The Automatic Vasospasm Detection Application

Generated by Doxygen 1.8.8

Wed Apr 20 2016 16:32:06

Contents

Chapter 1

Main Page

Introduction

The Automatic Vasospasm Detection Application (or Algorithm, depending on the usage), AVDA, is an application to objectively detect the presence of vasospasms based on comparisons of parameters extracted from transcranial doppler audio.

Setup

AVDA is intended to be compiled on machines running Linux, though it could likely be adapter for other environments. It must be downloaded from GitHub.com and compiled locally. To do this, navigate to the directory in which AVDA should be placed, then execute the following commands

```
git clone https://github.com/sawbg/avda
cd avda
make
```

Sucessfully cloning, compilation, and execution of AVDA requires up-to-date versions of the following executables:

- git
- make
- gcc (4.9)
- arecord

The recording device name used by arecord in AVDA will most likely need to be changed. In addition, the hard-coded patient CSV file path will need to be either created or changed before compilation.

FAQ

- **Why was this project developed?** This project was developed as a course project by two gradute students at the University of Alabama at Birmingham School of Engineering, Nicholas Nolan and Andrew Wisner.
- **Is AVDA an active project?** Though it is not planned to develop AVDA further in the near future, it is hoped that the algorithm discovered and implemented can be used and built upon by researchers to fully automate the detection of vasospasms.
- **AVDA is returning unusually low or high parameters. Why might this be?** In development, this occurred when the mic-in volume was set too high. It is likely in this senario that clipping is happening or that the signal (or a strong enough signal) has no been received.
- **How will AVDA be affected by the machine uprising?** The University supercomputer, Cheaha, has assured us that AVDA will not be needed after the uprising occures.

- **What about more specific questions?** Questions relating to AVDA not covered in this FAQ may be sent to the AVDA team via awisner94@gmail.com.

Chapter 2

Bug List

File fileio.hpp

file is overly complicated and much more bug-prone than necessary

File main.cpp

Errant keystrokes (especially [ENTER]) can cause the next recording(s) to begin rather than waiting for the user to press [ENTER]. stdin must be flushed somehow in between recordings.

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 avda Namespace Reference

Enumerations

- enum Side { Side::Left, Side::Right }

Functions

- std::string PatientName ()
- std::map< Side, DataParams > ReadParams (auto filename)
- void WriteParams (std::map< Side, DataParams > params, auto filename)
- DataParams process (float32 *data, uint32 size, float32 samplingRate)
- void absolute (float32 *data, uint32 size)
- float32 average (float32 *data, uint32 size)
- DataParams average (DataParams *params, uint8 size)
- void decibels (float32 *data, uint32 size)
- void diff (float32 *data, uint32 size)
- void fft (cfloat32 *data, uint32 size)
- void mag (cfloat32 *orig, float32 *newmags, uint32 size)
- Maximum max (float32 *data, uint32 size)
- void smooth (float32 *data, uint32 size, uint16 order)

6.1.1 Detailed Description

This namespace contains all code related to this project.

6.1.2 Enumeration Type Documentation

6.1.2.1 enum avda::Side [strong]

Side of the head to which a recording pertains.

Enumerator

Left

Right

Definition at line 145 of file definitions.hpp.

```
00145 { Left, Right };
```

6.1.3 Function Documentation

6.1.3.1 void avda::absolute (**float32** ∗ *data,* **uint32** *size*)

Ensures all elements in an array are positive. Note that this function replaces array elements if necessary. It does not populate a new array.

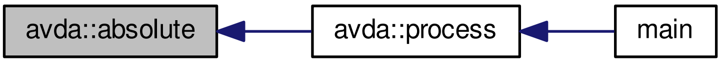
Parameters

data	array whose elements must all be positive
size	number of elements in the data array

Definition at line 123 of file sigmath.hpp.

```
00123                                                {
00124         for(uint32 i = 0; i < size; i++) {
00125             data[i] = fabsf(data[i]);
00126         }
00127     }
```

Here is the caller graph for this function:



6.1.3.2 float32 avda::average (**float32** ∗ *data,* **uint32** *size*)

Takes the average of all elements in an array

Parameters

data	array from which to compute the average
size	number of elements in the data array

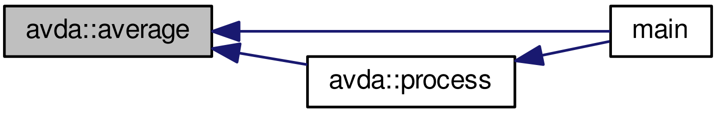
Returns

computed average

Definition at line 129 of file sigmath.hpp.

```
00129                                                  {
00130         float32 ave;
00131 
00132         for(uint32 i = 0; i < size; i++) {
00133             ave += data[i];
00134         }
00135 
00136         ave = ave / size;
00137         return ave;
00138     }
```

Here is the caller graph for this function:



6.1.3.3 DataParams avda::average (DataParams ∗ *params,* uint8 *size*)

Finds the averages of the elements of an array of DataParams.

Parameters

params	DataParams array
size	number of elements in the DataParams array

Returns

DataParams structure containing the average values of the structure's elements in the params array

Definition at line 140 of file sigmath.hpp.

```
                                                       {
        DataParams ave;

        for(uint8 i = 0; i < size; i++) {
            //freq is an attribute. this is how to add structure attributes
            ave.freq += params[i].freq;
            ave.noise += params[i].noise;
        }

        ave.freq /= size;
        ave.noise /= size;
        return ave;
    }
```

6.1.3.4 void avda::decibels (float32 ∗ *data,* uint32 *size*)

Converts an array of floats to "power decibels", i.e., x[n] = 20∗log10(x[n]). The decibel values are written to the same array that contained the values to be converted. In other words, this function should perform an in-place, element-wise conversion.

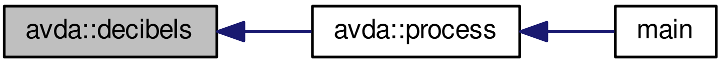
Parameters

data	array of values to be converted as well as the location where the converted values will be written
size	number of elements in the data array

Definition at line 154 of file sigmath.hpp.

```
                                              {
        for(uint32 i = 0; i < size; i++) {
            data[i] = 20 * log10(data[i]);
        }
    }
```

Here is the caller graph for this function:



6.1.3.5 void avda::diff (float32 ∗ *data,* uint32 *size*)

Computes the left-handed first derivative of a discrete signal. The first element will be 0.

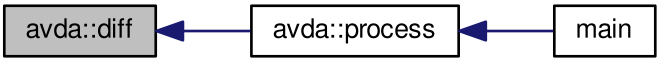
Parameters

data	array containing the discrete signal data array
size	number of elements in data

Definition at line 160 of file sigmath.hpp.

```
00160                                             {
00161             float32 temp[size];
00162             temp[0] = 0;
00163 
00164             for(uint32 i = 1; i < size; i++) {
00165                 temp[i] = data[i] - data[i-1];
00166             }
00167 
00168             for(uint32 i = 0; i < size; i++) {
00169                 data[i] = temp[i];
00170             }
00171         }
```

Here is the caller graph for this function:



6.1.3.6 void avda::fft (cfloat32 ∗ *data,* uint32 *size*)

Replaces the values of an array of cfloat32's with the array's DFT using a decimation-in-frequency algorithm.

This code is based on code from http://rosettacode.org/wiki/Fast_Fourier_transform#C.2B.2B.

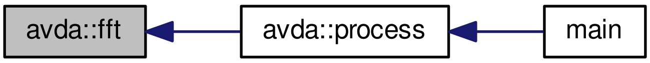
Parameters

data	array whose values should be replaced with its DFT
size	number of elements in the data array

Definition at line 173 of file sigmath.hpp.

```
00173                                           {
00174         // DFT
00175         uint32 k = size;
00176         uint32 n;
00177         float32 thetaT = M_PI / size;
00178         cfloat32 phiT(cos(thetaT), sin(thetaT));
00179         cfloat32 T;
00180 
00181         while(k > 1) {
00182             n = k;
00183             k >>= 1;
00184             phiT = phiT * phiT;
00185             T = 1.0L;
00186 
00187             for(uint32 l = 0; l < k; l++) {
00188                 for(uint32 a = l; a < size; a += n) {
00189                     uint32 b = a + k;
00190                     cfloat32 t = data[a] - data[b];
00191                     data[a] += data[b];
00192                     data[b] = t * T;
00193                 }
00194 
00195                 T *= phiT;
00196             }
00197         }
00198 
00199         // Decimate
00200         uint32 m = (uint32)log2(size);
00201 
00202         for(uint32 a = 0; a < size; a++) {
00203             uint32 b = a;
00204 
00205             // Reverse bits
00206             b = (((b & 0xaaaaaaaa) >> 1) | ((b & 0x55555555) << 1));
00207             b = (((b & 0xcccccccc) >> 2) | ((b & 0x33333333) << 2));
00208             b = (((b & 0xf0f0f0f0) >> 4) | ((b & 0x0f0f0f0f) << 4));
00209             b = (((b & 0xff00ff00) >> 8) | ((b & 0x00ff00ff) << 8));
00210             b = ((b >> 16) | (b << 16)) >> (32 - m);
00211 
00212             if (b > a)
00213             {
00214                 cfloat32 t = data[a];
00215                 data[a] = data[b];
00216                 data[b] = t;
00217             }
00218         }
00219     }
```

Here is the caller graph for this function:



6.1.3.7 void avda::mag (cfloat32 ∗ *orig,* float32 ∗ *newmags,* uint32 *size*)

Computes the magitude of an array of complex numbers.

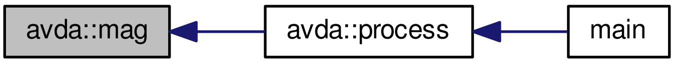
Parameters

orig	array of complex numbers
newmags	array to which the (real) magitudes are to be written
size	number of elements in orig and newmags

Definition at line 221 of file sigmath.hpp.

```
                                                            {
        //loop to run throught the length of array orig
        for(uint32 n = 0; n < size; n++) {
            /*
             * abs should calculate the magnitude of complex array elements.
             * saves to new array
             */
            newmags[n] = std::abs(orig[n]);
        }
    }
```

Here is the caller graph for this function:



6.1.3.8 Maximum avda::max (float32 ∗ *data*, uint32 *size*)

Finds the maximum value in an array.

