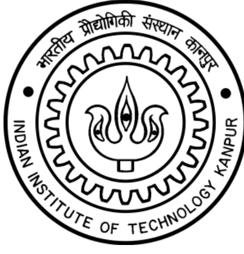

ONLINE HEALTH MONITORING OF ROTATING MACHINES

PROJECT REPORT

SUBMITTED IN THE FULFILLMENT OF THE REQUIREMENTS FOR THE COURSE EE393A

Project Mentor
Dr. Nishchal Kumar Verma
Assistant Professor,
Department of Electrical Engineering,
Indian Institute of Technology, Kanpur

SUBMITTED BY
Anirudh Kr. Agrawal
Roll no - 11098
3rd year Undergraduate Student,
Indian Institute of Technology, Kanpur

ACKNOWLEDGEMENT

I would like to thank Prof. **Nishchal Kumar Verma**, Assistant Professor, Department of Electrical Engineering, IIT Kanpur for his generous guidance and useful suggestions during the project. And also for making the project a learning experience for me.

I also thank Mr. Rahul Sevakula and Ms. Sonal Dixit for their mentorship during the project and insights into the problems faced.

Contents

Page No

1. Introduction 4
2. Previous Work 5
3. Present Work 7
4. Results 9
5. Conclusion 11
6. Future Scope 11
7. Source Codes 12

Introduction

Real-time machine fault diagnosis based on acoustic and vibrational readings from rotating machine has been a prime research interest for industries. Therefore we see lot of research in this direction in the last decade. Maintaining production rate with non-intrusive condition monitoring is a very useful method.

Condition Based Monitoring System for such rotating machine has been developed to be run on android platform on a smartphone. The smartphone acquires the acoustic and vibration data, this is then fed into recognition software already trained. The acquired data is then converted into a feature space chosen for this application consisting of various functions. Based on the recognition algorithm like Support Vector Machines or Neural Networks this can classified into one of the conditions defined earlier.

The smartphone can send this information through the internet or SMS regarding the condition of the machine thus providing a remote monitoring device. And thus the condition of the machine can be accessed remotely through an internet connected device anywhere in the world.

Android is a mobile operation system based on Dalvik Virtual Machine. Application can be coded Android in Java language. Android was chosen because of the ease of application development and deployment on an actual device.

Coding for this project was done on Eclipse v3.8 which is an IDE for Java language, and provides a simple interface for developing android applications.

After the data is taken its is preprocessed and then features are extracted from the signal these features represent the signal and qualitatively describe it. Based on these features extracted from the signal our machine learning system can recognize the system. Preprocessing refers to the process of removing noise from the data, checking the validity of the data and modeling it for suitable extraction of features from it.

Previous Work

Feature extraction from the incoming time signal is an important stage in machine learning system. In the previous work various time domain, frequency and time-frequency domain function were used listed below.

1. Time Domain , Absolute Mean, Root Mean Square, Variance, Skew, Kurtosis,
2. Frequency Domain: Energy of different bins in the frequency spectrum obtained after dividing the frequency spectrum into 8 bins. .
3. Continuous Wavelet Transform: After performing Continuous Wavelet Transform of the input signal various signal defining properties for example kurtosis, skew, mean were calculated.
4. Wavelet Packet Transform: In a recursive fashion the signal is transformed by the wavelet transform and at each step is further broken into half and again transformed, energy at each level is stored.

Java Language has few numbers of different libraries for signal processing but these are mostly limited to Fourier Transform and Discrete Cosine Transform. It seems that Java is not the industry standard for signal processing as much effort has not been placed into developing libraries for it.

This summer I worked on improving the preprocessing module for the Machine Learning System .The new preprocessing module developed consisted of a filter , followed by a clipping algorithm, after which smoothing of data is performed and lastly the smoothed data is normalized.

The clipping algorithm used a clustering algorithm to find an ideal segment in a feature space defined for this purpose, consisting of 18 features. K-means clustering was used as the clustering algorithm using Euclidean distance as the similarity measure. The number of appropriate cluster was decided using Global Silhouette as the decision criteria. According to the new clipping algorithm any new input signal was first segmented into segments of 5000 sample length with an overlap of 50% i.e 2500 samples. The segment which was closest to the ideal segment as found during clustering of the training dataset was chosen as the clipped segment.

Based on the comparison done between various smoothing techniques moving average filter with 5 sample length window was chosen was the smoothing algorithm for our preprocessing module. Also, modifications in standard max-min normalizations were found to become more robust against outliers in the data. This was done seeing that even a few outliers in one signal makes the normalization ineffective and reduces the efficiency of the machine learning algorithms. We found that the input signal had a Gaussian distribution, based on which decided to use the 3 sigma interval as our confidence range. Points outside this interval were not included in the calculation of the max and minimum of the data. The max and min found from the new dataset was used for max-min normalization.

