Processing Signal Viewer

System Documentation

Authors: Quintess Barnhoorn, Loes Erven, Kayla Gericke, Mignon Hagemeijer, Nick van der Linden, Peter van Olmen, Sander van ‘t Westeinde

Introduction

This is the system documentation for the Processing EEG signal viewer. The Processing signal viewer was developed in order to have functionality similar to the MATLAB signal viewer, but be runnable on multiple platforms. So far, the program has been implemented and tested for Windows desktops and Android tablets. In this version, display of the signal in time and

frequency domains has been implemented, and some but not all of the preprocessing options of the MATLAB version are available.

Getting started

Getting started with Processing

What is Processing

Processing a flexible open-source sketch software for data visualization using Java. It is a graphical library and has its own IDE, but can also be used in Java IDE Eclipse. We used Processing because it provided a flexible way to visualize continuously updating data on multiple platforms. Other platforms for applications were either not powerful enough, and platforms meant for mobile games turned out to be too clunky to use for our purposes.

Downloading Processing

Processing can be downloaded at https://processing.org/download/, no installation is necessary. For this project Processing 3+ is needed. Note: only Processing 3+ is still supported thus using older versions is highly inadvisable.

Using Processing

There are plenty of tutorials on the basic principles of Processing here:
https://processing.org/tutorials/

And an official tutorial on how to get started with Processing is available here:
https://processing.org/tutorials/gettingstarted/

A few important things to note when using Processing:

1. You always need to start your project with:
   void setup(){
   }
   This method is run when and only when starting the program. Here, you set up the basic properties of the environment such as screen size. There can only be one setup method and it should not be called anywhere.

2. draw() method:
   After your have added *setup()* you always need to add:
   void draw(){
   }

This function is automatically called right after the setup() and shouldn't be called anywhere else explicitly; it's continuously called as long as the program is running. The screen is updated after the end of each *draw()* call. There can be only one *draw()* method.

3. A very important thing to note is that Processing treats each file you make as essentially being in one big file. A consequence is that if you make variables in draw(), or setup() they will always be accessible everywhere, even when you set them to private. Try to avoid this as much as possible by minimizing your first file to the absolute bare necessities and then adding new, well encapsulated classes.

Importing required libraries

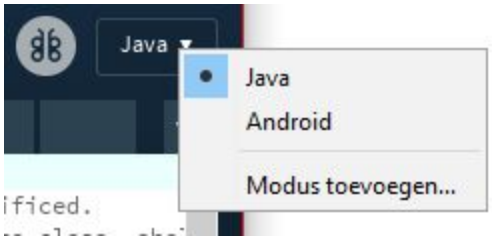
To import a library, go to Tools → Add tools → Libraries, then search for the library you want to import.
To run this program, you will need the following libraries:

- ControlP5
- Minim
- Ketai (Exclusively for the Android version)

Running the program

Picking a version

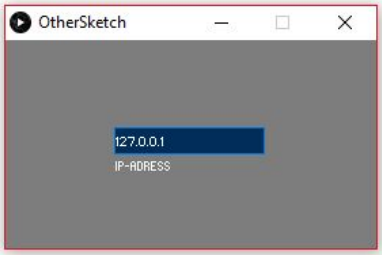
To switch between android and Java, there is a button in the upper right corner



Running on Desktop

1. Make sure you are in Java mode.
2. Start the signal proxy by running the debug_quickstart.bat file
    a. Make sure there are 3 windows popping up. Only then the buffer is running.
3. Open the EEGSignalViewer in Processing by pressing the button below:

4. A popup window will open, asking for an IP-address as shown in the picture below. Here, enter the IP address of the device on which the proxy is running. If you are running the buffer on the same device as Processing this should be the default IP address that has already been entered by default.



5. Hit enter and the program should run. You can now start selecting preprocessing and filter options and switch domains by clicking or tapping on the boxes.

Adding the buffer to Android

1. Install the Android APK app on your device (preferably somewhere you can easily find it later)
2. On your device, open the APK, it will download immediately
3. Go to the app, and select both server and Clients
4. Select *SignalProxyThreadke*

Running on Android

1. First make sure you have added the APK of the buffer to your device following the steps from the previous section.
2. Make sure Processing is in the Android mode. If not, switch to it as described in the *Picking a Version* section.
3. Become an Android developer :

a. Find your build number: this differs for each machine but for most it can be found in the tablet's or phone's settings under "software info" or "device info".
b. Click the build number 7 times.