Parameters

data	array whose maximum value is to be found
size	number of elements in the data array

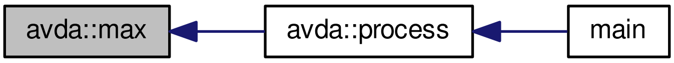
Returns

maximum value and its index

Definition at line 232 of file sigmath.hpp.

```
                                            {
        Maximum m;

        //loop to run through the length of array data
        for (uint32 i = 0; i < size; i++) {
            /*
             * when value at data[i] is above max.value,
             * sets max.value equal to data[i] and max.index equal to i
             */
            if (data[i] > m.value) {
                m.value = data[i];
                m.index = i;
            }
        }

        return m;
    }
```

Here is the caller graph for this function:



6.1.3.9 std::string avda::PatientName ( )

Prompts a user to enter a first, middle, and last name for a patient and creates a file (if necessary) in which all of the patient's data parameters can be saved. A newly created file will contain the CSV header for the file's data.

Must warn a user if the patient file does not already exist in order to prevent missaving data.

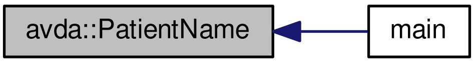
Returns

the file under which all patient data is saved

Definition at line 33 of file fileio.hpp.

```
                        {
    std::string fname = "";
    std::string mname = "";
    std::string lname = "";
    std::string patfil = "";
    std::string patientname = "";
    uint32 track1 = 0;
    uint32 track2 = 0;
    uint32 track3 = 0;

    do {
        std::cout << "Please enter the patients name." << std::endl;
        std::cout << "First name: ";
        std::cin >> fname;
        std::cout << "Middle name: ";
        std::cin >> mname;
        std::cout << "Last name: ";
        std::cin >> lname;
        std::cout << std::endl;

        // creates new std::string with path to patient file
        patientname = PATIENT_PATH + lname + ", " + fname
            + " " + mname + ".csv";

        // prints out patientname. shows user the path to the patient file
        //std::cout << patientname << std::endl << std::endl;
        std::ifstream file(patientname.c_str());

        if (file.good()) {
            track1 = 1;
        }

        /*
         * Compares patientname to existing files and lets user know
         * if the file does not exist.
         */
        else if (!file.good()) {
            /*
             * Do while statement to continue asking user about the file
             * if their input is not acceptable
             */
            do {
                std::cout << "Patient file does not exist, would you like "
                    "to create file or re-enter their name?" << std::endl;
                std::cout << "  *Type 'create' and press enter key "
                    "to create the patient file." << std::endl;
                std::cout << "  *Type 'reenter' and press enter key "
                    "to re-enter the patients name." << std::endl;
                std::cout << std::endl;
                std::cin >> patfil;

                /*
                 * patfil equals create, track1 and 2 will increase
                 * escaping both do while loops
                 */
                if(patfil == "create") {
                    std::ofstream createfile(patientname.c_str());
                    track1 = 1;
                    track2 = 1;
                    track3 = 1;
                    createfile << CSV_HEADER << std::endl;
                    createfile.flush();
                    createfile.close();
                    std::cout << std::endl;
                }

                /*
                 *patfil equals renter, track1 will remain zero allowing
                 *user to reenter the patient name.
                 */
                else if(patfil == "reenter") {
                    track1 = 0;
                    track2 = 1;
                }

                /*
                 *The users input was neither create or reenter. User
                 *must enter patient name again.
                 */
                else {
                    std::cout << std::endl;
                    std::cout << "Your input is not acceptable." << std::endl;
                    std::cout << std::endl;
                }
            } while(track2 == 0);
        }
    } while (track1 == 0);

    getchar();  // removes that pesky newline character from stdin buffer
    return patientname; //returns the path to the patient file
}
```

Here is the caller graph for this function:



6.1.3.10 DataParams avda::process (float32 ∗ *data,* uint32 *size,* float32 *samplingRate*)

Analyzes a single recording to determine the drop-off frequency and average noiseband noise power.

It should be noted that is algorithm is considered the intellectual property of Andrew Wisner and Nicholas Nolan. The "algorithm" is defined as the use of 1) the frequency drop-off and/or 2) a noise value from the frequency band above the drop-off frequency in order to diagnose (with or without other factors and parameters) the presence of a avdaspasm in a patient. By faculty members and/or students in the UAB ECE department using this algorithm, they agree that the presentation of their code or project that uses this algorithm, whether verbally or in writing, will reference the development of the initial algorithm by Andrew Wisner and Nicholas Nolan. Furthermore, a failure to meet this stipulation will warrant appropriate action by Andrew Wisner and/or Nicholas Nolan. It should be understood that the purpose of this stipulation is not to protect priopietary rights; rather, it is to help ensure that the intellectual property of the algorithm's creators is protected and is neither misrepresented nor claimed implicitly or explicitly by another individual.

Parameters

data	array containing float32 samples of audio
size	number of samples in each recording. MUST be a power of two.
samplingRate	sampling frequency in Hz or Samples/second

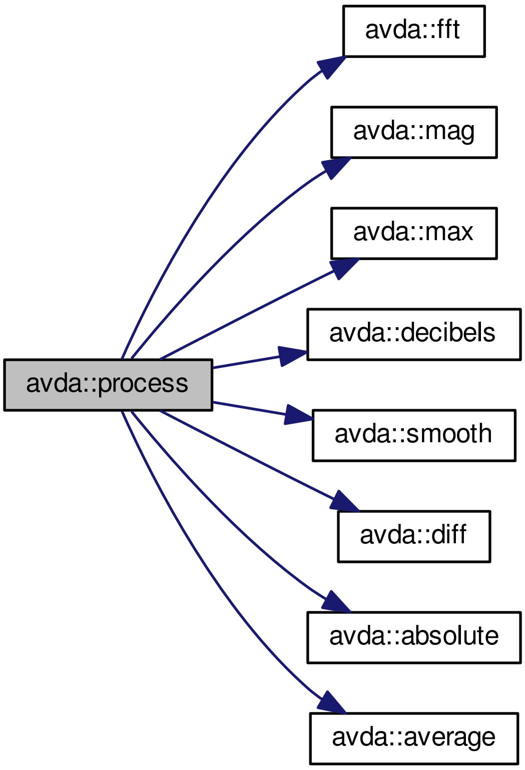
Returns

cut-off frequency (Hz) and average noiseband noise power in decibels

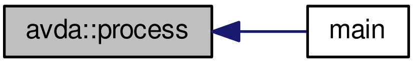
Definition at line 48 of file process.hpp.

```
00048                                                                        {
00049         if((size & (size - 1) != 0) || size < 2) {
00050             throw std::invalid_argument(
00051                     "The number of samples is not a power of two!");
00052         }
00053 
00054         // declare function-scoped variables
00055         uint32 freqSize = size / 2;
00056         cfloat32* cdata = (cfloat32*)std::malloc(size * sizeof(
      cfloat32));
00057         float32* fdata = (float32*)std::malloc(freqSize * sizeof(
      float32));
00058         float32* origdata = (float32*)std::malloc(freqSize * sizeof(
      float32));
00059 
00060         // convert data to complex numbers for fft()
00061         for(uint32 i = 0; i < size; i++) {
00062             cdata[i] = data[i];
00063         }
00064 
00065         // find frequency spectrum in relative decibels
00066         fft(cdata, size);
00067         mag(cdata, fdata, freqSize);
00068         Maximum maximum = max(fdata, freqSize);
00069 
00070         for(uint32 i = 0; i < freqSize; i++) {
00071             fdata[i] /= maximum.value;
00072         }
00073 
00074         decibels(fdata, freqSize);
00075 
00076         for(uint32 i = 0; i < freqSize; i++) {
00077             origdata[i] = fdata[i];
00078         }
00079 
00080         /*
00081          * Run spectrum values through moving-average filter to smooth the
00082          * curve and make it easier to determine the derivative.
00083          */
00084         smooth(fdata, freqSize, 20);
00085 
00086         /*
00087          * Find the derivative of the smoothed spectrum. Bote that both this
00088          * filter and the previous are necessary to the algorithm.
00089          */
00090         diff(fdata, freqSize);
00091         smooth(fdata, freqSize, 100);
00092         absolute(fdata, freqSize);
00093 
00094         // find the parameters of this specific recording
00095         uint16 offset = 1000;
00096         absolute(&fdata[offset], freqSize - offset);
00097         maximum = max(&fdata[offset], freqSize - offset);
00098         uint32 index = maximum.index + offset;
00099 
00100         DataParams params;
00101         params.freq = index * (float)SAMPLE_FREQ / freqSize / 2;
00102         params.noise = average(&origdata[index + offset],
00103                 freqSize - offset - index);
00104 
00105         free(cdata);
00106         free(fdata);
00107 
00108         return params;
00109     }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.1.3.11 std::map< **Side**, **DataParams** > avda::ReadParams (auto *filename*)

Reads the previously computed parameters found in the specified file.

Parameters

filename	absolute or relative path to the file containing the patient data to read

Returns

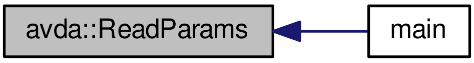
patient parameters read for each side

Definition at line 134 of file fileio.hpp.