As emphasized before that the feature space chosen for the machine learning algorithm is vital for efficiency. Time and frequency domain features have already been largely tested for tis particular condition based monitoring system. The only time-frequency domain level features chosen were the wavelet transform. The advantage that time-frequency features offer over the purely time or purely frequency feature is that many events occur at a particular instance which cannot be detected using frequency domain. And thus to determine the spectral properties of these rare occurrences we need time-frequency functions.

Present Work

There exist many more time-frequency domain transforms which can be used as features for the machine learning system. A number of them which could be potentially used in the fault recognitions system were coded by me in JAVA language for Android Platform to be included in the Smartphone condition monitoring application. I have listed them below:-

1. Short Time Fourier Transform: It is obtained taking Fourier transform of small windows of the time signal. Thus the spectral properties of the signal at various points of time can be found, giving us a better view of the input signal. It is defined as:

$$X(m,\omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega n} \tag{1}$$

2. Wigner-Ville Transform : Is the most important and simplest of the Cohen's class of Bilinear TFDs and is defined as :

$$W_x(t,\omega) = \int_{-\infty}^{\infty} x(t+\frac{\tau}{2})x^*(t-\frac{\tau}{2})e^{-i\omega\tau}d\tau \tag{2}$$

3. Pseudo Wigner-Ville Transform: Is the same as Wigner Ville Transform but with a smoothing kernel.
4. S-Transform: It is an advanced form of STFT and uses variable window length sizes at different frequencies and thus having better resolution. With fixed window length It is defined as

$$S_x(t,f) = \int_{-\infty}^{\infty} x(\tau)|f|e^{-\pi(t-\tau)^2 f^2} e^{-j2\pi f\tau}\, d\tau \tag{3}$$

5. Born-Jordan Transform : Is a shift invariant, kernel smoothed Wigner-ville distribution and is defined as :

$$BJD(t,\omega) = \int_{-\infty}^{\infty}\int_{t-|\alpha\tau|}^{t+|\alpha\tau|} \frac{1}{2\alpha\tau} z(u+\frac{\tau}{2})z^*(u-\frac{\tau}{2})e^{-j\omega\tau}\, du\, d\tau \tag{4}$$

It is also able to suppress cross terms.

6. Choi-Williams Transform : Is able to suppress the cross –terms of the Wigner Ville Transform by having an exponential kernel as shown below

$$\Phi(\theta, \tau) = \frac{\sqrt{\pi\sigma}}{|\tau|} e^{-\pi^2 \sigma t^2 / \tau^2} \tag{5}$$

And thus the distribution becomes

$$CWD(t,\omega) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \frac{\sqrt{\pi\sigma}}{|\tau|} e^{\frac{-\pi^2\sigma(t-u)^2}{\tau^2}} z(u+\frac{\tau}{2}) z^*(u-\frac{\tau}{2}) \times e^{-j\omega\tau} du d\tau \tag{6}$$

Apart from these following functions were also developed in the Java language by me

1. Autocorrelation: A function checks to the correlation in the signal with certain time lag between the samples.
2. Updated Morlet Transform

I have used a pre-existing library 'JTransform' for Discrete Cosine Transform and Common math Apache library for FFT. One advantage that JTransform offers over Common Apache functions is that it can even work with signals whose length is a power of 2. As required by some functions above.

All these codes were test for correctness by testing similar inputs to MATLAB implementations of these functions. The outputs of MATLAB and JAVA functions were found to match up to the 4 digits after the decimal. This level of accuracy should be enough for our applications.

After this I created the Java code for the preprocessing module for the Fault Recognition for Rotating Machines developed by me during summers. They are as listed below

1. HighPassFilter.java: A FIR filter was coded using the form of the filter.
2. Normalization.java:- This functions contained modifications over the normal max-min normalization by excluding some extreme points in calculation for max-min.
3. Featurespace.java:- this file takes in the input signal and returns a double array of size 18x1 containing the 18 features for that signal. These functions are :
    a. Root Mean Square

b. Zero Crossing Rate
c. Standard Deviation
d. Absolute Mean of Bins: Returns the absolute mean of bins of the input signal divided into 5 bins.
e. Time Centroid: Returns the time centroid of the input signal divided into 5 bins.
f. Spectral Centroid: Weighted mean of frequencies are returned after dividing signal into 5 bins in the frequency domain.

4. SegmentSignal.java – This function clips an incoming signal based on the ideal segment coordinates determined already.

The code for segmentsignal.java is under test, and will be completed by November end. Rest of the files are tested and verified by their MATLAB counterparts coded in June.

The work done in summers on the preprocessing module was presented in a paper titled 'Improvements in Preprocessing for Machine Fault Diagnosis' which is accepted in the upcoming conference ICIIS'13 to be held in December. This paper was written during the month of August as a part of the UGP.

