PuPl Manual, v1.1.0

Isaac Kinley

January 2, 2022

Contents

1 About PuPl

PuPl (**Pu**pillometry **P**ipe**l**iner) is an Octave-compatible library of Matlab functions for processing pupillometry data and a user interface that lets you run these functions without having to write any code.

1.1 Installation

PuPl's source code is hosted at `github.com/kinleyid/pupl`. You can download a compressed version of the repository and unzip it somewhere on your own computer (ideally, somewhere that's easy to navigate to from Matlab). Because the source code is all just text files, no further installation is required.

1.2 Workflow

Figure 1 illustrates the intended workflow. Every eye tracker is different, so you will need to figure out a processing pipeline that works well with your data. The way to do this is to first load data from a single participant and work out a pipeline iteratively, by processing the data, evaluating the readouts and visualizing how the data is being altered, and redoing the processing with different parameters.

Processing the data broadly involves processing the continuous data (i.e., the pupil size and gaze data streams, as covered in section 4), processing the event data (covered in section 6), and defining epochs and grouping them into sets (covered in section 7). After each of these steps, you should visualize the data (as covered in section 3) to make sure nothing unexpected is happening. If something unexpected does happen, PuPl has a handy "undo" option.

Once you've processed one recording the way you want to do all of them, you can export the command history as a script (as covered in section 9.2). This is a standard Matlab file that contains all the commands the user interface was running to process your data. Once you import the entire dataset, you can run this script to process all of it at once.

It is always a good idea to check the readouts and visualize the entire dataset to make sure the processing pipeline is working as intended for all recordings. Sometimes there will be a recording where the eye tracker was not well-calibrated or the participant was moving around or blinking rapidly. In these situations, you

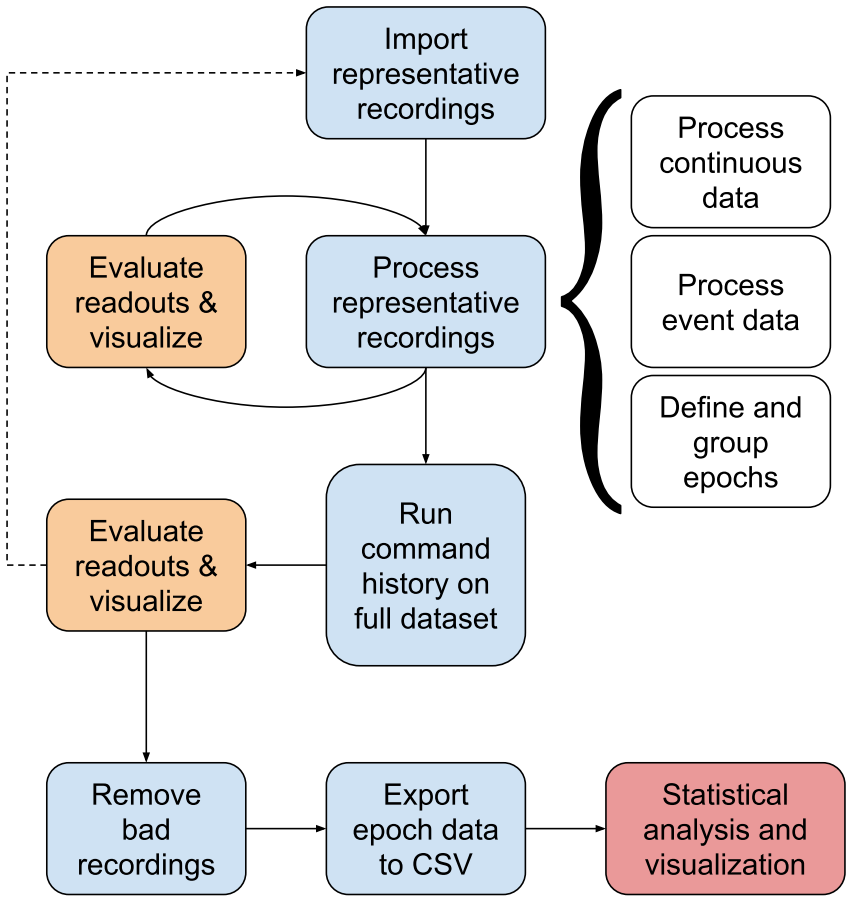

Figure 1: PuPl's intended workflow.

can either rework the processing pipeline to account for these recordings or you can simply exclude them from any further analysis.

Once you have processed the entire dataset, you can export the epoch data to a CSV table (as covered in section 8). By design, PuPl does not contain built-in statistical analysis tools (the better such tools became, the more closely they would approximate already-existing statistical software, so why not just stick to this pre-existing software to begin with?). Instead, the export tools are very flexible and should enable any type of analysis you want to do in your preferred statistical software. I have also included example analyses in R (ANOVA, linear mixed effects models, etc.; see appendix A)

2 Getting started

2.1 Initializing

Change Matlab's working directory to PuPl's home folder (which should contain `pupl.m`, `LICENSE.md`, etc.). You can do this either using Matlab's "Current Folder" panel or from the Command Window using the `cd` function. Run `pupl init` in the Command Window to add the source code to Matlab's path, check the web for a new version, initialize the global data variable (by default named `eye_data`), initialize the user interface, and load whichever add-ons are in the `add-ons/` folder. Any of these last 4 steps can be skipped by adding `noWeb`, `noGlobals`, `noUI`, and/or `noAddOns`, respectively, as command-line arguments to `pupl init`.

2.2 Configuring PuPl

Most users can probably safely skip this step. If you want to change any of the global options (e.g. how many timeline steps to keep track of for undo/redo operations, what the global data variable should be called, what extension PuPl should use to save data in its native format, whether to use single or double precision), these are set early in the file `pupl_init.m`.

2.3 Getting the example dataset

At `github.com/kinleyid/PuPl-worked-example`, you can find a worked example of how to use PuPl to reproduce published results from an experiment on sustained attention [15].

2.4 Importing raw data

Under `File > Import`, you will see options for importing raw data. If the format you use is not yet supported, please send an email to kinleyid@mcmaster.ca and I can look into adding support. Note: if you import raw data from a BIDS-formatted folder [3], event logs will not automatically also be imported. See 2.6 to find out how to import event logs from a BIDS-formatted folder. If you want a relatively quick way to organize your data according to the BIDS specification, I have another project that might help: `pypi.org/project/dastr/`.

2.4.1 A note on importing EyeLink EDF files

PuPl provides 2 methods of importing EyeLink's binary EDF files. The first (labeled the "Edf2Mat method") is based on a tool called Edf2Mat, the output of which is converted by the eyelink-import add-on to PuPl's expected format. To use this functionality, install Edf2Mat from `github.com/uzh/edf-converter` and add the main folder (containing @Edf2Mat) to the path. Note that Edf2Mat does not work in Octave.

The second method (labeled the "base method" on PuPl's user interface) attempts to read EDF files using plain Matlab code. This has been tested on all the EDF files I could find on the Open Science Framework (osf.io), but there is no guarantee that it will work for every EDF file out there (and in fact, it doesn't work on the EDF files here: osf.io/wrzjq). When it doesn't work, this will probably manifest as PuPl thinking the file is missing lots of data or contains no data at all.

If neither the "mex method" nor the "base method" works for your EDF file, there are 2 options:

1. Run SR Research's EDF2ASC tool on your files. This converts EDF files to human-readable text (.asc) files, which PuPl can then import.

2. Contact me (kinleyid@mcmaster.ca) and I can try to change PuPl to be able to read your EDF file.

2.5 Inspecting and manipulating data

Loaded data will appear on the user interface. Hovering your cursor over the name of recording will cause a quick summary to appear (sample rate, number of events, etc.). To inspect a recording in more detail, double click it in the workspace or run `openvar eye_data`. This will open Matlab's built-in tool for viewing and editing variables. See 3 to find out how to visualize your data.

Unselected recordings are ignored by the functions run by the user interface. To delete recordings from Matlab's workspace, go to `Edit > Remove recordings`. Data is also accessible in the Command Window through the global variable `eye_data`. See 9.1 for details.

2.6 Importing event logs

After you have loaded some data, you will be able to attach event logs to it. Go to `Trials > Event logs > [Import/Import from BIDS]` to see your options for importing raw event logs. Note that this step only establishes a correspondence between recordings and event logs—to actually load events from the event logs, go to `Trials > Event logs > Synchronize with eye data`. See 2.7 for details.

PuPl can load event logs saved as .mat files as long as these contain only one variable which is a Matlab struct array with field names "name" (the name of an event) and "time" (the time, in seconds, at which the event occurred). Any other field names will be treated as event variables.

2.7 Loading events from imported event logs

There are in general two situations when loading events from event logs. In the simpler and more common situation, the same computer records both eye tracker data and event log events, meaning all timestamps are recorded according the same system clock. If this is your situation, see 2.7.1. Alternatively, it could be that different computers are recording the eye tracker data and event log events, and that sync triggers are sent to both computers at the same time. If this is your situation, see 2.7.2.

2.7.1 By timestamps

Go to `Trials > Event logs > Synchronize with eye data > By timestamps` to select the events from the event logs you wish to load/attach to the eye data for further analysis.

2.7.2 By shared events

The different clocks are used to measure timestamps in the event log and the eye data may drift and/or be offset relative to one another. To solve this problem, PuPl uses linear regression to find a mapping between the two sets of sync markers. To do this, go to `Trials > Event logs > Synchronize with eye data > By shared events`. You will be able to specify which events are sync markers. After this, you will be able to select the events from the event logs that should be copied to the eye data and specify whether the pre-existing events in the eye data should be deleted.

3 Visualizing data

3.1 Plotting continuous

It is a good idea to visualize your data throughout processing to see how it is being altered. Under `Plot > Plot continuous` there are tools for plotting continuous datastreams.

3.2 Plotting gaze coordinates

To see how gaze positions are distributed for each recording, go to `Plot > Gaze scatterplot`.

3.3 Plotting pupil size

To see how pupil size measurements are distributed for each recording, go to `Plot > Pupil diameter histogram`

3.4 Visualizing pupil foreshortening error

The pupil foreshortens as it turns away from the eye tracker, being measured as smaller than it really is [4]. E.g. if the eye tracker is placed below a computer screen, the pupil will tend to be measured as smaller when looking at the top of the screen and either side. This will appear as a (roughly) linear trend in a plot of pupil size vs gaze y coordinate and a (roughly) quadratic trend in a plot of pupil size vs gaze x coordinate. Under `Plot > Pupil foreshortening error`, there are tools to examine whether there is a systematic relationship between gaze location and measured pupil size. The tools explained in 4.11 can correct for such a relationship.

3.4.1 Plotting pupil size as a function of single gaze dimension

To plot pupil size as a function of either gaze x or gaze y coordinate, go to `Plot > Pupil foreshortening error > Pupil size vs gaze [x/y]`.

3.4.2 Plotting pupil foreshortening error surface

To visualize pupil size across the whole gaze field, go to `Plot > Pupil foreshortening error > Error surface`. This divides the gaze field into a grid and computes the mean pupil size for points falling within a square centered on each node.

3.5 Plotting epochs

Once epochs have been defined (see 7), they can be plotted using `Plot > Plot epochs`.