4. Activate USB debug.
5. Connect the device to the computer with Processing.
6. Click on run. If your machine is not found automatically, select the machine using the file explorer on your computer.

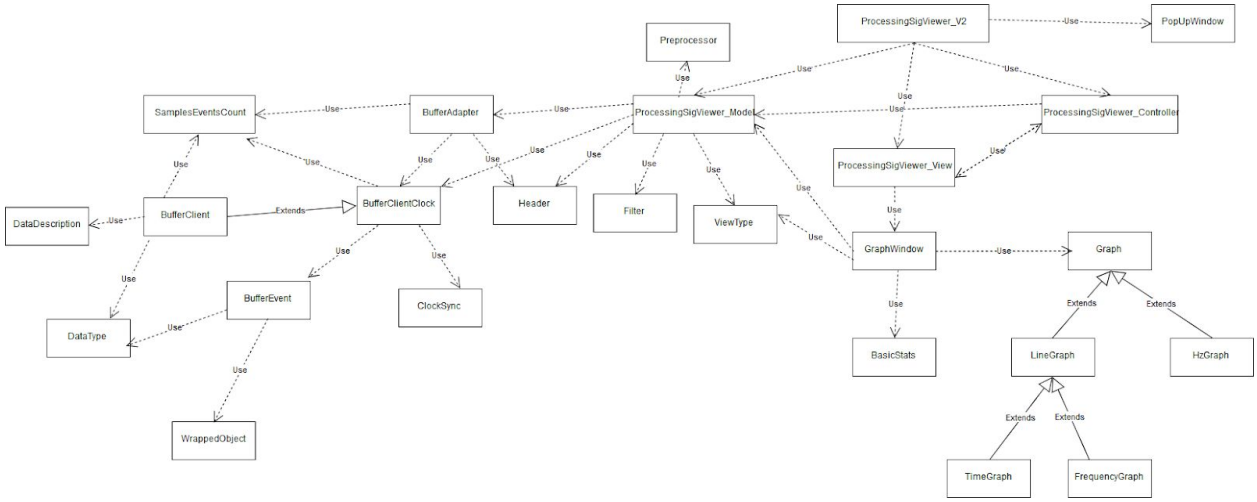
Running test data:

Sometimes you want your data to run with very specific test cases. If so there are a few lines of code that need to be changed such that the test data is run.

1. Uncomment or write the test you want to use in *public void updateTestData()* in the class ProcessingSigViewer_Model.
2. Comment out the lines in the GraphWindow class saying *model.updateData()* and uncomment the line saying *model.updateTestData()*.
3. When run, the signal viewer will display the data as defined in updateTestData() instead of the data from the buffer.

Getting started with the Processing Signal Viewer

Classes



- **BasicStats**: Handles simple mathematical operations.
- **BufferAdapter**: Updates the data; newer version of SignalViewer.
- **BufferClient**: A class that reads the data and passes it on to the graph.
- **BufferClientClock**: Extension of bufferclient to add ability to fill in a negative event sample number, based on the output of the system clock and tracking the mapping between clock-time and sample-time.
- **BufferEvent**: A class for wrapping buffer events.

- **Clocksync**: Handles the rate at which data is updated.
- **DataDescriptions**: Simple class for describing data blocks as used in GET_DAT and PUT_DAT requests.
- **DataType**: A class for defining data types and routines for conversion to Java objects.
- **Filter**: Enumeration of the available types of filters.
- **FrequencyGraph**: Used for drawing a LineGraph in the Frequency domain.
- **Graph**: Abstract class used for drawing a graph of any kind.
- **GraphWindow**: Handles the behaviour of the window containing the graphs and stores information common to all graphs.
- **Header**: A class for wrapping a buffer header structure.
- **HzGraph**: Used to visualize 50Hz line noise as a colour.
- **LineGraph**: Used for drawing a graph with a line connecting data points.
- **PopUpWindow**: Contains a class named OtherSketch, which makes the pop-up in the java version.
- **Preprocessor**: Handles things related to processing the data.
- **ProcessingSigViewer_Model**: Handles the internal state of the signal viewer, specifically the incoming data and what type of data that is.
- **ProcessingSigViewer_Controller**: Handles all external events.
- **ProcessingSigViewer_View**: Handles everything that needs to be shown to the user.
- **ProcessingSigViewer_V2**: This is the main file. It calls the Model, View and Controller.
- **SamplesEventsCount**: Simple class for describing data blocks as used in GET_DAT and PUT_DAT request.
- **SignalViewer**: Updates the data.
- **TimeGraph**: Used for drawing a LineGraph in the Time domain.
- **ViewType**: An enumeration of the different types of graph.
- **WrappedObject**: A class for wrapping relevant Java objects in a way that is easily convertible to FieldTrip data types.

