The Automatic Vasospasm Detection Application

Generated by Doxygen 1.8.8

Wed Apr 20 2016 04:24:03

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

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 occured 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

extra newline character inserted into stdin buffer after PatientName() is run

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.

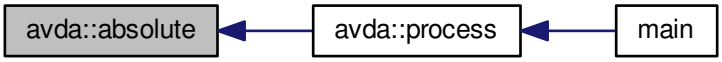
Parameters

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

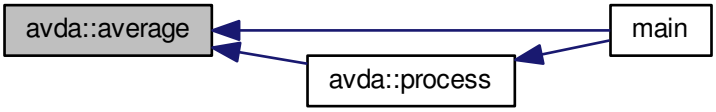
Returns

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.

```
00140                                                   {
00141         DataParams ave;
00142 
00143         for(uint8 i = 0; i < size; i++) {
00144             //freq is an attribute. this is how to add structure attributes
00145             ave.freq += params[i].freq;
00146             ave.noise += params[i].noise;
00147         }
00148 
00149         ave.freq /= size;
00150         ave.noise /= size;
00151         return ave;
00152     }
```

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.

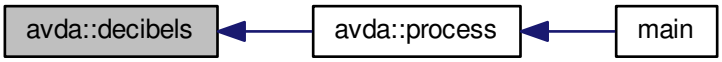
Parameters

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.

```
00154                                               {
00155         for(uint32 i = 0; i < size; i++) {
00156             data[i] = 20 * log10(data[i]);
00157         }
00158     }
```

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.

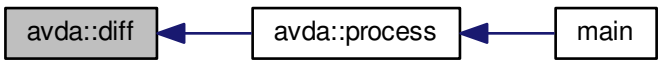
Parameters

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.

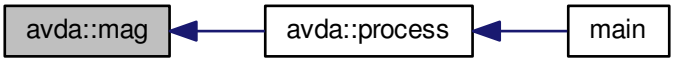
Parameters

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.

```
00221                                                       {
00222         //loop to run throught the length of array orig
00223         for(uint32 n = 0; n < size; n++) {
00224             /*
00225              * abs should calculate the magnitude of complex array elements.
00226              * saves to new array
00227              */
00228             newmags[n] = std::abs(orig[n]);
00229         }
00230     }
```

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.

```
00232                                           {
00233         Maximum m;
00234
00235         //loop to run through the length of array data
00236         for (uint32 i = 0; i < size; i++) {
00237             /*
00238              * when value at data[i] is above max.value,
00239              * sets max.value equal to data[i] and max.index equal to i
00240              */
00241             if (data[i] > m.value) {
00242                 m.value = data[i];
00243                 m.index = i;
00244             }
00245         }
00246
00247         return m;
00248     }
```

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

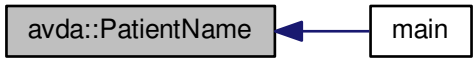
the file under which all patient data is saved

Definition at line 33 of file fileio.hpp.

```
00033                           {
00034         std::string fname = "";
00035         std::string mname = "";
00036         std::string lname = "";
00037         std::string patfil = "";
00038         std::string patientname = "";
00039         uint32 track1 = 0;
00040         uint32 track2 = 0;
00041         uint32 track3 = 0;
00042 
00043         do {
00044             std::cout << "Please enter the patients name." << std::endl;
00045             std::cout << "First name: ";
00046             std::cin >> fname;
00047             std::cout << "Middle name: ";
00048             std::cin >> mname;
00049             std::cout << "Last name: ";
00050             std::cin >> lname;
00051 
00052             // creates new std::string with path to patient file
00053             patientname = PATIENT_PATH + lname + ", " + fname
00054                 + " " + mname + ".csv";
00055 
00056             // prints out patientname. shows user the path to the patient file
00057             //std::cout << patientname << std::endl << std::endl;
00058             std::ifstream file(patientname.c_str());
00059 
00060             if (file.good()) {
00061                 track1 = 1;
00062             }
00063 
00064             /*
00065              * Compares patientname to existing files and lets user know
00066              * if the file does not exist.
00067              */
00068             else if (!file.good()) {
00069                 /*
00070                  * Do while statement to continue asking user about the file
00071                  * if their input is not acceptable
00072                  */
00073                 do {
00074                     std::cout << "Patient file does not exist, would you like "
00075                         "to create file or re-enter their name?" << std::endl;
00076                     std::cout << "  *Type 'create' and press enter key "
00077                         "to create the patient file." << std::endl;
00078                     std::cout << "  *Type 'reenter' and press enter key "
00079                         "to re-enter the patients name." << std::endl;
00080                     std::cout << std::endl;
```

```
00081                     std::cin >> patfil;
00082
00083                     /*
00084                      * patfil equals create, track1 and 2 will increase
00085                      * escaping both do while loops
00086                      */
00087                     if(patfil == "create") {
00088                         std::ofstream createfile(patientname.c_str());
00089                         track1 = 1;
00090                         track2 = 1;
00091                         track3 = 1;
00092                         createfile << CSV_HEADER << std::endl;
00093                         createfile.flush();
00094                         createfile.close();
00095                     }
00096
00097                     /*
00098                      *patfil equals renter, track1 will remain zero allowing
00099                      *user to reenter the patient name.
00100                     */
00101                     else if(patfil == "reenter") {
00102                         track1 = 0;
00103                         track2 = 1;
00104                     }
00105
00106                     /*
00107                      *The users input was neither create or reenter. User
00108                      *must enter patient name again.
00109                      */
00110                     else {
00111                         std::cout << std::endl;
00112                         std::cout << "Your input is not acceptable." << std::endl;
00113                         std::cout << std::endl;
00114                     }
00115                 }while(track2 == 0);
00116             }
00117         } while (track1 == 0);
00118
00119         return patientname; //returns the path to the patient file
00120     }
```

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 prioprietary 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

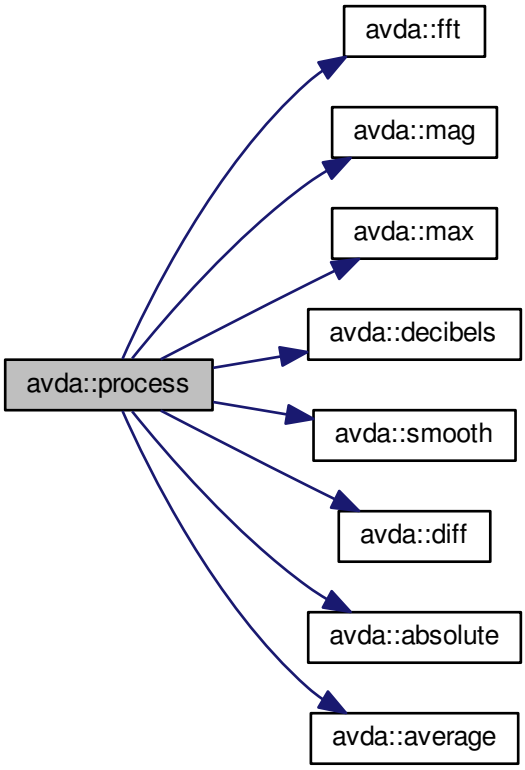
Returns

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

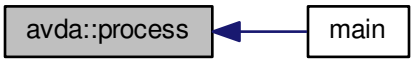
Definition at line 48 of file process.hpp.

```
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

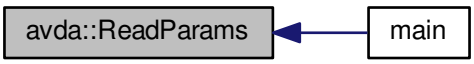
patient parameters read for each side

Definition at line 131 of file fileio.hpp.