3.6 Plotting epoch sets

Once epoch sets have been defined (see 7.5), they can be plotted using `Plot > Plot epoch sets`.

3.6.1 Line plots with error bars

Under `Plot > Plot epoch sets > Line plot with error bars`, you will see options for generating these types of plots. If you select more than one recording at a time (e.g. if you plot the data for one experimental condition), you will be able to weight the individual epochs either equally/by trial (each individual epoch has an equal influence over the appearance of the final plot) or by participant (each participant's epochs are averaged together before being averaged for the final plot, i.e., a participant with 50 valid epochs would have the same influence over the appearance of the final plot as a participant with 10 valid epochs.

3.6.2 Data matrix images

This type of plot is conceptually the same as EEGLAB's erpimage (see sccn.ucsd.edu/wiki/Chapter_08:_Plotting_ERP_images). Essentially, it's a grid/raster image where each row is a different epoch, each column is a different timepoint relative to the timelocking event, and the color of each grid cell/pixel indicates to the pupil size at that epoch and timepoint. To change the colours used, see www.mathworks.com/help/matlab/ref/colormap.html.

4 Processing continuous data

All functions for preprocessing raw data can be found under `Preprocess`.

4.1 Monocularizing recordings

If you have a lot of data from a high-quality eye tracker, you may not need to include both eyes in your analysis. To remove the pupil size data from one or the other, go to `Edit > Make recordings monocular`. If you are concerned about memory usage, you may also want to use low numerical precision and a limited undo/redo timeline—see 2.2 for more info.

4.2 Cropping recordings

There may be sections of data before, after, or between trials that are not of interest. To reduce memory usage, you can crop these sections using the tool under `Edit > Crop recordings`. If you are concerned about memory usage, you may also want to use low numerical precision and a limited undo/redo timeline—see 2.2 for more info.

4.3 Concatenating recordings

It is not unusual to record data from the same participant in several blocks, resulting in 2 or more separate data files for the same participant. To concatenate recordings together, go to `Edit > Concatenate recordings`. If the block number is of interest, you can add a suffix to every event from a given block (e.g. `_block1`, such that an event recorded as `trial1` would be renamed to `trial1_block1`).

4.4 Excluding recordings

If a participant moved around a lot during recording or if the eye tracker did not calibrate properly, the entire recording may be unusable. To exclude such recordings, go to `Preprocess > Exclude recordings`. Recordings can be excluded manually or on the following bases:

- High percent missing data
- High standard deviation of pupil size
- Low correlation between left and right pupil size measurements
- High number of rejected epochs (see 7)

4.5 Converting between pupil area and diameter measurements

Some eye trackers record pupil area, while others record pupil diameter. To convert between these, go to `Process > Convert pupil size`. To see which size measurement your tracker uses, read the y-axis on the plot of the continuous data (3.1)

4.6 Z-scoring data

To z-score the pupil size data, go to `Preprocess > Z-score pupil size data`.

4.7 Removing bad data samples

4.7.1 Trimming extreme pupil size measurements

Go to `Process > Trim data > Trim extreme dilation values` to remove pupil size measurements that are too high or low. In the window that opens, you can specify cutoff points as explained in B.2. Note: the gaze measurements corresponding to rejected pupil size measurements will also be removed. If you plan to run the same pipeline on multiple participants, it is best to specify relative cutoffs since different participants will likely have different average pupil size measurements (unless, of course, you z-score the data as in 4.6).

4.7.2 Trimming extreme gaze measurements

Go to `Process > Trim data > Trim extreme gaze values` to remove gaze measurements that are too extreme (e.g. off-screen or too far from a fixation cross). You can specify cutoff points as explained in B.2. Note: the pupil size measurements corresponding to rejected gaze measurements will also be removed. If you plan to run the same pipeline on multiple participants, it is best to create absolute gaze cutoffs since screen x and y values are likely to have the same meaning across participants.

4.7.3 Trimming extreme dilation speeds

Go to `Process > Trim data > Trim by extreme dilation speed` to trim datapoints that represent an abrupt change in pupil size [7].

4.7.4 Trimming isolated samples

Go to `Process > Trim data > Trim isolated samples` to remove temporally isolated islands of data that are unlikely to be meaningful or accurate measurements [7] [8]. See B.1 to understand how to specify the max length of these islands of isolated data and the maximum allowable distance from the nearest other datapoint.

4.8 Blinks

Blinks are problematic for 2 reasons: first, obviously, they represent lost data. Second, the pupil is partially obstructed shortly before and after the eye fully closes during blinks, making samples recorded near blinks unreliable. To deal with these issues, PuPl first identifies which data samples were recorded during blinks and then removes both these samples and, optionally, adjacent ones.

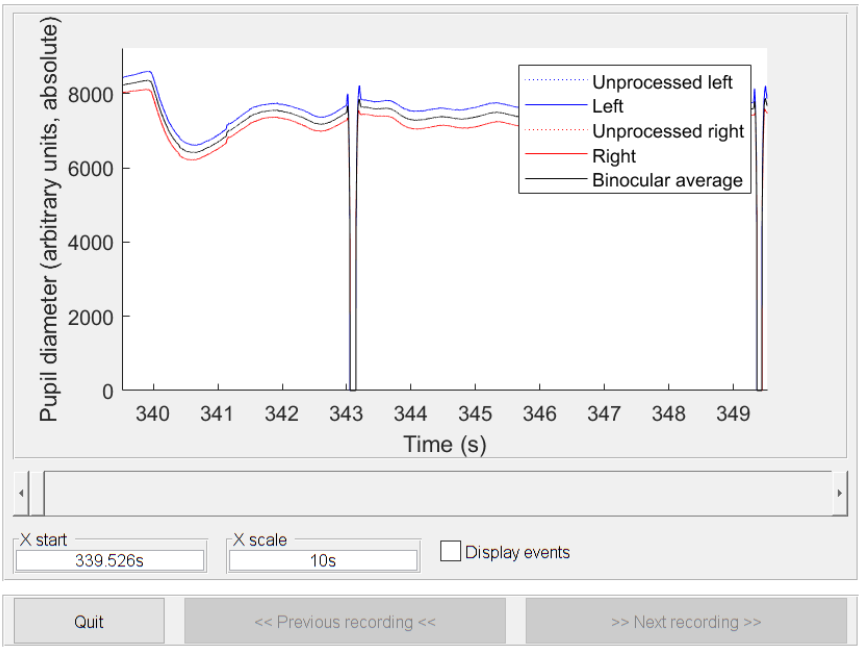

Figure 2: Data from a higher-end eye tracker

4.8.1 Identifying blinks

There are several methods for identifying blink samples, including the pupillometry noise [5] and velocity profile [9] methods. These are excellent methods if you're using a high-end eye tracker that records a relatively smooth drop-off and then increase in measured pupil size around blinks (see 2). However, lower-end eye trackers may not be able to do this (see 3). You should inspect your data (3.1) to see which case best matches yours. If your data does not contain the characteristic slopes used by these sophisticated algorithms, you may be best off using the simpler method of identifying blinks as sections of consecutive missing data.

Note that eye trackers are inconsistent in the way that they treat blink samples. Some set the pupil size to 0 during blinks (as in figure 2) while others treat these samples as missing (as in figure 3). The blink identification algorithms all require blink samples to be missing, so it is necessary to remove pupil size measurements below some threshold (a plausible value for the dataset in figure 2 might be 100; see 4.7.1).

4.8.2 Removing blink and possibly blink-adjacent samples

To remove blink samples and possibly blink-adjacent samples, go to `Blinks > Remove blink samples`. It is best to start by removing only the blink samples themselves (i.e. providing `0s` as input to the dialog box that asks how much data to remove immediately before and after blinks) and then inspecting the continuous data to decide whether blink-adjacent data also needs to be removed (see 3.1). If it does, click the same menu and provide a non-zero input to the dialog box this time (see B.1 for an explanation of valid inputs).

4.9 Filtering data

To apply a moving mean or median filter to pupil size or gaze data, go to `Process > Moving average filter`. If your data contains large jumps (as in figure 3), you may be best off using a moving median filter.

4.10 Downsampling

It is not uncommon to downsample data to save memory and accelerate subsequent processing. To do this, go to `Preprocess > Downsample`. If your data contains high-frequency artifacts (as in figure 3), make sure

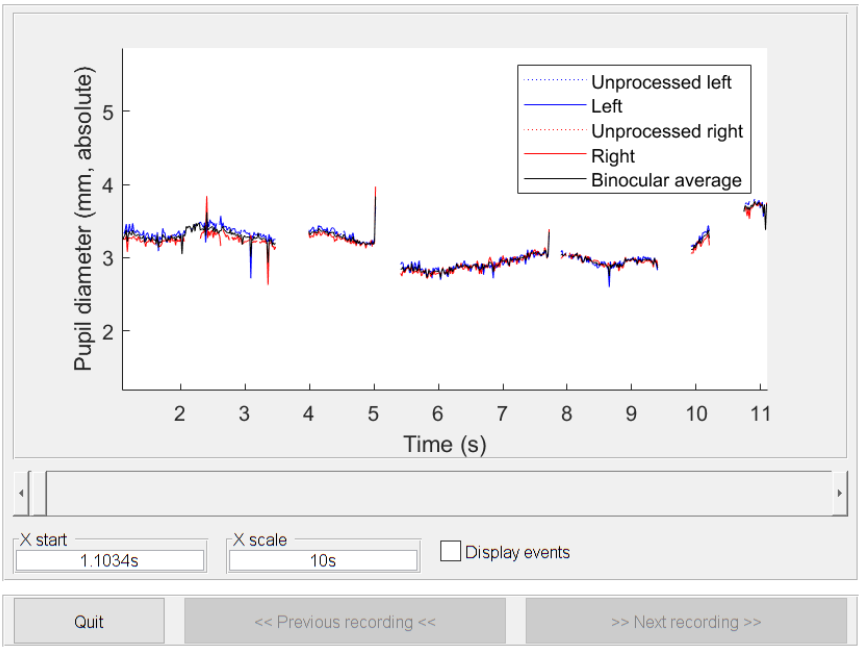

Figure 3: Data from a lower-end eye tracker

to first filter it (see 4.9) to avoid aliasing. If you are concerned about memory usage, you may also want to use low numerical precision and a limited undo/redo timeline—see 2.2 for more info.

4.11 Pupil foreshortening error correction

For an explanation of pupil foreshortening error (PFE), see 3.4 [4]. The PFE can be corrected using the options under `Process > Pupil foreshortening error correction`. However, since PFE correction makes use of gaze measurements, it is important to first ensure that these are accurate.

4.11.1 Identifying saccades and fixations

Algorithms available for identifying saccades and fixations can be found under `Process > Identify saccades and fixations` [11]. Note: data points themselves are not labelled as saccades or fixations, only the periods of time between them. The justification for this is as follows: if gaze samples 1, 2, and 3 are all at the same coordinates and gaze samples 4, 5, and 6 are at a far-away coordinate, it doesn't make sense to label any of those points themselves as a saccade. Clearly the participant was fixating at points 1, 2, and 3 and then again at points 4, 5, and 6, and the saccade took place between points 3 and 4.

