WHO SAID THAT?

New Mexico Adventures In Supercomputing Challenge

Final Report

April 7, 2004

Team 048

Manzano High School

Public Academy for the Performing Arts

Eldorado High School

Team Members:

Kellan Bethke

Sam Boling

William Laub

Teachers

Steven Schum, Manzano

David Dixon, Eldorado

EXECUTIVE SUMMARY

Our project was to separate mixed recordings of speech signals into the individual speakers. The method used to perform the separation is known of the Kurtosis Maximization Algorithm (KMA). It is based on an empirically observed property of speech signals. This property is that the kurtosis of individual speech signals is almost always higher that the kurtosis of a mixed speech signal. Kurtosis is a statistical measure of the “peakedness” of a distribution of data. During the project two C++ class were developed: a WAV class to handle .WAV formatted sound files and a distribution class to handle the calculation of statistical properties of data distributions.

The project was successful in separating two mixtures of two speakers into two individual speech signals. We found that the KMA is greatly affected by initial conditions, and that varying parameters of the algorithm also greatly affects its performance.

The range of uses for this type of program is wide: from improving hearing aid performance to keeping records of negotiations or heated debates, to intelligence gathering.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

This project report, “Who Said That?”, was written by Kellan Bethke, Sam Boling, and William Laub, Team 48 in the 2003-2004 Adventures in Supercomputing Challenge. We come from Manzano High School, the Public Academy for the Performing Arts, and El Dorado High School, and in our project we are looking into the problem of blind speech separation. Blind speech separation refers to the separation of individual sound sources from a mixed recording of multiple speakers, when nothing is known about the room, the positions of the speakers in relation to the microphones, the balance of the voices in the mixtures, or other aspects of the recordings that might affect separation. In our project, we are limiting the problem to two different mixtures of sound from two different sources. Our goal is to separate the mixtures into sound files resembling the original sound sources.

We elected to attempt to solve this problem for several reasons. Firstly, it sounded interesting and unique when it was mentioned during brainstorming. Also, we knew that programs that were capable of separating instrumental pieces from a musical score were available commercially, and we believed that no similar program existed to separate speech. We decided that it would be interesting to see if we could develop a program that would do this.

Another reason we chose this project was the wide array of possible implementations. Over the course of our work, we found that the project could be used for many things, including intelligence gathering, background noise elimination in hearing aids, and analysis of fast-paced negotiations or debates. However, along with all of its uses, the project has many difficulties. Ten years ago, this problem was thought impossible to solve. Also, we are using the algorithm known as KMA (Kurtosis Maximization Algorithm) for this project, which, in this implementation, has limitations that restrict the environment in which the sounds can be recorded. However, this algorithm is simpler than a more complex implementation would be, and we selected it over others to avoid working for another year in the Challenge. We believe this implementation to be completed and functioning adequately for our needs.

THEORY

Kurtosis is the statistical measure of the “peakedness” of a distribution of data. Kurtosis is defined as $E[x^4]/E[x^2]^2$ when measuring a data sample. In $E[x^4]$ the “E” refers to the expectation operator, and x is a distribution of data, and therefore, $E[x]$ is the average of all x samples.

In our project, kurtosis is a critically important measure. Since a WAV file is a distribution of data it also has a kurtosis measurement. Furthermore, it has been empirically shown that the kurtosis of an individual speaker is higher than the kurtosis of a mixture of speakers in most cases. Thus, if the algorithm causes the kurtosis of the resulting files to rise, the speakers will theoretically separate the mixtures and the process will be complete except for writing the separated sounds into new files.

The algorithm we intend to use to solve our problem is called the Kurtosis Maximization Algorithm or KMA. The KMA was developed by Professor Phillip De Leon of the Klipsch School of Electrical Engineering at New Mexico State University in Las Cruces. The KMA uses a gradient ascent method to maximize the kurtosis of separated signals. First of all, this algorithm does not take echoes or reverberation into account. If echoes are present, the algorithm may perform poorly. Also, the algorithm will fail if the Kurtosis of the mixture is higher than the kurtosis of the individual speakers. There may be problems if the Kurtosis of a separated speaker is higher than the speaker was originally.

Fig. 1 illustrates the problem we are solving. In Figure 1, the individual speakers are represented by s_1 and s_2. The sounds from these speakers are picked up by the two microphones and recorded as mixtures x_1 and x_2. The mixtures x_1 x_2 are then separated by the process box into new signals y_1 and y_2. y_1 and y_2 should be more or less equivalent to s_1 and s_2.

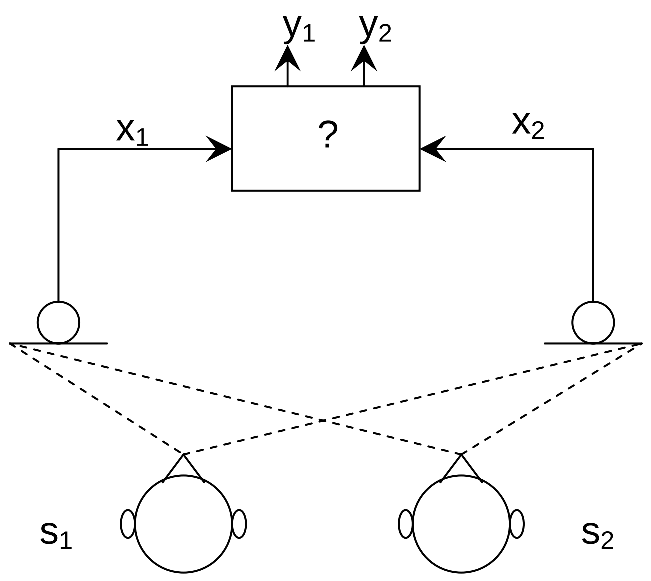


Fig. 1 Problem Illustration

The mixing and separating process can be represented by Fig. 2, where A and W are the mixing and separation matrices, respectively.