Results

For determining the ideal segment of the vibration data, clustering was done using k-means algorithm whence the number of cluster with highest Global Silhouette came to be 6. In fig 1. We can the time domain representation of each cluster.

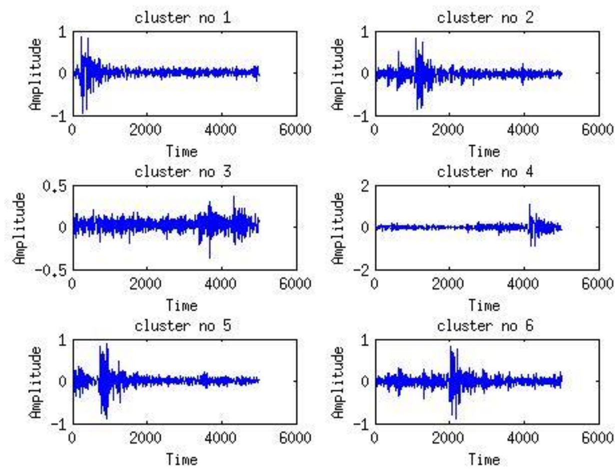

Fig 1. Time domain representation of various clutsters.

Out of this the cluster chosen as ideal segment was cluster 5 based on number of good features found using each of these clusters. Comparison of the normalization with vibration data done using classification accuracy to recognize the two states of the compressor healthy and LOV, following results were obtained.

Table 1: Comparison of classification accuracies between different normalizaitons

Process Used	Classification Accuracy
Max-Min Normalization	87 %
Modified Max-Min Normlization	92.2%

Based on this the modified max-min normalization was chosen.

Conclusion

Cluster Analysis for vibration data was performed and ideal segment was determined. Also the modified max-min normalization was verified for vibration data.

A Java library of time-frequency domain transform was developed. This library tested on i5 Processor have run time less than or equal to of the MATLAB run time. Preprocessing module is nearly developed for android platform.

Future Scope

There is still much work which could have been done, few ideas are listed below:-

1. The code currently does not take advantage of parallel architecture and thus can be modified to be able to run in parallel thus reducing the running time of the code.
2. Some functions were made in accordance to the need of the project and can be made more generalized.
3. Some other Time-Frequency transforms such as Signal Dependent TFD from cohen's class could be tried as a feature.

References

1. JTransform Library : jtransform.sourceforge.net
2. Stockwell, RG, L Mansinha, and RP Lowe (1996). Localization of the complex spectrum: the S transform, IEEE Transactions on Signal Processing 44 (4), p 998-1001.
3. Thomas M., Roshen J.,Lethakumary B.Comparison of WVD Based Time-Frequency Distributions. 2012

Source Codes

1. PseudoWignerVille.java

```
package time_frequency;

import edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D;

public class PseudoWignerVille {
	int length;				// Length of the input signal.
	int fs;					// Sampling frequency.
	int wlength;			//window_length

	public PseudoWignerVille(int length, int fs) {
		this.length = length;
		this.fs = fs;
		this.wlength = length/4 + 1 - (length%2);
	}
	/**
	 * Returns the gaussian window for the given frequency and time width.
	 * @param wlength Length of the window.
	 * @return Window elements.
	 */
	public double[] gauss_window(int wlength) {
		double[] gauss = new double[wlength];
		double constant = -5.29832;
		double step = 2.0/(wlength-1);
		for(int j = 0;j<wlength; j++) {
			gauss[j] = Math.exp(constant*(-1+j*step)*(j*step-1));
		}
		return gauss;
	}
	/**
	 * Returns the Pseudo Wigner-Ville for a real time signal.
	 * The difference between PWVD and WVD is the presence of a Window function.
	 * @param data
	 *			Input signal.
	 * @return
	 *			A array contanining the maxmimum values for each normalized frequency.
	 */
	private double[] pwvd(double[] data) {
		int taumax;
		double[] tfr = null;
		double[] result = new double[this.length];
		double[] h = gauss_window(this.wlength);
		//Normalizing the window
		double mid = h[h.length/2];
		for(int i =0;i<h.length;i++){
			h[i] /= mid;
		}
		int Lh = (this.wlength-1)/2;

		DoubleFFT_1D fft = new DoubleFFT_1D(length);
		for(int i=0;i<this.length;i++) {
			tfr = new double[2*this.length];
			taumax = Lh > i ? i : Lh;
			taumax = taumax > (this.length-i-1) ? (this.length-i-1) : taumax;
			if( taumax > this.length/2) { taumax = this.length/2;}
			for(int j=-taumax; j<0;j++) {
				tfr[2*(this.length+j)] = h[Lh+j]*data[i-j] * data[i+j];
```

```
			}
			for(int j=0; j<=taumax;j++) {
				tfr[2*j] = data[i-j]*data[i+j]*h[Lh+j];
			}
			fft.complexForward(tfr);

			// finding the maximum for each frequency.
			for(int k=0;k<this.length;k++) {
				if(tfr[2*k] > result[k]) result[k]  = tfr[2*k];
			}
		}
		return result;
	}
	/**
	 * The function for returning the extracted features.
	 * @param data
	 *			The time signal.
	 * @return
	 *			1 x 72 dimension feature values in double array.
	 */
	public double[] FV_pwvd(double[] data) {
		int bins = 72;
		double[] result = new double[bins];
		double[] energy = this.pwvd(data);
		int bin_length = energy.length/(bins*2);
		double sum = 0;
		for(int i=0;i<energy.length;i++) {
			sum += energy[i];
		}
		for(int i=1;i<=bins; i++) {
			for(int j = (i-1)*bin_length; j<i*bin_length;j++) {
				result[i-1] += energy[j];
			}
			result[i-1] /= sum;
		}
		return result;
	}
	}
```

