DeepSample

Developer Handbook

Andrew Moore, Alex Reno, Hue Truong

Contents

Making The Project 5
DeepSample Programming Tutorial 7
- Part 1: Loading an Audio File 7
- Part 2: Verifying an Audio Object Using DeepSampleTest 9
- Part 3: Sample Generation With a Single Audio Directory 10
- Part 4: Recursive Sample Generation 15
- Part 5: Analyzing a Database with DeepSample 17

Running the Prebuilt Binaries 18
- DeepSample 18
- DeepSampleTests 20
- SampleGenerator / genSamples 22

File Dependency Diagram 25
Class Reference 26
- AudioWave 26
  - Private Member Variables 26
    - Strings 26
      - fileName 26
    - Vectors 26
      - leftChannel 26
      - rightChannel 26
      - leftFFT 26
      - rightFFT 26
      - cepstrumData 26
      - zeroData 27
      - max 27
      - min 27
      - spectrumCData 27
      - spectrumFData 27
      - sourceFiles 27
    - Integers 27
      - channels 27
      - frames 27
  - Public Member Functions 28
    - Object Manipulation Functions 28
      - AudioWave 28
      - ~AudioWave 28

Initialization Functions 29
setCepstrumData 29
setChannels 29
setFrames 29
setName 30
setLeftFFT 30
setRightFFT 30
setYMaximums 31
setYMinimums 31
setSourceFiles 31
setZeroData 31
Update Functions 32
pushCepstrum 32
pushLeftChannel 32
pushRightChannel 33
pushSpectrumC 33
pushSpectrumF 33
pushZero 34
Getter Functions 34
Return Strings 34
getFileName 34
getSourceFile 34
Return Vectors 35
getLeftChannel 35
getLeftFFT 35
getRightChannel 35
getRightFFT 36
Return Complex 36
getChannelData 36
Return Double 37
getCepstrumDataPoint 37
getFFTDataPoint 37
getSpectrumCDataPoint 38
getSpectrumFDataPoint 38
getYMaximum 39
getYMinimum 39
getZeroDataPoint 40
Return Integer 40
getChannels 40

getChannelSize 40
getFrames 41
getCSize 41
getLeftSize 41
getLeftFFTSize 42
getRightSize 42
getRightFFTSize 42
getSCSize 43
getSFSize 43
getZSize 43
Fold 44
Private Member Variables 44
Vectors 44
dataFolds 44
Public Member Functions 44
Object Manipulation Functions 44
Fold 44
~Fold 44
Updater Functions 44
pushFold 44
Get Functions 45
getFold 45
getSize 45
Function Reference 46
Audio Segmentation Algorithms 46
ANN 46
Main Neural Net Functions 46
ANNI 46
getBestMatch 47
learningVectorQuantization 48
trainCodeBooks 48
Helper Functions 49
euclideanDistance 49
lvqHelper 50
prepareFolds 51
randomDatabase 52
audioHandler 53
convertSound 53
loadAudio 54

cepstrum 55
rCepstrum 55
windowHamming 55
FourierTransform 56
fft 56
inverseFFT 56
SpectrumCentroid 57
spectralCentroid 57
SpectrumFlux 57
spectralFlux 57
ZeroCrossing 58
zeroCross 58
Utilities 59
createString 59
graphAlg 60
generateScript 61
genTrainSet 61
normalize 62
plotter 63
printer 64
realify 64
timestamp 65
fileExists 65
sortDist 66
sign 67

Making the Project

DeepSample uses a Makefile to simplify the compilation process. There are several options that can be used to generate different working binaries. First an overview of the commands:

make default:
This command will generate all of the binaries: **DeepSampleTests, SampleGenerator, genSamples, and ANN.**

make all:
This command will also generate all of the binaries: **DeepSampleTests, SampleGenerator, genSamples, and ANN.**

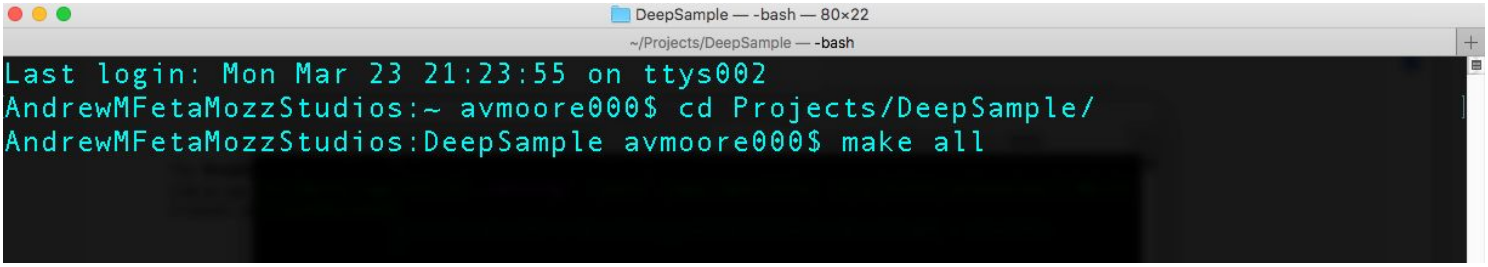
make test:
This command will only create the **DeepSampleTests** binary.

make sample:
This command will only create the **SampleGenerator** and **genSamples** binaries

make anni:
This command will create the **ANN** binary file.

make clean:
This command will clean up all binaries and text files, ignoring the user created results and plot directories.

To make the project, first navigate to the DeepSample project directory in your terminal. In the root directory of the project ~/DeepSample, enter the desired make command. For example, make all the binaries:



This will make all of the binaries, and place them in a build directory within the root directory of the project. You can then proceed to use them as normal.

DeepSample Programming Tutorial

DeepSample is a library of functions that allows the user to perform audio segmentation tasks. Right now it is able to handle the fast fourier transform, spectrum flux, spectrum centroid, zero crossing, and real cepstrum algorithms. The audio formats currently supported are **OGG Vorbis**, **FLAC**, and **WAV** file formats. The following section contains some basic example programs using the library.