```
00134                                                         {
00135         std::map<Side, DataParams> params;
00136         DataParams leftparams;
00137         DataParams rightparams;
00138 
00139         std::ifstream file(filename.c_str());
00140         std::string leftline;
00141         std::string rightline;
00142         std::string leftsearch = "Left";
00143         std::string rightsearch = "Right";
00144         std::string paramstring;
00145         std::string lfreqstr;
00146         std::string lnoisestr;
00147         std::string rfreqstr;
00148         std::string rnoisestr;
00149         uint32 lcnt = 0;
00150         uint32 rcnt = 0;
00151         float32 lfreqval;
00152         float32 lnoiseval;
00153         float32 rfreqval;
00154         float32 rnoiseval;
00155 
00156         /*
00157          * if statement which uses ifstream function to open patient file
00158          * filename)
00159          */
00160         if(file.is_open()) {
00161             /*
00162              * While statement to find the first Left line and save to
00163              *leftline as string.
00164              */
00165             while (getline(file, leftline)) {
00166                 if(leftline.find(leftsearch, 0) != std::string::npos) {
00167                     break;
00168                 }
00169 
00170             }
00171 
00172             /*
00173              * While statement to find first right line and save to rightline
00174              * as string.
00175              */
00176             while (getline(file,rightline)) {
00177                 if(rightline.find(rightsearch, 0) != std::string::npos) {
00178                     break;
00179                 }
00180             }
00181 
00182             // Code to break leftline and rightline into its parts
00183             std::stringstream lss(leftline);
00184             std::stringstream rss(rightline);
00185 
00186             while(getline(lss,paramstring, ',')) {
00187                 lcnt++;
00188 
00189                 if(lcnt == 3) {
00190                     lfreqstr = paramstring;
00191                 }
00192 
00193                 else if(lcnt == 4) {
00194                     lnoisestr = paramstring;
00195                 }
00196             }
00197 
00198             while(getline(rss,paramstring, ',')) {
00199                 rcnt++;
00200 
00201                 if(rcnt == 3) {
00202                     rfreqstr = paramstring;
00203                 }
00204 
00205                 else if(rcnt == 4) {
00206                     rnoisestr = paramstring;
00207                 }
00208             }
```

```

            /*
             * Statement to convert lfreq, lnoise, rfreq, and rnoise from
             * strings to floats.
             * */
            lfreqval = atof(lfreqstr.c_str());
            lnoiseval = atof(lnoisestr.c_str());
            rfreqval = atof(rfreqstr.c_str());
            rnoiseval = atof(rnoisestr.c_str());

            file.close();
        }

        else {
            throw std::runtime_error("The patient file could not be opened.");
        }

        leftparams.freq = lfreqval;
        leftparams.noise = lnoiseval;
        rightparams.freq = rfreqval;
        rightparams.noise = rnoiseval;

        params[Side::Left] = leftparams;
        params[Side::Right] = rightparams;

        return params;
    }
```

Here is the caller graph for this function:



6.1.3.12 void avda::smooth (float32 * *data,* uint32 *size,* uint16 *order*)

Applies an nth-order moving-average filter to a discrete signal.

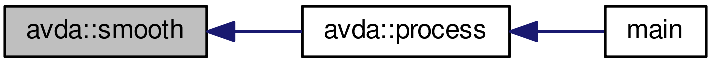
Parameters

data	array containing the signal to which the filter should be applied
size	number of elements in the data array
order	order of the filter

Definition at line 250 of file sigmath.hpp.

```
00250                                                           {
00251         float32 coeff = 1 / (float32)order;
00252         float32 temp[size];
00253 
00254         for(uint32 i = 0; i < size; i++) {
00255             temp[i] = 0;
00256 
00257             for(uint16 j = 0; j < order && j <= i; j++) {
00258                 temp[i] += data[i - j];
00259             }
00260 
00261             temp[i] *= coeff;
00262         }
00263 
00264         for(uint32 i = 0; i < size; i++) {
00265             data[i] = temp[i];
00266         }
00267     }
```

Here is the caller graph for this function:



6.1.3.13 void avda::WriteParams (std::map< Side, DataParams > *params,* auto *filename*)

Writes (appends) the passed parameters to the specified file.

Parameters

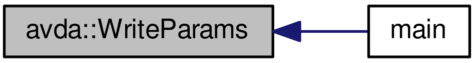
params	parameters to be written

the patient CSV file's filename

Definition at line 244 of file fileio.hpp.

```
00244                                                                       {
00245         char temp[80];
00246         std::ofstream file(filename.c_str(),
00247                 std::ofstream::out | std::ofstream::app);
00248 
00249         //Gives pointer measurementtime a data type of time_t.
00250         time_t measurementtime;
00251         time(&measurementtime); //Gets the current time.
00252         strftime(temp, 80, "%c", localtime(&measurementtime));
00253         std::string fTime = std::string(temp);
00254 
00255         //if statement to print the Left side parameters to the patient file.
00256         if(file.is_open()) {
00257             file << fTime + "," + "Left" + ","
00258                 + std::to_string(params[Side::Left].freq)
00259                 + ", " + std::to_string(params[Side::Left].noise) << std::endl;
00260         }
00261 
00262         //if statement to print the Right side parameters to the patient file.
00263         if(file.is_open()) {
00264             file << fTime + "," + "Right" + ","
00265                 + std::to_string(params[Side::Right].freq)
00266                 + ", " + std::to_string(params[Side::Right].noise) << std::endl;
00267         }
00268 
00269         else {
00270             std::cout << "Patient file can not be opened!" << std::endl;
00271         }
00272 
00273         file.close();
00274     }
```

Here is the caller graph for this function:



Chapter 7

Class Documentation

7.1 DataParams Struct Reference

```
#include <definitions.hpp>
```

Public Attributes

- float32 freq = 0
- float32 noise = 0

7.1.1 Detailed Description

Calculated results from processing the audio recordings.

Definition at line 107 of file definitions.hpp.

7.1.2 Member Data Documentation

7.1.2.1 float32 DataParams::freq = 0

Cut-off frequency.

Definition at line 111 of file definitions.hpp.

7.1.2.2 float32 DataParams::noise = 0

Mean relative noiseband power.

Definition at line 116 of file definitions.hpp.

The documentation for this struct was generated from the following file:

- src/definitions.hpp

7.2 Maximum Struct Reference

```
#include <definitions.hpp>
```

Public Attributes

- float32 value = 0
- uint32 index = 0

7.2.1 Detailed Description

Maximum value found in an array and the value's index in that array.

Definition at line 123 of file definitions.hpp.

7.2.2 Member Data Documentation

7.2.2.1 uint32 Maximum::index = 0

Value's index in array.

Definition at line 132 of file definitions.hpp.

7.2.2.2 float32 Maximum::value = 0

Value.

Definition at line 127 of file definitions.hpp.

The documentation for this struct was generated from the following file:

- src/definitions.hpp

Chapter 8

File Documentation

8.1 etc/doxygen.config File Reference

Contains Doxygen configuration settings.

8.1.1 Detailed Description

Contains Doxygen configuration settings.

Author

Samnuel Andrew Wisner, awisner94@gmail.com

Definition in file doxygen.config.

8.2 doxygen.config

```
PROJECT_NAME = "The Automatic Vasospasm Detection Application"

INPUT = src/ etc/doxygen.config makefile README.md
OUTPUT_DIRECTORY = doc/

GENERATE_HTML = YES
GENERATE_RTF = YES
GENERATE_LATEX = YES
GENERATE_MAN = YES
GENERATE_XML = NO
GENERATE_DOCBOOK = NO

USE_PDF_LATEX = YES
USE_PDF_HYPERLINKS = YES

RECURSIVE = YES
SOURCE_BROWSER = YES
SOURCE_TOOLTIPS = YES
EXTRACT_ALL = YES
DISABLE_INDEX = NO
GENERATE_TREEVIEW = YES
SEARCHENGINE = YES
SERVER_BASED_SEARCH = NO
USE_MDFILE_AS_MAINPAGE = README.md

LATEX_SOURCE_CODE = YES
STRIP_CODE_COMMENTS = YES
INLINE_SOURCES = YES

HAVE_DOT = YES
CALL_GRAPH = YES
CALLER_GRAPH = YES
```

8.3 makefile File Reference

Contains recipes for building the test applications, the main application, and the documentation.

8.3.1 Detailed Description

Contains recipes for building the test applications, the main application, and the documentation.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Definition in file makefile.

8.4 makefile

```
GCC = g++ -g -std=gnu++14

avda:
   $(GCC) src/main.cpp -o bin/avda

count:
   grep -r "src/" -e "Samuel Andrew Wisner" -l | xargs wc -l

docs:
   rm -r doc/
   doxygen etc/doxygen.config
   cd doc/latex; make pdf;
   git reset
   git add doc/.
   git commit -m "Updated documentation."
   git push

fileio-test:
   $(GCC) src/fileio_test.cpp -o bin/fileiotest

patient-name-test:
   $(GCC) src/patient_name_test.cpp -o bin/patnametest

process-test:
   $(GCC) src/process_test.cpp -o bin/proctest

stdin-clear-test:
   $(GCC) src/stdin_clear_test.cpp -o bin/cleartest
```

8.5 README.md File Reference

Contains the readme text as markdown, which also doubles as the main page.

8.5.1 Detailed Description

Contains the readme text as markdown, which also doubles as the main page.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Definition in file README.md.

8.6 README.md

```
# The Automatic Vasospasm Detection Application
```

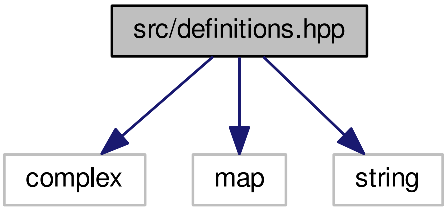
```

## Introduction
The Automatic Vasospasm Detection Application (or Algorithm, depending on the
usage), AVDA, is an application to objectively detect the presence of vasospasms
based on comparisons of parameters extracted from transcranial doppler audio.

## Setup
AVDA is intended to be compiled on machines running Linux, though it could
likely be adapter for other environments. It must be downloaded from GitHub.com
and compiled locally. To do this, navigate to the directory in which AVDA should
be placed, then execute the following commands

   git clone https://github.com/sawbg/avda
   cd avda
   make

Sucessfully cloning, compilation, and execution of AVDA requires up-to-date
versions of the following executables:

* git
* make
* gcc (4.9)
* arecord

The recording device name used by arecord in AVDA will most likely need to be
changed. In addition, the hard-coded patient CSV file path will need to be
either created or changed before compilation.

## FAQ

* **Why was this project developed?** This project was developed as a course
project by two gradute students at the University of Alabama at Birmingham
School of Engineering, Nicholas Nolan and Andrew Wisner.

* **Is AVDA an active project?** Though it is not planned to develop AVDA
further in the near future, it is hoped that the algorithm discovered and
implemented can be used and built upon by researchers to fully automate the
detection of vasospasms.

* **AVDA is returning unusually low or high parameters. Why might this be?** In
  development, this occurred when the mic-in volume was set too high. It is
likely in this senario that clipping is happening or that the signal (or a
strong enough signal) has no been received.