```
00131                                                   {
00132         std::map<Side, DataParams> params;
00133         DataParams leftparams;
00134         DataParams rightparams;
00135 
00136         std::ifstream file(filename.c_str());
00137         std::string leftline;
00138         std::string rightline;
00139         std::string leftsearch = "Left";
00140         std::string rightsearch = "Right";
00141         std::string paramstring;
00142         std::string lfreqstr;
00143         std::string lnoisestr;
00144         std::string rfreqstr;
00145         std::string rnoisestr;
00146         uint32 lcnt = 0;
00147         uint32 rcnt = 0;
00148         float32 lfreqval;
00149         float32 lnoiseval;
00150         float32 rfreqval;
00151         float32 rnoiseval;
00152 
00153         /*
00154          * if statement which uses ifstream function to open patient file
00155          * filename)
00156          */
00157         if(file.is_open()) {
00158             /*
00159              * While statement to find the first Left line and save to
00160              *leftline as string.
00161              */
00162             while (getline(file, leftline)) {
00163                 if(leftline.find(leftsearch, 0) != std::string::npos) {
00164                     break;
00165                 }
00166 
00167             }
00168 
00169             /*
00170              * While statement to find first right line and save to rightline
00171              * as string.
00172              */
00173             while (getline(file,rightline)) {
00174                 if(rightline.find(rightsearch, 0) != std::string::npos) {
00175                     break;
00176                 }
00177             }
00178 
00179             // Code to break leftline and rightline into its parts
00180             std::stringstream lss(leftline);
00181             std::stringstream rss(rightline);
00182 
00183             while(getline(lss,paramstring, ',')) {
00184                 lcnt++;
00185 
00186                 if(lcnt == 3) {
00187                     lfreqstr = paramstring;
00188                 }
00189 
00190                 else if(lcnt == 4) {
00191                     lnoisestr = paramstring;
00192                 }
00193             }
00194 
00195             while(getline(rss,paramstring, ',')) {
00196                 rcnt++;
00197 
00198                 if(rcnt == 3) {
00199                     rfreqstr = paramstring;
00200                 }
00201 
00202                 else if(rcnt == 4) {
00203                     rnoisestr = paramstring;
00204                 }
00205             }
```

```
00206
00207             /*
00208              * Statement to convert lfreq, lnoise, rfreq, and rnoise from
00209              * strings to floats.
00210              * */
00211             lfreqval = atof(lfreqstr.c_str());
00212             lnoiseval = atof(lnoisestr.c_str());
00213             rfreqval = atof(rfreqstr.c_str());
00214             rnoiseval = atof(rnoisestr.c_str());
00215
00216             file.close();
00217         }
00218
00219         else {
00220             throw std::runtime_error("The patient file could not be opened.");
00221         }
00222
00223         leftparams.freq = lfreqval;
00224         leftparams.noise = lnoiseval;
00225         rightparams.freq = rfreqval;
00226         rightparams.noise = rnoiseval;
00227
00228         params[Side::Left] = leftparams;
00229         params[Side::Right] = rightparams;
00230
00231         return params;
00232     }
```

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.

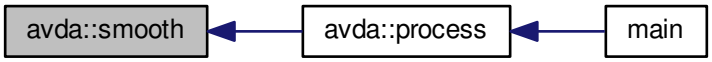
Parameters

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

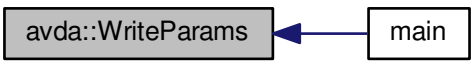
params	parameters to be written

the patient CSV file's filename

Definition at line 241 of file fileio.hpp.

```
00241                                                                    {
00242         char temp[80];
00243         std::ofstream file(filename.c_str(),
00244                 std::ofstream::out | std::ofstream::app);
00245
00246         //Gives pointer measurementtime a data type of time_t.
00247         time_t measurementtime;
00248         time(&measurementtime); //Gets the current time.
00249         strftime(temp, 80, "%c", localtime(&measurementtime));
00250         std::string fTime = std::string(temp);
00251
00252         //if statement to print the Left side parameters to the patient file.
00253         if(file.is_open()) {
00254             file << fTime + "," + "Left" + ","
00255                 + std::to_string(params[Side::Left].freq)
00256                 + ", " + std::to_string(params[Side::Left].noise) << std::endl;
00257         }
00258
00259         //if statement to print the Right side parameters to the patient file.
00260         if(file.is_open()) {
00261             file << fTime + "," + "Right" + ","
00262                 + std::to_string(params[Side::Right].freq)
00263                 + ", " + std::to_string(params[Side::Right].noise) << std::endl;
00264         }
00265
00266         else {
00267             std::cout << "Patient file can not be opened!" << std::endl;
00268         }
00269
00270         file.close();
00271     }
```

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

```
00001 PROJECT_NAME = "The Automatic Vasospasm Detection Application"
00002
00003 INPUT = src/ etc/doxygen.config makefile README.md
00004 OUTPUT_DIRECTORY = doc/
00005
00006 GENERATE_HTML = YES
00007 GENERATE_RTF = YES
00008 GENERATE_LATEX = YES
00009 GENERATE_MAN = YES
00010 GENERATE_XML = NO
00011 GENERATE_DOCBOOK = NO
00012
00013 USE_PDF_LATEX = YES
00014 USE_PDF_HYPERLINKS = YES
00015
00016 RECURSIVE = YES
00017 SOURCE_BROWSER = YES
00018 SOURCE_TOOLTIPS = YES
00019 EXTRACT_ALL = YES
00020 DISABLE_INDEX = NO
00021 GENERATE_TREEVIEW = YES
00022 SEARCHENGINE = YES
00023 SERVER_BASED_SEARCH = NO
00024 USE_MDFILE_AS_MAINPAGE = README.md
00025
00026 LATEX_SOURCE_CODE = YES
00027 STRIP_CODE_COMMENTS = YES
00028 INLINE_SOURCES = YES
00029
00030 HAVE_DOT = YES
00031 CALL_GRAPH = YES
00032 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

```
00001 GCC = g++ -g -std=gnu++14
00002
00003 avda:
00004    $(GCC) src/main.cpp -o bin/avda
00005
00006 count:
00007    grep -r "src/" -e "Samuel Andrew Wisner" -l | xargs wc -l
00008
00009 docs:
00010    rm -r doc/
00011    doxygen etc/doxygen.config
00012    cd doc/latex; make pdf;
00013    git reset
00014    git add doc/.
00015    git commit -m "Updated documentation."
00016    git push
00017
00018 fileio-test:
00019    $(GCC) src/fileio_test.cpp -o bin/fileiotest
00020
00021 patient-name-test:
00022    $(GCC) src/patient_name_test.cpp -o bin/patnametest
00023
00024 process-test:
00025    $(GCC) src/process_test.cpp -o bin/proctest
00026
00027 stdin-clear-test:
00028    $(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