2. STFT.java

```
1.	import org.apache.commons.math3.complex.Complex;
2.	import org.apache.commons.math3.transform.DftNormalization;
3.	import org.apache.commons.math3.transform.FastFourierTransformer;
4.	import org.apache.commons.math3.transform.TransformType;
5.
6.	public class STFT {
7.		/**
8.		 *
9.		 */
10.		private double[] h;
11.		private int length;
12.		private int wlength;
13.		public double[] result;
14.
15.		public STFT(int length) {
16.			this.length = length;
17.			this.wlength = length/4;
18.			this.wlength = this.wlength + 1 - (length%2);
19.			this.h = new double[this.wlength];
20.			this.h = create_window(this.wlength);
21.			this.result = new double[nextpow2(length)];
22.		}
```

```
23.         /**
24.          *
25.          * @param length
26.          * @return
27.          */
28.         private double[] create_window(int length){
29.                 double norm = 0;
30.                 double[] h = new double[length];
31.                 double factor = 2.0*Math.PI/length;
32.                 int half = (length-1)/2;
33.                 for(int i = -half;i<=half;i++){
34.                         h[i+half] = 0.3635819 + 0.4891775*Math.cos(i*factor) +
    0.1363995*Math.cos(2.0*i*factor) + 0.0106411*Math.cos(3.0*i*factor) ;
35.                         norm += h[i+half]*h[i+half];
36.                 }
37.                 norm = Math.sqrt(norm);
38.                 //Normalizing the window function
39.                 for(int i=0;i<length;i++){
40.                         h[i] /= norm;
41.                 }
42.                 return h;
43.         }
44.         /**
45.          *
46.          * @param data
47.          * @return
48.          */
49.         public double[] forward(double[] data){
50.                 Complex[] temp = new Complex[nextpow2(this.length)];
51.                 int n,taulower,tauupper,index;
52.                 Complex one = Complex.ONE;
53.                 int Lh = (this.wlength-1)/2;
54.                 double x;
55.                 FastFourierTransformer fft = new FastFourierTransformer(DftNormalization.STANDARD);
56.                 for(int m=0;m<this.length;m++){
57.                         taulower = Math.min(Lh,m);
58.                         tauupper = Math.min(Lh,this.length-1-m);
59.                         index = this.length-1;
60.                         for(int i=0;i<temp.length;i++) temp[i] = Complex.ZERO;
61.                         for(n=m-taulower;n<m;n++){
62.                                 temp[index+n-m] = one.multiply(data[n]*h[n-m+Lh]) ;
63.                         }
64.                         for(n=m;n<= m+tauupper;n++){
65.                                 temp[n-m] = one.multiply(data[n]*h[n-m+Lh]) ;
66.                         }
67.                         temp = fft.transform(temp,TransformType.FORWARD);
68.                         for(int i=0;i<temp.length;i++){
69.                                 x = temp[i].abs();
70.                                 if(x>this.result[i]){
71.                                         this.result[i] = x;
72.                                 }
73.                         }
74.                 }
75.                 return this.result;
76.         }
77.         /**
78.          *
79.          * @param data
80.          * @return
81.          */
82.         public double[] tfrstft(double[] data){
83.                 this.forward(data);
```

```
84.             int bin_length = this.result.length/72;
85.             double[] ans = new double[72];
86.             double total = 0;
87.             for(int i=1;i<73;i++){
88.                 for(int j = (i-1)*bin_length;j<i*bin_length;j++){
89.                     ans[i-1] += this.result[j];
90.                     total += this.result[j];
91.                 }
92.             }
93.             for(int j = 72*bin_length;j<this.result.length;j++) total += this.result[j];
94.             for(int i=0;i<72;i++) ans[i]/=total;
95.             System.out.printf("Total : %f\n", total);
96.             return ans;
97.         }
98.         /**
99.          *
100.         * @param x
101.         * @return
102.         */
103.        private int nextpow2(int x){
104.            int result = 1;
105.            while(result<x) result *=2;
106.            return result;
107.        }
108. }
```