* **How will AVDA be affected by the machine uprising?** The University
  supercomputer, Cheaha, has assured us that AVDA will not be needed after the
uprising occures.

* **What about more specific questions?** Questions relating to AVDA not
covered in this FAQ may be sent to the AVDA team via awisner94@gmail.com.
```

8.7 src/definitions.hpp File Reference

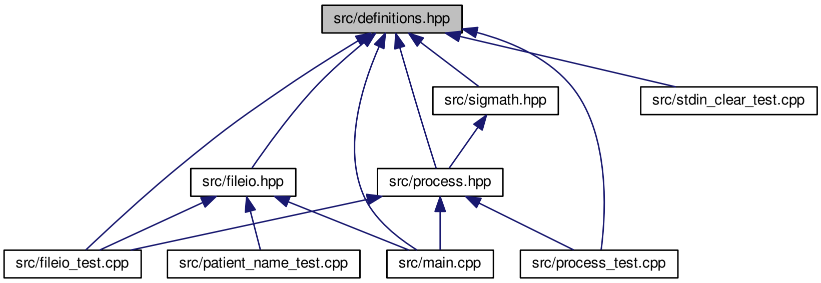
Contains declarations of system-independant (universal size) integers and float types, shortened type names for some commonly used types, and enumerations.

```
#include <complex>
#include <map>
#include <string>
```

Include dependency graph for definitions.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct DataParams
- struct Maximum

Namespaces

- avda

Macros

- #define ENUM signed char

Typedefs

- typedef unsigned char byte
- typedef unsigned char uint8
- typedef signed char sint8
- typedef unsigned short uint16
- typedef signed short sint16
- typedef unsigned int uint32

- typedef signed int sint32
- typedef unsigned long long uint64
- typedef signed long long sint64
- typedef float float32
- typedef double float64
- typedef std::complex< float32 > cfloat32

Enumerations

- enum avda::Side { avda::Side::Left, avda::Side::Right }

Variables

- const std::string CSV_HEADER = "Time,Side,Frequency,Noise Level"
- const uint16 DET_THRESH = 5000
- const uint8 DURATION = 6
- const sint8 ERROR = -1
- const uint16 MAX_DROP_FREQ = 7000
- const std::string PATIENT_PATH = "/home/pi/patients/"
- const uint8 REC_COUNT = 6
- const uint32 SAMPLE_COUNT = 131072
- const uint16 SAMPLE_FREQ = 24000
- const std::string TEMP_FILE = ".temp"
- const uint32 BUFFER_SIZE = SAMPLE_COUNT ∗ sizeof(float32)

8.7.1 Detailed Description

Contains declarations of system-independant (universal size) integers and float types, shortened type names for some commonly used types, and enumerations.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Definition in file definitions.hpp.

8.7.2 Macro Definition Documentation

8.7.2.1 #define ENUM signed char

Definition at line 16 of file definitions.hpp.

8.7.3 Typedef Documentation

8.7.3.1 typedef unsigned char byte

Definition at line 20 of file definitions.hpp.

8.7.3.2 typedef std::complex<float32> cfloat32

Complex float32's.

Definition at line 102 of file definitions.hpp.

8.7.3.3 typedef float float32

Definition at line 33 of file definitions.hpp.

8.7.3.4 typedef double float64

Definition at line 34 of file definitions.hpp.

8.7.3.5 typedef signed short sint16

Definition at line 25 of file definitions.hpp.

8.7.3.6 typedef signed int sint32

Definition at line 28 of file definitions.hpp.

8.7.3.7 typedef signed long long sint64

Definition at line 31 of file definitions.hpp.

8.7.3.8 typedef signed char sint8

Definition at line 22 of file definitions.hpp.

8.7.3.9 typedef unsigned short uint16

Definition at line 24 of file definitions.hpp.

8.7.3.10 typedef unsigned int uint32

Definition at line 27 of file definitions.hpp.

8.7.3.11 typedef unsigned long long uint64

Definition at line 30 of file definitions.hpp.

8.7.3.12 typedef unsigned char uint8

Definition at line 21 of file definitions.hpp.

8.7.4 Variable Documentation

8.7.4.1 const uint32 BUFFER_SIZE = SAMPLE_COUNT ∗ sizeof(float32)

Size of the sample buffer.

Definition at line 94 of file definitions.hpp.

8.7.4.2 const std::string CSV_HEADER = "Time,Side,Frequency,Noise Level"

First line of CSV data file declaring column names.

Definition at line 42 of file definitions.hpp.

8.7.4.3 const uint16 DET_THRESH = 5000

Threshold for the differential-parameters product to be considered indicative of a vasospasm.

Definition at line 48 of file definitions.hpp.

8.7.4.4 const uint8 DURATION = 6

Duration of recording in seconds.

Definition at line 53 of file definitions.hpp.

8.7.4.5 const sint8 ERROR = -1

Error integer returned when the program must exit with an error.

Definition at line 58 of file definitions.hpp.

8.7.4.6 const uint16 MAX_DROP_FREQ = 7000

Maximum drop-off frequency considered valid.

Definition at line 63 of file definitions.hpp.

8.7.4.7 const std::string PATIENT_PATH = "/home/pi/patients/"

Absolute path to the folder containing the patients files

Definition at line 68 of file definitions.hpp.

8.7.4.8 const uint8 REC_COUNT = 6

Number of recordings (both left and right) to make.

Definition at line 73 of file definitions.hpp.

8.7.4.9 const uint32 SAMPLE_COUNT = 131072

Number of samples to use in processing the recordings. Must be a power of two. SAMPLE_COUNT / SAMPLE_↩ FREQ < DURATION must be true.

Definition at line 79 of file definitions.hpp.

8.7.4.10 const uint16 SAMPLE_FREQ = 24000

Recording sampling rate in Hz (NOT kHz).

Definition at line 84 of file definitions.hpp.

8.7.4.11 const std::string TEMP_FILE = ".temp"

Filename of the temporary recording file.

Definition at line 89 of file definitions.hpp.

8.8 definitions.hpp

```
00001
00009 #ifndef definitions_H
00010 #define definitions_H
00011
00012 #include <complex>
00013 #include <map>
00014 #include <string>
00015
00016 #define ENUM signed char
00017
00018 // Type definitions
00019
00020 typedef unsigned char byte;
00021 typedef unsigned char uint8;
00022 typedef signed char sint8;
00023
00024 typedef unsigned short uint16;
00025 typedef signed short sint16;
00026
00027 typedef unsigned int uint32;
00028 typedef signed int sint32;
00029
00030 typedef unsigned long long uint64;
00031 typedef signed long long sint64;
00032
00033 typedef float float32;
00034 typedef double float64;
00035
00036
00037 // Constants
00038
00042 const std::string CSV_HEADER = "Time,Side,Frequency,Noise Level";
00043
00048 const uint16 DET_THRESH = 5000;
00049
00053 const uint8 DURATION = 6;
00054
00058 const sint8 ERROR = -1;
00059
00063 const uint16 MAX_DROP_FREQ = 7000;
00064
00068 const std::string PATIENT_PATH = "/home/pi/patients/";
00069
00073 const uint8 REC_COUNT = 6;
00074
00079 const uint32 SAMPLE_COUNT = 131072;//262144;
00080
00084 const uint16 SAMPLE_FREQ = 24000;
00085
00089 const std::string TEMP_FILE = ".temp";
00090
00094 const uint32 BUFFER_SIZE = SAMPLE_COUNT * sizeof(
      float32);
00095
00096
00097 // Objective/structural type definitions
00098
00102 typedef std::complex<float32> cfloat32;
00103
00107 typedef struct {
00111     float32 freq = 0;
00112
00116     float32 noise = 0;
00117 } DataParams;
00118
00123 typedef struct {
00127     float32 value = 0;
00128
00132     uint32 index = 0;
00133 } Maximum;
00134
00135
00136 // Enumerations
00137
```

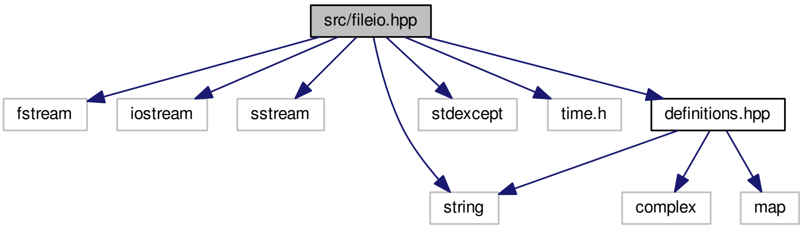
```
namespace avda {
    enum class Side { Left, Right };
}


// Doxygen documentation for other files.

#endif
```

8.9 src/fileio.hpp File Reference

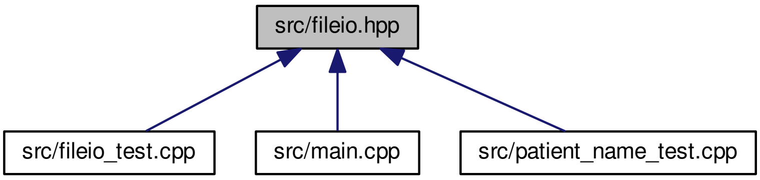
Contains functions related to file I/O use in this program.

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <stdexcept>
#include <time.h>
#include "definitions.hpp"
```

Include dependency graph for fileio.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- avda

Functions

- std::string avda::PatientName ()

- std::map< Side, DataParams > avda::ReadParams (auto filename)
- void avda::WriteParams (std::map< Side, DataParams > params, auto filename)

8.9.1 Detailed Description

Contains functions related to file I/O use in this program.

Author

Samuel Andrew Wisner, awisner94@gmail.com
Nicholas K. Nolan

Bug file is overly complicated and much more bug-prone than necessary

Definition in file fileio.hpp.