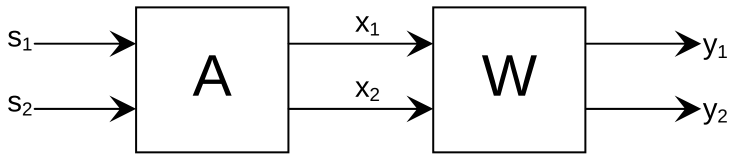


Fig. 2 Mixing and Separation

Since we only have the mixtures x1 and x2, we will be working on the right side of Fig. 2. This can be written mathematically as

$$[y] = [W][x]$$

where W is the separation matrix. From this, we see that

$$y_1 = W_{11}x_1 + W_{12}x_2 \text{ and that } y_2 = W_{21}x_1 + W_{22}x_2$$

Since the kurtosis of y is $E[y^4]/E[y^2]^2$, (where E is the expectation operator), we have an equation for the kurtosis of y_1 and y_2. The mathematics of the method from this point on is beyond our mathematic level. The result of using the gradient ascent method to maximize kurtosis is the following set of equations. (Ma, 6)

$$y(n)=W(n)x(n)$$

$$\sigma_i^2(n) = \lambda_2\sigma_i^2(n-1) + (1-\lambda_2)x_i^2(n)$$

$$r_{12}(n) = \lambda_2 r_{12}(n-1) + (1-\lambda_2)x_1(n)x_2(n)$$

$$\alpha_i = 4y_i^3(n)$$

$$\beta_i = -W_{i1}(n)r_{12}(n)x_1(n) - W_{i2}(n)\sigma_2^2(n)x_1(n) + W_{i1}(n)\sigma_1^2(n)x_2(n) + W_{i2}(n)r_{12}(n)x_2(n)$$

$$\gamma_i = [W_{i1}^2(n)\sigma_1^2(n) + 2W_{i1}(n)W_{i2}(n)r_{12}(n) + W_{i2}^2(n)\sigma_2^2(n)]^{-3}$$

$$C(n) = \begin{bmatrix} -\alpha_1\beta_1\gamma_1 w_{12}(n) & \alpha_1\beta_1\gamma_1 w_{11}(n) \\ -\alpha_2\beta_2\gamma_2 w_{22}(n) & \alpha_2\beta_2\gamma_2 w_{21}(n) \end{bmatrix}$$

$$\mathbf{W}(n+1) = \mathbf{W}(n) + \frac{\tilde{\mu}}{\|\mathbf{C}(n)\|_2^2}\mathbf{C}(n)$$

In the above equations, n represents the nth piece of data in the sound data distribution. The statistical measures are estimates based on data that is updated as it comes in, and λ is a “forgetfulness” factor. The forgetfulness factor allows the statistical measures to change as new data is received without being too heavily affected by previous data. In this way, the algorithm can adapt to changes in the mixtures of the recording, such as the speakers moving in relation to the microphones.

The factor μ is a step size and $||C||22$ is the correction matrix norm. μ and λ are factors that must be tuned for good performance. In addition, the initial guess for [W] also affects performance.

Normalization is an important component in the algorithm we are using. We must normalize both the sound data to separate the speech sources. For the sound data, normalization involves making all the amplitudes of the waveform between -1 and +1. For the algorithm, normalization means making the step size of the corrections to the separation matrix a reasonable size, so it doesn’t overshoot the correct value because the step is too large, and it doesn’t take steps that are too small to get anywhere in a reasonable timeframe.

In order to normalize the algorithm, we use divide the correction matrix by the largest absolute value of any single item in the correction matrix. In order to normalize the WAV file we divide all the data in the file by the single largest absolute value in the file.

THE PROGRAM

Our project's goal was to create a program or programs capable of separating two files containing mixed speech signals into two files containing individual speech signals. In order to do this, we wrote a "WAV" class, a "distribution" class, the main program, some auxiliary functions, and an auxiliary program, called Mix. These files are each important to the program and have various uses.

The program that we wrote in our project is able to use and manipulate .WAV files, mix them together, and separate mixed files. One executable file takes two .WAV files and mixes them, and another takes two .WAV files and separates them into two individual speech signals. It does not take very long to run this program, usually approximately 5 seconds for either executable, after which the files are written out and can be played outside of the program.

We wrote a "WAV" class for this program, which is a file in the C++ programming language that allows for data to be stored and recognized as a .WAV file within the program, and for pieces of data involved in that file to be called and manipulated. This is used to read .WAV files for the program. The class identifies the unique properties of the .WAV file, tells the program where to find them, and allows the program to change those properties. The properties we included in this class are organized into "private" and "public" properties. Private properties can be accessed only by the WAV class itself. Public functions can be accessed by any program using the WAV class.

Fig 3A: Properties of the WAV class:

`TotalLengthOfPackage`	The length of the entire .WAV file in bytes.
`NumberOfChannels`	shows whether the .WAV file is mono or stereo.
`SampleRate`	The rate that samples are played when the .WAV file is played back.
`BytesPerSecond`	The number of bytes that are played per second.
`BytesPerSample`	How many bytes are in each sample.
`BitsPerSample`	Equal to 8 times `BytesPerSample.`
`LengthOfData`	The length of the data section in bytes.
`*ptrData`	Points to the location where the data begins.
`riffStr`	The string of characters "RIFF" that appears in every .WAV file.
`fmtStr`	The string of characters "fmt " that appears in every .WAV file.
`wavStr`	The string of characters "WAVE" that appears in every .WAV file.
`dataStr`	The string of characters "data" that appears in every .WAV file.
`fmtLength`	The length of he format block.
`audioFmt`	The format information.
`NumberOfSamples`	The number of samples.
`BytesPerChannel`	The number of bytes that are used for each channel.
`NumberOfData`	The number of data members.

Fig. 3B: Public Functions of the WAV class:

```
unsigned long getLengthOfPackage() const
unsigned short getNumberOfChannels() const
unsigned long getSampleRate() const
unsigned long getBytesPerSecond() const
unsigned short getBytesPerSample() const
unsigned short getBitsPerSample() const
unsigned long getLengthOfData() const
unsigned long getNumberOfData() const
short* getData() const
bool read(string nameOfFile)
void setLengthOfPackage(unsigned long)
void setNumberOfChannels(unsigned short)
void setSampleRate(unsigned long)
void setBytesPerSecond(unsigned long)
void setBytesPerSample(unsigned short)
void setBitsPerSample(unsigned short)
void setLengthOfData(unsigned long)
bool setData(short*,long)
bool add(const WAV&, double factor = 1.0)
bool subtract(const WAV&, double factor = 1.0)
```

Please note that this class is only capable of handling cononical .WAV files.

We also wrote a "distribution" class to handle large groups of data. It is used when we need to look at the data as a whole, or various properties, such as average and kurtosis, that apply to the entire group. Its properties and functions are as follows.

Fig. 4A : Private Properties and Functions of the distribution class:

`meanVal`	The mean of the data.
`meanVal2`	The mean of all data members squared.
`meanVal3`	The mean of all data members cubed.
`meanVal4`	The mean of all data members to the fourth power.
`varyVal`	The variance of the data.
`skewVal`	The skewness of the data.
`kurtVal`	The kurtosis of the data.
`sizeVal`	The number of data members.
`dataVal`	A pointer to the array of data.
`statCall`	A boolean value that says whether or not stats() has been called.
`maxVal`	The highest value in the data.
`Norm ()`	A boolean value that says whether or not the data has been normalized.
`bool stats ()`	This function calculates all properties except for maxVal.
`bool maxCalc ()`	This function calculates maxVal.