```
00001 # The Automatic Vasospasm Detection Application
```

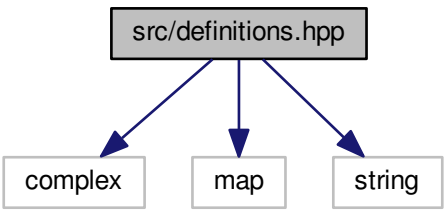
```
00002
00003 ## Introduction
00004 The Automatic Vasospasm Detection Application (or Algorithm, depending on the
00005 usage), AVDA, is an application to objectively detect the presence of vasospasms
00006 based on comparisons of parameters extracted from transcranial doppler audio.
00007
00008 ## Setup
00009 AVDA is intended to be compiled on machines running Linux, though it could
00010 likely be adapter for other environments. It must be downloaded from GitHub.com
00011 and compiled locally. To do this, navigate to the directory in which AVDA should
00012 be placed, then execute the following commands
00013
00014    git clone https://github.com/sawbg/avda
00015    cd avda
00016    make
00017
00018 Sucessfully cloning, compilation, and execution of AVDA requires up-to-date
00019 versions of the following executables:
00020
00021 * git
00022 * make
00023 * gcc (4.9)
00024 * arecord
00025
00026 ## FAQ
00027
00028 * **Why was this project developed?** This project was developed as a course
00029 project by two gradute students at the University of Alabama at Birmingham
00030 School of Engineering, Nicholas Nolan and Andrew Wisner.
00031
00032 * **Is AVDA an active project?** Though it is not planned to develop AVDA
00033 further in the near future, it is hoped that the algorithm discovered and
00034 implemented can be used and built upon by researchers to fully automate the
00035 detection of vasospasms.
00036
00037 * **AVDA is returning unusually low or high parameters. Why might this be?** In
00038   development, this occured when the mic-in volume was set too high. It is
00039 likely in this senario that clipping is happening or that the signal (or a
00040 strong enough signal) has no been received.
00041
00042 * **How will AVDA be affected by the machine uprising?** The University
00043   supercomputer, Cheaha, has assured us that AVDA will not be needed after the
00044 uprising occures.
00045
00046 * **What about more specific questions?** Questions relating to AVDA not
00047 covered in this FAQ may be sent to the AVDA team via awisner94@gmail.com.
```

8.7 src/definitions.hpp File Reference

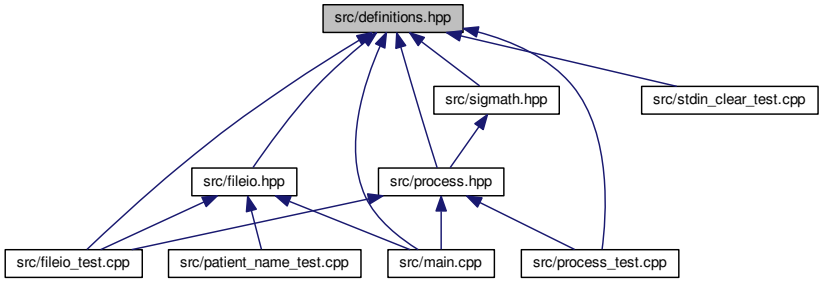
Contains declarations of system-independant (universal size) integers and float types, shortened type names for some commonly used types, and enumerations.

```
#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>
```

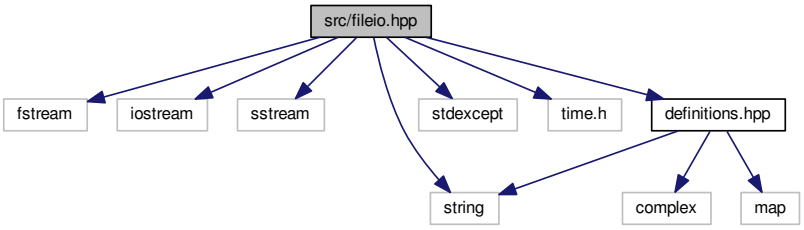
```
00014 #include <string>
00015
00016 #define ENUM signed char
00017
00018 // Type definitions
00019
00020 typedef unsigned char byte;
00021 typedef unsigned char uint8;
00022 typedef signed char sint8;
00023
00024 typedef unsigned short uint16;
00025 typedef signed short sint16;
00026
00027 typedef 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
00141 namespace avda {
00145     enum class Side { Left, Right };
00146 }
00147
00148
00149 // Doxygen documentation for other files.
00150
00171 #endif
```

8.9 src/fileio.hpp File Reference

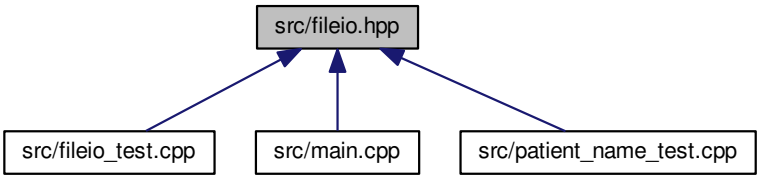
Contains functions related to file I/O use in this program.