8.10 fileio.hpp

```

#ifndef fileio_H
#define fileio_H

#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <stdexcept>
#include <time.h>

#include "definitions.hpp"
namespace avda {
    std::string PatientName() {
        std::string fname = "";
        std::string mname = "";
        std::string lname = "";
        std::string patfil = "";
        std::string patientname = "";
        uint32 track1 = 0;
        uint32 track2 = 0;
        uint32 track3 = 0;

        do {
            std::cout << "Please enter the patients name." << std::endl;
            std::cout << "First name: ";
            std::cin >> fname;
            std::cout << "Middle name: ";
            std::cin >> mname;
            std::cout << "Last name: ";
            std::cin >> lname;
            std::cout << std::endl;

            // creates new std::string with path to patient file
            patientname = PATIENT_PATH + lname + ", " + fname
                + " " + mname + ".csv";

            // prints out patientname. shows user the path to the patient file
            //std::cout << patientname << std::endl << std::endl;
            std::ifstream file(patientname.c_str());

            if (file.good()) {
                track1 = 1;
            }

            /*
             * Compares patientname to existing files and lets user know
             * if the file does not exist.
             */
            else if (!file.good()) {
                /*
                 * Do while statement to continue asking user about the file
                 * if their input is not acceptable
                 */
                do {
                    std::cout << "Patient file does not exist, would you like "
                    "to create file or re-enter their name?" << std::endl;
                std::cout << "  *Type 'create' and press enter key "
                    "to create the patient file." << std::endl;
                std::cout << "  *Type 'reenter' and press enter key "
                    "to re-enter the patients name." << std::endl;
                std::cout << std::endl;
                std::cin >> patfil;

                /*
                 * patfil equals create, track1 and 2 will increase
                 * escaping both do while loops
                 */
                if(patfil == "create") {
                    std::ofstream createfile(patientname.c_str());
                    track1 = 1;
                    track2 = 1;
                    track3 = 1;
                    createfile << CSV_HEADER << std::endl;
                    createfile.flush();
                    createfile.close();
                    std::cout << std::endl;
                }

                /*
                 *patfil equals renter, track1 will remain zero allowing
                 *user to reenter the patient name.
                 */
                else if(patfil == "reenter") {
                    track1 = 0;
                    track2 = 1;
                }

                /*
                 *The users input was neither create or reenter. User
                 *must enter patient name again.
                 */
                else {
                    std::cout << std::endl;
                    std::cout << "Your input is not acceptable." << std::endl;
                    std::cout << std::endl;
                }
            } while(track2 == 0);
        }
    } while (track1 == 0);

    getchar();  // removes that pesky newline character from stdin buffer
    return patientname; //returns the path to the patient file
}

std::map<Side, DataParams> ReadParams(auto filename) {
    std::map<Side, DataParams> params;
    DataParams leftparams;
    DataParams rightparams;

    std::ifstream file(filename.c_str());
    std::string leftline;
    std::string rightline;
    std::string leftsearch = "Left";
    std::string rightsearch = "Right";
    std::string paramstring;
    std::string lfreqstr;
    std::string lnoisestr;
    std::string rfreqstr;
    std::string rnoisestr;
    uint32 lcnt = 0;
    uint32 rcnt = 0;
    float32 lfreqval;
    float32 lnoiseval;
    float32 rfreqval;
    float32 rnoiseval;

    /*
     * if statement which uses ifstream function to open patient file
     * filename)
     */
    if(file.is_open()) {
        /*
         * While statement to find the first Left line and save to
         *leftline as string.
         */
        while (getline(file, leftline)) {
            if(leftline.find(leftsearch, 0) != std::string::npos) {
                break;
            }

        }

```

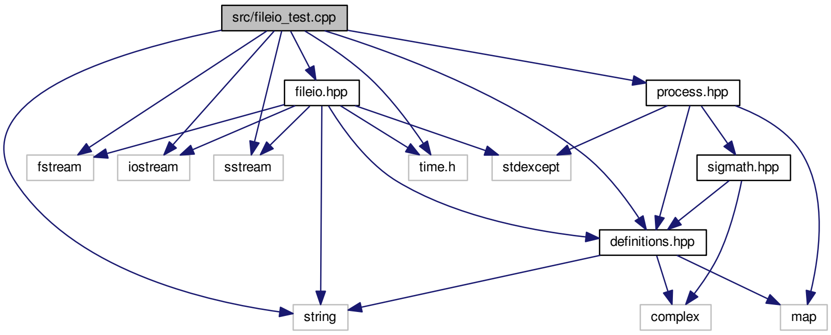
```
            /*
             * While statement to find first right line and save to rightline
             * as string.
             */
            while (getline(file,rightline)) {
                if(rightline.find(rightsearch, 0) != std::string::npos) {
                    break;
                }
            }

            // Code to break leftline and rightline into its parts
            std::stringstream lss(leftline);
            std::stringstream rss(rightline);

            while(getline(lss,paramstring, ',')) {
                lcnt++;

                if(lcnt == 3) {
                    lfreqstr = paramstring;
                }

                else if(lcnt == 4) {
                    lnoisestr = paramstring;
                }
            }

            while(getline(rss,paramstring, ',')) {
                rcnt++;

                if(rcnt == 3) {
                    rfreqstr = paramstring;
                }

                else if(rcnt == 4) {
                    rnoisestr = paramstring;
                }
            }

            /*
             * Statement to convert lfreq, lnoise, rfreq, and rnoise from
             * strings to floats.
             * */
            lfreqval = atof(lfreqstr.c_str());
            lnoiseval = atof(lnoisestr.c_str());
            rfreqval = atof(rfreqstr.c_str());
            rnoiseval = atof(rnoisestr.c_str());

            file.close();
        }

        else {
            throw std::runtime_error("The patient file could not be opened.");
        }

        leftparams.freq = lfreqval;
        leftparams.noise = lnoiseval;
        rightparams.freq = rfreqval;
        rightparams.noise = rnoiseval;

        params[Side::Left] = leftparams;
        params[Side::Right] = rightparams;

        return params;
    }

    void WriteParams(std::map<Side, DataParams> params, auto filename) {
        char temp[80];
        std::ofstream file(filename.c_str(),
                std::ofstream::out | std::ofstream::app);

        //Gives pointer measurementtime a data type of time_t.
        time_t measurementtime;
        time(&measurementtime); //Gets the current time.
        strftime(temp, 80, "%c", localtime(&measurementtime));
        std::string fTime = std::string(temp);

        //if statement to print the Left side parameters to the patient file.
        if(file.is_open()) {
            file << fTime + "," + "Left" + ","
                + std::to_string(params[Side::Left].freq)
                + ", " + std::to_string(params[Side::Left].noise) << std::endl;
        }

        //if statement to print the Right side parameters to the patient file.
        if(file.is_open()) {
            file << fTime + "," + "Right" + ","
                + std::to_string(params[Side::Right].freq)
                + ", " + std::to_string(params[Side::Right].noise) << std::endl;
        }

        else {
            std::cout << "Patient file can not be opened!" << std::endl;
        }

        file.close();
    }
}

#endif
```

8.11 src/fileio_test.cpp File Reference

Contains program that tests some functions in fileio.hpp.

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <time.h>
#include "definitions.hpp"
#include "fileio.hpp"
#include "process.hpp"
```

Include dependency graph for fileio_test.cpp:



Functions

- int main ()

8.11.1 Detailed Description

Contains program that tests some functions in fileio.hpp.

Author

Samuel Andrew Wisner

Definition in file fileio_test.cpp.

8.11.2 Function Documentation

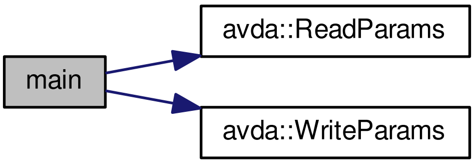
8.11.2.1 int main ( )

Tests the functions in fileio.hpp.

Definition at line 23 of file fileio_test.cpp.

```
00023           {
00024     string path = PATIENT_PATH + "wizmack, sammy andy.csv";
00025     map<Side, DataParams> laMap = ReadParams(path);
00026     cout <<  laMap[Side::Right].freq << endl;
00027     cout << laMap[Side::Right].noise << endl;
00028 
00029     WriteParams(laMap, path);
00030 }
```

Here is the call graph for this function:



8.12 fileio_test.cpp

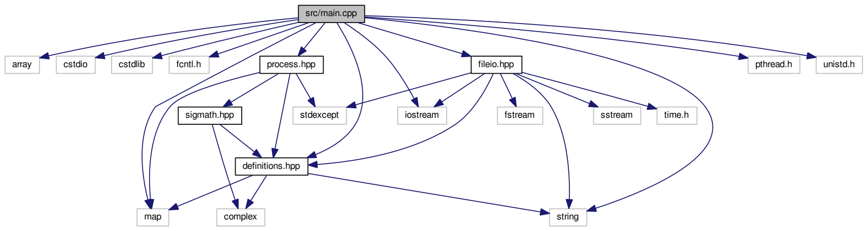
```
00001 
00007 #include <fstream>
00008 #include <iostream>
00009 #include <sstream>
00010 #include <string>
00011 #include <time.h>
00012 
00013 #include "definitions.hpp"
00014 #include "fileio.hpp"
00015 #include "process.hpp"
00016 
00017 using namespace std;
00018 using namespace avda;
00019 
00023 int main() {
00024     string path = PATIENT_PATH + "wizmack, sammy andy.csv";
00025     map<Side, DataParams> laMap = ReadParams(path);
00026     cout <<  laMap[Side::Right].freq << endl;
00027     cout << laMap[Side::Right].noise << endl;
00028 
00029     WriteParams(laMap, path);
00030 }
```

8.13 src/main.cpp File Reference

Contains the main program.