Part 1: Loading an Audio File

The loading of an audio file is the simplest program that can be written using the DeepSample library. It merely takes in an audio file and converts it to a numerical representation, then writes that representation to a file.

```
void main()
{
	AudioWave wave(“test”, 2);
	string inputFile;
	string path;
	string audioDir;
	string sanName;
	int channels;
	bool debug, fullPrecision;

	inputFile = “sample.ogg”;
	path = “pathToOutputFiles”;
	audioDir = “pathToAudioOutputDirectory”;
	sanName = “sample.ogg”;
	channels = 2;
	debug = 1;
	fullPrecision = 1;

	loadAudio(wave, inputFile, audioDir, sanName, channels, fullPrecision,
		path, debug);

	return;
}
```

Example 1: Loading An Audio File With DeepSample

In example 1 we are loading an audio file. The first step is to tell the program the pertinent details of the task, which are loaded via command line arguments that correspond to the following variables:

wave - An AudioWave object that will store the resulting vectors

inputFile - A string describing the audiofile to load, in this case it is pointing to sample.ogg

audioDir - The path to the audio file directory

sanName - The name of the audio file without any path information

channels - The number of channels in the audio file.

fullPrecision - The precision of the data output.

path - This is the path where the various algorithms will output log files, create databases, and save plots.

debug - Whether to generate debug information on the run.

These variables are initialized and passed to the loadAudio function, which acts as a wrapper function for the audio converter. Note that loadAudio is a void function, and stores its results in an AudioWave object that is passed, in this case the object is wave. Note that if debug mode is turned off, the data generated by this program is not accessible by the user. The main purpose of the audio loading function is to prepare an audio file for further analysis. This program is not useful alone, so we will build on the concepts in the next section.

Part 2: Verifying an Audio Object using DeepSampleTest

Continuing with the example from above, in this section we will test the AudioObject using the DeepSampleTest program. The purpose of this is to make sure the audio samples you are working with are generating data in the proper format for use with DeepSample. This is an important step when using the program with new data, as you need to be sure that the audio segmentation algorithms are creating meaningful input for DeepSample.

```
int main(int argc, char** argv)
{
	vector<AudioWave> waves;
	string inputFile = “sample.ogg”;
	string outputFile = sampleOut.txt”;
	string filePath “/path/to/output/”;
	bool debug = 1;
	int channels = 2;

	AudioWave w("test",2);
	waves.push_back(w);

	// Grab the arguments from the command line

	// Create a directory for results
	if(mkdir(filePath.c_str(), 0777) == -1)
		cout << "Error creating directory.  " << filePath
			<< " already exists." << endl;
	else
		cout << "Results directory created at "
			<< filePath << endl;

	audioTest(waves[0], inputFile, "", "", channels, 1, outputFile,
			filePath, debug);

	return 0;
}
```

Example 2: Verifying an audio sample with DeepSample

The initial setup of Example 2 is very similar to Example 1 in the previous section. The user passes in the details of the samples via command line arguments that are then parsed into variables. The main difference is that instead of calling the

loadAudio function we are making a call to audioTest. The audioTest function is part of an extensive test suite that comes prepackaged with DeepSample, whose purpose is to perform a sanity check on the sample generation process. This helps ensure that the databases being used to test DeepSample are valid.

Part 3: Sample Generation With Single Audio Directory

After you have verified that the directory you'll be using is valid with the DeepSample test suite, it is time to actually generate the databases. The simplest method of doing this is to use the **SampleGenerator** binary to create sample databases, however it is possible to access the library in a custom manner. The following code listing uses the SampleGeneration library to generate a Zero Crossing database.

```
int main(int argc, char** argv)
{

    string audioDir;                // Directory for converted audio files.
    string fileName;                // Name for output file
    string inputDirectory;          // Location of input files
    string path;                    // Path for file output.
    string resultsDirectory;        // Name of directory for storing results.

    ostringstream converter;        // Used to convert dir entr. to string

    ifstream inFile;                // Stream pointer for data input.

    bool debug;                     // Flag that controls debug output.
    bool plot;                      // Flag that controls plotting
    bool startGeneration;           // Starts the sample generation.
    bool fullPrecision;             // The precision for output.
    bool save;                      // Toggle saving
    int channels;                   // Holds the number of channels
    int test;                       // Specifies which tests are being run.
    int graphType;                  // Specifies the type of graph to plot.

    fs::path p;                     // Will hold an iterable directory path.

    vector<string> audioNames;      // Names of all audio files to analyze.
    vector<string> scrubbedNames   ; // Names of file without path info
```

```
// Grab variable values from command line and store for program use.

if (startGeneration)
{

    // Create directories for results
    if(mkdir((path.c_str()),0777) == -1)
        cout << “Error: Directory already exists.” << endl;
    else
    {
        if (mkdir((path + "/ConvertedAudio").c_str(), 0777) == -1)
        {
            cout << timeStamp() << ":  ConvertedAudio directory not “
                 << “created." << endl;
        }

        if (mkdir((path + "/ZeroCross").c_str(), 0777) == -1)
        {
            cout << timeStamp() << ":  ZeroCross directory not created."
                 << endl;
        }

        if (mkdir((path + "/Databases").c_str(), 0777) == -1)
        {
            cout << timeStamp() << ":  Databases directory not created"
                 << endl;
        }

        if (plot)
        {
            if (mkdir((path + "/Plots").c_str(), 0777) == -1)
            {
                cout << timeStamp() << ":  Plots directory not created"
                     << endl;
            }
        }

        if (debug)
        {
            if (mkdir((path + "/Debug").c_str(), 0777) == -1)
            {
                cout << timeStamp() << ":  Debug directory not created"
                     << endl;
            }
        }
    } // Finished making directories

    // Grab the audio file names from the input directory
```

```
p = inputDirectory;
if (fs::is_directory(inputDirectory))
{
    for (auto &entry:
    boost::make_iterator_range(fs::directory_iterator(p)))
    {
        converter << entry;

        string temp = converter.str();
        string converted = "";

        // Loop through and grab the current file name, removing
        // quotes
        for (int i = 0; i < temp.size(); i++)
        {
            if (temp[i] != '\"')
                converted += temp[i];
        }

        audioNames.push_back(converted);
        converter.str("");
        converter.clear();

        string tempName = "";

        // Get just the audio file name, returns backwards string
        for (int i = converted.size() - 1; i > 0; i--)
        {
            if (converted[i] != '/')
                tempName += converted[i];
            else
                break;
        }

        converted = "";

        // Reverse the backwards audio file name.
        for (int i = tempName.size() - 1; i >= 0; i--)
            converted += tempName[i];

        scrubbedNames.push_back(converted);

        if (debug)
            cout << "\tFile Name = " << converted << endl;
    }
} // Finished grabbing audio file names
```

```
if (startGeneration)
{
    cout << timeStamp() << ":  Processing Audio Files..." << endl;

    // Run algorithms for each file in the audio sample directory
    for (int i = 0; i < audioNames.size(); i++)
    {
        AudioWave wave("test", channels);  // An AudioWave object

        loadAudio(wave, audioNames[i],"ConvertedAudio",
                  scrubbedAudioNames[i], channels, fullPrecision,
                  path, debug);

        zeroCross(wave, fileName, path, debug);

        // If plotting, plot
        if (plot)
        {
            // Generate Y maximums and minimums

            wave.setYMaximums();
            wave.setYMinimums();

            // Set up source files for plotting
            wave.setSourceFiles();

            // Plot the data

            plotter(wave, graphType, 3, fileName, path, debug);
        }
        // Done with plotting

        // If saving, create/update the database files
        if (save)
        {
            int tChannel = wave.getChannels();
            vector<complex<double> > tempFFT;
            string audioName = "\u03bc " + wave.getFileName();

            for (int i = 0; i < wave.getChannels(); i++)
            {
                // Zero Cross Database

                database.open((path +
                "/Databases/stereoZeroCross.txt").c_str(), ios::app);

                database << audioName << endl;
```

```
                        bool end = 0;

                        for (int j = 0; j < tChannel; j++)
                        {
                            for (int k = 0; k < wave.getZSize(j+1); k++)
                                database << wave.getZeroDataPoint(j+1,k)
                                         << " ";

                            database << endl;
                        }

                        database << endl;
                        database.close();

                    }
                }
                // Done with saving, creating/updating databases.
            } // Finished algorithm loop
        } // End second generation
    } // End generation

    return 0;
}
```