```
#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

```
00001
00009 #ifndef fileio_H
00010 #define fileio_H
00011
00012 #include <fstream>
00013 #include <iostream>
00014 #include <sstream>
00015 #include <string>
00016 #include <stdexcept>
00017 #include <time.h>
00018
00019 #include "definitions.hpp"
00020
00021 namespace avda {
00033     std::string PatientName() {
00034         std::string fname = "";
00035         std::string mname = "";
00036         std::string lname = "";
00037         std::string patfil = "";
00038         std::string patientname = "";
00039         uint32 track1 = 0;
00040         uint32 track2 = 0;
00041         uint32 track3 = 0;
00042
00043         do {
00044             std::cout << "Please enter the patients name." << std::endl;
00045             std::cout << "First name: ";
00046             std::cin >> fname;
00047             std::cout << "Middle name: ";
00048             std::cin >> mname;
00049             std::cout << "Last name: ";
00050             std::cin >> lname;
00051
00052             // creates new std::string with path to patient file
00053             patientname = PATIENT_PATH + lname + ", " + fname
00054                 + " " + mname + ".csv";
00055
00056             // prints out patientname. shows user the path to the patient file
00057             //std::cout << patientname << std::endl << std::endl;
00058             std::ifstream file(patientname.c_str());
00059
00060             if (file.good()) {
00061                 track1 = 1;
00062             }
00063
00064             /*
00065              * Compares patientname to existing files and lets user know
00066              * if the file does not exist.
00067              */
00068             else if (!file.good()) {
00069                 /*
00070                  * Do while statement to continue asking user about the file
00071                  * if their input is not acceptable
00072                  */
00073                 do {
00074                     std::cout << "Patient file does not exist, would you like "
00075                         "to create file or re-enter their name?" << std::endl;
00076                     std::cout << "  *Type 'create' and press enter key "
00077                         "to create the patient file." << std::endl;
00078                     std::cout << "  *Type 'reenter' and press enter key "
00079                         "to re-enter the patients name." << std::endl;
00080                     std::cout << std::endl;
00081                     std::cin >> patfil;
00082
00083                     /*
00084                      * patfil equals create, track1 and 2 will increase
00085                      * escaping both do while loops
00086                      */
00087                     if(patfil == "create") {
```

```
00088                       std::ofstream createfile(patientname.c_str());
00089                       track1 = 1;
00090                       track2 = 1;
00091                       track3 = 1;
00092                       createfile << CSV_HEADER << std::endl;
00093                       createfile.flush();
00094                       createfile.close();
00095                   }
00096
00097                   /*
00098                    *patfil equals renter, track1 will remain zero allowing
00099                    *user to reenter the patient name.
00100                    */
00101                   else if(patfil == "reenter") {
00102                       track1 = 0;
00103                       track2 = 1;
00104                   }
00105
00106                   /*
00107                    *The users input was neither create or reenter. User
00108                    *must enter patient name again.
00109                    */
00110                   else {
00111                       std::cout << std::endl;
00112                       std::cout << "Your input is not acceptable." << std::endl;
00113                       std::cout << std::endl;
00114                   }
00115               }while(track2 == 0);
00116           }
00117       } while (track1 == 0);
00118
00119       return patientname; //returns the path to the patient file
00120   }
00121
00131   std::map<Side, DataParams> ReadParams(auto filename) {
00132       std::map<Side, DataParams> params;
00133       DataParams leftparams;
00134       DataParams rightparams;
00135
00136       std::ifstream file(filename.c_str());
00137       std::string leftline;
00138       std::string rightline;
00139       std::string leftsearch = "Left";
00140       std::string rightsearch = "Right";
00141       std::string paramstring;
00142       std::string lfreqstr;
00143       std::string lnoisestr;
00144       std::string rfreqstr;
00145       std::string rnoisestr;
00146       uint32 lcnt = 0;
00147       uint32 rcnt = 0;
00148       float32 lfreqval;
00149       float32 lnoiseval;
00150       float32 rfreqval;
00151       float32 rnoiseval;
00152
00153       /*
00154        * if statement which uses ifstream function to open patient file
00155        * filename)
00156        */
00157       if(file.is_open()) {
00158           /*
00159            * While statement to find the first Left line and save to
00160            *leftline as string.
00161            */
00162           while (getline(file, leftline)) {
00163               if(leftline.find(leftsearch, 0) != std::string::npos) {
00164                   break;
00165               }
00166
00167           }
00168
00169           /*
00170            * While statement to find first right line and save to rightline
00171            * as string.
00172            */
00173           while (getline(file,rightline)) {
00174               if(rightline.find(rightsearch, 0) != std::string::npos) {
00175                   break;
00176               }
00177           }
00178
00179           // Code to break leftline and rightline into its parts
00180           std::stringstream lss(leftline);
00181           std::stringstream rss(rightline);
00182
00183           while(getline(lss,paramstring, ',')) {
```

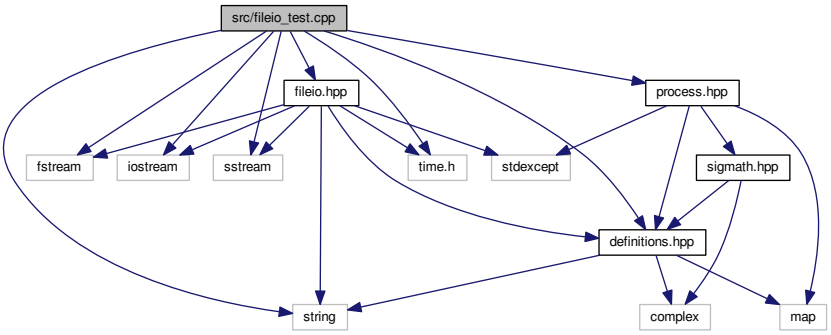
```
00184                 lcnt++;
00185
00186                 if(lcnt == 3) {
00187                     lfreqstr = paramstring;
00188                 }
00189
00190                 else if(lcnt == 4) {
00191                     lnoisestr = paramstring;
00192                 }
00193             }
00194
00195             while(getline(rss,paramstring, ',')) {
00196                 rcnt++;
00197
00198                 if(rcnt == 3) {
00199                     rfreqstr = paramstring;
00200                 }
00201
00202                 else if(rcnt == 4) {
00203                     rnoisestr = paramstring;
00204                 }
00205             }
00206
00207             /*
00208              * Statement to convert lfreq, lnoise, rfreq, and rnoise from
00209              * strings to floats.
00210              * */
00211             lfreqval = atof(lfreqstr.c_str());
00212             lnoiseval = atof(lnoisestr.c_str());
00213             rfreqval = atof(rfreqstr.c_str());
00214             rnoiseval = atof(rnoisestr.c_str());
00215
00216             file.close();
00217         }
00218
00219         else {
00220             throw std::runtime_error("The patient file could not be opened.");
00221         }
00222
00223         leftparams.freq = lfreqval;
00224         leftparams.noise = lnoiseval;
00225         rightparams.freq = rfreqval;
00226         rightparams.noise = rnoiseval;
00227
00228         params[Side::Left] = leftparams;
00229         params[Side::Right] = rightparams;
00230
00231         return params;
00232     }
00233
00241     void WriteParams(std::map<Side, DataParams> params, auto filename) {
00242         char temp[80];
00243         std::ofstream file(filename.c_str(),
00244                 std::ofstream::out | std::ofstream::app);
00245
00246         //Gives pointer measurementtime a data type of time_t.
00247         time_t measurementtime;
00248         time(&measurementtime); //Gets the current time.
00249         strftime(temp, 80, "%c", localtime(&measurementtime));
00250         std::string fTime = std::string(temp);
00251
00252         //if statement to print the Left side parameters to the patient file.
00253         if(file.is_open()) {
00254             file << fTime + "," + "Left" + ","
00255                 + std::to_string(params[Side::Left].freq)
00256                 + ", " + std::to_string(params[Side::Left].noise) << std::endl;
00257         }
00258
00259         //if statement to print the Right side parameters to the patient file.
00260         if(file.is_open()) {
00261             file << fTime + "," + "Right" + ","
00262                 + std::to_string(params[Side::Right].freq)
00263                 + ", " + std::to_string(params[Side::Right].noise) << std::endl;
00264         }
00265
00266         else {
00267             std::cout << "Patient file can not be opened!" << std::endl;
00268         }
00269
00270         file.close();
00271     }
00272 }
00273
00274 #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

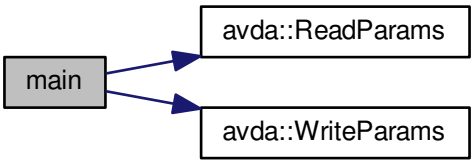
8.11.2.1 int main ( )

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

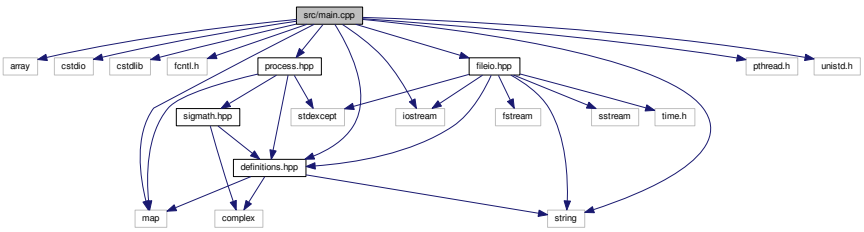
```
00001
00007 #include <fstream>
00008 #include <iostream>
00009 #include <sstream>
00010 #include <string>
00011 #include <time.h>
00012
00013 #include "definitions.hpp"
00014 #include "fileio.hpp"
00015 #include "process.hpp"
00016
00017 using namespace std;
00018 using namespace avda;
00019
00023 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 extra newline character inserted into stdin buffer after PatientName() is run