4.11.2 Mapping to fixations

After identifying fixation periods, the data during these periods can be reassigned to their centroids (spatial means; [11]) using `Process > Map gaze data to fixation centroids`.

4.11.3 Geometric PFE correction

The pupil foreshortening error can be corrected using straightforward geometry if the coordinates of the experimental setup are provided [4]. First, gaze coordinates need to be converted from units of pixels to units of millimeters; go to `Process > Pupil foreshortening error correction > Convert gaze units from pixels to millimeters` to do this. Then, you can add the coordinates of the experimental setup under `Process > Pupil foreshortening error correction > Add coordinates of experimental setup`. A dialog box will appear explaining the coordinate system to use when providing these coordinates. Once you

have done so, go to `Process > Pupil foreshortening error correction > Geometric PFE correction` to apply the geometric correction.

4.11.4 Regression-based PFE correction

If the physical dimensions of the experimental setup, you may want to do multivariate linear regression-based PFE correction [1]. This method models pupil size P as

$$P = \beta_0 + \beta_1 G_x + \beta_2 G_y \tag{1}$$

where G_x is the gaze x position and G_y is the gaze y position. The regression-based estimate is subtracted from the measured pupil size and then the mean measured pupil size is added to the result (so that the regression-corrected pupil size is on the same scale as the original). To use this method, go to `Preprocess > Pupil foreshortening error correction > Multivariate linear detrend`.

4.11.5 Piecewise quartic PFE correction

Pupil size can be corrected on the basis of horizontal gaze position (ignoring vertical gaze position) using piecewise quartic detrending [2]. This method models pupil size P as

$$P = \beta_0 + \beta_1 G_x + \beta_2 G_x^2 + \beta_3 G_x^3 + \beta_4 G_x^4 \tag{2}$$

where G_x is the gaze x position. In the reference paper this method comes from, the authors fit two such quartic models, one on either side of a clear peak in measured pupil size as a function of gaze x position. To find this peak, PuPl divides the horizontal gaze axis into a one-dimensional grid with 100 cells and computes the mean pupil size at each of these. Two quartic models are created, one on either side of the middle of the cell where the mean pupil size is greatest. The quartic model estimates are subtracted from the measured pupil size and then the mean pupil size is added to the result (so that the corrected pupil size is on the same scale as the original). To use this method, go to `Preprocess > Pupil foreshortening error correction > Piecewise quartic detrend in gaze x`.

4.12 Interpolating missing data

To interpolate missing pupil size and gaze data using either linear interpolation or cubic spline interpolation, go to `Process > Interpolate missing data`. Dialog boxes will open asking for the maximum length of time and the maximum distance to interpolate over (see B.1 and B.2 for explanations of how to provide input for these, respectively). Note: it is a good idea to be conservative when specifying the maximum gaze distance to interpolate over since, if saccades take place during long periods of missing data, linear interpolation will make these look like periods of smooth pursuit.

5 Saving data

To save data, go to `File > Save as`. PuPl can save data either to a binary format or a text format. A separate file save dialog will open for each active recording.

5.1 Saving to binary

PuPl can save data to a binary format with extension `.pupl`. This format is space-efficient and can be loaded very quickly. The `.pupl` is merely an alias for version 6 `.mat` files, which are readable using Octave.

5.2 Saving to text

PuPl can also save data to a text format. This creates two files for each recording: a `.txt` tab-separated table for the continuous data and a `.csv` file for the event data. Note that no information about epochs is saved in the text format. The text format is intended as a standardized human-readable format for raw eye tracker data.

5.3 Batch saving data

To save all active data to the same folder using filenames corresponding to recording names, go to `File > Batch save`. This will open a single folder selection dialog.

5.4 Saving data to BIDS format

If you loaded your data from a BIDS-formatted folder, you can likewise save it to one under `File > BIDS > Save` (e.g. you may want to save your formatted data to a `sourcedata/` folder for quick loading later using `File > BIDS > Load`).

6 Processing event data

6.1 Event variables

Event variables refer to variables such as numbers or strings that are associated with events. By default, events only have two attributes of interest: their name and their time of occurrence. However, you may want to add other attributes, such as whether an event was the onset of a congruent or incongruent Stroop trial. The following sections explain how to do this.

6.1.1 Reading event variables from event names

Often, event variables are encoded in event names. E.g. the onset of a congruent trial Stroop trial may be called `trial_cong1` and the onset of an incongruent trial may be called `trial_cong0`. To read these variables, we use regular expression captures, which search for for a given pattern in a piece of text and return anything that matches.

In the example dataset (see 2.3), events have names like `Response:2 Hit=2/3,FA=1`. The numbers after `Hit=` indicate the number of detected gaps and the number of gaps total and the number after `FA=` indicates the number of false alarms, all of which are of interest.

Regular expressions capture information using round brackets. For example, the regular expression `Hit=(.)/(.)` would look for the string `Hit=`, then get the character immediately following that, then skip over the following backslash and get the following character. I.e. running that regular expression on `Hit=1/2` would return `1` and `2`.

You can read event variables from event names in this way using the tool under `Trials > Event variables > Read event variables from event names`. You will be able to:

1. select a subset of events to read event variables from (e.g., all those containing the string `Hit=`; see B.4)
2. provide a regular expression that will capture your variables of interest (e.g., `Hit=(.)/(.)`)
3. name the resulting event variable(s) (e.g., `#nhits; #maxhits`; note that event variable names are stored as fields in a Matlab structure and so must be valid structure field names, i.e., they must start with a letter and contain only letters, numbers, and underscores)
4. specify whether these should be converted to numbers or left as strings (we would want to convert `#nhits` and `#maxhits` to numbers)

6.1.2 Manually entering event variables

It is possible (though not efficient) to manually enter event variables using Matlab's variable viewer. This can be done by running `openvar eye_data`and then clicking on the structures under `event`. By populating a new column in the table that appears, you will be adding a new event variable.

6.1.3 Computing event variables

If you want to compute higher-order event variables (e.g., hit rate = n. hits / max. possible hits), go to `Trials > Event variables > Compute event variables`. You will be able to:

1. select a subset of events on which to compute your higher-level event (e.g., all those where `#nhits` is not empty; see B.4)
2. specify an expression that will return a new event variable (e.g., `#nhits / #maxhits`)
3. specify the name of the resulting event variable (e.g. `#HR`)
4. specify whether the resulting variable should be converted to a string or converted to a number (we would select "Numeric" for `#HR`

6.1.4 Homogenizing event variables within trials

Usually, there will be information about a participant's response encoded in the name of the event that marks that response (e.g. key pressed, hit or miss). However, often we want to use trial onsets, not responses, as timelocking events for event-related pupillometry (see 7). To deal with this scenario, it makes sense to read the response-related event variables from the names of the events that mark responses and then copy them to all the other events occurring within their trials (including the events that mark a trial's onset).

In the example dataset, we want `#HR` to be an attribute not only of the events that summarize the preceding trials, but also of the events that indicate the onset of a trial (which we will later use as timelocking events; see 7).

To accomplish this, go to `Trials > Event variables > Homogenize event variables within trials`. You only need to specify which events mark the onset of a trial and which events mark the end of a trial (in the example dataset, these are events whose names contain the string `Scene` and events whose names contain the string `Response`, respectively).

6.1.5 Mapping string event variables to numeric event variables

For a long time, Matlab did not treat strings as atomic/scalar data types, meaning that, for example, the expression `'wxyz' == 'abc'` does not just return `false` but an error. Matlab introduced string scalars in r2016b (meaning that the expression `"hello" == "hello"`, note the double quotes, returns `true`) and PuPl tries to convert string event variables to this data type.

However, if you are using Octave or an older version of Matlab, then to compare strings, you have to use the `strcmp` function. If you want to select events on the basis of a event variable which has been stored as a string (see B.4), it can be cumbersome to repeatedly type `strcmp`.

One solution to this is to map each possible value of a string event variable to a different number. For example, you might have a event variable called `#congruency` that takes on the values `'con'` or `'inc'` to specify whether the corresponding Stroop trial is congruent or incongruent. You could map these to the values 1 and 0 and name the resulting variable `#iscongruent`.

To do this, go to `Trials > Event variables > Make string event variables numeric`.

6.1.6 Computing reaction times

To compute reaction times, go to `Trials > Event variables > Compute reaction times` to specify which events mark trial onsets and which trials mark responses. Reaction times are stored in the event variable `#rt` and are associated with both the trial onset events and the response events.

6.2 Defining compound events

Often, we want to analyze responses to specific serial combinations of events (e.g. a trial of a certain type followed by a response of a certain type). To define these "compound events", go to `Trials > Find compound events`.

The steps involved in this are the following:

1. specify the name of the compound event you wish to define
2. specify the "primary" events (these are the events whose onset times will be used as the onset times of your compound event)
3. specify the "secondary events" that, if they occur in some serial position and/or time window relative to the time-locking events, will indicate the presence of your compound event
4. optionally, specify a time window centered on the primary events in which the secondary events must occur (e.g. Start: `-2s`, End: `0s`; see B.1 for an explanation of how to specify this window; if none is specified, it defaults to the entire duration of the recording)
5. optionally, specify the relative serial positions at one of which the secondary events must occur (these can be specified using Matlab's colon operator, e.g. `-3:-1`; if left empty, any serial position is allowed)
6. optionally, specify a criterion, based on event variables, that will determine whether a secondary event has been found. You can use event variables from both primary and secondary events to define this criterion. Event variables prefixed with `#1` (e.g. `#1rt`) will be read from the primary event and event variables prefixed with `#2` (e.g. `#2rt`) will be read from the candidate secondary event (e.g. `#2rt >= #1rt` sets the criterion that secondary events must have reaction times at least as long as their primary events; if left empty, no such event variable-based criterion will be used)
7. specify whether compound events should be marked based on the presence or the absence of the secondary events within the specified window and/or serial position list
8. specify what should be done when a compound event is found (options are to rename the primary event or add a event variable to the primary event)

If you choose to add a event variable to the primary event, the procedure for doing this is the same as in 6.1.3, with one difference: event variables prefixed with `#1` (e.g. `#1rt`) will be read from the primary event and event variables prefixed with `#2` will be read from the first found secondary event.

6.2.1 Example: computing reaction times the long way

To compute reaction times using compound events, you would:

1. set the primary events to be trial onsets
2. set the secondary events to be responses
3. leave the relative time window empty, since there is not reason not to compute reaction times that are greater than, e.g., 3 seconds
4. set the relative serial positions to `1` (assuming there are no other events that occur between trial onset and responses)
5. specify that compound events should be marked by the presence rather than the absence of the secondary events
6. specify that you want to add a event variable to the primary events
7. specify the expression to compute said event variable as `#2time - #1time` (the time of the response minus the time of the trial onset)
8. give the resulting event variable the name `#rt`
9. specify that `#rt` is a numeric variable

7 Epochs

"Epochs" are windows of data defined around events of interest, usually events signalling stimulus onset, that contain the pupil dilation response.

7.1 Defining epochs

Epochs can be defined using the tool under `Trials > Define epochs`. The "timelocking" events are the ones according to which epochs will be lined up when they are averaged together.