3. Strans.java

```
public class STrans {

    private int length;
    private int sampling_rate;
    private int freqsamplingrate;
    private int minfreq,maxfreq;
    private double factor;

    public STrans(int length) {
        this.length = length;
        this.minfreq = 0;
        this.maxfreq = length / 2;
        this.sampling_rate = 1;
        this.freqsamplingrate = 1;
        this.factor = 1;
    }
    /**
     * Returns the gaussian window for the given frequency and time width.
     * @param wlength Length of the window.
     * @return Window elements.
     */
    public double[] gauss_window(int wlength,int freq,double factor) {
        double[] gauss = new double[wlength];
        double constant = -2*factor*Math.PI*Math.PI/(freq*freq);
        for (int i = 0;i < wlength; i++) {
            gauss[i] = Math.exp(i*i*constant) + Math.exp((wlength-i)*(wlength-i)*constant);
        }
        return gauss;
    }
    /**
     * The Stockwell transform function.
     * @param data the time series
```

```
	 * @return the maximum for each frequency.
	 */
	private double[] strans(double[] data) {
		int freq_range = this.maxfreq - this.minfreq;
		double[] result = new double[freq_range+1];
		double[] window = new double[this.length];
		double[] data_inverse = new double[2*this.length];

		//Calculating the Fourier Transform of the
		double[] data_fft = new double[2*this.length];
		for(int i=0;i<this.length;i++)data_fft[2*i]= data[i];
		DoubleFFT_1D fft = new DoubleFFT_1D(this.length);
		fft.complexForward(data_fft);

		//Starting Calculation of S matrix
		double max,mean = 0;
		for(int i=0;i<this.length;i++){ mean+=data[i];}
		result[0] = mean/this.length;
		// Start the computation for various frequencies values
		for(int f = this.freqsamplingrate; f <= freq_range; f+=this.freqsamplingrate){
			window = this.gauss_window(this.length,this.minfreq+f,this.factor);
			for(int j = 0;j < this.length;j++) {
				data_inverse[2*j] = data_fft[2*((j+f)%this.length)]*window[j];
				data_inverse[2*j+1] = data_fft[2*((j+f)%this.length)+1]*window[j];
			}
			fft.complexInverse(data_inverse, true);
			max = 0;
			for(int i = 0;i < this.length;i++) {
				double abs = Math.sqrt(data_inverse[2*i]*data_inverse[2*i] +
data_inverse[2*i+1]*data_inverse[2*i+1]);
				if(abs>max) max = abs;
			}
			result[f] = max;
		}
		return result;
	}
	/**
	 * Feature Extraction function for the Strans.
	 * @param data The timeseries.
	 * @return bin energies, after dividing into 72 bins.
	 */
	public double[] st(double[] data) {
		int freq_range = this.maxfreq - this.minfreq;
		int bin_length = 100;
		double[] result = new double[36];
		double sum = 0;
		double[] senergy = this.strans(data);
		for(int i=0;i<= freq_range;i++){
			sum += senergy[i];
		}
		for(int i = 1;i <= 36;i++) {
			result[i-1] = 0;
			for(int j =(i-1)*bin_length ;j < i*bin_length; j++) {
				result[i-1] += senergy[j];
			}
			result[i-1] /= sum;
		}
		return result;
	}
	}
```

4. WignerVille.java

```
public class WignerVille {
	int length;
	int sampling_frequency ;

	public WignerVille(int length,int fs) {
		this.length = length;
		this.sampling_frequency = fs;
	}
	/**
	 * Returns the Wigner-Ville for a real time signal.
	 * @param data
	 *		Input signal.
	 * @return
	 *		A array contanining the maxmimum values for each normalized frequency.
	 */
	private double[] wv(double[] data) {
		int taumax;
		double[] tfr = null;
		double[] result = new double[this.length];

		DoubleFFT_1D fft = new DoubleFFT_1D(length);
		for(int i=0;i<this.length;i++) {
			tfr = new double[2*this.length];
			taumax = i > (this.length-i-1) ? (this.length-i-1) : i;
			if( taumax > this.length/2) { taumax = this.length/2;}
			for(int j=-taumax; j<0;j++) {
				tfr[2*(this.length+j)] = data[i-j] * data[i+j];
			}
			for(int j=0; j<=taumax;j++) {
				tfr[2*j] = data[i-j]*data[i+j];
			}
			fft.complexForward(tfr);

			// finding the maximum for each frequency.
			for(int k=0;k<this.length;k++) {
				if(tfr[2*k] > result[k]) result[k]  = tfr[2*k];
			}
		}
		return result;
	}
	/**
	 * The function for returning the extracted features.
	 * @param data
	 *		The time signal.
	 * @return
	 *		1 x 72 dimension feature values in double array.
	 */
	public double[] FV_wvd(double[] data) {
		int bins = 72;
		double[] result = new double[bins];
		double[] energy = this.wv(data);
		int bin_length = energy.length/(bins*2);
		double sum = 0;
		for(int i=0;i<energy.length;i++) {
			sum += energy[i];
		}
		for(int i=1;i<=bins; i++) {
			for(int j = (i-1)*bin_length; j<i*bin_length;j++) {
				result[i] += energy[j];
			}
			result[i] /= sum;
		}
```

```
			return result;
		}
}
```