Fig. 4B : Public Functions of the distribution class:

```
double mean ()         |
double mean2 ()        |
double mean3 ()        |
double mean4 ()        |---  These return their various private properties.
double variance ()     |
double skew ()         |
double kurtosis ()     |
long size ()          _|
double* getData ()
bool setData (const double*, const long)
bool normalize (double)
bool setSize (long)
bool maximum ()
bool zeroMean ()
bool trim (long)
bool userSet (double, long)
bool getItem(long)

distribution ()
distribution (const double*, const long)
```

Our main program uses these two classes to manipulate the data in .WAV files such that they are mixed or separated. The algorithm iteratively corrects the inverse matrix to the mixing matrix, then uses that matrix to separate sounds. After sound separation, the results are inserted into new WAV objects and written to disk.

We also wrote several auxiliary functions that are used within the program.

`double* STF(const short* Unconv, long Length)` Converts an array of short to doubles.

`short* FTS(const double* Unconv, long Length)` Converts an array of doubles to shorts.

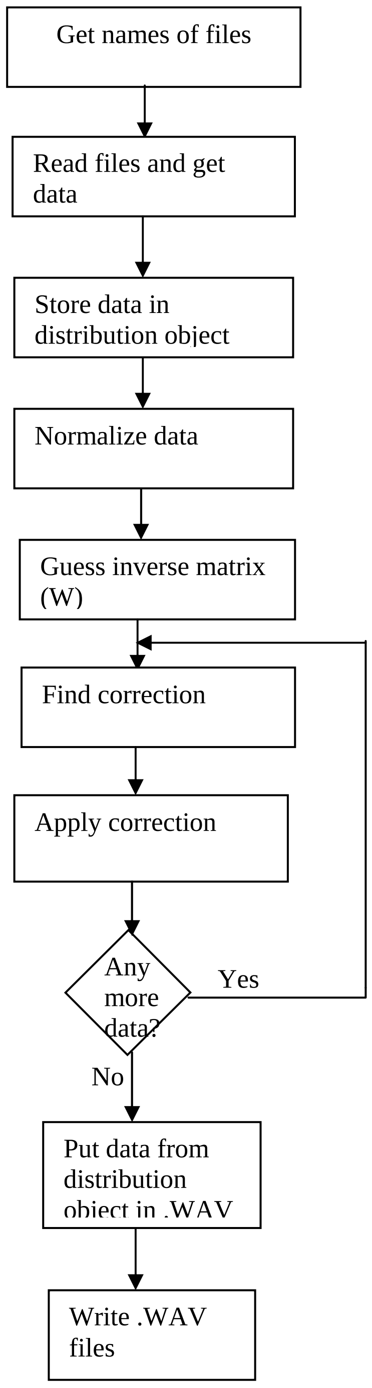
`double* madd(double* W, double* C, short n)` Adds two matrices together.

`double norm(double* Mat, short n)` Finds the norm of the matrix.

`bool mscale (double* M, double s, short n)` Performs scalar multiplication on a matrix.

Finally, we wrote an auxiliary program, called Mix, to mix two .WAV files together. It reads in the two WAV files, then it adds them together in the desired proportions. Together, these components form the project. The entire source listing for all the project components may be found in the appendix.

Fig. 5 Flowchart



RESULTS

We used two pairs of source mixtures to test our program. The first pair of mixtures were supplied by Professor De Leon. These mixtures were called x1 and x2. We also recorded two sources and mixed them on our own. The mixtures we made were called xx1 and xx2. The separated sounds from x1 and x2 were called y1 and y2, and the separated sounds from our own mixtures were called yy1, and yy2. The original sources that we mixed to create xx1 and xx2 were called s1 and w1. As you can see from Table 1 below, we were able to successfully maximize the kurtosis and mix different sounds..

After about three seconds most of y2 had been separated from y1. At about nine seconds, there were no traces of y2 left in y1. y2 was separated slightly faster than y1, and after seven seconds, y1 had been completely separated from y2. By using y1 and y2 as mixtures and running the program on them two more times, the three second separation in y1 was reduced to about one second.

The parameters we used in the algorithm were taken from Ma (p. 9-11). We tried varying these pages (μ, λ, initial [W]) but found that the values in Ma provided optimal performance.

Table 1: Mixture and separation Kurtosis

File Name	Kurtosis
x1.wav	14.555
x2.wav	20.0745
y1.wav	32.6707
y2.wav	26.4053
s1.wav	15.4127
w1.wav	10.5393
xx1.wav	9.2573
xx2.wav	8.19572
yy1.wav	11.8801
yy2.wav	26.3232

CONCLUSION

There is some more work that needs to be done on the program. Some mixtures either began to lose separation or didn't separate at all, while others separated perfectly. Several attempts to separate xx1 and yy1 failed even though the kurtosis was maximized. When s1 and w1 were mixed .4:1 and 1:.4, the separation was very successful, but when the separation was 1:1 and 1:1, the separation failed completely. We have also found that repeating separations on separated sounds can increase the separation or, in one case, heavily distort the separated sounds so that the kurtosis approaches three (a Gaussian distribution).

We found that the choice of algorithm parameters (μ, λ, and initial [W]) greatly affect performance

The course of action we took in our project was by no means the only one available. First of all, in our project, we created a program that would separate sources in real-time, or, in other words, we separate the sources as the data is read. We actually began by attempting to separate the sources for the whole recording all at one time, but in a test of the program, which supposedly had no errors or warnings and should have worked brilliantly, we found that it did not work, and we have not yet determined why.

Furthermore, our intention in the project was to be able to separate as many sources as we have microphones, however, we were not able to extend the numbers beyond two speakers and two microphones because of time constraints.

Also, our program cannot handle echoes well and reverberation in recordings, and we would have liked to have made our speech separation algorithm capable of handling echoes and reverberation, for it would make the algorithm far more effective and more suited for practical use. We did not attempt to expand on the algorithm in such a way that it could handle echoes and reverberation because it would have required a much, much more advanced algorithm which none of us are equipped to program, or even understand well, and again, time cut us short.

Finally, our program requires the number of speakers and the number of microphones to be congruous, and it would have been spectacular if we could have devised a program that could separate speakers when there are fewer microphones than speakers. This would give the program limitless use and the project could be no more successful than that.

ACKNOWLEDGEMENTS

We owe many thanks to several individuals who have been involved with our project and who have given us immeasurable amounts of help. It is safe to say that, without these individuals we would be nowhere near where we are now. The first is Mr. Tom Laub, our coach and supervisor. He has given us a vast amount of aid in the project, he has helped cart us from place to place, he aids us in staying on task, he aids us in the programming when we are having difficulty, and he has given countless hours of his time for the benefit of the project. Due to all of this we would like to extend a special thanks to Mr. Tom Laub.