Epochs can be fixed- or variable-length. Fixed-length epochs are those that are defined around a single event of interest (for example, from 1 second prior to stimulus onset until 5 seconds after).

Variable-length epochs are those that are defined between 2 events (for example, from 1 second prior to stimulus onset to 1 second after a participant's response). For these epochs, there will be an event type that signals an epoch's onset and an event type that signals an epoch's end (either one of these could be the timelocking event). When variable-length epochs are being added together, shorter ones are padded with NaN ("non a number", i.e., missing) values to make them the same length as the longest epoch being averaged. If the timelocking event is the one that signals an epoch's onset, these NaNs will be added to the end. If the timelocking event is the one that signals an epoch's end, these NaNs will be added to the beginning.

You can specify the timelocking events (as explained in B.4) and the relative starts and ends of the time windows centred on the timelocking events (as explained in B.1).

In the example dataset, epochs should be defined from `0s` to `29.5s` relative to each event whose name contains the string `Scene`.

7.2 Baseline-correcting epochs

Due to variability in pre-trial pupil size measurements, you may wish to baseline correct your epochs (however, be aware that baseline pupil size can contain valuable information—for example, pre-decision pupil size could reflect bias toward exploration or exploitation [6]).

Pupil size can be baseline corrected by subtracting the baseline mean, computing the percent change from baseline mean, or z-scoring based or baseline statistics (i.e. subtracting baseline mean and normalizing by baseline standard deviation).

To do this, go to `Trials > Baseline correction and normalization`. The mapping from baseline periods to epochs can be one-to-one (e.g. baselines periods are the 200 ms before each event onset), one-to-all (e.g. one single baseline period occurs at the beginning of the experiment), or one-to-some (e.g. one baseline period occurs before a block of 5 epochs). If the one-to-some mapping is selected, epochs can be baseline-corrected using the most recent prior or the earliest subsequent baseline period. Like epochs, baseline periods can be of fixed or variable length.

7.3 Normalizing epochs

Different people have different ranges in pupil size, so it is sometimes valuable to normalize changes in pupil size from baseline according to some reference epoch (e.g. one in which the pupillary light reflex is evaluated [14]).

To do this, go to `Trials > Baseline correction and normalization` and specify which trial epochs are to be corrected on the basis of which reference epochs.

7.4 Rejecting epochs

Epochs can be marked for rejection using the tools under `Trials > Epoch rejection` on the following bases:

- proportion missing data
- presence of extreme pupil size measurements
- presence of blinks (see 4.8)
- presence of saccades (see 4.11.1)
- event variables (see B.4)

It is also possible to reject epochs on the basis of the amount of data missing within a specific window (e.g. the baseline period) using `Trials > Epoch rejection > Reject by proportion missing data within window`.

7.5 Defining epoch sets

It is useful to average together epochs of multiple types to get reliable estimates of pupillary response. E.g. you may want to examine responses to both a particular stimulus and a category of stimuli to which it belongs. To define sets of epochs for later analysis, go to `Trials > Define epoch sets`. Epochs can be selected as in B.4.

8 Exporting for statistical analysis

By design, PuPl does not contain its own built-in statistical analysis tools. Instead, it contains a flexible set of data export options that allow for subsequent analysis by dedicated statistical software such as the R programming language [10], which are far more comprehensive that PuPl could ever hope to be.

8.1 Exporting statistics

To export statistics (e.g. mean pupil diameter) as a spreadsheet, go to `Experiment > Write statistics to spreadsheet`. You have the option to compute statistics on individual epochs or on averages of epochs within epochs sets.

You can compute multiple statistics within different time windows. For example, you may want to compute the mean in a baseline period (e.g., from `-1s` to `0s` relative to timelocking events) and the max pupil size during the actual trial (e.g., from `0s` to `5s`).

Computing statistics on individual epochs enables the use of linear mixed effects models, which are powerful tools for detecting effects of interest while ignoring noise (e.g. between-subjects differences). These models enable you to, for example, measure effects on baseline pupil size while still being able to perform baseline correction (by simply adding baseline mean as a predictor term in a mixed effects model of pupil dilation response).

8.2 Exporting data

You can export epochs to a spreadsheet either as raw data or as downsampled/binned data by going to, respectively, `Experiment > Export undownsampled data` or `Experiment > Export downsampled/binned data`.

9 Using PuPl programmatically

9.1 Processing data in the command window

The central data structure used by PuPl is by default called `eye_data`. This is the variable that the user interface operates on. After initialization, this variable will appear in Matlab's global workspace. You can manipulate it the way you would any other variable using the Command Window.

If you are about to alter the data variable and want the option to return it to its present state, run `pupl cache`. This saves a copy of the data that can be reinstated using `Edit > Undo`.

If you process data in the Command Window and want to update the appearance of the user interface (e.g. if you have defined epochs in the Command Window and want the `Plot epochs` menu to be enabled), run `pupl redraw`.

9.2 Automating processing pipelines

When a processing function is run, it saves the equivalent command as a string to the `history` field of the data. All of these commands together constitute a processing pipeline. To export the processing history to a Matlab script, go to `Tools > Save processing history as script`. To view the processing history within the command window, run `pupl history`. To run a processing pipeline on the currently active data, go to `Tools > Run processing script`.

When you run processing functions using the user interface, you will see that the readouts specify which functions are being run. You can type `help <function name>` into the command window (e.g., `help pupl_blink_id`) to see what these functions take as input. You can also type `edit <function name>` to view the source code.

A Worked example

To demonstrate how PuPl works, I have prepared a worked example based on the data from [15]. I am extremely grateful to Drs Zhao and Chait for their help in understanding the dataset. The following explains how to reproduce the results of experiment in their paper.

A.1 Understanding the study

Participants in the sample dataset completed a sustained auditory attention task in which they heard one or more perceptually distinct tone sequences. They were asked to monitor one of these tone sequences for 25 seconds, listening for silent gaps. In the easy condition, there was only one such tone stream. In the medium difficulty condition, there was one target stream and one distractor stream. In the hard condition, there were two distractor streams and one target stream. Further details can be found in [15].

The expectation (borne out, as we'll see) was that participants would exhibit larger pupil sizes in the medium and hard conditions over the 25-second trials. The data for each participant was recorded in 6 blocks, with each block containing 12 trials for a total of 72 trials for each participant (24 at each difficulty level) and 198 individual recordings.

A.2 Getting the data

The data is stored at `osf.io/mdhgp/`. If you plan to analyze the entire dataset, you can download the `data-raw/` folder as a zip file. Alternatively, if you just want to follow along with the processing of the first participant's data, download the first 12 recordings (cS1_b1.edf, cS1_b2.edf, ... , cS1_b6.edf and cS2_b1.edf, cS2_b2.edf, ... , cS2_b6.edf), which are blocks 1–6 for participants 1 and 2. You can put them in the `data-raw/` folder in PuPl's `example/` directory.

If you want to convert the data to BIDS format, you can run the script example/code/toBIDS.py in PuPl's home folder, which relies on dastr, another project of mine (`pypi.org/project/dastr/`).

To import the data, go to `File > Import > From EyeLink EDF`. Even if you have downloaded the entire dataset, the best approach is to first load the first few recordings and use them to arrive at a general-purpose processing pipeline. Going through the iterative trial-and-error process of developing a pipeline takes too long with all the recordings loaded into Matlab's memory (then again, maybe you have more patience/a faster computer than me).

Once you've imported the data from raw, you may wish to save it as binary .pupl files, which can be loaded in very quickly. See 5 to learn how to do this.

A.3 Preparing the data

First, remove the data from the right eye (see 4.1). Then, concatenate the recordings together (see 4.3). Since we want to eventually use the processing history as a macro for processing the entire dataset (see 9.2), it's best to select the first block using the regular expression "b1", the second block using the regular expression "b2", the third using "b3", etc. (see B.3).

You're welcome to add suffixes to the new recordings or to the events in each block, but I did not in this example. The output in the Command Window should appear as follows:

```
Concatenating as follows:
    cS1_b1 = cS1_b1 + cS1_b2 + cS1_b3 + cS1_b4 + cS1_b5 + cS1_b6
    cS2_b1 = cS2_b1 + cS2_b2 + cS2_b3 + cS2_b4 + cS2_b5 + cS2_b6
```

A.4 Processing the continuous data

A.4.1 Trimming 0 pupil size samples

By inspecting the continuous data (see 3.1 and figure 2), you will see that blink samples are measured as 0 rather than missing. If you were going to use the "consecutive missing data" blink detection method, you would trim these samples (i.e. set them to missing/NaN; see 4.7.1). The upper pupil size cutoff can be set to `inf` to avoid cutting any data from the higher end of the pupil size distribution (see figure 4). However, in this example we will be using the pupillometry noise method [5], which does not require that blink samples are set to missing/NaN.

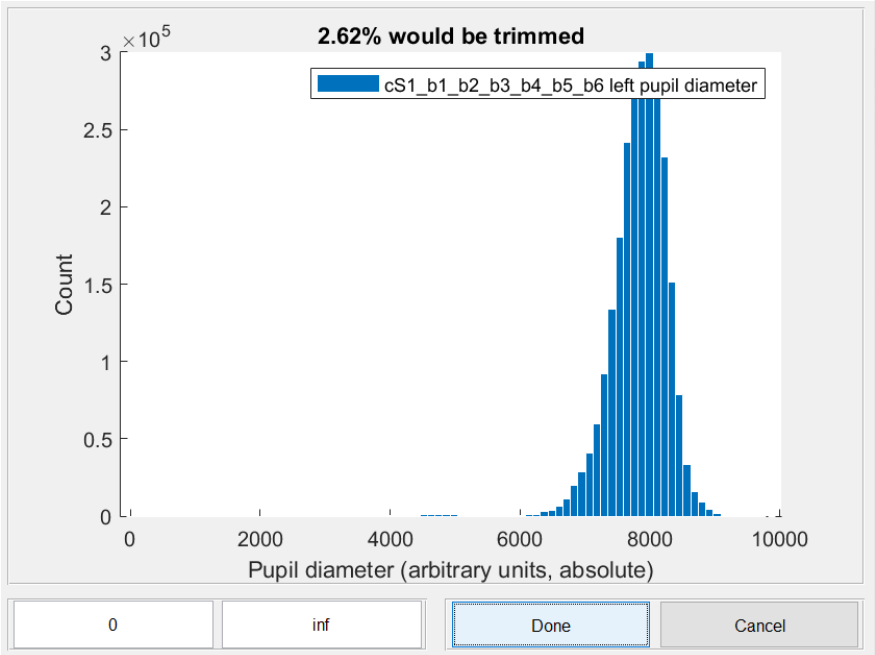

Figure 4: The user dialog for trimming extreme pupil size measurements

A.4.2 Correcting blinks

Use the pupillometry noise method [5] to identify blinks. You can see by inspecting the continuous data (see 3.1) that this does a good job of identifying blinks (Fig. 5).