Definition in file main.cpp.

8.13.2 Function Documentation

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

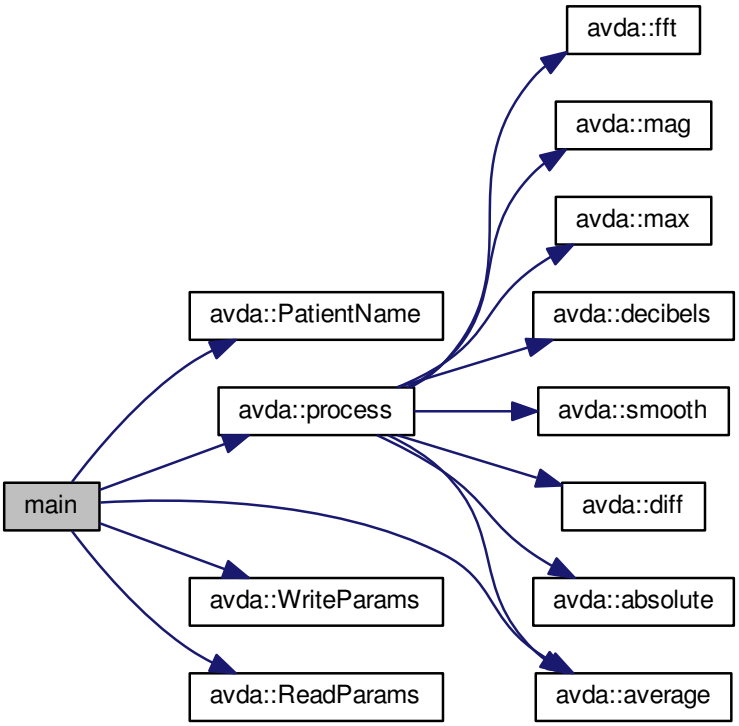
The main program for this project. It will detect avdaspasms over a period of days.

Definition at line 30 of file main.cpp.

```
00030                                {
00031     // Recorded audio buffer
00032     float32* buffer = (float32*)std::malloc(BUFFER_SIZE);
00033     bool cont = true;  // whether to continue in the recording loop
00034     DataParams params[REC_COUNT];  // holds DataParam's from recordings
00035     string filename = PatientName();  // generate name for patient's file
00036     map<Side, DataParams> results;  // parameters by side
00037 
00038     // arecord command
00039     const string recCommand = string("arecord -t raw -d ")
00040         + to_string(DURATION) + string(" -D plughw:1,0 -f FLOAT -q -r ")
00041         + to_string(SAMPLE_FREQ) + string(" ") + TEMP_FILE;
00042 
00043     // Recording
00044     while(cont) {
00045         for(uint8 i = 0; i < REC_COUNT; i++) {
00046             // prompt
00047             cout << "Press [ ENTER ] to begin analysis for the "
00048                 << (i < REC_COUNT / 2 ? "left" : "right") << " side, depth #"
00049                 << (((i >= REC_COUNT / 2) ? (i - REC_COUNT / 2) : i) + 1)
00050                 << " ";
00051             getchar();  // wait for ENTER to be pressed
00052             cout << "Analyzing..." << endl;
00053 
00054             system(recCommand.c_str());
00055             usleep(DURATION*1000000 + 1500000);  // sleep DURATION + 1.5 seconds
00056 
00057             int file = open(TEMP_FILE.c_str(), O_RDONLY);  // open temp file
```

```
00058             int retRead = read(file, buffer, BUFFER_SIZE);  // copy to buffer
00059             close(file);  // close temp file
00060             remove(TEMP_FILE.c_str());  // delete temp file
00061 
00062             // if something goes wrong reading the temp file, program exits
00063             if(file < 0 || retRead < BUFFER_SIZE) {
00064                 cerr << "An error occurred reading the doppler audio! "
00065                     "The program will now exit." << endl;
00066                 return ERROR;
00067             }
00068 
00069             // process and store parameters
00070             params[i] = process(buffer, SAMPLE_COUNT,
      SAMPLE_FREQ);
00071             cout << "The analysis is complete." << endl << endl;
00072         }
00073 
00074         // calculate averaged parameters
00075         results[Side::Left] = average(params, REC_COUNT / 2);
00076         results[Side::Right] = average(&params[REC_COUNT / 2], REC_COUNT / 2);
00077 
00078         cout << "Analysis is complete." << endl << endl;
00079 
00080         // print averaged side analysis
00081         for(int i = 0; i < 2; i++) {
00082             Side side = (Side)i;
00083             cout << (side == Side::Left ? "[LEFT]" : "[RIGHT]") << endl;
00084             cout << "Drop-off frequency: " << (uint16)(results[side].freq + 0.5)
00085                 << " Hz" << endl;
00086             cout << "Average relative noiseband power: "
00087                 << (sint16)(results[side].noise - 0.5) << " dB" << endl <<endl;
00088         }
00089 
00090         cont = results[Side::Left].freq > MAX_DROP_FREQ
00091             || results[Side::Right].freq > MAX_DROP_FREQ;
00092 
00093         if(cont) {
00094             cout << "An error in aquisition of the doppler audio has occurred! "
00095                 "Ensure the connection from the doppler machine to this device "
00096                 "is secure and the connection uninterruptable." << endl << endl;
00097         }
00098     }
00099 
00100     free(buffer);  // free buffer to prevent memory leak
00101     WriteParams(results, filename);
00102 
00103     // examine likelihood of avdaspasm
00104     try {
00105         map<Side, DataParams> baseParams = ReadParams(filename);
00106         map<Side, bool> comparison;
00107 
00108         for(uint8 i = 0; i < 2; i++) {
00109             Side side = (Side)i;
00110             float comp = fabs(results[side].freq - baseParams[side].freq)
00111                 * fabs(baseParams[side].noise - results[side].noise);
00112             comparison[side] = comp > DET_THRESH;
00113         }
00114 
00115         string which;
00116 
00117         if(comparison[Side::Left] && !comparison[Side::Right]) {
00118             which = "The left";
00119         } else if(!comparison[Side::Left] && comparison[Side::Right]) {
00120             which = "The right";
00121         } else if (comparison[Side::Left] && comparison[Side::Right]) {
00122             which = "Both";
00123         } else {
00124             which = "Neither";
00125         }
00126 
00127         cout << which << " side seems to show evidence of a vasospasm." << endl;
00128     } catch(runtime_error ex) {
00129         cout << "These values will be stored as the baseline parameters to "
00130             "which all future parameters are compared." << endl;
00131     }
00132 }
```