5. ChoiWilliamsDistribution.java

```
public class ChoiWilliamsDistribution {

	private int length;
	private int fs;
	public ChoiWilliamsDistribution(int length, int fs) {
		this.length = length;
		this.fs = fs;
	}
	/**
	 * Returns the window based on the for the given frequency and time width.
	 * @param wlength Length of the window.
	 * @param name the window function to be generate
	 * @return Window values at different values.
	 */
	public double[] generate_window(int wlength, String name) {
		double[] h = new double[wlength];
		if(name == "Gauss") {
			double constant = -5.29832;
			double step = 2.0/(wlength-1);
			for(int j = 0;j<wlength; j++) {
				h[j] = Math.exp(constant*(-1+j*step)*(j*step-1));
			}
		}
		else if(name == "Nuttal") {
			double factor = 2.0*Math.PI/wlength;
			int half = (wlength-1)/2;
			for(int i = -half;i<=half;i++){
				h[i+half] = 0.3635819 + 0.4891775*Math.cos(i*factor) +
		0.1363995*Math.cos(2.0*i*factor) + 0.0106411*Math.cos(3.0*i*factor) ;
			}
		}
		return h;
	}
	public double[] cwd(double[] data) {
		//Declaring constants
		int taumax,temp;
		int pointsmax,pointsmin;
		double sigma = 1.0;
		double tempd,R;
		double[] tfr = null;
		double[] result = new double[this.length];
		DoubleFFT_1D fft = new DoubleFFT_1D(length);

		//Generating Windows and kernel functions
		temp = this.length/4 + 1 - this.length % 2;
		double[] hw = generate_window(temp,"Nuttal");
		temp = this.length/10 + 1 - this.length % 2;
		double[] gw = generate_window(temp,"Nuttal");

		//Normalizing the gaussian window
		double mid = hw[hw.length/2];
		for(int i =0;i<hw.length;i++){
			hw[i] /= mid;
		}
		int Lg = (gw.length-1)/2;
		int Lh = (hw.length -1)/2;
```

```
		// Generating the Kernel function
		double normfac = 16.0*Math.PI/sigma;
		double spreadfac = 16.0/sigma;
		taumax = Math.min(this.length/2, Lh);
		double[][] CWDker = new double[gw.length][taumax];
		for(int i=-Lg;i<=Lg;i++) {
			for(int j= 1;j<=taumax;j++) {
				CWDker[i+Lg][j-1] = Math.exp(-1*i*i/(spreadfac*j*j))*gw[i+Lg];
			}
		}

		//Main computation for Choi Williams starts
		for (int i = 0;i < this.length;i++) {
			tfr = new double[2*this.length];
			taumax = Math.min(Lh , i + Lg  );
			taumax = Math.min(taumax , (this.length - i + Lg));
			taumax = Math.min(taumax,this.length/2);
			tfr[0] = data[i]*data[i];

			for(int tau = 1;tau<=taumax;tau++) {
				//pointsmin = pointsmax = Math.min(Lg, tau);
				pointsmin = pointsmax = Lg;
				pointsmax = Math.min(pointsmax, i-tau);
				pointsmin = Math.min(pointsmin, this.length-i-tau-1);

				tempd = 0;
				for(int j = Lg-pointsmin; j <= Lg+ pointsmax; j++ ) {
					tempd += CWDker[j][tau-1];
				}
				R = 0;
				for(int j=-pointsmin;j<=pointsmax;j++) {
					R += CWDker[Lg+j][tau-1]*data[i+tau-j]*data[i-tau-j];
				}
				tfr[2*(tau)] = hw[Lh + tau]*R/tempd;
				R = 0;
				for(int j=-pointsmin;j<=pointsmax;j++) {
					R += CWDker[Lg+j][tau-1]*data[i-tau-j]*data[i+tau-j];
				}
				tfr[2*(this.length-tau)] = hw[Lh-tau]*R/tempd;
			}
			fft.complexForward(tfr);
			for(int k=0;k<this.length;k++) {
				if(result[k] < tfr[2*k]) result[k] = tfr[2*k];
			}
		}
		return result;
	}
	/**
	* Calculates the feature values using CW distribution for extraction.
	 * @param data Time signal
	 * @return Extracted Feature Values.
	 */
	public double[] tfrcw(double[] data) {
		int bins = 36;
		double[] result = new double[bins];
		double[] energy = this.cwd(data);
		int bin_length = energy.length/(bins*2);
		double sum = 0;
		for(int i=0;i<energy.length;i++) {
			sum += energy[i];
		}
```

```
		for(int i=1;i<=bins; i++) {
			for(int j = (i-1)*bin_length; j<i*bin_length;j++) {
				result[i-1] += energy[j];
			}
			result[i-1] /= sum;
		}
		return result;
	}
}
```