The readout appears as follows:

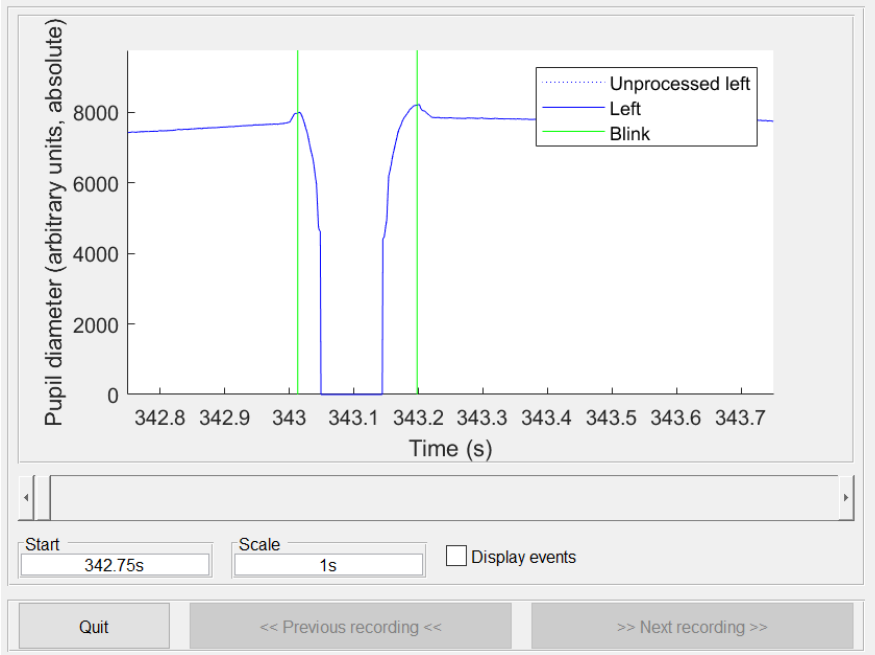

Figure 5: A blink has been identified.

```
Identifying blinks by pupillometry noise (method "noise")
        Running pupl_blink_id...
    cS1_b1...
According to method "noise":
    4.892967% of data marked as blinks using this method
    508 blinks in 43.27 minutes of recording (11.74 blinks/min)
In total:
    4.892967% of data marked as blinks
    508 blinks in 43.27 minutes of recording (11.74 blinks/min)
    done
    cS2_b1...
According to method "noise":
    14.164103% of data marked as blinks using this method
    1565 blinks in 43.38 minutes of recording (36.08 blinks/min)
In total:
    14.164103% of data marked as blinks
    1565 blinks in 43.38 minutes of recording (36.08 blinks/min)
    done
Done
```

As you can see in figure 5, there are sometimes small increases in measured pupil size around the usual blink-adjacent drops in measured pupil size. To deal with this, you can remove blink samples and trim 50ms of blink-adjacent data on either side (see 4.8.2). Note that, if this were an experiment examining phasic pupil dilation responses with shorter trials, you would be probably be best off not removing any blink-adjacent data, as this can impact your results quite significantly [5].

The readout for this appears as follows:

```
Removing from 50ms immediately before blinks to 50ms immediately after
     Running pupl_blink_rm...
    cS1_b1...
        4.892967% of data are blink samples
        1.956909% of data are blink-adjacent
    done
    cS2_b1...
        14.164103% of data are blink samples
```

```
        6.013298% of data are blink-adjacent
    done
Done
```

After this, linearly interpolate (see 4.12) over a max. of 400ms and across a maximum gap of 1 standard deviations (i.e. `1*sd`; see B.2). If you inspect the data (see 3.1), you will see that this does a good job of correcting for blinks (see figure 6).

The readout for this step appears as follows:

```
Interpolating max. 400ms of missing data using linear interpolation
        Running pupl_interp...
    cS1_b1...
interpolating max. 400 missing data points
        left (max jump 380.84): 6.11808% of data interpolated
        right (max jump 317.68): 6.11808% of data interpolated
    done
    cS2_b1...
interpolating max. 400 missing data points
        left (max jump 224.03): 16.30088% of data interpolated
        right (max jump 285.91): 16.30088% of data interpolated
    done
Done
```

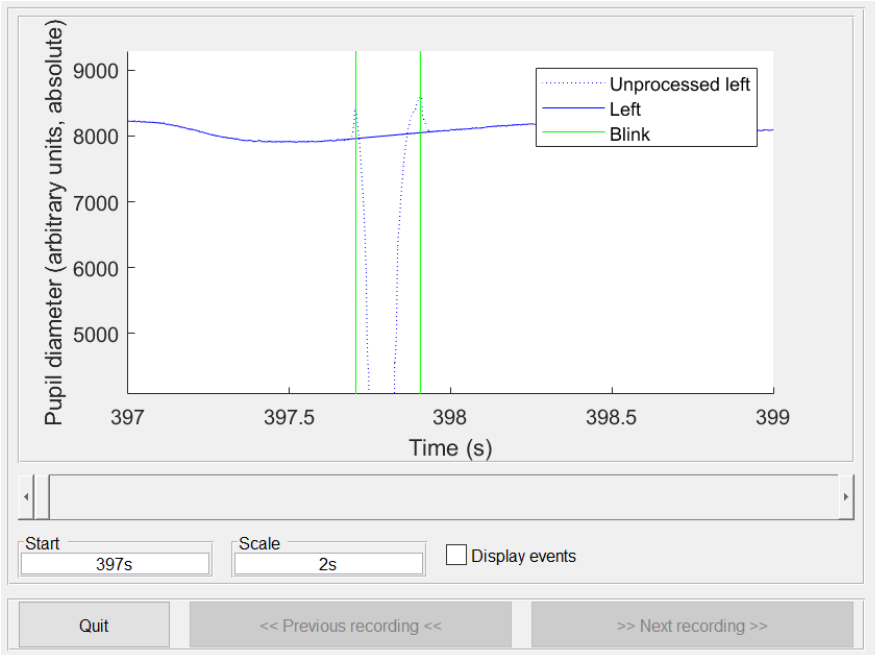

Figure 6: Bink-corected data

A.4.3 Filtering

To deal with any high-frequency artifacts in the data, use a Hann window moving average filter with a width of 150ms (see 4.9). This step smooths the data to reduce the risk of aliasing in subsequent downsampling steps (see figure 7).

The readout for this step appears as follows:

```
Applying hann-window moving mean filter of width 150ms
        Running pupl_filt...
    cS1_b1...
filter width is 150 data points
        Filtering left...[........]
        Filtering right...[........]
    done
    cS2_b1...
filter width is 150 data points
        Filtering left...[........]
        Filtering right...[........]
    done
Done
```

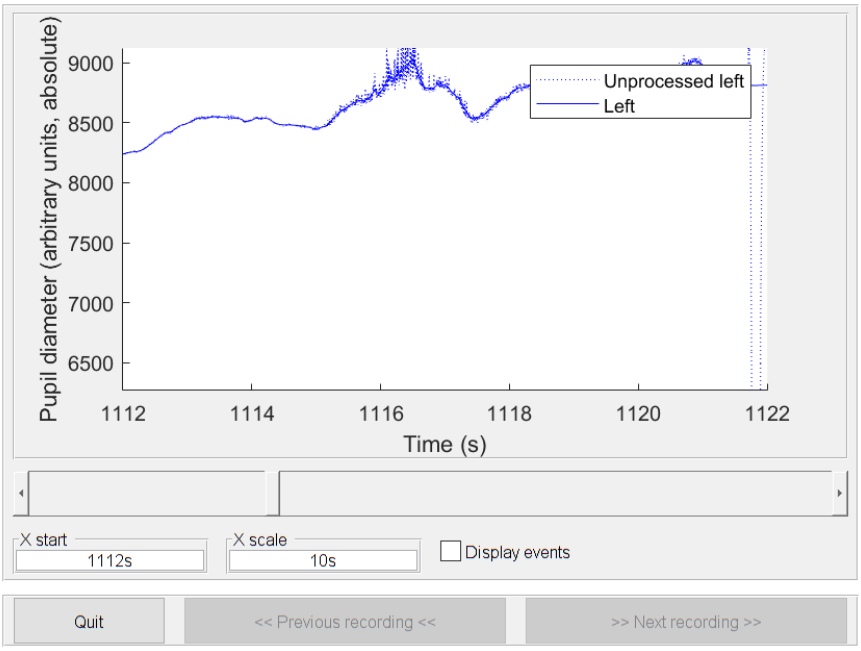

Figure 7: A high-frequency artifact has been smoothed

A.4.4 Downsampling

To speed up subsequent analysis, you can downsample by a factor of 50 to obtain data with a new sample rate of 20 Hz (see 4.10).

The readout for this step appears as follows:

```
Downsampling by a factor of 50
        Running pupl_downsample...
    cS1_b1...
Previous sample rate: 1000.000000 Hz
New sample rate: 20.000000 Hz
    done
    cS2_b1...
Previous sample rate: 1000.000000 Hz
New sample rate: 20.000000 Hz
    done
Done
Redraw
```

A.5 Working with trials

A.5.1 Reading event variables

Trials have the following structure in this experiment:

1. `Trial:[n] [stimulus information]`
2. `Trial:[n] EndStim`
3. `Response:[n] Hit=[n. hits]/[max. hits],FA=[n. false alarms]`

We want to compute the hit rate and the number of false alarms in order to exclude trials where participants may not have been on task.

To compute the number of false alarms, read event variables from events whose names contain the word `Response` (i.e., use the regular expression "Response" to select the events to which event variables will be added). The regular expression to use for reading false alarms from the event names is `FA=(\d+)` (see B.3) and the resulting event variable can be named `#FA` and will be numeric (see 6.1).

The readout should appear as follows:

```
Adding the following event variables:
    #FA (numeric)
        Running pupl_evar_add...
    cS1_b1...
72 non-empty event variables read from 72 events
    done
    cS2_b1...
72 non-empty event variables read from 72 events
    done
Done
```

To compute the hit rate, we need to first read the number of hits and the max. possible hits. To do this, read event variables from events whose names contain the word `Response` again. This time, the regular expression to use for reading from event names is `Hit=(\d)/(\d)` and the event variables can be named `#n_hits` and `#max_hits` (both numeric).

The readout should appear as follows:

```
Adding the following event variables:
    #n_hits (numeric)
    #max_hits (numeric)
        Running pupl_evar_add...
    cS1_b1...
144 non-empty event variables read from 72 events
    done
    cS2_b1...
144 non-empty event variables read from 72 events
    done
Done
```

Next, to actually compute hit rate, we divide the number of hits by the max possible hits. To do this, we compute an event variable (see 6.1.3) for events whose names again contain the word `Response`. The expression to compute the new event variable will be `#n_hits / #max_hits` and the resulting event variable can be called `#HR` (it will be numeric).

The readout for this step will appear as follows:

```
Adding the following event variables:
    #HR (numeric)
        Running pupl_evar_add...
    cS1_b1...
72 non-empty event variables read from 72 events
    done
```

```
    cS2_b1...
72 non-empty event variables read from 72 events
    done
Done
```

A.5.2 Homogenizing event variables within trials

So far, the event variables that have been derived are attached to the `Response` events. However, we want to attach these to the corresponding `Trial` events, which will be used as timelocking events for epoching.