Another individual who has greatly helped and influenced our project is Dr. DeLeon of the Klipsch School of Electrical Engineering, a department of New Mexico State University in Las Cruces, New Mexico. Dr. De Leon devoted an entire afternoon of his time to meet with us and share his progress, successes, and discoveries, show us his algorithms and much of his other work, and teach us the functionality of the algorithms, how they work, and why they work. Furthermore, he, for quite some time now, has been sending us papers, theses, and other documents relevant and helpful to our project. We located Dr. De Leon in a web search for information and possible methods for our project. We are all extremely thankful for all of his help, and we are also thankful that he teaches in Las Cruces rather than in New York or another distant place.

We would also like to offer thanks to the interim presentation evaluators who evaluated our presentation, especially Mr. Michael Trahan who also evaluated our written interim. We greatly appreciate their time that they donated to us and the improvement of our group dynamics and our ability to show our knowledge of our project. We realize that there is a price on their time and are truly thankful for the time they spent on us.

REFERENCES

1. De Leon, Philip, “Speech Separation by Kurtosis Maximization”, New Mexico State University, January 25th, 2004
2. Ma, Yunsheng, “Generalization, Extension, and Enhancement of the Kurtosis Maximization Algorithm for Co-Channel Speech Separation”, New Mexico State University, December 2000
3. Weber, Timothy John, “The Wav File Format” (htttp://www.lightlink.com/tjweber/stripwav/wav.html)
4. Dale, Nell, Weems, Chip, Headington, Mark, Programming and Problem Solving with C++ 3rd edition, Jones and Bartlett publishers, 2002
5. Dale, Nell, A Laboratory Course in C++, 3rd edition, Jones and Bartlett publishers, 2002

APPENDIX

Program Listing

Main Program:

```
#include<iostream>
#include "wav.h"
#include <fstream>
#include <string>
#include "datadist.h"
#include <cmath>
#include <cstdlib>
#include <iomanip>

using namespace std;

double* STF(const short*, long);
short* FTS(const double*, long);
double* correct(distribution*, distribution*, short);
bool mscale (double*, double, short);
double norm(double* Mat, short n);
bool madd(double* W, double* C, short n);
double* winit(short);

int main()
{

//whitespace is futile
  cout << "WHitE5P4cE 15 PhUt1l3 " << endl << endl;


//get input

  short ns = 2;
  string* filen = new string[ns];
  for (short i =0; i<ns; ++i)
  {
  cout << "Input mixture " << i+1 << " file name: ";
  cin >> filen[i];
  cout << endl;
  }

//read mixture matrix

  WAV* mix = new WAV[ns];
  for (i = 0; i<ns; ++i)
  {
    if (!mix[i].read(filen[i]))
    {
      cout << "Read failure on " << filen[i] << endl;
      return 42;
    }
  }

//printing header information

  for (i = 0; i<ns; ++i)
  {
    cout << endl << "Header info for mixture# " << i+1 << endl;
    mix[i].print();
  }

//filling distributions

  distribution* x = new distribution[ns];

  for (i=0; i<ns; ++i)
  {
    if ( !x[i].setData(STF(mix[i].getData(), mix[i].getNumberOfData() ),
mix[i].getNumberOfData() ) )
    {
      cout << "Failed to fill distribuion " << i << "with data from " <<
filen[i] << endl;
      return 42;
    }
  }

//x information print

  for (i = 0; i<ns; ++i)
  {
    cout << endl << "Stats of distribution x[" << i+1 << "]:" << endl;
    cout << "         Size: " << x[i].size() << endl;
    cout << "         Mean: " << x[i].mean() << endl;
    cout << "     Kurtosis: " << x[i].kurtosis() << endl;
    cout << "     Variance: " << x[i].variance() << endl;
    cout << "         Skew: " << x[i].skew() << endl;
    cout << "Maximum Value: " << x[i].maximum() << endl;
  }

//triming x's

  long slod = pow(2,31)-1;
  cout << "slod starting value: " << slod << endl;
  for (i = 0 ; i<ns ; ++i)
    if (x[i].size() < slod) slod = x[i].size();

  for (i = 0; i<ns; ++i)
    if(!x[i].trim(slod))
    {
      cout << "Unable to trim distribution " << i << endl;
      return 42;
    }

//trim x information print

  cout << "Maximum Value Found: " << slod << endl;
  for (i = 0; i<ns; ++i)
  {
    cout << endl << "Stats of distribution x[" << i+1 << "] after trimming:"
<< endl;
    cout << "         Size: " << x[i].size() << endl;
    cout << "         Mean: " << x[i].mean() << endl;
    cout << "     Kurtosis: " << x[i].kurtosis() << endl;
    cout << "     Variance: " << x[i].variance() << endl;
    cout << "         Skew: " << x[i].skew() << endl;
    cout << "Maximum Value: " << x[i].maximum() << endl;
  }

//zero mean and normalize

  for (i = 0; i<ns; ++i)
  {
    if ( !x[i].zeroMean() )
    {
      cout << "Failed to zero mean of distribution " << i << endl;
      return 42;
    }
    if ( !x[i].normalize() )
    {
      cout << "Failed to normalize distribution " << i << endl;
      return 42;
    }
  }

//zero and norm x information print

  for (i = 0; i<ns; ++i)
  {
    cout << endl << "Stats of distribution x[" << i+1 << "] after zeroing and
normalizing:" << endl;
    cout << "         Size: " << x[i].size() << endl;
    cout << "         Mean: " << x[i].mean() << endl;
    cout << "     Kurtosis: " << x[i].kurtosis() << endl;
    cout << "     Variance: " << x[i].variance() << endl;
    cout << "         Skew: " << x[i].skew() << endl;
    cout << "Maximum Value: " << x[i].maximum() << endl;
  }

//declaring separation matrix and initialize

  double* W = new double [4];
  W[0] = 1;
  W[1] = 0.1;
  W[2] = 0.1;
  W[3] = -1;

//print first W

  cout << endl << "First W:" << endl;
  for (i = 0; i<ns; ++i)
  {
    for (short j = 0; j<ns; ++j)
    {
      cout << setw(10) << W[i*ns+j] << " ";
```

```
        }
        cout << endl << endl;
      }

      double x1, x2;
      double w11, w12, w21, w22;
      double r12 = 0.01;
      double alpha1, alpha2, beta1, beta2, gamma1, gamma2;
      double sigma2_x1 = 0.01;
      double sigma2_x2 = 0.01;
      double lambda2 = 0.99995;
      double alambda2 = 1 - lambda2;
      double* C = new double [ns*ns];
      double  Cnorm = 0.0;
      double Mu = 0.00005;
      double y1,y2;
      distribution* y = new distribution[ns];
      for (i = 0; i < ns; ++i)
      {
          y[i].zeroIt(slod);
      }

      for (long wicky=0; wicky<slod; ++wicky)
      {
          x1 = x[0].getItem(wicky);
          x2 = x[1].getItem(wicky);

          w11 = W[0];
          w12 = W[1];
          w21 = W[2];
          w22 = W[3];

          y1 = (w11 * x1) + (w12 * x2);
          y[0].userSet(y1, wicky);
          y2 = (w21 * x1) + (w22 * x2);
          y[1].userSet(y2, wicky);

          sigma2_x1 = lambda2 * sigma2_x1 + alambda2 * x1 * x1;
          sigma2_x2 = lambda2 * sigma2_x2 + alambda2 * x2 * x2;
          r12 = lambda2 * r12 + alambda2 * x1 * x2;

          alpha1 = pow (y1, 3);
          alpha2 = pow (y2, 3);

          beta1 = (-x1*w11*r12) - (x1*w12*sigma2_x2) + (x2*w11*sigma2_x1) +
(x2*w12*r12);
          beta2 = (-x1*w21*r12) - (x1*w22*sigma2_x2) + (x2*w21*sigma2_x1) +
(x2*w22*r12);

          gamma1 = pow (w11*w11*sigma2_x1 + 2.0*w11*w12*r12 + w12*w12*sigma2_x2,
-3);
          gamma2 = pow (w21*w21*sigma2_x1 + 2.0*w21*w22*r12 + w22*w22*sigma2_x2,
-3);

          C[0] = -alpha1 * beta1 * gamma1 * w12;
          C[1] =  alpha1 * beta1 * gamma1 * w11;
          C[2] = -alpha2 * beta2 * gamma2 * w22;
          C[3] =  alpha2 * beta2 * gamma2 * w21;

// Get norm of C

          Cnorm = norm(C, ns);

// Scale C by Mu/Cnorm

          mscale (C, Mu / Cnorm, ns);

// Add scaled C to W

          madd (W, C, ns);

      }

//print final W

      cout << endl << "Final W:" << endl;
      for ( i = 0; i<ns; ++i)
      {
        for (short j = 0; j<ns; ++j)
        {
          cout << setw(10) << W[i*ns+j] << " ";
        }
        cout << endl << endl;
      }

// Print out kurtosis of separated sounds

      for ( i=0; i<ns; ++i)
      {
        cout << endl << " Statistics for y[" << i << "]" << endl;
        cout << "    Mean = " << y[i].mean() << endl;
        cout << "Kurtosis = " << y[i].kurtosis() << endl;
        cout << " Maximum = " << y[i].maximum() << endl;
      }
      cout << endl;

//denormalize

      for (i = 0; i<ns; ++i)
      {
        if (!y[i].normalize(32767.0))
        {
          cout << "Failed to denormalize distribution " << i << endl;
          return 42;
        }
      }

// Print out kurtosis of separated sounds after denormalization

      for ( i=0; i<ns; ++i)
      {
        cout << endl << " Statistics for y[" << i << "] after denormalizing" <<
endl;
        cout << "    Mean = " << y[i].mean() << endl;
        cout << "Kurtosis = " << y[i].kurtosis() << endl;
        cout << " Maximum = " << y[i].maximum() << endl;
      }
      cout << endl;

    //new WAV data

      WAV* nwav = new WAV[ns];
      double* tempd;
      short* temps;
      string newname;
      char* a = new char[2];
      string aa;

    //build wav files

      for (i = 0; i<ns; ++i)
      {

        nwav[i].setaudioFmt(1);
        nwav[i].setNumberOfChannels(1);
        nwav[i].setBitsPerSample(mix[i].getBitsPerSample());
        nwav[i].setBytesPerSample(mix[i].getBytesPerSample());
        nwav[i].setSampleRate(mix[i].getSampleRate());

    nwav[i].setBytesPerSecond(nwav[i].getBytesPerSample()*nwav[i].getSampleRate()
    );
        nwav[i].setfmtLength(mix[i].getfmtLength());
        nwav[i].setLengthOfData(y[i].size()*mix[i].getBytesPerSample());
        nwav[i].setLengthOfPackage(nwav[i].getLengthOfData()+36);
        tempd = y[i].getData();
        temps = FTS(tempd,y[i].size());
        nwav[i].setData(temps,y[i].size());
        itoa((i+1), a, 10);
        aa.assign(a);
        newname = "yy" + aa + ".wav";
        if (!nwav[i].write(newname))
        {
          cout << "unable to write y" << i <<".wav"<< endl;
          return 42;
        }
        nwav[i].print();
        delete[] tempd;
        delete[] temps;


      }

      return 0;
    }
```