6. BornJordan.java

```
public class BornJordan {
	/**
	 * Length of the signal.
	 */
	private int length;
	/**
	 * The sampling frequency of the signal.
	 */
	private int fs;
	/**
	 * Constructor for the class.
	 * @param length Length of the input signal.
	 * @param fs Sampling frequency of the signal.
	 */
	public BornJordan(int length,int fs) {
		this.length = length;
		this.fs = fs;
	}
	/**
	 * Returns the window based on the for the given frequency and time width.
	 * @param wlength Length of the window.
	 * @param name the window function to be generate
	 * @return Window values at different values.
	 */
	public double[] generate_window(int wlength, String name) {
		double[] h = new double[wlength];
		if(name == "Gauss") {
			double constant = -5.29832;
			double step = 2.0/(wlength-1);
			for(int j = 0;j<wlength; j++) {
				h[j] = Math.exp(constant*(-1+j*step)*(j*step-1));
			}
		}
		else if(name == "Nuttal") {
			double factor = 2.0*Math.PI/wlength;
			int half = (wlength-1)/2;
			for(int i = -half;i<=half;i++){
				h[i+half] = 0.3635819 + 0.4891775*Math.cos(i*factor) +
0.1363995*Math.cos(2.0*i*factor) + 0.0106411*Math.cos(3.0*i*factor) ;
			}
		}
		return h;
	}
	/**
	 * The function calculated Born Jordan Distribution for the given data.
	 * It returns the maximum value obtained for each frequency.
	 * @param data Time series input signal.
	 * @return A vector of length N containing max for each normalized frequencies.
	 */
	public double[] bj(double[] data) {
```

```
		int taumax,temp;
		int pointsmax,pointsmin;
		double tempd,R;
		double[] tfr = null;
		double[] result = new double[this.length];
		DoubleFFT_1D fft = new DoubleFFT_1D(length);

		// Generating the windows.
		temp = this.length/4 + 1 - this.length % 2;
		double[] gw = generate_window(temp,"Gauss");
		temp = this.length/10 + 1 - this.length % 2;
		double[] nw = generate_window(temp,"Nuttal");

		//Normalizing the gaussian window
		double mid = gw[gw.length/2];
		for(int i =0;i<gw.length;i++){
			gw[i] /= mid;
		}

		int Lg = (gw.length-1)/2;
		int Ln = (nw.length -1)/2;

		//Main computation for born jordan starts
		for (int i = 0;i < this.length;i++) {
			tfr = new double[2*this.length];
			taumax = Math.min(Lg , i + Ln  );
			taumax = Math.min(taumax , (this.length - i + Ln));
			taumax = Math.min(taumax,this.length/2);
			tfr[0] = data[i]*data[i];

			for(int tau = 1;tau<=taumax;tau++) {
				pointsmin = pointsmax = Math.min(Ln, tau);
				pointsmax = Math.min(pointsmax, i-tau);
				pointsmin = Math.min(pointsmin, this.length-i-tau-1);

				tempd = 0;
				for(int j = Ln-pointsmin; j <= Ln+ pointsmax; j++ ) {
					tempd += nw[j];
				}
				R = 0;
				for(int j=-pointsmin;j<=pointsmax;j++) {
					R += nw[Ln+j]*data[i+tau-j]*data[i-tau-j];
				}
				tfr[2*(tau)] = gw[Lg + tau]*R/tempd;
				R = 0;
				for(int j=-pointsmin;j<=pointsmax;j++) {
					R += nw[Ln+j]*data[i-tau-j]*data[i+tau-j];
				}
				//if(tau == 0) System.out.printf("%d :sum = %f\n", i,tempd);
				tfr[2*(this.length-tau)] = gw[Lg-tau]*R/tempd;
			}
			fft.complexForward(tfr);
			for(int k=0;k<this.length;k++) {
					//System.out.printf("%d : %f\n", k+1,tfr[2*k]);
					if(result[k] < tfr[2*k]) result[k] = tfr[2*k];
			}
		}
		return result;
	}
	/**
	 * Calculates the feature values using BJ distribution for extraction.
	 * @param data Time signal
```

```
 * @return
 */
	public double[] tfrbj(double[] data) {
		int bins = 36;
		double[] result = new double[bins];
		double[] energy = this.bj(data);
		int bin_length = energy.length/(bins*2);
		double sum = 0;
		for(int i=0;i<energy.length;i++) {
			sum += energy[i];
		}
		for(int i=1;i<=bins; i++) {
			for(int j = (i-1)*bin_length; j<i*bin_length;j++) {
				result[i-1] += energy[j];
			}
			result[i-1] /= sum;
		}
		return result;
	}
}
```