Therefore we will homogenize event variables within trials (see 6.1.4). Trial onsets will be marked by events whose names contain the word `Scene` and trial ends will be marked by events whose names contain the word `Response`.

The readout for this step will appear as follows:

```
Homogenizing event variables within trials...
The following events mark the beginning of a trial:
    "Scene" (regular expression)
The following events mark the end of a trial:
    "Scene" (regular expression)
Adding the event variable #trial_idx
        Running pupl_evar_hg...
    cS1_b1...
5 event variables homogenized across 72 trials
    done
    cS2_b1...
5 event variables homogenized across 72 trials
    done
Done
```

A.5.3 Epoching

The next step is to define epochs (see 7). Epochs will be defined relative to events containing the word `Scene` and will be fixed length, lasting from the occurrence of these events (i.e., `0s` relative to their onsets) til `29.5s` afterward. This type of epoch can be called "trial." After epoching, you will be able to visualize individual epochs (as in figure 8; see 3.5).

The readout for this step appears as follows:

```
Defining epochs called "trial"
Epochs begin at 0s relative to the timelocking events:
    "Scene" (regular expression)
Epochs end at 29.5s relative to the timelocking events
        Running pupl_epoch...
    cS1_b1...
Defining epochs...72 new epochs defined, 72 epochs defined total
Sorting epochs by onset time...done
    done
    cS2_b1...
Defining epochs...72 new epochs defined, 72 epochs defined total
Sorting epochs by onset time...done
    done
Done
```

A.5.4 Baseline correction

Epochs are baseline corrected by subtracting the average pupil diameter between `4s` and `4.5` after the event signalling the onset of a trial (i.e. baselines are fixed-length and there will be a one-to-one mapping between baseline periods and epochs; see 7.2).

To select all epochs for baseline correction, use the regular expression ".", which matches any character.

The readout for this step appears as follows:

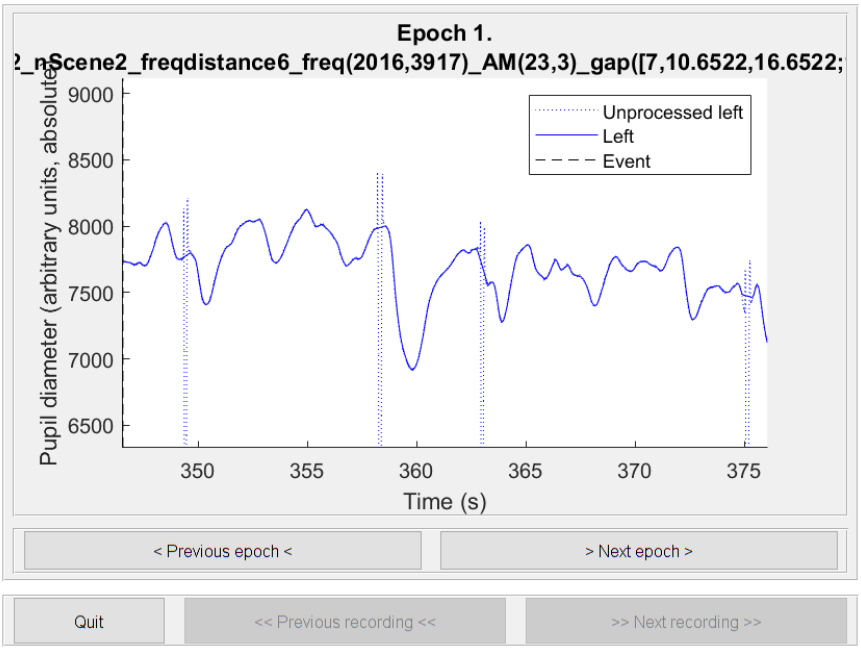

Figure 8: Plot of a single epoch

```
Performing baseline correction using method:
    subtract baseline mean
one:one mapping from baselines to epochs
        Running pupl_baseline...
    cS1_b1...
    done
    cS2_b1...
    done
Done
```

A.5.5 Rejecting epochs

Epochs will be rejected if they contain more than 1 false alarm or if the hit rate is any less than 100%. We will therefore be rejecting epochs on the basis of event variables (see 7.4). Epochs will be rejected according to the criterion `#HR < 1 | #FA > 1`.

The readout from this step will appear as follows:

```
        Running pupl_epoch_reject...
    cS1_b1...
7 new epochs rejected, 7 epochs rejected in total
    done
    cS2_b1...
31 new epochs rejected, 31 epochs rejected in total
    done
Done
```

A.5.6 Defining epoch sets

The next step is to group epochs by difficulty level (easy, medium, or hard). The difficulty level is encoded in the names of the trial onsets: `Trial:[n]...Scene[x]...` where `x` is 1 for easy, 2 for medium, and 3 for hard. Create the epoch set `Easy` out of epochs containing the string `Scene1`; the set `Medium` out of the epochs containing the string `Scene2`; and the set `Hard` out of the epochs containing the string `Scene3` (see

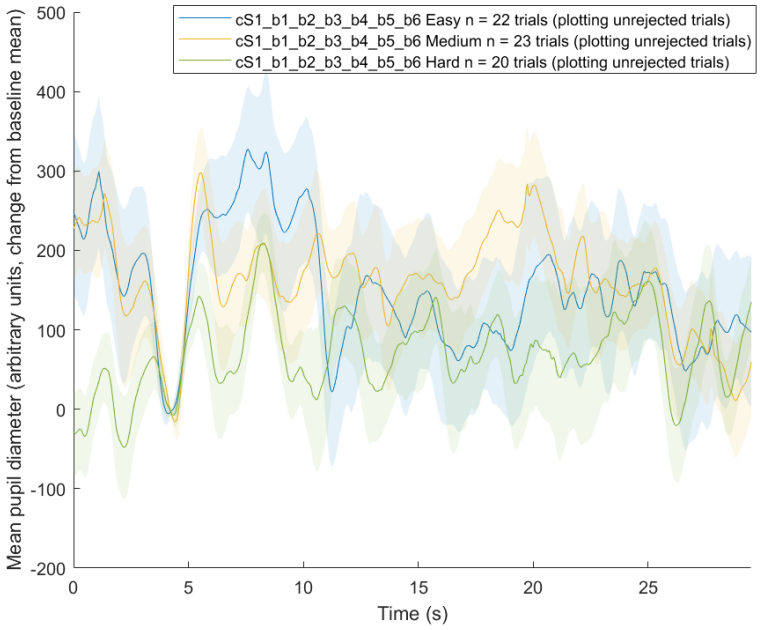

Figure 9: The epoch sets for participant 1

7.5). Each epoch set should contain 24 trials. Once you've done this, you can plot these epoch sets to obtain figure 9 (see 3.6.1).

The readout for this step appears as follows:

```
        Running pupl_epochset...
    cS1_b1...
Set Easy contains 24 epochs
Set Medium contains 24 epochs
Set Hard contains 24 epochs
    done
    cS2_b1...
Set Easy contains 24 epochs
Set Medium contains 24 epochs
Set Hard contains 24 epochs
    done
Done
```

A.6 Processing the entire dataset

The processing steps outlined above define a general-purpose pipeline that can be re-used on the entire dataset. To do this, first export the processing history as a script (see 9.2). It should match the annotated script `code/preprocess.m`.

Next, remove the processed data from PuPl's workspace (see 2.5) and import all of the EDF files in `data-raw/` (this step takes around a half hour on my laptop, since there are 198 in total).

Once the data has been imported, you can process it all at once using the script you exported (see 9.2). This step is also fairly time-consuming for me, but hopefully you have a more powerful computer.

Once the analysis script has finished running, you can plot the epoch sets (grand average by condition, using the default unlabeled condition that contains all recordings; see 3.6.1) to obtain figure 10.

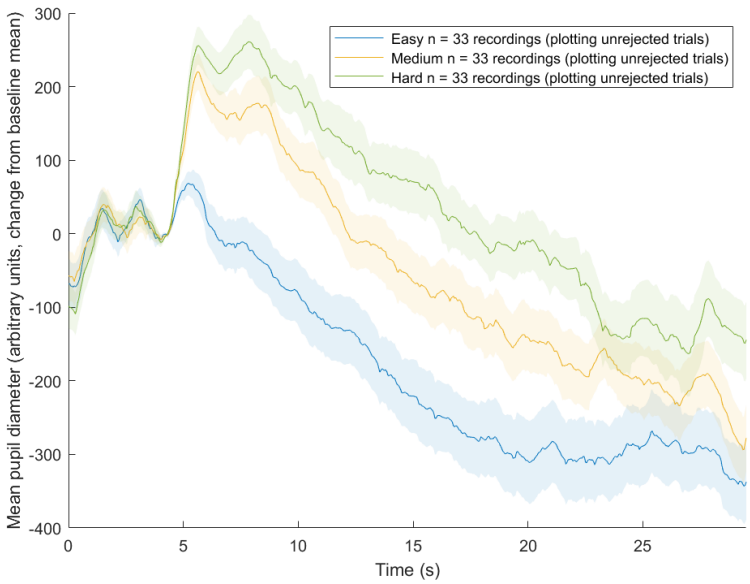

Figure 10: The epoch sets for the entire dataset

A.7 Exporting and analyzing the data

To actually statistically analyze the data, you have to export it to a csv table (see 8).

We want to know whether the pupil diameter over the course of the 25-second trial differs depending on the difficulty level (just imagine it's not blatantly obvious from figure 10).

The sample R code in the file `code/analysis.R` will run the statistical analyses and generate the plots in this section.

A.7.1 Exporting epoch-wise measures for statistical analysis

For standard statistical analysis, we use a participant's average pupil size within trials of a certain type as the unit of analysis. To do this, go to `Experiment > Write statistics to spreadsheet` and specify that epoch set averages (rather than individual epochs) should be analyzed.

Specify that the data of interest occurs between `4.5s` and `29.5s` into the trial. Name this statistics window "trial" and specify that the mean pupil size should be computed in it (as well as whatever other statistics you may be interested in).

Save the table to `export/stats-basic.csv`, which will look like this:

recording	...	epoch_set	trial_mean
cS1_b1_b2_b3_b4_b5_b6	...	Easy	150.4734
cS1_b1_b2_b3_b4_b5_b6	...	Medium	161.2859
cS1_b1_b2_b3_b4_b5_b6	...	Hard	79.7393
cS2_b1_b2_b3_b4_b5_b6	...	Easy	-22.309
cS2_b1_b2_b3_b4_b5_b6	...	Medium	-14.9397
...	...	...	...

If we had named the window of interest "attention", the last column would be called "attention_mean", and if we had computed the median pupil size in this window, it would be called "attention_median".