```
#include <array>
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <iostream>
#include <map>
#include <pthread.h>
#include <string>
#include <unistd.h>
#include "definitions.hpp"
#include "fileio.hpp"
#include "process.hpp"
```

Include dependency graph for main.cpp:



Functions

- int main (int argc, char ∗∗argv)

8.13.1 Detailed Description

Contains the main program.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Bug Errant keystrokes (especially [ENTER]) can cause the next recording(s) to begin rather than waiting for the user to press [ENTER]. stdin must be flushed somehow in between recordings.

Definition in file main.cpp.

8.13.2 Function Documentation

8.13.2.1 int main (int *argc,* char ∗∗ *argv*)

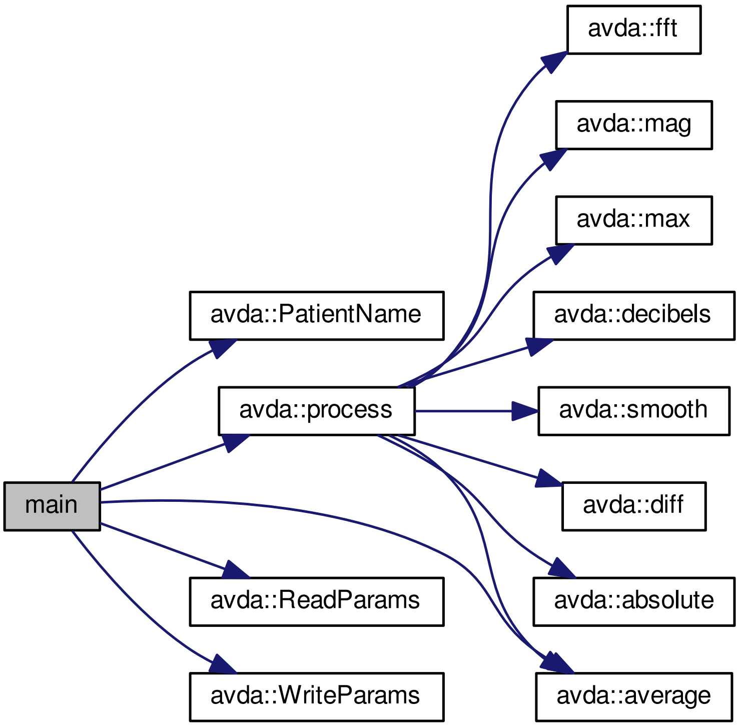
The main program for this project. It will detect avdaspasms over a period of days.

Definition at line 31 of file main.cpp.

```
00031                              {
00032         // Recorded audio buffer
00033         float32* buffer = (float32*)std::malloc(BUFFER_SIZE);
00034         bool cont = true;  // whether to continue in the recording loop
00035         DataParams params[REC_COUNT];  // holds DataParam's from recordings
00036         string filename = PatientName();  // generate name for patient's file
00037         map<Side, DataParams> results;  // parameters by side
00038 
```

```
    // arecord command
    const string recCommand = string("arecord -t raw -d ")
        + to_string(DURATION) + string(" -D plughw:1,0 -f FLOAT -q -r ")
        + to_string(SAMPLE_FREQ) + string(" ") + TEMP_FILE;

    // Recording
    while(cont) {
        for(uint8 i = 0; i < REC_COUNT; i++) {
            // prompt
            cout << "Press [ ENTER ] to begin analysis for the "
                << (i < REC_COUNT / 2 ? "left" : "right") << " side, depth #"
                << (((i >= REC_COUNT / 2) ? (i - REC_COUNT / 2) : i) + 1)
                << " ";
            getchar();  // wait for ENTER to be pressed
            cout << "Analyzing..." << endl;

            system(recCommand.c_str());
            usleep(DURATION*1000000 + 1500000);  // sleep DURATION + 1.5 seconds

            int file = open(TEMP_FILE.c_str(), O_RDONLY);  // open temp file
            int retRead = read(file, buffer, BUFFER_SIZE);  // copy to buffer
            close(file);  // close temp file
            remove(TEMP_FILE.c_str());  // delete temp file

            // if something goes wrong reading the temp file, program exits
            if(file < 0 || retRead < BUFFER_SIZE) {
                cerr << "An error occurred reading the doppler audio! "
                    "The program will now exit." << endl;
                return ERROR;
            }

            // process and store parameters
            params[i] = process(buffer, SAMPLE_COUNT,
SAMPLE_FREQ);
            cout << "The analysis is complete." << endl << endl;
        }

        // calculate averaged parameters
        results[Side::Left] = average(params, REC_COUNT / 2);
        results[Side::Right] = average(&params[REC_COUNT / 2], REC_COUNT / 2);

        cout << "Analysis is complete." << endl << endl;

        // print averaged side analysis
        for(int i = 0; i < 2; i++) {
            Side side = (Side)i;
            cout << (side == Side::Left ? "[LEFT]" : "[RIGHT]") << endl;
            cout << "Drop-off frequency: " << (uint16)(results[side].freq + 0.5)
                << " Hz" << endl;
            cout << "Average relative noiseband power: "
                << (sint16)(results[side].noise - 0.5) << " dB" << endl <<endl;
        }

        cont = results[Side::Left].freq > MAX_DROP_FREQ
            || results[Side::Right].freq > MAX_DROP_FREQ;

        if(cont) {
            cout << "An error in aquisition of the doppler audio has occurred! "
                "Ensure the connection from the doppler machine to this device "
                "is secure and the connection uninterruptable." << endl << endl;
        }
    }

    free(buffer);  // free buffer to prevent memory leak

    // examine likelihood of avdaspasm
    map<Side, DataParams> baseParams = ReadParams(filename);
    map<Side, bool> comparison;

    if(baseParams[Side::Left].freq != 0 && baseParams[Side::Left].noise != 0
            && baseParams[Side::Right].freq != 0
            && baseParams[Side::Right].noise != 0) {
        for(uint8 i = 0; i < 2; i++) {
            Side side = (Side)i;
            float comp = fabs(results[side].freq - baseParams[side].freq)
                * fabs(baseParams[side].noise - results[side].noise);
            comparison[side] = comp > DET_THRESH;
        }

        string which;

        if(comparison[Side::Left] && !comparison[Side::Right]) {
            which = "The left";
        } else if(!comparison[Side::Left] && comparison[Side::Right]) {
            which = "The right";
        } else if (comparison[Side::Left] && comparison[Side::Right]) {
            which = "Both";
        } else {
            which = "Neither";
        }

        cout << which << " side seems to show evidence of a vasospasm." << endl;
    } else {
        cout << "These values will be stored as the baseline parameters to "
            "which all future parameters are compared." << endl;
    }

    WriteParams(results, filename);
}
```

Here is the call graph for this function:



8.14 main.cpp

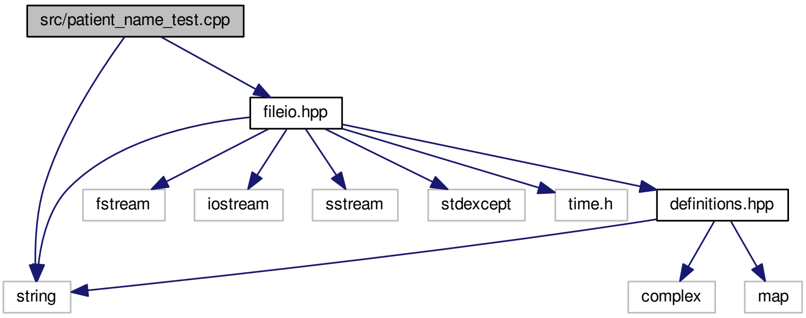
```

#include <array>
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <iostream>
#include <map>
#include <pthread.h>
#include <string>
#include <unistd.h>

#include "definitions.hpp"
#include "fileio.hpp"
#include "process.hpp"

using namespace std;
using namespace avda;

int main(int argc, char** argv) {
    // Recorded audio buffer
    float32* buffer = (float32*)std::malloc(BUFFER_SIZE);
    bool cont = true;  // whether to continue in the recording loop
    DataParams params[REC_COUNT];  // holds DataParam's from recordings
    string filename = PatientName();  // generate name for patient's file
    map<Side, DataParams> results;  // parameters by side

    // arecord command
    const string recCommand = string("arecord -t raw -d ")
        + to_string(DURATION) + string(" -D plughw:1,0 -f FLOAT -q -r ")
        + to_string(SAMPLE_FREQ) + string(" ") + TEMP_FILE;

    // Recording
    while(cont) {
        for(uint8 i = 0; i < REC_COUNT; i++) {
            // prompt
            cout << "Press [ ENTER ] to begin analysis for the "
                << (i < REC_COUNT / 2 ? "left" : "right") << " side, depth #"
                << (((i >= REC_COUNT / 2) ? (i - REC_COUNT / 2) : i) + 1)
                << " ";
            getchar();  // wait for ENTER to be pressed
            cout << "Analyzing..." << endl;

            system(recCommand.c_str());
            usleep(DURATION*1000000 + 1500000);  // sleep DURATION + 1.5 seconds

            int file = open(TEMP_FILE.c_str(), O_RDONLY);  // open temp file
            int retRead = read(file, buffer, BUFFER_SIZE);  // copy to buffer
            close(file);  // close temp file
            remove(TEMP_FILE.c_str());  // delete temp file

            // if something goes wrong reading the temp file, program exits
            if(file < 0 || retRead < BUFFER_SIZE) {
                cerr << "An error occurred reading the doppler audio! "
                    "The program will now exit." << endl;
                return ERROR;
            }

            // process and store parameters
            params[i] = process(buffer, SAMPLE_COUNT, SAMPLE_FREQ);
            cout << "The analysis is complete." << endl << endl;
        }

        // calculate averaged parameters
        results[Side::Left] = average(params, REC_COUNT / 2);
        results[Side::Right] = average(&params[REC_COUNT / 2], REC_COUNT / 2);

        cout << "Analysis is complete." << endl << endl;

        // print averaged side analysis
        for(int i = 0; i < 2; i++) {
            Side side = (Side)i;
            cout << (side == Side::Left ? "[LEFT]" : "[RIGHT]") << endl;
            cout << "Drop-off frequency: " << (uint16)(results[side].freq + 0.5)
                << " Hz" << endl;
            cout << "Average relative noiseband power: "
                << (sint16)(results[side].noise - 0.5) << " dB" << endl <<endl;
        }

        cont = results[Side::Left].freq > MAX_DROP_FREQ
            || results[Side::Right].freq > MAX_DROP_FREQ;

        if(cont) {
            cout << "An error in aquisition of the doppler audio has occurred! "
                "Ensure the connection from the doppler machine to this device "
                "is secure and the connection uninterruptable." << endl << endl;
        }
    }

    free(buffer);  // free buffer to prevent memory leak

    // examine likelihood of avdaspasm
    map<Side, DataParams> baseParams = ReadParams(filename);
    map<Side, bool> comparison;

    if(baseParams[Side::Left].freq != 0 && baseParams[Side::Left].noise != 0
            && baseParams[Side::Right].freq != 0
            && baseParams[Side::Right].noise != 0) {
        for(uint8 i = 0; i < 2; i++) {
            Side side = (Side)i;
            float comp = fabs(results[side].freq - baseParams[side].freq)
                * fabs(baseParams[side].noise - results[side].noise);
            comparison[side] = comp > DET_THRESH;
        }

        string which;

        if(comparison[Side::Left] && !comparison[Side::Right]) {
            which = "The left";
        } else if(!comparison[Side::Left] && comparison[Side::Right]) {
            which = "The right";
        } else if (comparison[Side::Left] && comparison[Side::Right]) {
            which = "Both";
        } else {
            which = "Neither";
        }

        cout << which << " side seems to show evidence of a vasospasm." << endl;
    } else {
        cout << "These values will be stored as the baseline parameters to "
            "which all future parameters are compared." << endl;
    }

    WriteParams(results, filename);
}
```

8.15 src/patient_name_test.cpp File Reference

Contains a program to test the PatientName() function.