Distribution Class Header:

```
#include <iostream>

using namespace std;

class distribution
{

private:

      double meanVal, meanVal2, meanVal3, meanVal4;
      double varyVal;
      double skewVal;
      double kurtVal;
      long sizeVal;
      double* dataVal;
      bool statCall;
      double maxVal;
      bool stats ();
      bool maxCalc ();

public:
      double mean ();
      double mean2 ();
      double mean3 ();
      double mean4 ();
      double variance ();
      double skew ();
      double kurtosis ();
      long size ();
      double* getData ();
      bool setData (const double*, const long);
      bool normalize (double factor = 1.0);
      bool setSize (long);
      bool denormalize (double);
      double maximum ();
      bool zeroMean ();
      bool trim (long);
      bool userSet (double, long);
      double getItem(long);
  bool zeroIt(long);
  bool scale (double);
  bool add (distribution&);
  bool copy (distribution&);

      distribution ();
      distribution (const double*, const long);
  ~distribution();
};
```

Distribution Class Implementation:

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include "datadist.h"

using namespace std;


distribution::distribution () /*default constructor*/
{
    meanVal = 0.0;    //initiate values to 0
    varyVal = 0.0;
    skewVal = 0.0;
    kurtVal = 0.0;
    maxVal  = 0.0;
    sizeVal = 0;
    dataVal = 0;
    statCall = false;
}


distribution::distribution (const double * dist, const long distSize) //other
constructor
{
    long i;
    sizeVal = distSize;
    dataVal = new double[sizeVal];
    for (i = 0; i < sizeVal; i++)
    {
        dataVal[i] = dist[i];
    }
    stats ();
    maxCalc ();
}

distribution::~distribution()
{
  if (dataVal) delete[] dataVal;
}


bool distribution::copy (distribution& indist)
{
  if (dataVal) delete[] dataVal;
  dataVal = indist.getData();
  sizeVal = indist.size();
  stats();
  maxCalc();
  return true;
}


bool distribution::maxCalc ()
{
    for (long i = 0; i < sizeVal; ++i)
    {
        if (fabs(dataVal[i]) > maxVal) maxVal = fabs(dataVal[i]);
    }
    return true;
}

