >> 8);   // little Endian

        // send to sound card if buffer is full
        if (bufferSize >= buffer.length) {
            line.write(buffer, 0, buffer.length);
            bufferSize = 0;
        }
    }

    /**
     * Write an array of samples (between -1.0 and +1.0) to standard
audio. If a sample
     * is outside the range, it will be clipped.
     */
    public static void play(double[] input) {
        for (int i = 0; i < input.length; i++) {
            play(input[i]);
        }
    }

    /**
     * Read audio samples from a file (in .wav or .au format) and
return them as a double array
     * with values between -1.0 and +1.0.
     */
    public static double[] read(String filename) {
        byte[] data = readByte(filename);
        int N = data.length;
        double[] d = new double[N/2];
        for (int i = 0; i < N/2; i++) {
            d[i] = ((short) (((data[2*i+1] & 0xFF) << 8) +
(data[2*i] & 0xFF))) / ((double) MAX_16_BIT);
        }
        // Added by me.
        System.out.println("The array has "+d.length+" samples");

        return d;
    }




    /**
     * Play a sound file (in .wav, .mid, or .au format) in a
background thread.
     */
    public static void play(String filename) {
        URL url = null;
        try {
            File file = new File(filename);
            if (file.canRead()) url = file.toURI().toURL();
        }
        catch (MalformedURLException e) { e.printStackTrace(); }
        // URL url = StdAudio.class.getResource(filename);
        if (url == null) throw new RuntimeException("audio " +
filename + " not found");
        AudioClip clip = Applet.newAudioClip(url);
        clip.play();
    }

    /**
     * Loop a sound file (in .wav, .mid, or .au format) in a
background thread.
     */
    public static void loop(String filename) {
        URL url = null;
        try {
            File file = new File(filename);
            if (file.canRead()) url = file.toURI().toURL();
        }
        catch (MalformedURLException e) { e.printStackTrace(); }
        // URL url = StdAudio.class.getResource(filename);
        if (url == null) throw new RuntimeException("audio " +
filename + " not found");
        AudioClip clip = Applet.newAudioClip(url);
        clip.loop();
    }


    // return data as a byte array
    private static byte[] readByte(String filename) {
        byte[] data = null;
        AudioInputStream ais = null;
        try {

            // try to read from file
            File file = new File(filename);
            if (file.exists()) {
                ais = AudioSystem.getAudioInputStream(file);
                data = new byte[ais.available()];
                ais.read(data);
            }

            // try to read from URL
            else {
                URL url = StdAudio.class.getResource(filename);
                ais = AudioSystem.getAudioInputStream(url);
                data = new byte[ais.available()];
                ais.read(data);
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
            throw new RuntimeException("Could not read " +
filename);
        }

        return data;
    }
    /**
     * Save the double array as a sound file (using .wav or .au
format).
     */
    public static void save(String filename, double[] input) {

        // assumes 44,100 samples per second
        // use 16-bit audio, mono, signed PCM, little Endian
        AudioFormat format = new AudioFormat(SAMPLE_RATE, 16, 1,
true, false);
        byte[] data = new byte[2 * input.length];
        for (int i = 0; i < input.length; i++) {
            int temp = (short) (input[i] * MAX_16_BIT);
            data[2*i + 0] = (byte) temp;
            data[2*i + 1] = (byte) (temp >> 8);
        }

        // now save the file
        try {
            ByteArrayInputStream bais = new
ByteArrayInputStream(data);
            AudioInputStream ais = new AudioInputStream(bais,
format, input.length);
            if (filename.endsWith(".wav") ||
filename.endsWith(".WAV")) {
                AudioSystem.write(ais, AudioFileFormat.Type.WAVE,
new File(filename));
            }
            else if (filename.endsWith(".au") ||
filename.endsWith(".AU")) {
                AudioSystem.write(ais, AudioFileFormat.Type.AU, new
File(filename));
            }
            else {
                throw new RuntimeException("File format not
supported: " + filename);
            }
        }
        catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }
    }
}
```

```
/*****************************************************************
***
    * sample test client

*****************************************************************
**/

    // create a note (sine wave) of the given frequency (Hz), for the
given
    // duration (seconds) scaled to the given volume (amplitude)
/*
    private static double[] note(double hz, double duration, double
amplitude) {
        int N = (int) (StdAudio.SAMPLE_RATE * duration);
        double[] a = new double[N+1];
        for (int i = 0; i <= N; i++)
            a[i] = amplitude * Math.sin(2 * Math.PI * i * hz /
StdAudio.SAMPLE_RATE);
        return a;
    }

    /**
     * Test client - play an A major scale to standard audio.
     */
/*
    public static void main(String[] args) {

        // 440 Hz for 1 sec
        double freq = 440.0;
        for (int i = 0; i <= StdAudio.SAMPLE_RATE; i++) {
            StdAudio.play(0.5 * Math.sin(2*Math.PI * freq * i /
StdAudio.SAMPLE_RATE));
        }

        // scale increments
        int[] steps = { 0, 2, 4, 5, 7, 9, 11, 12 };
        for (int i = 0; i < steps.length; i++) {
            double hz = 440.0 * Math.pow(2, steps[i] / 12.0);
            StdAudio.play(note(hz, 1.0, 0.5));
        }


        // need to call this in non-interactive stuff so the program
doesn't terminate
        // until all the sound leaves the speaker.
        StdAudio.close();

        // need to terminate a Java program with sound
        System.exit(0);
    }
}
*/
```