Class descriptions

PApplets

A PApplet is a special kind of Java program that creates a new window and draws graphics on it.

ProcessingSigViewer

This file creates the main window of the program, all the objects on it, and the behaviour of those objects.

settings() is called first, before the program is run. When it is run, setup() is called, which creates the model, view, controller and the OtherSketch (the popup window for entering a IP address.

draw() is called directly after setup(), and is executed in a loop, which continuously tells the view to update what is seen on the screen.

controlEvent() is called whenever a controller value is changed or a tab is activated.

OtherSketch

This file creates the pop-up window in the java version. This pop-up contains a textfield in which the IP-address must be filled in.

The constructor is called from setup() from ProcessingSigViewer, and launches the window as a pop-up. Like in ProcessingSigViewer, settings() is called first, before the window is run. When it is run, setup() is called, which creates the text field. draw() is called directly after setup(), and is executed in a loop. exit() is called whenever the pop-up is closed. input() is called whenever the text in the textfield in the pop-up gets altered.

MCV Classes

ProcessingSigViewer_Model

setBufferClientClock() is called first, and initializes an instance of BufferClientClock.
setIP() and makeBuffer() is called when an ip-address has been typed in. setIP() is a simple setter for the variable that stores the IP-address. makeBuffer() initializes an instances of Buffer and Header, and adjusts the size of the array that stores the data to the corresponding amount of channels.
updateData() is called when an instance of GraphWindow is initialized or when drawGraphs() is called from GraphWindow. It updates the data and processes it according to the current viewtype.

ProcessingSigViewer_Controller

ControlEvent() is a big function that handles the behaviour for every possible event by calling the appropriate functions in the model or view. If the event is one for changing a filter, cutoff or viewtype, the appropriate variables in the model are changed so the preprocessing is done using that filter, cutoff or viewtype. If the event is the popup giving an IP address, the Buffer is initialized in the model, and the GraphWindow is initialized in the view.

ProcessingSigViewer_View

First makeSetup() is called from setup() in ProcessingSigViewer. This functions calls addOptionsButtons() and addTabs(), which add the buttons, text fields and tabs to switch between different view types.

drawView() is continuously called from draw() in ProcessingSigViewer. It calls drawBoxes() and writeText(), and drawGraphs() from a GraphWindow instance if it has been initialized. The initialization of the GraphWindow is done through a call from the controller, from an event in the IP-address textfield.

Graph Classes

GraphWindow

GraphWindow handles the behaviour of the window containing the graphs and stores information common to all graphs. For instance, it knows what the viewType of the model is, and what the ranges of the axes of graphs are, things graphs themselves do not know and are told by the GraphWindow. Upon initialization, GraphWindow gets the data from the model and calls generateGraphs(), which creates the graphs. Note that the initialization of GraphWindow requires that the buffer in the model has been initialized. GenerateGraph also calls positionGraphs(), which adjusts the size and coordinates of the graphs to fit on the screen. drawGraphs() gets called from drawView in the view, and draws/updates each graph. If the graphs are of HzGraph, drawLegend() is called to draw a legend next to the graph.

Graph

An abstract class that defines the behaviour common to all graphs. drawGraphs(), from GraphWindow, calls updateGraph, updateYRange and drawGraph every time it is called. UpdateGraph and updateYRange are setters for the variables representing data and the y-range of the graph respectively. drawGraph is an abstract method that is defined differently for each child class of Graph - each graph knows how to draw itself. This class only defines behaviour that is common to all types of graphs.

LineGraph

An abstract class for defining behaviour for drawing graphs that use a line to represent data points. For LineGraphs, drawGraph draws a line between each datapoint.

TimeGraph

Used for drawing a LineGraph in the Time domain. It is virtually identical to FrequencyGraph, except that in the TimeGraph the range of the x axis is hard coded as being between -4 and 0 seconds.

FrequencyGraph

Used for drawing a LineGraph in the Frequency domain. It is virtually identical to TimeGraph, except that in the FrequencyGraph the range dynamically depends on the frequency cutoffs.