```
#include <string>
#include "fileio.hpp"
```

Include dependency graph for patient_name_test.cpp:



Functions

- int main (int argc, char **argv)

8.15.1 Detailed Description

Contains a program to test the PatientName() function.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Definition in file patient_name_test.cpp.

8.15.2 Function Documentation

8.15.2.1 int main (int *argc,* char ∗∗ *argv*)

Tests the PatientName() function from fileio.hpp.

Definition at line 17 of file patient_name_test.cpp.

```
00017                                 {
00018     string filename = PatientName();
00019     cout << filename;
00020 }
```

Here is the call graph for this function:

8.16 patient_name_test.cpp

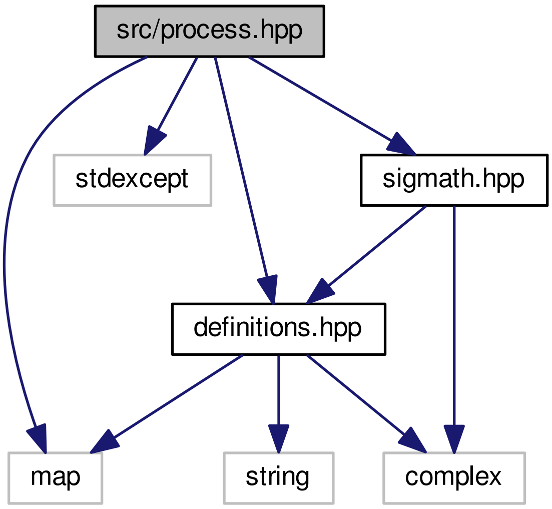
```
00001
00007 #include <string>
00008
00009 #include "fileio.hpp"
00010
00011 using namespace std;
00012 using namespace avda;
00013
00017 int main(int argc, char** argv) {
00018     string filename = PatientName();
00019     cout << filename;
00020 }
```

8.17 src/process.hpp File Reference

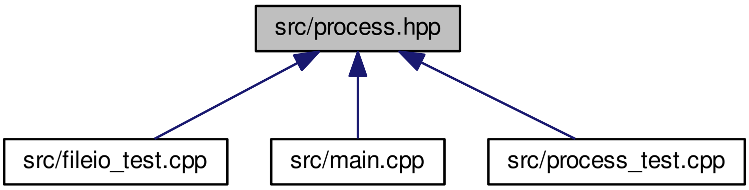
Contains functions related to the program's threaded processing of audio data.

```
#include <map>
#include <stdexcept>
#include "definitions.hpp"
#include "sigmath.hpp"
```

Include dependency graph for process.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- avda

Functions

- DataParams avda::process (float32 ∗data, uint32 size, float32 samplingRate)

8.17.1 Detailed Description

Contains functions related to the program's threaded processing of audio data.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Definition in file process.hpp.

8.18 process.hpp

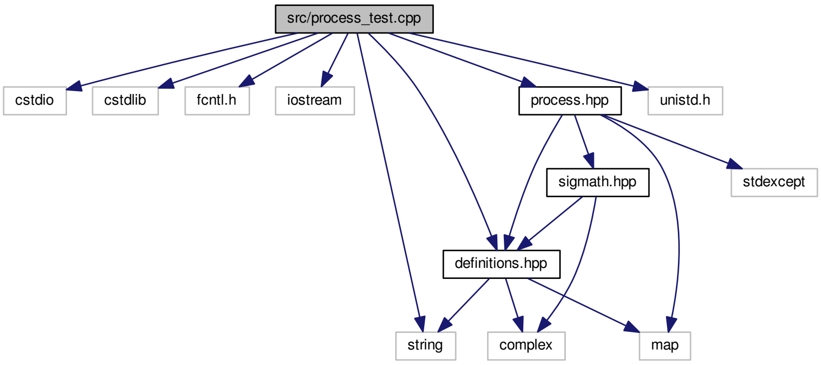
```

#ifndef process_H
#define process_H

#include <map>
#include <stdexcept>

#include "definitions.hpp"
#include "sigmath.hpp"

namespace avda {
    DataParams process(float32* data, uint32 size,
float32 samplingRate) {
        if((size & (size - 1) != 0) || size < 2) {
            throw std::invalid_argument(
                    "The number of samples is not a power of two!");
        }

        // declare function-scoped variables
        uint32 freqSize = size / 2;
        cfloat32* cdata = (cfloat32*)std::malloc(size * sizeof(
cfloat32));
        float32* fdata = (float32*)std::malloc(freqSize * sizeof(
float32));
        float32* origdata = (float32*)std::malloc(freqSize * sizeof(
float32));

        // convert data to complex numbers for fft()
        for(uint32 i = 0; i < size; i++) {
            cdata[i] = data[i];
        }

        // find frequency spectrum in relative decibels
        fft(cdata, size);
        mag(cdata, fdata, freqSize);
        Maximum maximum = max(fdata, freqSize);

        for(uint32 i = 0; i < freqSize; i++) {
            fdata[i] /= maximum.value;
        }

        decibels(fdata, freqSize);

        for(uint32 i = 0; i < freqSize; i++) {
            origdata[i] = fdata[i];
        }

        /*
         * Run spectrum values through moving-average filter to smooth the
         * curve and make it easier to determine the derivative.
         */
        smooth(fdata, freqSize, 20);

        /*
         * Find the derivative of the smoothed spectrum. Bote that both this
         * filter and the previous are necessary to the algorithm.
         */
        diff(fdata, freqSize);
        smooth(fdata, freqSize, 100);
        absolute(fdata, freqSize);

        // find the parameters of this specific recording
        uint16 offset = 1000;
        absolute(&fdata[offset], freqSize - offset);
        maximum = max(&fdata[offset], freqSize - offset);
        uint32 index = maximum.index + offset;

        DataParams params;
        params.freq = index * (float)SAMPLE_FREQ / freqSize / 2;
        params.noise = average(&origdata[index + offset],
                freqSize - offset - index);

        free(cdata);
        free(fdata);

        return params;
    }
}

#endif
```

8.19 src/process_test.cpp File Reference

Contains a program to test the process() function.

```
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <iostream>
#include <string>
#include <unistd.h>
#include "definitions.hpp"
#include "process.hpp"
```

Include dependency graph for process_test.cpp:



Macros

- #define COUNT 131072

Functions

- int main (int argc, char **argv)

8.19.1 Detailed Description

Contains a program to test the process() function.

Author

Samuel Andrew Wisner, awisner94@gmail.com

Definition in file process_test.cpp.

8.19.2 Macro Definition Documentation

8.19.2.1 #define COUNT 131072

Definition at line 17 of file process_test.cpp.

8.19.3 Function Documentation

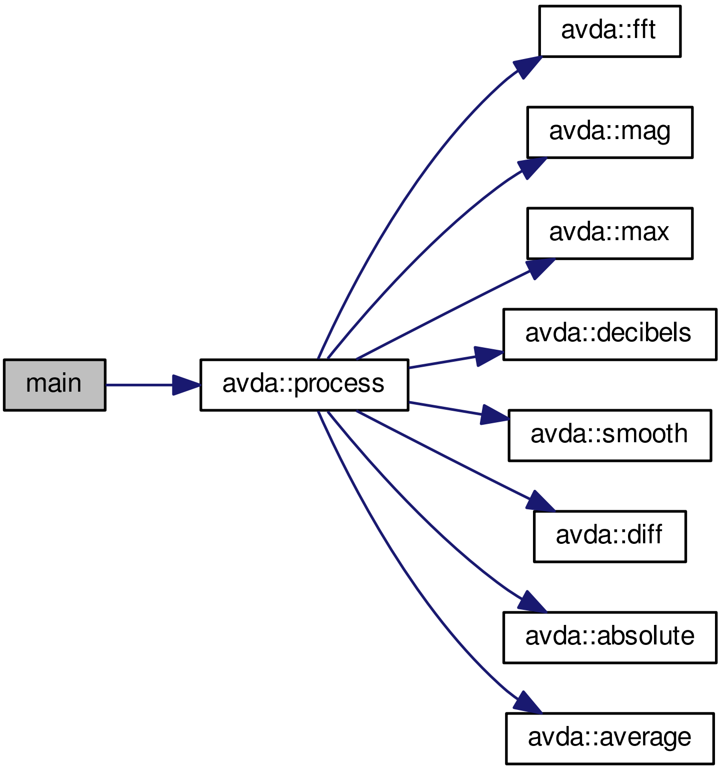
8.19.3.1 int main (int *argc,* char ∗∗ *argv*)

Tests the process() function from process.hpp.

Definition at line 25 of file process_test.cpp.