Example 3: Zero Cross Sample Database Generation

Some notable parts of the above example are the save functionality as well as the plotting. When the program is not told to save, the sample database will not be generated. The same can be said for the plotter. Also note that the plotting function uses gnuplot, without this program the plotter will fail.

Part 4: Recursive Sample Generation

After you have verified that the directory you'll be using is valid with the DeepSample test suite, it is time to actually generate the databases. The following code listing uses a recursive method to generate sample databases from an audio directory.

```
int main()
{
	string fileName;		// Name for output file
	string inputDirectory;	// Location of input files
	string path;			// Path for file output.
	string command;			// Command for calling sample generator.
	ostringstream converter;	// For converting directory entr. to strings
	bool debug;			// Flag that controls debug output.
	bool plot;			// Flag that controls plotting functionality.
	bool fullPrecision;		// The precision for output.
	bool save;			// Toggle saving
	int channels;			// Holds the number of channels in the file.
	int graphType;			// Specifies the type of graph to plot.
	fs::path p;			// Will hold an iterable directory path.

	command = "./SampleGenerator ";
	fileName = "outputLogName";
	inputDirectory = "path/to/audioSample/directory";
	path = "path/for/result/output";
	debug = 1;
	plot = 1;
	fullPrecision = 1;
	save = 1;
	channels = 1;
	graphType = 1;
	p = inputDirectory;

	if (fs::is_directory(inputDirectory))
	{
		for (auto &entry:
		     boost::make_iterator_range(fs::directory_iterator(p)))
		{
			converter << entry;
		command += path + " " + converter.str() + " " + fileName +
				"";
			command += to_string(channels) + " " +
			to_string(fullPrecision);
		command += " " + to_string(debug) + " " +
```

```
                to_string(plot) + " " + to_string(graphType);
                command += " " + to_string(save);

                cout << command << endl << endl;

            system(command.c_str());

            converter.str("");
            converter.clear();
            command = "./SampleGenerator ";
            }
        }

        return 0;
}
```

Example 4: Recursive Sample Generation

Example 4 uses the boost filesystem to recursively generate samples using the **SampleGenerator** binary. If the boost functionality is removed, **SampleGenerator** is still able to generate sample files from an entire directory of audio files, but it is unable to generate samples from any child directories within the root sample directory. For a more extensive look at the recursive generation process, look at the *driver.cpp* file located in the *src* directory of DeepSample. This is the source file for the genSamples binary, which is a recursive implementation of the sample generation process.

Part 5: Analyzing a Database with DeepSample

The last and arguably most important function of DeepSample is the analysis of the sample databases with an artificial neural network. While it is possible to use the prebuilt **ANN** binary to do this, you may also call the **ANN** in your own programs by using the ANNI function call. The following code listing shows an example of this:

```
int main(int argc, char** argv)
{

    string path;                // Path for file output.
    string sampleName;          // A string containing the name of the sample
    int folds;                  // Contains the number of folds to divide data
    double learnRate;           // The learning rate for the neural network.
    int epochs;                 // The number of epochs to train the network
    int codeBooks;              // The number of codebooks to generate
    int alg;                    // The algorithm to run against.
    bool debug;                 // Flag that controls debug output.
    int channels;               // Holds the number of channels in the file.



    // Grab the command line arguments and initialize the needed variables.

    // Call ANNI
    ANNI(sampleName, folds, learnRate, epochs, codeBooks, alg, channels, path,
         debug);

    return 0;
}
```

Example 5: Using the ANNI function

As can be seen from Example 5, it is fairly simple to call the ANN function from your own code. All that is needed is to follow some basic steps to prepare the needed arguments.

For more detailed information on the capabilities of DeepSample, please see the function reference.

Running the Prebuilt Binaries

DeepSample

The **DeepSample** binary is a neural network that can be used to analyze databases containing the results of the **SampleGenerator** binary. It operates by performing a best match comparison between a database of known samples, and a test database. In this iteration of **DeepSample** it must be given a prebuilt library of known data, and does not add to the library. The ability to incorporate new samples into the database will be present in future versions of the library. **DeepSample** has built in help that can be accessed by running it without arguments:

./DeepSample

This will output a list of commands that can be given to **DeepSample.** The following section goes into the various options in a bit more detail.