HzGraph

This graph does not draw the axis labels as the other types of graphs do, and its drawGraph() function makes it draw itself as a coloured rectangle. It is not fully implemented yet, a more precise description of what it needs to be done to finish it can be found in the *Future additions* section.

Static Classes

Preprocessor

Handles the behaviour of preprocessing options and filters.

Procfft() gets called when updating a frequency graph and transforms the data into the frequency domain. procfftNoise() gets called when updating a HzGraph and returns the average amplitudes for given frequency pairs. ProcSpectralFilter gets called in other types of graphs and returns the data between two given frequencies. Center() and car() get called if updateData() is called from the model, and the corresponding filter is selected with the checkboxes on the left.

BasicStats

Handles basic mathematical operations that are useful anywhere in the program. It has functions for statistical operations for computing the mean, median and standard deviation, array functions for flattening, conversion of double to float arrays and computing the range of an array, functions that apply operations to each element of an array such as multiplication, computing the natural logarithm and ensuring values are below a maximum, and operations on numbers such as computing the nearest power of two and rounding a number.

Enum Classes

Filter

An enumeration of the different kinds of spatial filters and preprocessing options. This class will need to be updated as the adapt filters get implemented. It is used by the controller and the model to communicate and store which filters should be applied to the data.

ViewType

An enumeration of the different kinds of graph. Each element in the enum corresponds to its own tab. It is used by the model and GraphWindow to store what type of graphs should currently be drawn.

Java Files

BufferAdapter

Updates the data; newer version of SignalViewer.

connect() gets called from the constructor, which in turn gets called from the makeBuffer() function in the model, and forms a connection with the buffer.
update_data() gets called from updateData() in the model, and updates the data. The results of this function get adjusted to the current viewtype in the rest of the updateData() function. If it detects a reset, or the connection to the buffer fails, it returns null. If it gets too many data points to update, it jumps ahead.

BufferClient

A class that reads the data and passes it on to the model. connect() is a boolean that tries to connect to the buffer and returns whether it worked or not. If the connection is lost, reconnect() is called, which prints out a message to notify the user that the program is reconnecting and calls connect(). The model gets and sets the IP address from the BufferClient, and gets the buffer from the BufferClient. Importantly, the model tells this class to update the data every timestep.

BufferClientClock

An extension to BufferClient. It is used for ensuring the sample time is identical to the actual elapsed time according to the computer’s clock.

BufferEvent

A class for wrapping buffer events as WrappedObjects. The constructor saves any datapoint or data-array as a WrappedObject, together with another WrappedObject of a String, describing the type of the datapoint or array.

ClockSync

Helps handling the rate at which data is updated. Is used by BufferClientClock to synchronize with the clock on the computer.

DataDescription

Simple class for describing data blocks as used in GET_DAT and PUT_DAT requests

DataType

A class for defining data types and routines for conversion to Java objects. It has only one function, getObject(), which returns data of a certain type of a given ByteBuffer

Header

A class for wrapping a buffer header structure. Gives us information about the received data. The information stored in the buffer that is most relevant to us specifically is the labels of each channel and the sampling frequency of each channel.

SamplesEventCount

Simple class for describing data blocks as used in GET_DAT and PUT_DAT request. Very similar to DataDescription. This class is used by BufferAdapter to represent the current status of the data.

WrappedObject

A class for wrapping relevant Java objects in a way that is easily convertible to FieldTrip data types. BufferEvent uses this class to wrap buffer events.

Version differences between Android and Windows

Processing is a programming environment in which it is possible to create programs for different environments. However, not all libraries are accessible within every mode. Due to this, we had to create different versions for the different modes. Most of the code could be reused in the other version, so there are only some small differences.

IP Addresses

For Windows, as long as you are running the program at the same server as where you are running the buffer, you can use the localhost as IP-address: 127.0.0.1. The localhost is the same for every computer and thus could theoretically be hardcoded. However, on tablets or mobile phones, this is not the case. We created a textfield in which you have to fill in the ipv4 of your device (the BufferBCIApp displays the IP-address you should use). When ENTER is pressed, the sigViewer starts and the graphs are displayed.