Here is the call graph for this function:



8.14 main.cpp

```
00001
00009 #include <array>
00010 #include <cstdio>
00011 #include <cstdlib>
00012 #include <fcntl.h>
00013 #include <iostream>
00014 #include <map>
00015 #include <pthread.h>
00016 #include <string>
00017 #include <unistd.h>
00018
00019 #include "definitions.hpp"
00020 #include "fileio.hpp"
00021 #include "process.hpp"
00022
00023 using namespace std;
00024 using namespace avda;
00025
00030 int main(int argc, char** argv) {
00031     // Recorded audio buffer
00032     float32* buffer = (float32*)std::malloc(BUFFER_SIZE);
00033     bool cont = true;  // whether to continue in the recording loop
00034     DataParams params[REC_COUNT];  // holds DataParam's from recordings
00035     string filename = PatientName();  // generate name for patient's file
00036     map<Side, DataParams> results;  // parameters by side
00037
00038     // arecord command
00039     const string recCommand = string("arecord -t raw -d ")
00040         + to_string(DURATION) + string(" -D plughw:1,0 -f FLOAT -q -r ")
00041         + to_string(SAMPLE_FREQ) + string(" ") + TEMP_FILE;
00042
00043     // Recording
00044     while(cont) {
```

```
00045         for(uint8 i = 0; i < REC_COUNT; i++) {
00046             // prompt
00047             cout << "Press [ ENTER ] to begin analysis for the "
00048                 << (i < REC_COUNT / 2 ? "left" : "right") << " side, depth #"
00049                 << (((i >= REC_COUNT / 2) ? (i - REC_COUNT / 2) : i) + 1)
00050                 << " ";
00051             getchar();  // wait for ENTER to be pressed
00052             cout << "Analyzing..." << endl;
00053
00054             system(recCommand.c_str());
00055             usleep(DURATION*1000000 + 1500000);  // sleep DURATION + 1.5 seconds
00056
00057             int file = open(TEMP_FILE.c_str(), O_RDONLY);  // open temp file
00058             int retRead = read(file, buffer, BUFFER_SIZE);  // copy to buffer
00059             close(file);  // close temp file
00060             remove(TEMP_FILE.c_str());  // delete temp file
00061
00062             // if something goes wrong reading the temp file, program exits
00063             if(file < 0 || retRead < BUFFER_SIZE) {
00064                 cerr << "An error occurred reading the doppler audio! "
00065                     "The program will now exit." << endl;
00066                 return ERROR;
00067             }
00068
00069             // process and store parameters
00070             params[i] = process(buffer, SAMPLE_COUNT,
      SAMPLE_FREQ);
00071             cout << "The analysis is complete." << endl << endl;
00072         }
00073
00074         // calculate averaged parameters
00075         results[Side::Left] = average(params, REC_COUNT / 2);
00076         results[Side::Right] = average(&params[REC_COUNT / 2], REC_COUNT / 2);
00077
00078         cout << "Analysis is complete." << endl << endl;
00079
00080         // print averaged side analysis
00081         for(int i = 0; i < 2; i++) {
00082             Side side = (Side)i;
00083             cout << (side == Side::Left ? "[LEFT]" : "[RIGHT]") << endl;
00084             cout << "Drop-off frequency: " << (uint16)(results[side].freq + 0.5)
00085                 << " Hz" << endl;
00086             cout << "Average relative noiseband power: "
00087                 << (sint16)(results[side].noise - 0.5) << " dB" << endl <<endl;
00088         }
00089
00090         cont = results[Side::Left].freq > MAX_DROP_FREQ
00091             || results[Side::Right].freq > MAX_DROP_FREQ;
00092
00093         if(cont) {
00094             cout << "An error in aquisition of the doppler audio has occurred! "
00095                 "Ensure the connection from the doppler machine to this device "
00096                 "is secure and the connection uninterruptable." << endl << endl;
00097         }
00098     }
00099
00100     free(buffer);  // free buffer to prevent memory leak
00101     WriteParams(results, filename);
00102
00103     // examine likelihood of avdaspasm
00104     try {
00105         map<Side, DataParams> baseParams = ReadParams(filename);
00106         map<Side, bool> comparison;
00107
00108         for(uint8 i = 0; i < 2; i++) {
00109             Side side = (Side)i;
00110             float comp = fabs(results[side].freq - baseParams[side].freq)
00111                 * fabs(baseParams[side].noise - results[side].noise);
00112             comparison[side] = comp > DET_THRESH;
00113         }
00114
00115         string which;
00116
00117         if(comparison[Side::Left] && !comparison[Side::Right]) {
00118             which = "The left";
00119         } else if(!comparison[Side::Left] && comparison[Side::Right]) {
00120             which = "The right";
00121         } else if (comparison[Side::Left] && comparison[Side::Right]) {
00122             which = "Both";
00123         } else {
00124             which = "Neither";
00125         }
00126
00127         cout << which << " side seems to show evidence of a vasospasm." << endl;
00128     } catch(runtime_error ex) {
00129         cout << "These values will be stored as the baseline parameters to "
00130             "which all future parameters are compared." << endl;
```