Program Use:

./DeepSample [trainPath] [testPath] [folds] [learnRate]
[epochs] [codeBooks] [alg {0,1,2,3,4,5]
[channels {1,2}] [resultsPath] [debug {0,1}]

trainPath:

A string indicating the path for the training data.

testPath:

A string indicating the path to the test data.

folds:

Integer indicating the number of folds to break data into.

learnRate:

Floating point indicating the learning rate.

epochs:

An integer indicating the number of epochs to train over.

codeBooks

An integer indicating the number of codebooks to generate from the dataset.

alg:

Integer indicating which algorithm to run against

0 - All algorithms
1 - Fast Fourier Transform
2 - Zero Cross
3 - Spectrum Flux
4 - Cepstrum
5 - Spectrum Centroid

channels:

An integer indicating the number of channels in the audio samples.

resultsPath:

The path for saving data output.

debug:

A flag that controls debug output.

0 - No debug
1 - Debug

When the **DeepSample** binary is run, it outputs the results to a log file in the user specified directory, this file is always named ANNResults.txt. It works by running a comparison on all of the samples found in the test directory, and generates an association for the test database based on the known data directory.

DeepSampleTests

The **DeepSampleTests** binary contains a suite of test functions that can be used to verify the functionality of the DeepSample library, as well as to experiment around with the algorithms effects on different input files. **DeepSampleTests** has built in help that can be accessed by running it without arguments:

./DeepSampleTests

This will output a list of commands that can be given to **DeepSampleTests**. The following section goes into the various options in a bit more detail.

Program Use:

./DeepSampleTests [resultsDirectory] [inputFile] [outputFile] [channels] [debugMode] [tests]

resultsDirectory:

This is a user specified directory where output will be stored. If the directory does not exist it will be created. The directory will be placed within the directory your program is being run.

inputFile:

The audio file for analysis. As of this writing, DeepSample has support for **OGG Vorbis**, **FLAC,** and **WAV** format files.

outputFile:

The name of the file for the main output of the program. This will include all non-debug output.

channels:

The number of channels in the audio file. This is important for allowing the program to work with monaural and stereo sound properly.

fullPrecision:

Used to toggle full precision decimals on and off.

1 = Enable
0 = Disable

save:

Used to toggle the saving of data files on and off.

1 = enable
0 = disable

debug:

Used to toggle debug mode on and off.

1 = enable
0 = disable.

tests:

This number will tell DeepSample which tests you wish to run. The options are as follows:

0 - Run all available tests.
1 - Run Audio Test
2 - Run FFT Test
3 - Run Zero Cross Test
4 - Run Spectrum Flux Test
5 - Run Cepstrum Test
6 - Run Spectrum Centroid Test
7 - Run ANN Test

SampleGenerator / genSamples

The **SampleGenerator** binary can be used to generate databases for training ANNI from a given set of audio files. It does not perform any testing of the functions and is meant as a utility allowing users to quickly create training sets for ANNI. In addition, **SampleGenerator** does not support recursive sample generation in child directories, it only reads audio files within the parent directory. For recursive sample generation, refer to the next section for the **genSamples** binary.

Similar to **DeepSampleTests**, **SampleGenerator** has built in help functionality that is accessible by running the program without any arguments:

./SampleGenerator

This will output information on using the program. This information is described in more detail in the following section.

Program Use:

./SampleGenerator [resultsDirectory] [inputDirectory] [outputFileName] [channels] [debugMode] [plot]

resultsDirectory:

This is a user specified directory where output will be stored. If the directory does not exist it will be created. The directory will be placed within the directory your program is being run.

inputDirectory:

The directory containing the audio files for analysis. As of this writing, DeepSample has support for **OGG Vorbis**, **FLAC,** and **WAV** format files.

outputFileName:

A prefix that will be used for the output file. This will contain all non-debug general output of the main program.

channels:

The number of channels in the audio file. This is important for allowing the program to work with monaural and stereo sound properly.

1 = Monaural
2 = Stereo

fullPrecision:

Used to toggle full precision decimals on and off.

1 = Enable
0 = Disable

debugMode:

Used to toggle debug mode on and off.

1 = Enable
2 = Disable

plot:

Toggles graph plotting on and off.

1 = Plot graphs
2 = No graphing

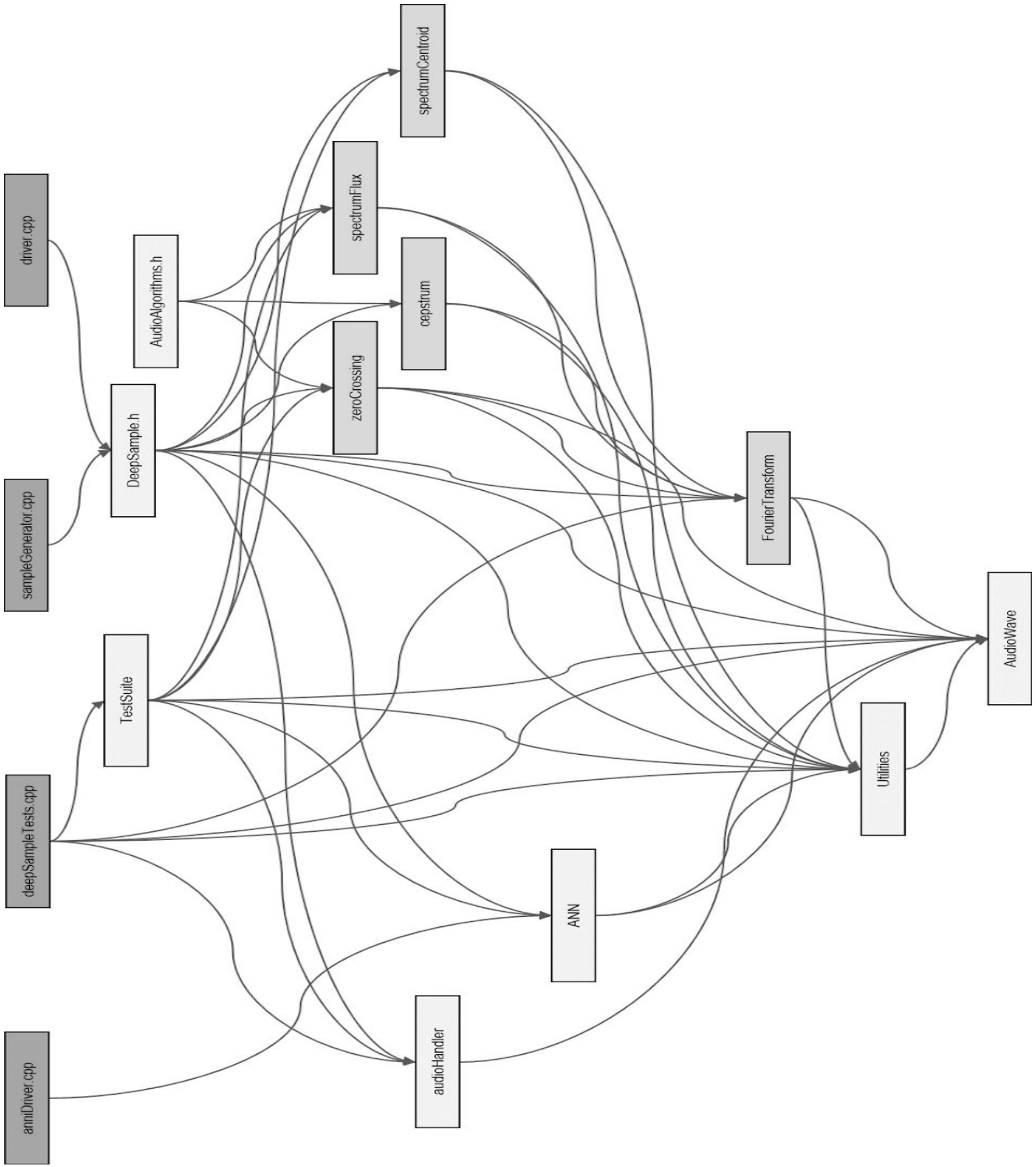
save:

Used to toggle the saving of data files on and off.

1 = Enable
2 = Disable

Alternatively, the **genSamples** binary is run exactly the same as the **SampleGenerator** binary, but allows the samples to be generated recursively, meaning that it will not ignore child directories. As of this version it only supports single layer recursion, in the form Parent -> Child directory.

File Dependency Diagram



Class Reference

AudioWave

Private Member Variables

string **fileName:**

A string indicating the file for data output.

vector<complex<double> > **leftChannel:**

A vector of complex doubles representing the left channel.

vector<complex<double> > **rightChannel:**

A vector of complex doubles representing the right channel.

vector<complex<double> > **leftFFT:**

A vector of complex doubles containing the fourier transform of the left channel.

vector<complex<double> > **rightFFT:**

A vector of complex doubles containing the fourier transform of the right channel.

vector<vector<double> > **cepstrumData:**

A 2D vector of doubles containing the cepstrum results for each channel.

vector<vector<double> > **zeroData:**

A 2D vector of doubles containing the zero cross results for each channel.

vector<double> **max:**

A vector of doubles containing the maximum values of each data vector.

vector<double> **min:**

A vector of doubles containing the minimum values of each data vector.

vector<double> **spectrumCData:**

A vector of doubles containing the spectrum centroid results for each channel.

vector<double> **spectrumFData:**

A vector of doubles containing the spectrum flux results for each channel.

vector<vector<string> > **sourceFiles;**

A vector of strings containing file names used for graphing data.

int **channels:**

An integer indicating the number of channels.

int **frames**:

An integer indicating the number of frames.

Public Member Functions

Object Manipulation Functions

function **AudioWave**(audioName, chan)

The constructor for an AudioWave object.

Parameters

- **audioName** - A string indicating the full path to an audio file.
- **chan** - An integer indicating the number of channels in the audio file.

Returns: wave - An AudioWave object.

Return Type: object

function **~AudioWave**()

The destructor for an AudioWave object.

Parameters

Returns:

Return Type:

Initialization Functions

function **setCepstrumData**()

Initializes the cepstrumData member variable.

Parameters

Returns:

Return Type: void

function **setChannels**(chan)

Initializes the channels member variable.

Parameters

- **chan** - An integer indicating the number of channels in the audio file.

Returns:

Return Type: void

function **setFrames**(num)

Initializes the frames member variable.

Parameters

- **num** - An integer indicating the number of frames in the audio file.

Returns:

Return Type: void

function **setName**(audioName)

Initializes the fileName member variable.

Parameters

- **audioName** - A string indicating the full path to an audio file.

Returns:

Return Type: void

function **setLeftFFT**(fft)

Initializes the leftFFT member variable.

Parameters

- **fft** - A vector of complex doubles representing a FFT.

Returns:

Return Type: void

function **setRightFFT**(fft)

Initializes the rightFFT member variable.

Parameters

- **fft** - A vector of complex doubles representing a FFT.

Returns:

Return Type: void

function **setYMaximums()**

Sets the maximum values of the data vectors

Parameters

Returns:

Return Type: void

function **setYMinimums()**

Sets the minimum values of the data vectors.

Parameters

Returns

Return Type: void

function **setSourceFiles**()

Initializes the sourceFiles member variable.

Parameters

Returns:

Return Type: void

function **setZeroData**()

Initializes the zeroData member variable.

Parameters

Returns:

Return Type: void

Update Functions

function **pushCepstrum**(chan, data)

Add a value to cepstrumData member variable.

Parameters

- **chan** - An integer indicating the channel to add data to.
- **Data** - A double containing the data to add.

Returns:

Return Type: void

function **pushLeftChannel**(data)

Add a value to the leftChannel member variable.

Parameters

- **data** - A complex double containing the data to add.

Returns:

Return Type: void

function **pushRightChannel**(data)

Add a value to the rightChannel member variable.

Parameters

- **data** - A complex double containing the data to add.

Returns:

Return Type: void

function **pushSpectrumC**(data)

Add a value to the spectrumCData member variable.

Parameters

- **data** - A double containing the data to add.

Returns:

Return Type: void

function **pushSpectrumF**(data)

Add a value to the spectrumCData member variable.

Parameters

- **data** - A double containing the data to add.

Returns:

Return Type: void

function **pushZero**(chan, data)

Add a value to the zeroData member variable.