bool distribution::stats ()
{
    long i;
    long n = sizeVal;
    double sum = 0.0;
    for (i = 0; i < sizeVal; ++i)
    {
        sum += dataVal[i];
    }
    maxCalc ();
    meanVal = sum / n;
    sum = 0.0;
    double sum1 = 0.0;
    double sum2 = 0.0;
    double sum3 = 0.0;
    double diff = 0.0;
    double origDiff = 0.0;
    double chgData = 0.0;
    meanVal2 = 0.0;
    meanVal3 = 0.0;
    meanVal4 = 0.0;
    for (i = 0; i < sizeVal; ++i)
    {
        chgData = dataVal[i];
        chgData *= chgData;
        meanVal2 += chgData;
        chgData *= dataVal[i];
        meanVal3 += chgData;
        chgData *= dataVal[i];
        meanVal4 += chgData;
        diff = dataVal[i] - meanVal;
        origDiff = diff;
        sum += diff;
        diff *= diff;
        sum1 += diff;
        diff *= origDiff;
        sum2 += diff;
        diff *= origDiff;
        sum3 += diff;
    }
    meanVal2 /= n;
    meanVal3 /= n;
    meanVal4 /= n;
    varyVal = (sum1 - sum * sum) / (n-1);
    skewVal = kurtVal = 0.0;
    if (varyVal > 0.0)
    {
    skewVal = (sum2 / (n * varyVal * sqrt (varyVal)));
    kurtVal = (sum3 / (n * varyVal * varyVal));
    }
    statCall = true;
    return true;
}
double distribution::mean ()
{
    if (!statCall) stats ();
    return meanVal;
}
double distribution::mean2 ()
{
    if (!statCall) stats ();
    return meanVal2;
}
double distribution::mean3 ()
{
    if (!statCall) stats ();
    return meanVal3;
}
double distribution::mean4 ()
{
    if (!statCall) stats ();
    return meanVal4;
}
double distribution::variance ()
{
    if (!statCall) stats ();
    return varyVal;
}
double distribution::skew ()
{
    if (!statCall) stats ();
    return skewVal;
}
double distribution::kurtosis ()
{
    if (!statCall) stats ();
    return kurtVal;
}
long distribution::size ()
{
    return sizeVal;
}
double* distribution::getData ()
{
    long i;
    double* temp = new double[sizeVal];
    for (i = 0; i < sizeVal; ++i)
    {
        temp[i] = dataVal[i];
    }
    return temp;
}
bool distribution::setData (const double* dataSet, const long sizeAmt)
{
    if (dataVal) delete[] dataVal;
    dataVal = new double[sizeAmt];
    sizeVal = sizeAmt;
    for (long i = 0; i < sizeVal; ++i)
    {
        dataVal[i] = dataSet[i];
    }
    return true;
}
double distribution::maximum ()
{
    maxCalc ();
    return maxVal;
}

bool distribution::normalize( double factor )
{
    maxCalc();
    double maxVal2 = 0.0;
    if (maxVal != 0.0)
    {
      for (long i = 0; i < sizeVal; ++i)
      {
        dataVal[i] = dataVal[i] / maxVal * factor;
        if ( fabs(dataVal[i]) > maxVal2 ) maxVal2 = fabs(dataVal[i]);
      }
    }
    else
      return false;
    maxVal = maxVal2;
    stats ();
    return true;
}

bool distribution::denormalize(double denorm)
{
    normalize();
    if (maxVal != 0.0)
      for (long i = 0; i < sizeVal; ++i) dataVal[i] = dataVal[i] * denorm;
    else
      return false;
    stats ();
    return true;
}

bool distribution::zeroMean()
{
    if (statCall == false) stats ();
    for (long i = 0; i < sizeVal; ++i)
    {
        dataVal[i] -= meanVal;
    }
    return true;
}

bool distribution::trim(long trimSize)
{
    if (trimSize > sizeVal) return false;
    sizeVal = trimSize;
    stats ();
    maxCalc();
    return true;
}

bool distribution::userSet (double newData, long pos)
{
    dataVal[pos] = newData;
    return true;
}

double distribution :: getItem(long pos)
{
    if (pos > sizeVal)
      {
        cout << "Error: position doesn't exist." << endl;
        return 0;
      }
    return dataVal[pos];
}

bool distribution::zeroIt(long inSize)  //Makes all distribution elements
false/0
{
    if (dataVal) delete [] dataVal;
    dataVal = new double [inSize];
    sizeVal = inSize;
    for (long i = 0; i < sizeVal; ++i)
    {
      dataVal [i] = 0.0;
    }
    sizeVal = inSize;
    meanVal = 0.0;
    meanVal2 = 0.0;
    meanVal3 = 0.0;
    meanVal4 = 0.0;
        varyVal = 0.0;
        skewVal = 0.0;
        kurtVal = 0.0;
    statCall = false;
    maxVal = 0.0;
    return true;
}
bool distribution::scale (double fact)
{
    for (long i = 0; i < sizeVal; ++i)
    {
        dataVal[i] = dataVal[i] * fact;
    }
    return true;
}
bool distribution::add (distribution& inDist)
{
    if(sizeVal != inDist.size()) return false;
    for (long i = 0; i < sizeVal; ++i)
    {
        dataVal[i] = dataVal [i] + inDist.getItem(i);
    }
    return true;
```

```
}
```

WAV Class Header:

```
#include <iostream>
#include <fstream>
#include <cstddef>
#include <string>
using namespace std;