7. UpdatedMorletTransform.java

```
public class UpdatedMorletTransform {
	private double[] umorl = null;

	public UpdatedMorletTransform() {
		double a=16, b=0.02, b1=0.5, a1=0.9;
		umorl = new double[200];
		for(int i=0;i<umorl.length;i++) {
			umorl[i] = Math.exp(-1*b1*b1*(i+1-b)*(i+1-b)/(a*a));
			umorl[i] *= Math.pow(Math.sin(Math.PI*(i+1-b)/a),2);
		}
	}

	public double[] FV_umorl(double[] data) {
		double[] cdata = conv(data,this.umorl);
		double[] result = new double[5];
		result[0] = std(cdata);
		result[1] = entropy(cdata);
		result[2] = kurt(cdata) ;
		result[3] = skew(cdata);
		result[4] = Math.pow(result[0], 2);
		return result;
	}
	private double[] conv(double[] data, double[] morl) {
		// TODO Auto-generated method stub
		int m = data.length, n = morl.length;
		double[] w = new double[m + n - 1];
		for (int k = 1; k <= m + n - 1; k++) {
			int ji = Math.max(1, k + 1 - n), jf = Math.min(k, m);
			w[k - 1] = 0;
			for (int i = ji; i <= jf; i++) {
				w[k - 1] += data[i-1] * morl[k - i];
			}
		}
		return w;
	}
	/* Calculate Kurtosis of data */
	public static double kurt(double[] data) {
		double avg = mean(data);
		double numerator = 0, denominator = 0;
```

```
//   double stds=std(data);
                for (int i = 0; i < data.length; i++) {
                        numerator += Math.pow(data[i] - avg, 4);
                        denominator += Math.pow(data[i] - avg, 2);
                }
                numerator = numerator / data.length;
                denominator = Math.pow(denominator / data.length, 2);
                return numerator / denominator;
  }
  /* Calculate Skewness of data */
  public static double skew(double[] data)
  {
                double avg = mean(data);
                double numerator = 0, denominator = 0;

                for (int i = 0; i < data.length; i++) {
                        numerator += Math.pow(data[i] - avg, 3);
                        denominator += Math.pow(data[i] - avg, 2);
                }

                numerator = numerator / data.length;
                denominator = Math.pow(denominator / data.length, 3.0 / 2.0);
                return numerator / denominator;

  }
  /* Calculate Standard Deviation */
  private double std(double[] morc) {
                double mu = mean(morc);
                double sumsq = 0.0;
                for (int j = 0; j < morc.length; j++)
                        sumsq += (mu - morc[j]) * (mu - morc[j]);
                return Math.sqrt((sumsq) / (morc.length - 1));
  }
  private static double mean(double[] morc) {
                double mean = 0;
                for(int i=0;i<morc.length;i++) mean += morc[i];
                mean /= morc.length;
                return mean;
  }
  /* Calculating entropy of the data | E = -sum(p*log2(p)) */
  private double entropy(double[] morc) {
                double[] copmorc = new double[morc.length];
                for (int i = 0; i < morc.length; i++) {
                        copmorc[i] = (double) (Math.round(morc[i] * 1000000)) / 1000000.0;
                }
                Arrays.sort(copmorc);
                // Finding Frequency Distributions
                int len = 256;
                double width = (double) 1.0 / 255.0;
                int[] freq = new int[len];
                double cp = width;
                int start = 0, j = 0;
                for (int i = 0; i < freq.length; i++) {
                        for (j = start; j < copmorc.length; j++) {
                                if (Double.compare((double) copmorc[j], cp) <= 0) {
                                        freq[i]++;
                                } else {
                                        cp = cp + width;
                                        start = j;
                                        break;
                                }
                        }
```

```
                }
                freq[len-1] += copmorc.length-1 -j;
                // Finding probabilities
                double[] prob = new double[len];
                for (int n = 0; n < len; n++) {
                        prob[n] = (double)freq[n] / copmorc.length;
                }
                // Finally calculating the entropy
                double ent = 0.0;
                double b = Math.log(2.0);
                for (int t = 0; t < len; t++) {
                        if(prob[t]!=0)
                                ent += prob[t] * Math.log(prob[t]) /b;
                }
                return (-1 * ent);
        }
}
```