Parameters

- **chan** - An integer indicating the channel to add the data to.
- **data** - A double containing the data to add.

Returns:

Return Type: void

Get Functions

function **getFileName**()

Return fileName member variable.

Parameters

Returns: fileName

Return Type: string

function **getSourceFile**(chan, index)

Parameters

- **chan** - An integer indicating the channel to access.
- **index** - An integer indicating the source file to look up.

Returns:

Return Type: string

function **getLeftChannel**()

Return the leftChannel member variable.

Parameters

Returns: leftChannel

Return Type: vector<complex<double> >

function **getLeftFFT**()

Return the leftFFT member variable.

Parameters

Returns: leftFFT

Return Type: vector<complex<double> >

function **getRightChannel**()

Return the rightChannel member variable.

Parameters

Returns: rightChannel

Return Type: vector<complex<double> >

function **getRightFFT**()

Return the rightFFT member variable.

Parameters

Returns: rightFFT

Return Type: vector<complex<double> >

function **getChannelData**(chan, index)

Return a valuefrom specified channel.

Parameters

- **chan** – An integer indicating the channel to access.
- **index** – An integer indicating which index to read from.

Returns: value – A complex double containing the data at the specified index.

Return Type: complex<double>

function **getCepstrumDataPoint**(chan, index)

Return a value from cepstrumData

Parameters

- **chan** - An integer indicating the channel to access.
- **index** - An integer indicating which index to read from.

Returns: dataPoint - A double containing the data at the specified index.

Return Type: double

function **getFFTDataPoint**(chan, index)

Return a value from an FFT vector.

Parameters

- **chan** - An integer indicating the channel to access.
- **index** - An integer indicating which index to read from.

Returns: value - A double containing the data at the specified index.

Return Type: double

function **getSpectrumCDataPoint(**chan)

Returns a value from the spectrumCData member variable.

Parameters

- **chan** - An integer indicating the channel to access.

Returns: dataPoint - A double containing the data at the specified index.

Return Type: double

function **getSpectrumFDataPoint**(chan)

Returns a value from the spectrumFData member variable.

Parameters

- **chan** - An integer indicating the channel to access.

Returns: dataPoint - A double containing the data at the specified index.

Return Type: double

function **getYMaximum(**alg, chan)

Parameters

- **alg** - An integer indicating the algorithm to look up
- **chan** - An integer indicating the channel to look up.

Returns: maxi - A double containing the maximum of the given dataset.

Return Type: double

function **getYMinimum(**alg, chan)

Parameters

- **alg** - An integer indicating the algorithm to look up
- **ahan** - An integer indicating the channel to look up.

Returns: mini - A double containing the minimum of the given dataset.

Return Type: double

function **getZeroDataPoint**(chan, index)

Returns a value from the zeroData member variable.

Parameters

- **chan** - An integer indicating the channel to access.
- **index** - An integer indicating which index to read from.

Returns: value - A double containing the data at the specified index.

Return Type: double

function **getChannels**()

Returns the channel member variable.

Parameters

Returns: channel - An integer indicating the number of channels.

Return Type: int

function **getChannelSize**(chan)

Returns the size of a specific channel.

Parameters

- **chan** - An integer indicating which channel's size to look up.

Returns: cSize - An integer indicating the size of the specified channel.

Return Type: int

function **getFrames**()

Returns the frames member variable.

Parameters

Returns: frames - An integer indicating the number of Frames.

Return Type: int

function **getCSize**(chan)

Returns the size of the cepstrumData member variable by specific channel.

Parameters

- **chan** - An integer indicating the channel to look up

Returns: cSize - An integer indicating the size of the channel's cepstrumData.

Return Type: int

function **getLeftSize**()

Returns the size of the leftChannel member variable.

Parameters:

Returns: lSize - An integer indicating the size of the leftChannel member variable.

Return Type: int

function **getLeftFFTSize**()

Return the size of the left FFT.

Parameters:

Returns: size - An integer indicating the size of the left FFT

Return Type: int

function **getRightSize**()

Returns the size of the rightChannel member variable.

Parameters:

Returns: rSize - An integer indicating the size of the rightChannel member variable.

Return Type: int

function **getRightFFTSize**()

Return the size of the right FFT.

Parameters:

Returns: size - An integer indicating the size of the right FFT

Return Type: int

function **getSCSize**()

Returns the size of the spectrumCData member variable.

Parameters:

Returns: sSize - An integer indicating the size of the spectrumData member variable.

Return Type: int

function **getSFSize**()

Returns the size of the spectrumFData member variable.

Parameters:

Returns: sSize - An integer indicating the size of the spectrumFData member variable.

Return Type: int

function **getZSize**(chan)

Returns the size of the zeroCrossData member variable.

Parameters

- **chan** - An integer indicating the channel to access.

Returns: zSize - An integer indicating the size of the zeroCrossData member variable.

Return Type: int

Fold

Private Member Variables

vector<vector<double> > **dataFolds**:

An N-dimensional vector of doubles that holds a fold of data.

Public Member Functions

Object Manipulation Functions

function **fold**()

A class constructor that creates a Fold object.

function **~fold**()

A class destructor that cleans up a Fold object.

Update Functions

function **pushFold**(data)

Adds new data to the Fold object by pushing it onto the dataFolds member variable.

Parameters

- **data** - A vector of doubles containing the data to be added.

Returns:

Return Type: void

Get Functions

function **getFold**(index)

Returns a fold from the dataFold member variable

Parameters

- **index** - An integer specifying the fold to retrieve.

Returns: fold - A 2D vector representing a fold of Data.

Return Type: vector<double>

function **getSize**()

Returns the size of the fold as an integer.

Parameters

Returns: size - An integer indicating the size of the Fold.

Return Type: int

Function Reference

ANN

Main Neural Net Functions

function **ANNI**(trainPath, testPath, folds, learnRate, epochs, codebooks, alg, channels, resultsPath, debug)

ANNI is the implementation of an artificial neural network (ANN) that is being used to analyze and classify audio files by musical genre.

Parameters

- **trainPath** - A string indicating the path for training data
- **testPath** - A string indicating the test data location.
- **folds** - An integer indicating the number of folds to create from the wave object.
- **learnRate** - A double indicating the learning rate
- **epochs** - An integer indicating the number of epochs to train over.
- **codebooks** - An integer indicating number of books
- **alg** - An integer indicating which algorithm to run ANNI on.
- **channels** - An integer describing the number of channels in the audio file.
- **resultsPath** - A string containing the path for output files.
- **debug** - A boolean flag that controls the debug output.