class WAV
{
private:
  unsigned long TotalLengthOfPackage; //total length of wav package less 4
bytes
  unsigned short NumberOfChannels; //left/right channel or stereo
  unsigned long SampleRate; //sampling rate in hz (bytes/second)
  unsigned long BytesPerSecond;
  unsigned short BytesPerSample; //number of bytes per sample
  unsigned short BitsPerSample; //number of bits per sample
  //BitsPerSample = BytesPerSample * 8
  unsigned long LengthOfData; //length of the data of the wav file
  short *ptrData; //pointer to data
  string riffStr, fmtStr, wavStr, dataStr;
  long fmtLength;
  short audioFmt;
  long NumberOfSamples;
  long BytesPerChannel;
  long NumberOfData;
public:
  WAV(); //constructor
  WAV(const WAV&); //copy constructinator
  ~WAV(); //destructor
  bool read(string nameOfFile); //read function
  bool write(string); //write function
  ostream& print(ostream& = cout) const;
  WAV& copy(); //copy function
//accessor functions start here
  unsigned long getLengthOfPackage() const; // accessor function for
TotalLengthOfPackage
  unsigned short getNumberOfChannels() const; // accessor function for shtuff
  unsigned long getSampleRate() const; // accessor function for shtuff
  unsigned long getBytesPerSecond() const;
  unsigned short getBytesPerSample() const; // accessor function
  unsigned short getBitsPerSample() const; // accessor function
  long getfmtLength() const;// accessor function
  unsigned long getLengthOfData() const; // accessor function
  unsigned long getNumberOfData() const; // accessor function
  short* getData() const; // accessor function
//mutator functions start here
  void setLengthOfPackage(unsigned long);
  void setaudioFmt(short);
  void setNumberOfChannels(unsigned short);
  void setSampleRate(unsigned long);
  void setBytesPerSecond(unsigned long);
  void setBytesPerSample(unsigned short);
  void setBitsPerSample(unsigned short);
  void setfmtLength(long);
  void setLengthOfData(unsigned long);
  bool setData(short*,long);
//Operator functions
  bool add(const WAV&, double factor = 1.0);
  bool subtract(const WAV&, double factor = 1.0);
};
```

WAV Class Implementation:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstddef>
#include "wav.h"
#include <cmath>
using namespace std;

WAV::WAV() //constructor
{
  TotalLengthOfPackage = 0;
  NumberOfChannels = 0;
  SampleRate = 0;
  BytesPerSecond = 0;
  BytesPerSample = 0;
  BitsPerSample = 0;
  LengthOfData = 0;
  NumberOfSamples = 0;
  BytesPerChannel = 0;
  ptrData = NULL;
  NumberOfData = 0;
  NumberOfSamples = 0;
  BytesPerChannel = 0;
  riffStr = "RIFF";
  fmtStr = "fmt ";
  wavStr = "WAVE";
  dataStr = "data";
}

//copy constructor

WAV::WAV(const WAV& xWAV)
{
  TotalLengthOfPackage = xWAV.getLengthOfPackage();
  NumberOfChannels = xWAV.getNumberOfChannels();
  SampleRate = xWAV.getSampleRate();
  BytesPerSecond = xWAV.getBytesPerSecond();
  BytesPerSample = xWAV.getBytesPerSample();
  BitsPerSample = xWAV.getBitsPerSample();
  LengthOfData = xWAV.getLengthOfData();

  NumberOfSamples = LengthOfData/BytesPerSample;
  ptrData = xWAV.getData();
  NumberOfSamples = LengthOfData/BytesPerSample;
  NumberOfData = NumberOfSamples*NumberOfChannels;
  BytesPerChannel = BytesPerSample/NumberOfChannels;
}

WAV::~WAV() /*destructor*/
{
  if (ptrData != NULL) delete[] ptrData;
}

//read function
```

```
bool WAV::read(string nameOfFile)
{
  ifstream input;
    input.open (nameOfFile.c_str(), ios::in|ios::binary);

    if (input == NULL)
    {
      cout << "WAV::read(): open failed" << endl;
      return false;
    }

  char *temp = new char[5];
  temp[4] = 0;
  input.read(temp, 4);
  riffStr.assign(temp);
  input.read ((char*)&TotalLengthOfPackage, 4);
  input.read (temp, 4);
  wavStr.assign(temp);
  input.read(temp, 4);
  fmtStr.assign(temp);
  input.read ((char*)&fmtLength, 4);
  input.read ((char*)&audioFmt, 2);
  input.read ((char*)&NumberOfChannels, 2);
  input.read ((char*)&SampleRate, 4);
  input.read ((char*)&BytesPerSecond, 4);
  input.read ((char*)&BytesPerSample, 2);
  input.read ((char*)&BitsPerSample, 2);
  input.read (temp, 4);
  dataStr.assign(temp);
  input.read ((char*)&LengthOfData, 4);
  NumberOfSamples = LengthOfData/BytesPerSample;
  NumberOfData = NumberOfSamples*NumberOfChannels;
  BytesPerChannel = BytesPerSample/NumberOfChannels;
  delete[]temp;
  ptrData = new short[NumberOfData];

  for ( long i=0; i<NumberOfData; ++i)
  {
    input.read((char*)&ptrData[i],BytesPerChannel);
  }

  input.close();
  return true;
}

//write function

bool WAV::write(string outFileName)
{
  ofstream output;
  output.open (outFileName.c_str(), ios::out|ios::binary);

    if (output == NULL)
    {
      return false;
    }
  output.write(riffStr.c_str(), 4);
  output.write ((char*)&TotalLengthOfPackage, 4);
  output.write(wavStr.c_str(), 4);
  output.write(fmtStr.c_str(), 4);
  output.write ((char*)&fmtLength, 4);
  output.write ((char*)&audioFmt, 2);
  output.write ((char*)&NumberOfChannels, 2);
  output.write ((char*)&SampleRate, 4);
  output.write ((char*)&BytesPerSecond, 4);
  output.write ((char*)&BytesPerSample, 2);
  output.write ((char*)&BitsPerSample, 2);
  output.write (dataStr.c_str(), 4);
  output.write ((char*)&LengthOfData, 4);

  for ( long i=0; i<NumberOfData; ++i)
  {
    output.write((char*)&ptrData[i],BytesPerChannel);
  }

  output.close();
  return true;
}

//print function

ostream& WAV::print (ostream& outStream) const
{
  outStream << "Riff string = " << riffStr << endl;
  outStream << "Length = " << TotalLengthOfPackage << endl;
  outStream << "WAVE string = " << wavStr << endl;
  outStream << "Format string = " << fmtStr << endl;
  outStream << "Format length = " << fmtLength << endl;
  outStream << "Audio format = " << audioFmt << endl;
  outStream << "Number of channels = " << NumberOfChannels << endl;
  outStream << "Sample Rate = " << SampleRate << endl;
  outStream << "Bytes per second = " << BytesPerSecond << endl;
  outStream << "Bytes per sample = " << BytesPerSample << endl;
  outStream << "Bits per sample = " << BitsPerSample << endl;
  outStream << "Data string = " << dataStr << endl;
  outStream << "Length of data = " << LengthOfData << endl;
  outStream << "Data Location = " << ptrData << endl << endl;
  return outStream;
}

WAV& WAV::copy(); //copy function

//accessor functions start here

unsigned long WAV::getLengthOfPackage() const // accessor function for
TotalLengthOfPackage
{
  return TotalLengthOfPackage;
}

unsigned short WAV::getNumberOfChannels()const // accessor function for
shtuff
{
  return NumberOfChannels;
}

unsigned long WAV::getSampleRate() const // accessor function for shtuff
{
  return SampleRate;
}

unsigned long WAV::getBytesPerSecond() const
{