Android Keyboard

Whereas you can use your keyboard for the Windows version, the Android version has no external keyboard. It will need a piece of software that makes a keyboard appear and disappear when clicked on the right place on the screen. Luckily, Processing has already some libraries for

this. We decided to use the ketai keyboard, which comes with other functions like *keyPressed()* and *mousePressed(). mousePressed()* activates the keyboard when clicked on the screen. When using the SignalViewer, the keyboard only appear when clicked on a textArea. *keyPressed()* knows on which key of the keyboard you clicked and is a kind of controlEvent. It states what happens when a key is pressed. To make this work, all the textFields are replaced with textAreas, and within the *draw()* function the text within these areas is set to *visible*.

Known Bugs

This section is intended as a guide for future developers, to list some of the most pressing known bugs of the program that need fixing. None of these bugs prevent effective use of the product, but do pose major or minor inconveniences to users or developers.

Data size needs to be power of 2

The fast fourier transform requires the size of the input data to be a power of 2 to work properly. This has been remedied by displaying only 4 seconds of data, but the program cannot handle displaying arbitrary amounts of data. The suggested fix of padding the data with zeros until its size was a power of 2 did not work.

Slow updating

The size of the graphs sometimes updates too slowly, leading to graphs that are zoomed in too far. This could probably be fixed by creating a replacement for the *outOfRange()* function in *GraphWindow*, that indicates whether too much or too little data outside the current range of the graphs and if that is the case, the y range for the graphs needs to be recalculated. A better way needs to be implemented for indicating whether the y range needs to be adapted. This is, however, only a minor inconvenience, since it can be resolved by switching between view types or selecting a preprocessing option while running the program.

Catching user input not handled

Currently, user input is not caught if it is wrong or nonsensical. This is a problem for using the IP address pop up and the text fields in the program, since entering wrong values causes the program to crash.

Older tablets break down

For unknown reasons, the program breaks down when used on older tablets. This might be due to the program simply being too heavy for those tablets to handle, but this has not been extensively tested due to older tablets to test on were unavailable to us at the time.

Android keyboard back press does not register properly

Back presses do not register properly on android when typing in text fields. The back press has to be used multiple times to delete a single character.

Future additions

This section describes functionality that is present in the MATLAB signal viewer, but not in this version of the Processing signal viewer. The section is intended as a guide and to-do list for any future developers of this project.

ViewTypes

50Hz

Already implemented for the most part. The view is already given and the class is there it only needs to be connected to the real data. The following functionalities still need to be implemented:

- A function to give the average power over the signal; one value per channel
- A way to change the colour depending on the “badness” of the data, based on the average decibel.

NoiseFrac

- A parent class of *HzGraph* that is a child of *Graph* called *ColourGraph* needs to be implemented.
- A child of *ColourGraph* called *NoiseGraph* needs to be implemented.
- A *ViewType* Enum for NoiseFrac needs to be created.
- Functions that compute the fraction of the signal that should be recognised as noise need to be implemented in *Preprocessor*.

Spectrogram

- A child class of Graph called *SpectGraph* needs to be implemented.
- A *ViewType* Enum for Spectrogram needs to be created.
- An appropriate library for displaying spectrograms needs to be selected and applied. Such libraries are available for Processing, but only for sound signals as far as we know.
- Functions that transform the data into an appropriate format that can be used as input for the spectrogram need to be implemented in *Preprocessor*, presumably functions that transform the data into sound.

Power

- A parent class of *HzGraph* that is a child of *Graph* called *ColourGraph* needs to be implemented.
- A child of *ColourGraph* called *PowerGraph* needs to be implemented.
- A *ViewType* Enum for Power needs to be created.
- Functions that compute the power of the frequency in the past range, averaged over all frequencies need to be implemented in *Preprocessor*.

Offset

- A parent class of *HzGraph* that is a child of *Graph* called *ColourGraph* needs to be implemented.
- A child of *ColourGraph* called *OffsetGraph* needs to be implemented.
- A *ViewType* Enum for Offset needs to be created.
- Functions to compute the average deviation from the global average need to be implemented in *Preprocessor*.

Preprocessing & Filters

Each of the following filters needs to be implemented as a function in the *Preprocessor* class, and those functions need to be called when the appropriate button is pressed using the *ProcessingSigViewer_Controller*.

Spatial Filters

- Slap

Adapt filter

- Whiten
- RM Artch
- Rm EMG

Pre-processing

- Detrending

Reading out Cap Files

- A class or function needs to be created that transforms the data from the cap files into screen coordinates.

- The current *positionGraphs()* function in the *GraphWindow* class needs to be deleted and replaced with a *positionGraphs()* function that takes into account these screen coordinates.