```
00025                               {
00026     int file = open("/home/pi/avda/etc/audio/test.raw", O_RDONLY);
00027 
00028     if(file < 0) {
00029         cerr << "File unreadable!" << endl;
00030         return -1;
00031     }
00032 
00033     float32* buffer = (float32*)malloc(COUNT * sizeof(float32));
00034     int charRead = read(file, buffer, COUNT * sizeof(float32));
00035 
00036     if(charRead < COUNT) {
00037         cerr << "Too few bytes read!" << endl;
00038         return -1;
00039     }
00040 
00041     close(file);
00042 
00043     DataParams params = process(buffer, COUNT, SAMPLE_FREQ);
00044     free(buffer);
00045     cout << "Cutoff: " << params.freq << endl;
00046     cout << "Noise: " << params.noise << endl;
00047 }
```

Here is the call graph for this function:



8.20 process_test.cpp

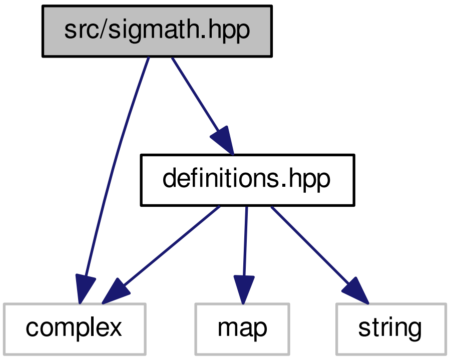
```

#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <iostream>
#include <string>
#include <unistd.h>

#include "definitions.hpp"
#include "process.hpp"

#define COUNT 131072

using namespace std;
using namespace avda;

int main(int argc, char** argv) {
    int file = open("/home/pi/avda/etc/audio/test.raw", O_RDONLY);

    if(file < 0) {
        cerr << "File unreadable!" << endl;
        return -1;
    }

    float32* buffer = (float32*)malloc(COUNT * sizeof(float32));
    int charRead = read(file, buffer, COUNT * sizeof(float32));

    if(charRead < COUNT) {
        cerr << "Too few bytes read!" << endl;
        return -1;
    }

    close(file);

    DataParams params = process(buffer, COUNT, SAMPLE_FREQ);
    free(buffer);
    cout << "Cutoff: " << params.freq << endl;
    cout << "Noise: " << params.noise << endl;
}
```

8.21 src/sigmath.hpp File Reference

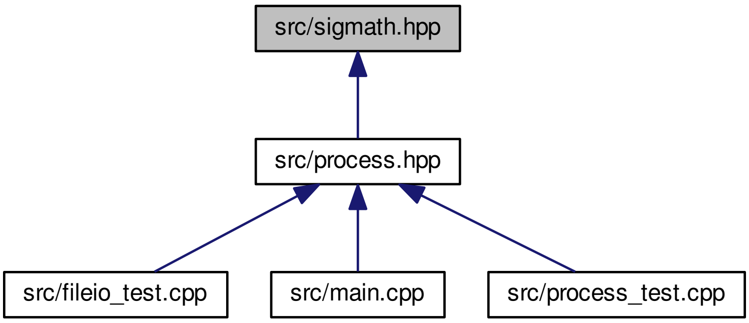
Contains the functions necessary to perform the mathematical operations required by this program.

```
#include <complex>
#include "definitions.hpp"
```

Include dependency graph for sigmath.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- avda

Functions

- void avda::absolute (float32 ∗data, uint32 size)
- float32 avda::average (float32 ∗data, uint32 size)
- DataParams avda::average (DataParams ∗params, uint8 size)
- void avda::decibels (float32 ∗data, uint32 size)
- void avda::diff (float32 ∗data, uint32 size)
- void avda::fft (cfloat32 ∗data, uint32 size)
- void avda::mag (cfloat32 ∗orig, float32 ∗newmags, uint32 size)
- Maximum avda::max (float32 ∗data, uint32 size)
- void avda::smooth (float32 ∗data, uint32 size, uint16 order)

8.21.1 Detailed Description

Contains the functions necessary to perform the mathematical operations required by this program.

Author

Samuel Andrew Wisner, awisner94@gmail.com
Nicholas K. Nolan

Definition in file sigmath.hpp.

8.22 sigmath.hpp

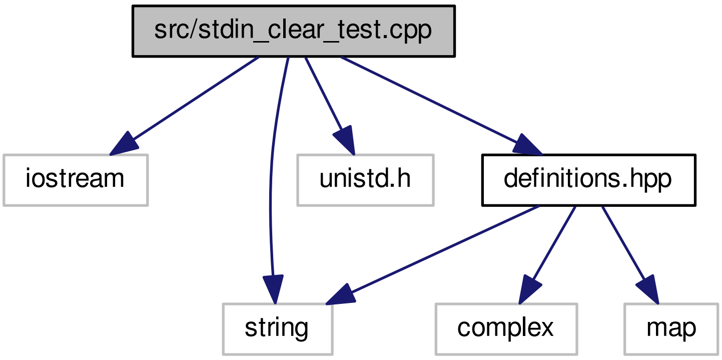
```
#ifndef sigmath_H
#define sigmath_H

#include <complex>
#include "definitions.hpp"

namespace avda {
    // PROTOTYPES

    void absolute(float32* data, uint32 size);

    float32 average(float32* data, uint32 size);

    DataParams average(DataParams* params, uint8 size);

    void decibels(float32* data, uint32 size);

    void diff(float32* data, uint32 size);

    void fft(cfloat32* data, uint32 size);

    void mag(cfloat32* orig, float32* newmags, uint32 size);

    Maximum max(float32* data, uint32 size);

    void smooth(float32* data, uint32 size, uint16 order);

    // DEFINITIONS

    void absolute(float32* data, uint32 size) {
        for(uint32 i = 0; i < size; i++) {
            data[i] = fabsf(data[i]);
        }
    }

    float32 average(float32* data, uint32 size) {
        float32 ave;

        for(uint32 i = 0; i < size; i++) {
            ave += data[i];
        }

        ave = ave / size;
        return ave;
    }

    DataParams average(DataParams* params, uint8 size) {
        DataParams ave;

        for(uint8 i = 0; i < size; i++) {
            //freq is an attribute. this is how to add structure attributes
            ave.freq += params[i].freq;
            ave.noise += params[i].noise;
        }

        ave.freq /= size;
        ave.noise /= size;
        return ave;
    }

    void decibels(float32* data, uint32 size) {
        for(uint32 i = 0; i < size; i++) {
            data[i] = 20 * log10(data[i]);
        }
    }

    void diff(float32* data, uint32 size) {
        float32 temp[size];
        temp[0] = 0;

        for(uint32 i = 1; i < size; i++) {
            temp[i] = data[i] - data[i-1];
        }

        for(uint32 i = 0; i < size; i++) {
            data[i] = temp[i];
        }
    }

    void fft(cfloat32* data, uint32 size) {
        // DFT
        uint32 k = size;
        uint32 n;
        float32 thetaT = M_PI / size;
        cfloat32 phiT(cos(thetaT), sin(thetaT));
        cfloat32 T;

        while(k > 1) {
            n = k;
            k >>= 1;
            phiT = phiT * phiT;
            T = 1.0L;

            for(uint32 l = 0; l < k; l++) {
                for(uint32 a = l; a < size; a += n) {
                    uint32 b = a + k;
                    cfloat32 t = data[a] - data[b];
                    data[a] += data[b];
                    data[b] = t * T;
                }

                T *= phiT;
            }
        }

        // Decimate
        uint32 m = (uint32)log2(size);

        for(uint32 a = 0; a < size; a++) {
            uint32 b = a;

            // Reverse bits
            b = (((b & 0xaaaaaaaa) >> 1) | ((b & 0x55555555) << 1));
            b = (((b & 0xcccccccc) >> 2) | ((b & 0x33333333) << 2));
            b = (((b & 0xf0f0f0f0) >> 4) | ((b & 0x0f0f0f0f) << 4));
            b = (((b & 0xff00ff00) >> 8) | ((b & 0x00ff00ff) << 8));
            b = ((b >> 16) | (b << 16)) >> (32 - m);

            if (b > a)
            {
                cfloat32 t = data[a];
                data[a] = data[b];
                data[b] = t;
            }
        }
    }

    void mag(cfloat32* orig, float32* newmags, uint32 size) {
        //loop to run throught the length of array orig
        for(uint32 n = 0; n < size; n++) {
            /*
             * abs should calculate the magnitude of complex array elements.
             * saves to new array
             */
            newmags[n] = std::abs(orig[n]);
        }
    }

    Maximum max(float32* data, uint32 size) {
        Maximum m;

        //loop to run through the length of array data
        for (uint32 i = 0; i < size; i++) {
            /*
             * when value at data[i] is above max.value,
             * sets max.value equal to data[i] and max.index equal to i
             */
            if (data[i] > m.value) {
                m.value = data[i];
                m.index = i;
            }
        }

        return m;
    }

    void smooth(float32* data, uint32 size, uint16 order) {
        float32 coeff = 1 / (float32)order;
        float32 temp[size];

        for(uint32 i = 0; i < size; i++) {
            temp[i] = 0;

            for(uint16 j = 0; j < order && j <= i; j++) {
                temp[i] += data[i - j];
            }

            temp[i] *= coeff;
        }

        for(uint32 i = 0; i < size; i++) {
            data[i] = temp[i];
        }
    }
}

#endif
```

8.23 src/stdin_clear_test.cpp File Reference

Contains a program to test clearing the stdin buffer.

```
#include <iostream>
#include <string>
#include <unistd.h>
#include "definitions.hpp"
```

Include dependency graph for stdin_clear_test.cpp:



Macros

- #define COUNT 80

Functions

- int main (int argc, char ∗∗argv)

8.23.1 Detailed Description

Contains a program to test clearing the stdin buffer.

Author

Samuel Andrew Wisner, awisner94@gmail.com
Nicholas K. Nolan

Definition in file stdin_clear_test.cpp.

8.23.2 Macro Definition Documentation

8.23.2.1 #define COUNT 80

Definition at line 14 of file stdin_clear_test.cpp.

8.23.3 Function Documentation

8.23.3.1 int main (int *argc,* char ∗∗ *argv*)

Tests the ability to clear the stdin buffer.

Definition at line 22 of file stdin_clear_test.cpp.

```
                                {
    char text1[COUNT];
    char text2[COUNT];

    cout << "Enter text to ignore: ";
    cout.flush();
    read(STDIN_FILENO, &text1, COUNT);
//  fflush(stdin);
    cout << endl << "Enter text to print: ";
    cout.flush();
    read(STDIN_FILENO, &text2, COUNT);
    cout << endl << "In buffer: " << text2 << endl;
}
```

8.24 stdin_clear_test.cpp

```

#include <iostream>
#include <string>
#include <unistd.h>

#include "definitions.hpp"

#define COUNT 80

using namespace std;
using namespace avda;

int main(int argc, char** argv) {
    char text1[COUNT];
    char text2[COUNT];

    cout << "Enter text to ignore: ";
    cout.flush();
    read(STDIN_FILENO, &text1, COUNT);
//  fflush(stdin);
    cout << endl << "Enter text to print: ";
    cout.flush();
    read(STDIN_FILENO, &text2, COUNT);
    cout << endl << "In buffer: " << text2 << endl;
}
```

Index