  return BytesPerSecond;
}

unsigned short WAV::getBytesPerSample() const // accessor function
{
  return BytesPerSample;
}

unsigned short WAV::getBitsPerSample() const // accessor function
{
  return BitsPerSample;
}

long WAV::getfmtLength() const// accessor function
{
  return fmtLength;
}

unsigned long WAV::getLengthOfData() const// accessor function
{
  return LengthOfData;
}

unsigned long WAV::getNumberOfData() const// accessor function
{
  return NumberOfData;
}

short* WAV::getData() const// accessor function
{
  short* temp = new short[NumberOfData];
  for (long i = 0; i < NumberOfData; ++i)
  {
    temp[i] = ptrData[i];
  }
  return temp;
}

//mutator functions start here

void WAV::setLengthOfPackage(unsigned long Length)
{
  TotalLengthOfPackage = Length;
}

void WAV::setaudioFmt(short Fmt)
{
  audioFmt=Fmt;
}

void WAV::setNumberOfChannels(unsigned short Number)
{
  NumberOfChannels = Number;
}

void WAV::setSampleRate(unsigned long Rate)
{
  SampleRate = Rate;
}

void WAV::setBytesPerSecond(unsigned long bps)
{
    BytesPerSecond = bps;
}

void WAV::setBytesPerSample(unsigned short Bytes)
{
  BytesPerSample = Bytes;
}

void WAV::setBitsPerSample(unsigned short Bits)
{
  BitsPerSample = Bits;
}

void WAV::setfmtLength(long flength)
{
  fmtLength=flength;
}

void WAV::setLengthOfData(unsigned long LengthOD)
{
  LengthOfData = LengthOD;
}

bool WAV::setData(short* Data, long Length)
{
  if ( ptrData ) delete[] ptrData;
  ptrData = new short[Length];
  NumberOfData = Length;
  for ( long i=0; i<NumberOfData; ++i)
  {
    ptrData[i] = Data[i];
  }
  NumberOfSamples = LengthOfData/BytesPerSample;
  NumberOfData = NumberOfSamples*NumberOfChannels;
  BytesPerChannel = BytesPerSample/NumberOfChannels;
  return true;
}

//Operator functions

bool WAV::add(const WAV& a, double f)
{
```

```
//set up WAV lengths

  short *ptrLongest = a.getData();
  double longf = f;
  long longest = a.getNumberOfData();
  short* ptrShortest = ptrData;
  double shortf = 1.0;
  long shortest = NumberOfData;

//correct Wave lengths

    if(shortest>longest)
      {
        shortest = longest;
        longest = NumberOfData;
        ptrShortest = ptrLongest;
        ptrLongest = ptrData;
        shortf = longf;
        longf = 1.0;
      }

//Allocate new data space

  double* Sum = new double[longest];

//Adding

  double Large = 0.0;
    for(long i = 0; i<shortest; ++i)
      {
         Sum[i] = (ptrLongest[i]*longf) + (ptrShortest[i]*shortf);
           if(Large<abs(Sum[i]))
             {
               Large=Sum[i];
             }
      }

    for(i = shortest; i<longest; ++i)
     {
      Sum[i] = ptrLongest[i]*longf;
        if(Large<abs(Sum[i]))
          {
            Large = Sum[i];
          }
     }

//Scaling

  for(i=0;i<longest;++i)
    {
      ptrLongest[i] = Sum[i]/Large*32767;
    }
    delete[] Sum;

// If ptrSoundData not already tmplong then delete prtSoundData.
// This will also delete tmpshort, else delete tmpshort
```

```
  if ( ptrData != ptrLongest )
  {
    delete[] ptrShortest;
    ptrData = ptrLongest;
  }
  else
    delete[] ptrShortest;

//Final formatting

  NumberOfData = longest;
  LengthOfData = NumberOfData*BytesPerChannel;
  TotalLengthOfPackage = LengthOfData+36;
  NumberOfSamples = LengthOfData/BytesPerSample;

  return true;
}

bool WAV::subtract(const WAV& a, double f)
{
   return(this->add(a,-f));
}
```

```
Auxiliary Functions:


//Converts an array of shorts to an array of doubles

double* STF(const short* Unconv, long Length)
      {
            double* Converted = new double[Length];
            for(long Pos=0; Pos < Length; ++Pos)
                  {
                        Converted[Pos] = Unconv[Pos];
                  }
            return Converted;
      }

//Converts an array of doubles to an array of shorts

short* FTS(const double* Unconv, long Length)
      {
            short* Converted = new short[Length];
            for(long Pos=0; Pos < Length; ++Pos)
                  {
                        Converted[Pos] = Unconv[Pos];
                  }
            return Converted;
      }



//Calculates The Correction

#include "datadist.h"
#include <cmath>

using namespace std;

double E31 (distribution*, distribution*);
double E11 (distribution*, distribution*);

double* correct(distribution* y, distribution* x, short n)
{


  double* C = new double[n*n];
  short pos;

// begin correction looping

   for(short i=0; i<n; ++i)
    {
      for(short j = 0; j<n; ++j)
        {
          pos = i*n+j;

          // Cij = 4 * ( E[y^3*x]*E[y^2] -
          //             E[y^4]*E[y*x] ) /
          //             E[y^2]^3
          C[pos] = 4 * ( E31(&y[i],&x[j]) * y[i].mean2()  -
                         y[i].mean4() * E11(&y[i],&x[j])  ) /
                         pow(y[i].mean2(),3);

      }
     }


      return C;
}

//E31

double E31 (distribution* y, distribution* x)
{
  double res = 0;
  double Y,X;
  long n = y->size();

  for(long i = 0; i<n; ++i)
    {
      Y = y->getItem(i);
      X = x->getItem(i);

      res = res + Y*Y*Y*X;
    }

  res = res/n;
  return res;
}

//E11

double E11 (distribution* y, distribution* x)
{
  double res = 0;
  double Y,X;
  long n = y->size();

  for(long i = 0; i<n; ++i)
    {
      Y = y->getItem(i);
      X = x->getItem(i);

      res = res + Y*X;
    }

  res = res/n;
  return res;
}



//Adds two matrices

bool madd(double* W, double* C, short n)
```

```
{

  for(short i=0; i<n*n; ++i)
   {
     W[i]=W[i]+C[i];
   }

  return true;

}



//Calculates the norm of a matrix

#include <cmath>

double norm(double* Mat, short n)
{

  double nmat = 0;

  for(short i = 0; i<n*n; ++i)
    {

      if ( fabs(Mat[i]) > nmat ) nmat = fabs(Mat[i]);
    }

  return nmat;

}



//Multiplies each element in a matrix by a scalar

bool mscale (double* M, double s, short n)
{

  for(short i =0; i<n*n; ++i)
  {
    M[i]=M[i]*s;
  }

  return true;

}
```

Mixing Program:

```
#include<iostream>
#include "wav.h"
#include <fstream>
#include <string>
#include <cmath>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main()
{

//whitespace is futile
  cout << "WHitE5P4cE 15 PhUt1l3 " << endl << endl;

  WAV s1;
  WAV w1;
  string sl;
  string wl;
  double sf;
  double wf;

//Get source input

  cout << "Input 1st source name:";
  cin >> sl;
  cout << endl;
  cout << "Input 2nd source name:";
  cin >> wl;
  cout << endl;

//Get mixing factors

  cout << "Input 1st source mixing factor:";
  cin >> sf;
  cout << endl;
  cout << "Input 2nd source mixing factor:";
  cin >> wf;
  cout << endl;

//Read sources

  s1.read(sl);
  w1.read(wl);

//Mix sources

  s1.add(w1);
  w1.add(s1);

//Writes mixtures

  s1.write("xx1.wav");
  w1.write("xx2.wav");

  return 0;
}
```