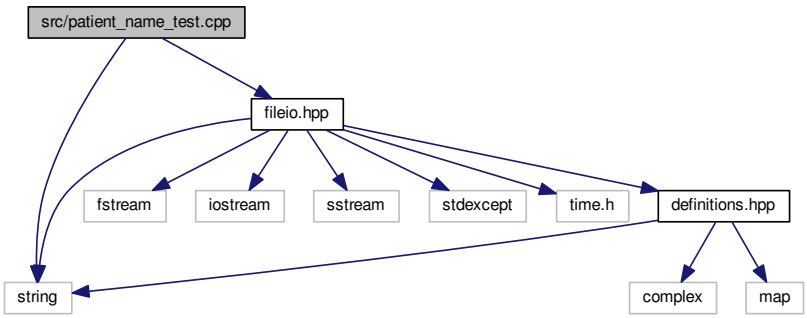
```
00131     }
00132 }
```

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

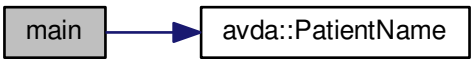
8.15.2.1 int main (int *argc*, char ** *argv*)

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

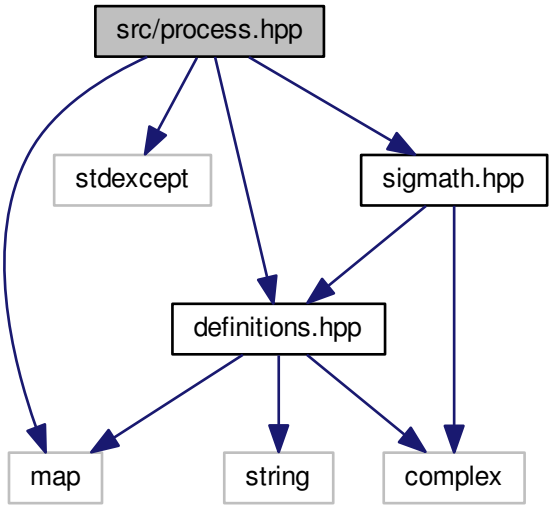
```
00001
00007 #include <string>
00008
00009 #include "fileio.hpp"
00010
00011 using namespace std;
00012 using namespace avda;
00013
00017 int main(int argc, char** argv) {
00018     string filename = PatientName();
00019     cout << filename;
00020 }
```

8.17 src/process.hpp File Reference

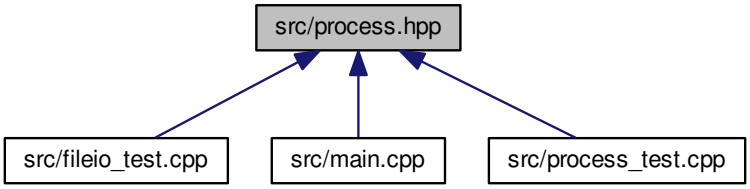
Contains functions related to the program's threaded processing of audio data.

```
#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

```
00001
00008 #ifndef process_H
00009 #define process_H
00010
00011 #include <map>
00012 #include <stdexcept>
00013
00014 #include "definitions.hpp"
00015 #include "sigmath.hpp"
00016
00017 namespace avda {
00048     DataParams process(float32* data, uint32 size,
      float32 samplingRate) {
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);
```

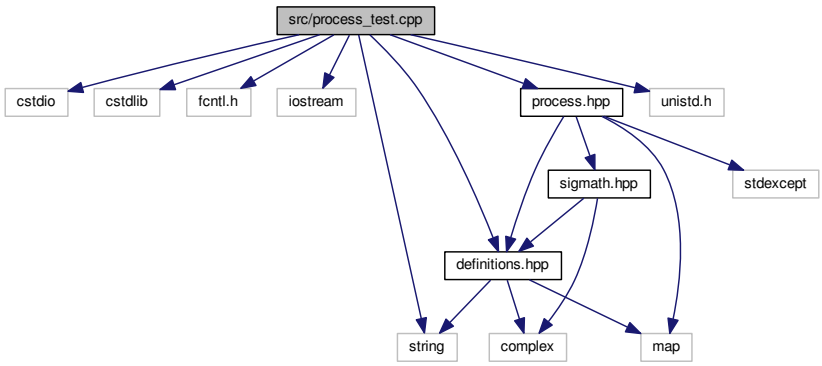
```
00107
00108         return params;
00109     }
00110 }
00111
00112 #endif
```

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

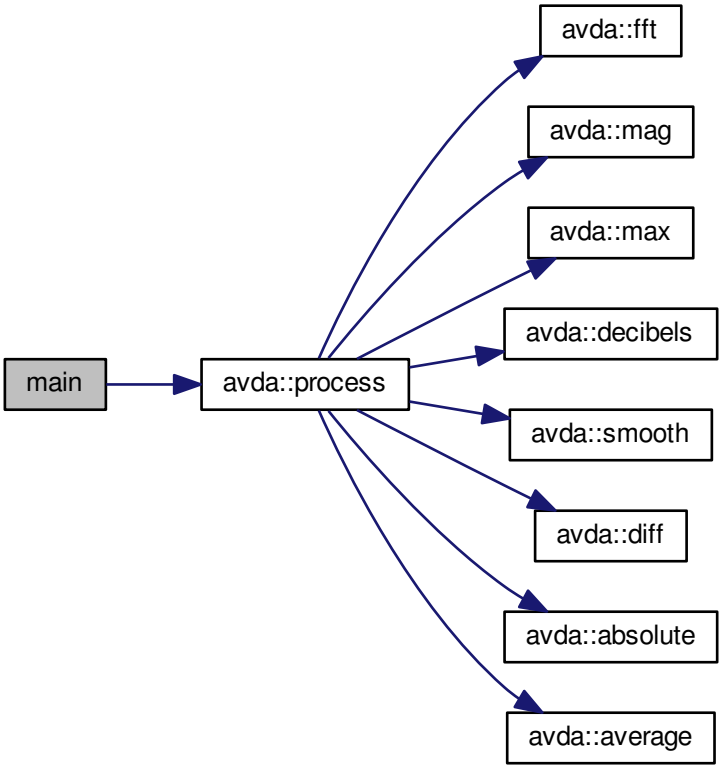
8.19.3.1 int main (int *argc*, char ** *argv*)

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