Returns:

Return Type: void

function **getBestMatch**(knownData, testRow, fileName, path, debug)

Finds the best matching genre for a new audio file by performing a comparison against a database of known files. This is an overloaded function.

Parameters

- **knownData** - Either a vector of floats or a vector of doubles containing the known dataset for use in the comparison.
- **testRow** - Either a vector of floats or a vector of doubles containing the data to be analyzed.
- **fileName** - A string containing the name of the output file
- **path** - A string containing the path for output files.
- **debug** - A boolean flag that controls the debug output.

Returns: match - An integer describing the category the testRow best matches.

Return Type: int

function **learningVectorQuantization**(trainSet, samples, BMUNames,
sampleNames, algorithm, currChan,
codeBooks, learnRate, epochs,
fileName, path, debug)

Determines the effectiveness of a training set, and makes a predictive match based on the trained data.

Parameters

- **trainSet** - An n-dimensional vector of doubles representing the training set.
- **samples** - An n-dimensional vector of doubles representing the sample set.
- **BMUNames** - A vector of strings containing the names for the BMU options
- **sampleNames** - A vector of strings containing the names of the samples
- **algorithm** - A string containing the algorithm being used
- **currChan** - An integer indicating the current channel.
- **codebooks** - An integer indicating the number of codebooks to use for training and analysis.
- **learnRate** - A double indicating the learning rate to apply to the algorithm.
- **epochs** - An integer indicating the number of epochs to train over.
- **fileName** - A string indicating the name of the output file.
- **path** - A string containing the path for output files.
- **debug** - A boolean flag that controls the debug output.

Returns:

Return Type: void

function **trainCodeBooks**(trainSet, &codeBookSet, nBooks, lRate, epochs, fileName, path, debug)

Generates a user specified number of codebooks from a set of known data. These codebooks will be used in the matching algorithm.

Parameters

- **trainSet** - An n-dimensional vector of doubles containing the known data for generating the training set.
- **&codeBookSet** - An n-dimensional vector that will store the training set.
- **nBooks** - An integer describing the number of codebooks to generate.
- **lRate** - A double describing the learning rate to use during training.
- **epochs** - An integer describing the number of learning generations
- **fileName** - A string containing the name of the output file.
- **path** - A string containing the path to the output directory
- **debug** - A boolean flag that controls the debug output.

Returns:

Return Type: void

Helper Functions

function **euclideanDistance**(fileName, row1, row2, debug)

Calculates the euclidean distance between row1 and row2.

Parameters

- **fileName** - A string containing the name of the output file.
- **row1** - A vector of floats containing the first row
- **row2** - A vector of floats containing the second row
- **debug** - A boolean flag that controls the debug output.

Returns: distance - A double containing the euclidean distance between the rows.

Return Type: double

function **lvqHelper**(algorithm, testFile, resultsOutput, folds, learnRate, epochs, codeBooks, channels, path, debug)

Parameters

- **algorithm** - A string containing the name of the algorithm being run against.
- **testFile** - A string indicating the name of the file to test
- **resultsOutput** - A string containing the full path to the output file.
- **folds** - An integer indicating the number of folds to create from the wave object.
- **learnRate** - A double indicating the learning rate to apply to the algorithm.
- **epochs** - An integer indicating the number of epochs to train over.
- **codebooks** - An integer indicating the number of codebooks to use for training and analysis.
- **channels** - An integer describing the number of channels in the audio file.
- **path** - A string containing the path for output files.
- **debug** - A boolean flag that controls the debug output.

Returns:

Return Type: void

function **prepareFolds**(folding, folds, alg, currChan, channels, &folded, path, debug)

Breaks a given dataset into the correct number of folds for analysis.

Parameters:

- **folding** - A boolean flag indicating a fold is being prepared
- **folds** - An integer indicating the number of folds to create
- **alg** - An integer specifying the algorithm being worked on
- **currChan** - An integer specifying the current channel being manipulated.
- **channels** - An integer indicating the number of channels in the audio file.
- **&folded** - An n-dimensional vector of doubles that will hold the folded vector
- **path** - A string containing the path to the output directory
- **debug** - A boolean flag that controls the debug output.

Returns:

Return Type: void

function **randomDatabase**(database, &trainSet, path, debug)

Parameters

- **database** - An n-dimensional vector of doubles containing known data points for generating the training set.
- **&trainSet** - An n-dimensional vector of doubles that will hold the randomly generated training set.
- **path** - A string containing the path to the output directory
- **debug** - A boolean flag that controls the debug output.

Returns:

Return Type: void

audioHandler

function **convertSound**(&wave, fileName, audioDir, sanName,
channels, fullPrecision, path,
debug)

Takes an audio file and converts it to a numerical representation of the waves.

Parameters

- **&wave** - An AudioWave object.
- **fileName** - A string indicating the audio file to load.
- **audioDir** - A string indicating the path to the audio file directory.
- **sanName** - A string indicating the name of the audio file without path information.
- **channels** - An integer indicating the number of channels in the audio file.
- **fullPrecision** - A boolean flag specifying the precision of the output.
- **path** - A string indicating the path for output files.
- **debug** - A boolean flag that controls debug output.

Returns:

Return Type: void

function **loadAudio**(&wave, fileName, audioDir, sanName, channels, fullPrecision, path, debug)

Wrapper function for convertSound

Parameters

- **&wave** - An AudioWave object.
- **fileName** - A string indicating the audio file to load.
- **audioDir** - A string indicating the path to the audio file directory.
- **sanName** - A string indicating the name of the audio file without path information.
- **channels** - An integer indicating the number of channels in the audio file.
- **fullPrecision** - A boolean flag specifying the precision of the output.
- **path** - A string indicating the path for output files.
- **debug** - A boolean flag that controls debug output.

Returns:

Return Type: void

cepstrum

function **rCepstrum**(x)

Perform the cepstrum segmentation algorithm on the given input in accordance to the real cepstrum equation.

Parameters

- **x** - A vector of complex doubles describing an audio wave.
- **windowSize** - Size of hamming window.