```
00001
00007 #include <cstdio>
00008 #include <cstdlib>
00009 #include <fcntl.h>
00010 #include <iostream>
00011 #include <string>
00012 #include <unistd.h>
00013
00014 #include "definitions.hpp"
00015 #include "process.hpp"
00016
00017 #define COUNT 131072
00018
00019 using namespace std;
00020 using namespace avda;
00021
00025 int main(int argc, char** argv) {
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;
```

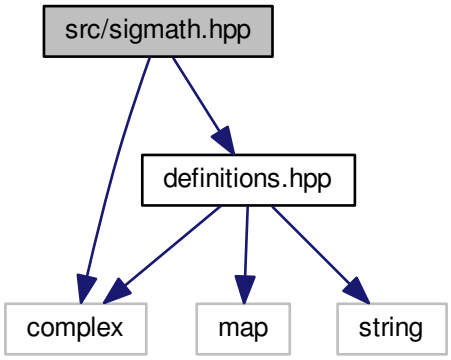
```
00039     }
00040
00041     close(file);
00042
00043     DataParams params = process(buffer, COUNT, SAMPLE_FREQ);
00044     free(buffer);
00045     cout << "Cutoff: " << params.freq << endl;
00046     cout << "Noise: " << params.noise << endl;
00047 }
```

8.21 src/sigmath.hpp File Reference

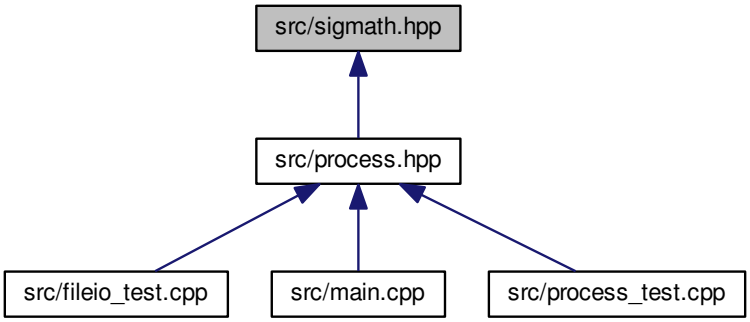
Contains the functions necessary to perform the mathematical operations required by this program.

```
#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

```
00001
00009 #ifndef sigmath_H
00010 #define sigmath_H
00011
00012 #include <complex>
00013 #include "definitions.hpp"
00014
00015 namespace avda {
00016     // PROTOTYPES
00017
00026     void absolute(float32* data, uint32 size);
00027
00037     float32 average(float32* data, uint32 size);
00038
00049     DataParams average(DataParams* params, uint8 size);
00050
00062     void decibels(float32* data, uint32 size);
00063
00072     void diff(float32* data, uint32 size);
00073
00085     void fft(cfloat32* data, uint32 size);
00086
00096     void mag(cfloat32* orig, float32* newmags, uint32 size);
00097
00107     Maximum max(float32* data, uint32 size);
00108
00119     void smooth(float32* data, uint32 size, uint16 order);
00120
00121     // DEFINITIONS
00122
00123     void absolute(float32* data, uint32 size) {
00124         for(uint32 i = 0; i < size; i++) {
00125             data[i] = fabsf(data[i]);
00126         }
00127     }
00128
00129     float32 average(float32* data, uint32 size) {
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     }
00139 
00140     DataParams average(DataParams* params, uint8 size) {
00141         DataParams ave;
00142 
00143         for(uint8 i = 0; i < size; i++) {
00144             //freq is an attribute. this is how to add structure attributes
00145             ave.freq += params[i].freq;
00146             ave.noise += params[i].noise;
00147         }
00148 
00149         ave.freq /= size;
00150         ave.noise /= size;
00151         return ave;
00152     }
00153 
00154     void decibels(float32* data, uint32 size) {
00155         for(uint32 i = 0; i < size; i++) {
00156             data[i] = 20 * log10(data[i]);
00157         }
00158     }
00159 
00160     void diff(float32* data, uint32 size) {
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     }
00172 
00173     void fft(cfloat32* data, uint32 size) {
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         }
```

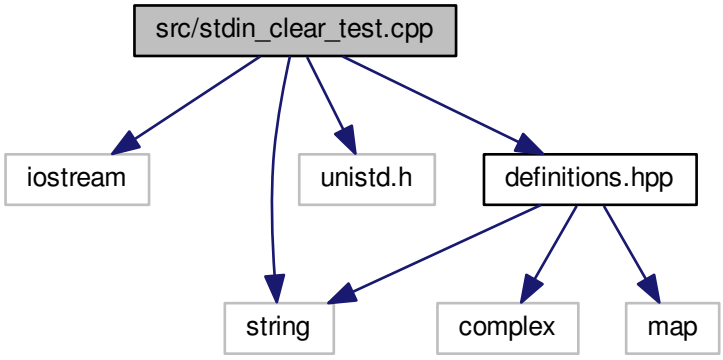
```
00219     }
00220
00221     void mag(cfloat32* orig, float32* newmags, uint32 size) {
00222         //loop to run throught the length of array orig
00223         for(uint32 n = 0; n < size; n++) {
00224             /*
00225              * abs should calculate the magnitude of complex array elements.
00226              * saves to new array
00227              */
00228             newmags[n] = std::abs(orig[n]);
00229         }
00230     }
00231
00232     Maximum max(float32* data, uint32 size) {
00233         Maximum m;
00234
00235         //loop to run through the length of array data
00236         for (uint32 i = 0; i < size; i++) {
00237             /*
00238              * when value at data[i] is above max.value,
00239              * sets max.value equal to data[i] and max.index equal to i
00240              */
00241             if (data[i] > m.value) {
00242                 m.value = data[i];
00243                 m.index = i;
00244             }
00245         }
00246
00247         return m;
00248     }
00249
00250     void smooth(float32* data, uint32 size, uint16 order) {
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     }
00268 }
00269
00270 #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.

```
00022                                {
00023     char text1[COUNT];
00024     char text2[COUNT];
00025
00026     cout << "Enter text to ignore: ";
00027     cout.flush();
00028     read(STDIN_FILENO, &text1, COUNT);
00029 //  fflush(stdin);
00030     cout << endl << "Enter text to print: ";
00031     cout.flush();
00032     read(STDIN_FILENO, &text2, COUNT);
00033     cout << endl << "In buffer: " << text2 << endl;
00034 }
```

8.24 stdin_clear_test.cpp

```
00001
00008 #include <iostream>
00009 #include <string>
00010 #include <unistd.h>
00011
00012 #include "definitions.hpp"
00013
00014 #define COUNT 80
00015
00016 using namespace std;
00017 using namespace avda;
00018
00022 int main(int argc, char** argv) {
00023     char text1[COUNT];
00024     char text2[COUNT];
00025
00026     cout << "Enter text to ignore: ";
00027     cout.flush();
00028     read(STDIN_FILENO, &text1, COUNT);
00029 //  fflush(stdin);
00030     cout << endl << "Enter text to print: ";
00031     cout.flush();
00032     read(STDIN_FILENO, &text2, COUNT);
00033     cout << endl << "In buffer: " << text2 << endl;
00034 }
```

Index