Returns: *vector* containing the results

Return Type: vector<double>

function **windowHamming**(n)

Creates a hamming window to be used by the cepstrum algorithm.

Parameters

- **n** - A vector of numbers to be used for the window

Returns: windowSignal - A vector of numbers describing the Window

Return Type: vector<complex<double> >

FourierTransform

function **fft**(&wave, save, fileName, path, debug)

A C++ implementation of the Cooley-Tukey Fast Fourier Transform (FFT) algorithm. Fourier transformations are used primarily in signal processing to indicate the frequency in a signal, and its proportion throughout said signal.

Parameters

- **&wave** - An AudioWave object.
- **save** - A boolean flag specifying whether to save data to file.
- **fileName** - A string indicating the file for data output.
- **path** - A string indicating the path for output files.
- **debug** - A boolean flag that controls the debug output.
-

Returns:
Return Type: void

function **inverseFT**(&x, fileName, debug)

Regenerates the audio file based on wave input.

Parameters

- **&x** - A vector of complex doubles representing the fft of an audio file. Must be passed by reference.
- **fileName** - A string containing the name of the output file.
- **debug** - A boolean flag that controls the debug output

Returns:
Return Type: void

spectrumCentroid

function **spectralCentroid**(&wave, fileName, path, debug)

Calculates the spectral centroid between each frame of a given wave.

Parameters

- **&wave** - An AudioWave object
- **fileName** - A string indicating the file for data output.
- **path** - A string indicating the path for output files.
- **debug** - A boolean flag that controls debug output.

Returns:

Return Type: void

spectrumFlux

function **spectralFlux**(&wave, fileName, path, debug)

Calculates the spectral flux between each frame of a given audio file.

Parameters

- **&wave** - An AudioWave object.
- **fileName** - A string indicating the file for data output.
- **path** - A string indicating the path for output files.
- **debug** - A boolean flag that controls debug output.

Returns:

Return Type: void

zeroCross

function **zeroCross**(&wave, fileName, path, debug)

Calculates the zero cross of a given audio file.

Parameters

- **&wave** - An AudioWave object.
- **fileName** - A string indicating the file for data output.
- **path** - A string indicating the path for output files.
- **debug** - A boolean flag that controls debug output.

Returns:

Return Type: void

Utilities

function **createString**(data, fieldWidth)

Generates a string from a given input. This function is Overloaded.

Parameters

- **data** - An integer, double, or boolean to be converted
- **fieldWidth** - An integer specifying the width of the data field.

Returns: newString - A string containing the converted data

Return Type: string

function **graphAlg**(wave. filePrefix, alg, fileName, path, debug)

Plots the results of an algorithm to file.

Parameters

- **wave** - An AudioWave object
- **filePrefix** - A string indicating the prefix for the plot file.
- **alg** - An integer indicating the algorithm to graph.
- **filName** - A string indicating the name of the output file.
- **path -** A string containing the path for output files
- **debug** - A boolean flag that controls debug output.

Returns:

Return Type: void

function **generateScript**(title, xlabel, ylabel, outFileName, sourceFile, channel)

Automates the generation of a gnuplot script file.

Parameters

- **title** - A string containing the title of the graph.
- **xlabel** - A string containing the label for the x-axis.
- **ylabel** - A string containing the label for the y-axis
- **outFileName** - A string specifying the name of the file to output the graph to.
- **sourceFile** - A string specifying the name of the source data file.
- **channel** - An integer specifying which audio channel is being graphed.

Returns:

Return Type: void

function **genTrainSet**(source, &sink, exclude)

Parameters

- **source** - An n-dimensional vector of doubles containing the source for training
- **&sink** - An n-dimensional vector of doubles that will hold the training set
- **exclude** - An integer indicating the index to skip when creating the set.

Returns:

Return Type: void

function **normalize**(data, &normals, frames, channel, path, debug)

Normalizes a vector.

Parameters

- **data** - A vector of complex doubles describing the audio wave
- **&normals** - A vector of doubles that will contain the normalized vector. Must be passed by reference.
- **frames** - An integer specifying the number of frames to break the data into.
- **channel** - An integer specifying the channel that is being normalized.
- **path** - A string containing the path for output files.
- **debug** - A boolean flag that controls the debug output

Returns:

Return Type: void

function **plotter**(sourceFile, plotFileName, graphType, alg, channel, path)

Graph a given data file.

Parameters

- **sourceFile** - A string containing the name of the file to plot
- **plotFileName** - A string containing the name of the file to save the plot to.
- **graphType** - An integer denoting the type of graph to create.
- **alg** - An integer specifying the algorithm that called the plotter.
- **channel** - An integers specifying the channel being plotted.
- **path** - A string containing the path for output files.

Returns:

Return Type: void

function **printer**(fileName, value, algo, begin, end)

Formats and outputs text to a file.

Parameters

- **fileName** - A string containing the name of the output file.
- **value** - A string to be added to the output file
- **algo** - An integer specifying the algorithm that called the printer
- **begin** - An integer describing the beginning of the printed range
- **end** - An integer describing the end of the printed range

Returns:

Return Type: void

function **realify**(wave, &reals, outputFile, path, debug)

Parameters

- **wave** - An AudioWave object containing the audiowave being analyzed
- **&reals** - A 2D vector of doubles that will contain the FFT magnitudes
- **outputFile** - A string describing the name of the file for output.
- **path** - A string containing the path for output files.
- **debug** - A boolean flag that controls the debug output

Returns:

Return Type: void

function **timestamp**()

Returns the current system time

Parameters:

Return: currentTime - A string containing the current system timestamp.

Return Type: string

function **fileExists**(fileName)

Determines the existence of a file.

Parameters

- **fileName** - A string containing the name of the file to check.

Returns: boolean value denoting existence of file

Return Type: bool

function **sortDist**(v1, v2)

Sorts a list of vectors from greatest to least euclidean Distance.

Parameters

- **v1** - The first vector to sort
- **v2** - The second vector to sort

Returns: isSorted - a boolean declaring the success of the function.

Return Type: bool

function **sign**(test)

Determines the sign of a given number.

Parameters

- **test** - A double containing the number to test.

Returns: result - An integer specifying the sign of the input.

Return Type: int