It is very easy to run an ANOVA on data that is formatted this way. The script `analysis.R` contains code to generate the following ANOVA table:

```
Analysis of Variance Table

Response: trial_mean
          Df  Sum Sq Mean Sq F value    Pr(>F)
epoch_set  2  909080  454540  10.283 8.986e-05 ***
Residuals 96 4243475   44203
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

You can also run a linear model to get the coefficients associated with each level of difficulty:

```
Call:
lm(formula = trial_mean ~ 0 + epoch_set, data = stats_basic)

Residuals:
    Min      1Q  Median      3Q     Max
-397.94 -166.32   -2.27  136.68  648.19

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
epoch_setEasy    -200.11      36.60  -5.468 3.61e-07 ***
epoch_setMedium   -61.00      36.60  -1.667   0.0989 .
epoch_setHard      33.18      36.60   0.906   0.3670
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 210.2 on 96 degrees of freedom
Multiple R-squared:  0.2587,Adjusted R-squared:  0.2355
F-statistic: 11.16 on 3 and 96 DF,  p-value: 2.376e-06
```

Finally, you can generate figure 11.

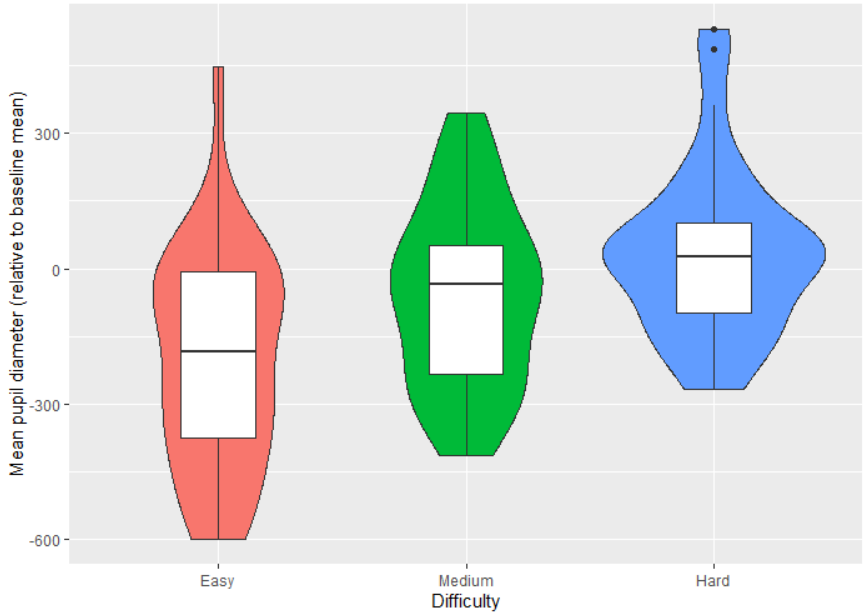

Figure 11: Mean pupil diameter by difficulty level

A.7.2 Exporting for mixed effects analysis

For mixed effects models, the unit of analysis is the mean pupil size during a single trial. To export this type of data, specify that each epoch should be analyzed individually.

Again, define the window of interest as `4.5s` into the trial until `29.5s` into the trial, name this window "trial", and specify that the mean pupil size should be computed in this window. Save the resulting table to `export/stats-long.csv`, which will appear as follows:

recording	...	trial_idx	rejected	evar_FA	...	evar_HR	...	set_Easy	set_Hard	set_Medium	trial_mean
cS1_b1_b2_b3_b4_b5_b6	...	1	0	0	...	1	...	0	0	1	70.1129
cS1_b1_b2_b3_b4_b5_b6	...	2	1	1	...	0.66667	...	0	1	0	188.5881
cS1_b1_b2_b3_b4_b5_b6	...	3	0	0	...	1	...	0	1	0	51.1425
cS1_b1_b2_b3_b4_b5_b6	...	4	0	0	...	1	...	1	0	0	-295.538
cS1_b1_b2_b3_b4_b5_b6	...	5	0	0	...	1	...	1	0	0	241.4843
...	...	...	...	...	...	...	...	...	...	...	...

Each row corresponds to a different trial and the columns contain, among other information, each event variable that was defined (prefixed by "evar_") as well as each epoch set (the columns indicating epoch set membership constitute a design matrix). Finally, "trial_mean" contains the mean pupil diameter over the trial from the corresponding row.

The script `analysis.R` contains code to convert this design matrix into a single factor called "difficulty" and run a linear mixed effects model on the data using the Analysis of Factorial Experiments (afex) package [12]:

```
Type III Analysis of Variance Table with Satterthwaite's method
            Sum Sq Mean Sq NumDF  DenDF F value    Pr(>F)
difficulty 17116939 8558470     2 1724.9  44.432 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As with the standard linear model, you can also get estimates of the coefficients for each difficulty level (these end up being quite close to the linear model's):

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method ['lmerModLmerTest']
Formula: trial_mean ~ 0 + set_Easy + set_Medium + set_Hard + (1 | recording)
   Data: subset(stats_long, !rejected)

REML criterion at convergence: 26364.3

Scaled residuals:
    Min      1Q  Median      3Q     Max
-4.1244 -0.5964 -0.0412  0.5418  6.1894

Random effects:
 Groups    Name        Variance Std.Dev.
 recording (Intercept)  28423   168.6
 Residual              192619   438.9
Number of obs: 1754, groups:  recording, 33

Fixed effects:
           Estimate Std. Error      df t value Pr(>|t|)
set_Easy    -202.85      33.65   43.88  -6.029 3.09e-07 ***
set_Medium   -53.49      34.43   47.99  -1.554    0.127
set_Hard      42.17      36.44   59.92   1.157    0.252
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation of Fixed Effects:
           st_Esy st_Mdm
set_Medium 0.746
set_Hard   0.705  0.694
```

A.7.3 Exporting wide-format downsampled data

If you want to analyze the epoch data sample-by-sample, it is best to export it (downsampled, so that the size is not unmanageable) to a wide-format table (i.e. each row contains an epoch or epoch mean). To do this, go to `Experiment > Export downsampled/binned data` and specify that you want to export epoch set averages. Specify the following downsampling parameters:

Start: `0s`
Width: `500ms`
Step size: (leave empty)
End: `29.5s`

This will compute pupil size in averages of 500-millisecond chunks, which will result in a smaller output file that if no downsampling occurred.

Export the data in wide format as a file called `export/ds-wide.csv`. This file will appear as follows:

recording	...	epoch_set	t1_0_0.45	t2_0.5_0.95	t3_1_1.45	...
cS1_b1_b2_b3_b4_b5_b6	...	Easy	230.0912	250.1837	269.2823	...
cS1_b1_b2_b3_b4_b5_b6	...	Medium	235.3199	234.7713	251.0524	...
cS1_b1_b2_b3_b4_b5_b6	...	Hard	-28.7279	-8.3199	42.9608	...
cS2_b1_b2_b3_b4_b5_b6	...	Easy	-61.4581	-27.7703	-84.3681	...
cS2_b1_b2_b3_b4_b5_b6	...	Medium	-58.9277	-39.4377	-57.4729	...
...	...	...	..	...	...	

The data columns are those that begin with a "t". The format of the data columns is as follows:

```
t<sample number>_<start of downsampled window>_<end of downsampled window>
```

That is, the column `t1_0_0.45` contains the average pupil from 0 ms into the epoch til 450 ms into the epoch, for the particular recording and epoch set corresponding to that row. The script `analysis.R` contains an example of how to read these window limits using a regular expression.

With the table formatted this way, it is easy to write a loop over the data columns and compute, e.g., a different ANOVA for each sample. The script `analysis.R` shows how to do this, collecting the ANOVA outputs to generate figure 12.

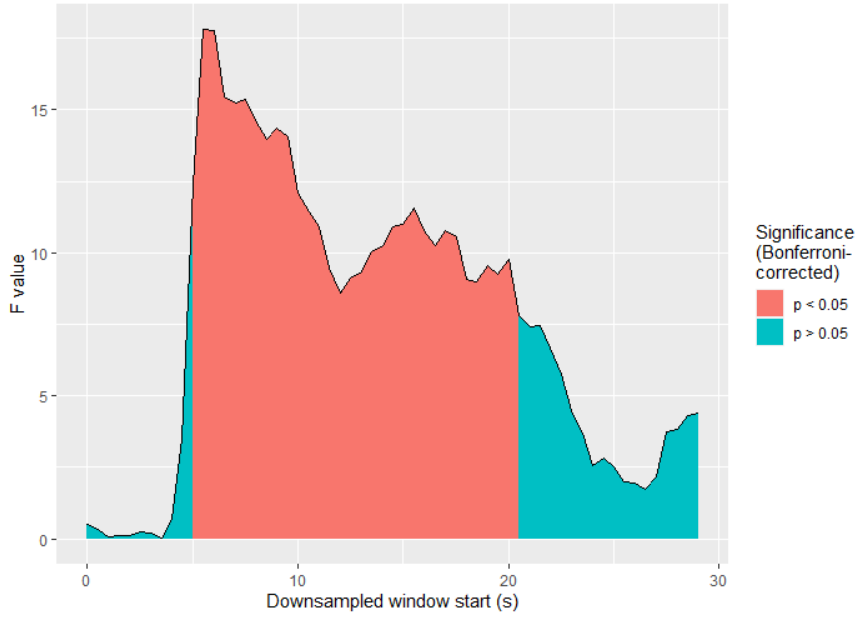

Figure 12: The results of running a separate ANOVA on each sample.

A.7.4 Exporting long-format downsampled data

Long-format data (i.e. each row corresponds to a single data point) is well-suited to generating plots in R. To export long-format data (downsampled, so that the size is manageable), go to `Experiment > Export downsampled/binned data` and specify that you want to export epoch averages (exporting individual epochs generates a table that's approx. 140,000 lines long).

Output the data in long format as a file called `export/ds-long.csv`, which will appear as follows:

recording	...	epoch_set	win_start	win_end	pupil_diameter
cS1_b1_b2_b3_b4_b5_b6	...	Easy	0	0.45	230.0912
cS1_b1_b2_b3_b4_b5_b6	...	Easy	0.5	0.95	250.1837
cS1_b1_b2_b3_b4_b5_b6	...	Easy	1	1.45	269.2823
cS1_b1_b2_b3_b4_b5_b6	...	Easy	1.5	1.95	192.6164
cS1_b1_b2_b3_b4_b5_b6	...	Easy	2	2.45	149.7066
...	...	...	...	...	...

This is essentially a rotation of the wide-format table. The window starts and ends are now in their own columns rather than being encoded in the data column names.

The script `analysis.R` contains code to generate figures 13 and 14 from this table. You can even combine figures 12 and 13 to generate figure 15. Users who are good at ggplot [13] will no doubt be able to generate even more sophisticated visualizations.

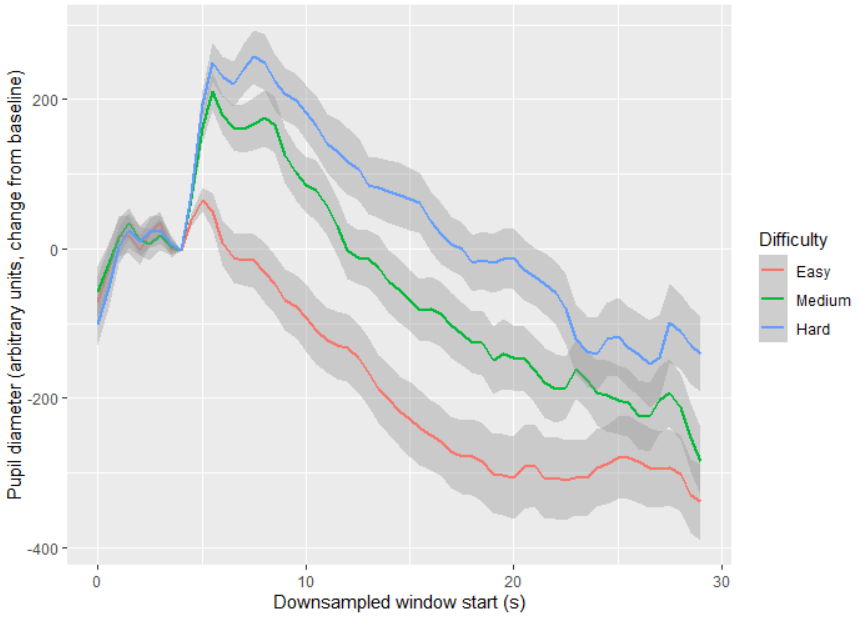

Figure 13: Mean pupil size over the course of a trial, by difficulty. Error bars reflect the standard error of the mean.

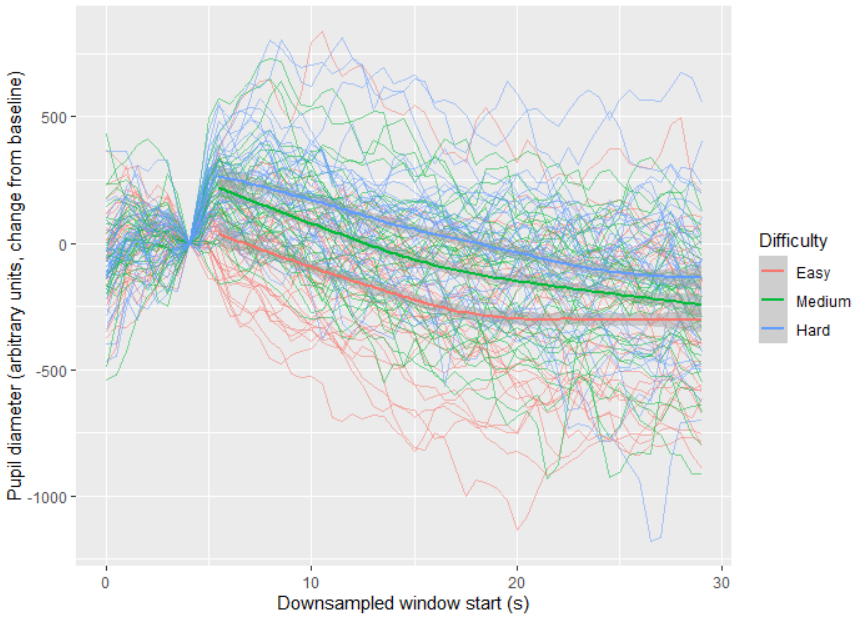

Figure 14: Pupil size over the course of a trial, by difficulty, showing each participant's average plus a LOESS estimate of the average by difficulty level.

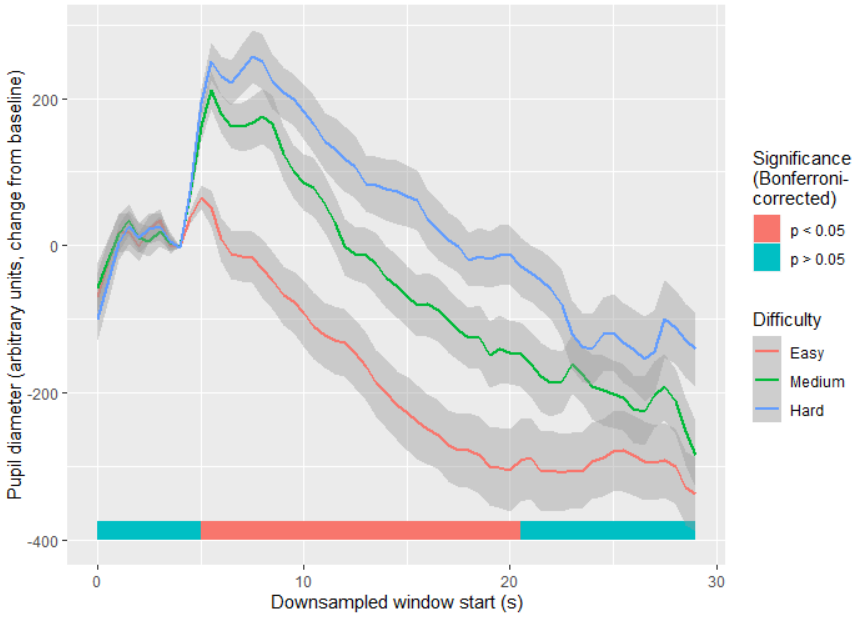

Figure 15: Mean pupil size over the course of a trial, by difficulty. The bar at the bottom indicates the sections where the 3 curves are significantly different (by ANOVA, Bonferroni-corrected).

B Recurring user dialogs

B.1 Specifying durations

You may use units and arithmetic to specify lengths of time. Valid units are as follows:

- `ms`: milliseconds
- `s`: seconds
- `m`: minutes
- `dp/d`: datapoints

For example, `1s + 2 dp - 100 ms - 2d` translates to one second plus two datapoints minus 100 milliseconds minus two datapoints.

B.2 Specifying relative quantities

You can specify relative quantities by writing any valid Matlab expression using `$` to refer to the data in question. E.g. `max($)` would return the max. You can also use the following shortcuts to compute common statistics (note that the backticks are to avoid conflicts with other Matlab expressions):

- `‘mu`: mean
- `‘md`: median
- `‘mn`: min
- `‘mx`: max
- `‘sd`: standard deviation
- `‘vr`: variance
- `‘iq`: interquartile range
- `‘madv`: median absolute deviation
- `%`: percentile (e.g. `15%` computes the 15th percentile)

E.g. `‘mn - 2‘sd` would compute the mean minus two standard deviations. Note that this would be much faster than `mean($) - 2*std($)`, because the latter requires converting the data to a string, substituting that string for every instance of `$`, and evaluating the resulting expression.

B.3 Regular expressions (regexp)

Many of PuPl's user dialogs contain input fields that are intended for regular expressions. A regular expression is a string of characters that specifies a search pattern. For example, the regular expression “abc” would look for any instance of “abc” (“abcdef” would be a match, “acb” would not).

Regular expressions can have special characters to make searches very specific (or very general). A few useful examples:

- a period matches any character. So the regular expression “a.” looks for an “a” followed by any character (“abcdef” and “acb” would both match, “cba” would not because the “a” is not followed by another character).
- a plus means “one or more instances of the immediately preceding pattern.” For example, “ab+” would look for “a” followed by one or more instances of “b” (“abb” and “abc” would match, “acb” would not).
- an asterisk means “zero or more instances of the immediately preceding pattern.” For example, “ab*” would match “abb”, “abc”, and “acb”.
- “\d” matches any digit from (0-9).

- square brackets match any character within them. For example, "a[bc]" looks for an "a" followed by either a "b" or a "c" ("abc" and "acb" would both match, "aa" would not).

A full list of Matlab's regular expression rules can be found here: mathworks.com/help/matlab/ref/regexp.html#btn_p45_sep_shared-expression.

Regular expressions can be frustrating at times, but they're extraordinarily useful once you get the hang of them. When I'm struggling to pick the right regular expression, I find it helps to remember that the computer is doing exactly what I'm telling it to.

B.4 Selecting events

Many of PuPl's tools require you to select a subset of events. (e.g. timelocking events for epoching; see 7). When this occurs, a window like the one depicted in figure 16 will appear. This window allows you to select subsets of events by serial position, by regular expression, or by an event variable filter (see 6.1). The "selected" column shows which events from a given recording are selected using the current parameters. You can toggle the recording currently being previewed using the "previous recording" and "next recording" buttons.

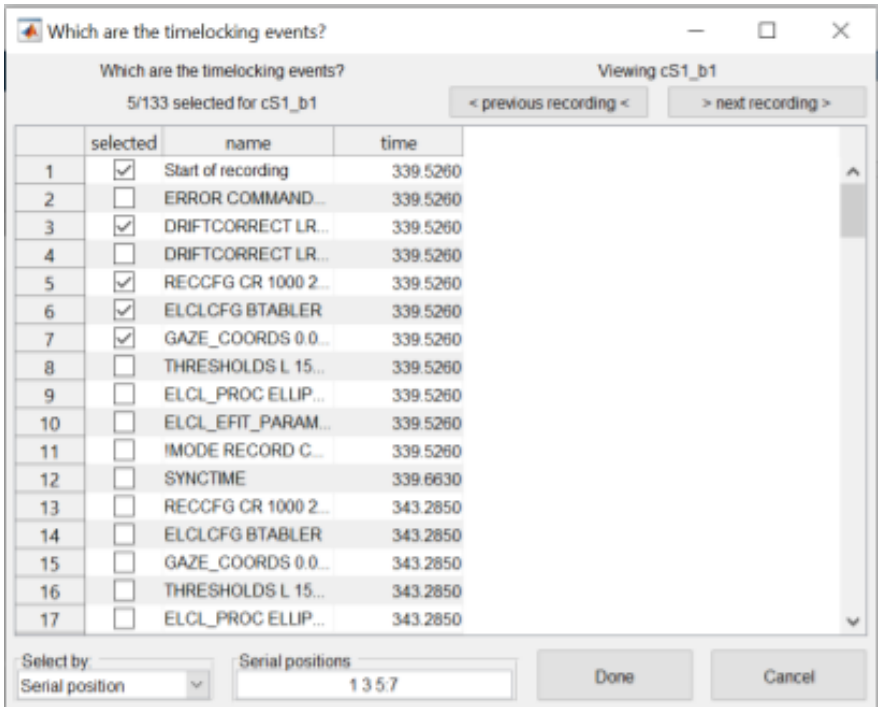

Figure 16: The window that allows users to select a subset of events.

B.4.1 Event selection by serial position

To select events by their serial position, write an expression in the text box labeled "Serial positions" that could be used to index a one-dimensional array (see https://www.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html). For example, `1:10 12` would select events 1 to 10 and event 12; `end-2:end` would select the last 3 events.

B.4.2 Event selection by regular expression

To select events whose names match a certain pattern, put that pattern in the box labeled "Regular expression" (see B.3). E.g. if you wanted to select all events whose names include the word `Response`, you would

write `Response` in that box (then press enter to highlight the events that include that word).

B.4.3 Event selection by event variable filter

To select events that meet some criterion defined by their event variables (see 6.1), put an expression that returns `true` if that criterion is met in the box labeled "Trial var filter". To refer to a event variable, prefix its name with a hash symbol. For example, if you wanted to select events with reaction times (see 6.1.6) greater than 500ms, you would write `#rt > 0.5` in that box (then press enter to highlight the events that meet that criterion). Entering `regexp(#name, 'x')` would be the same as writing `x` in the regexp filter box.

B.4.4 Testing filters

To see which events would be selected by a given filter, go to `Trials > Event variables > Test filter`. This lets you ensure a filter is working as expected before using it in data processing.

B.4.5 Using filters

Which events actually get selected depends on which button you press. If you click "Use highlighted", all of the highlighted events will be selected, regardless of what's written in the text boxes. If you click "Use regexp filter", the regular expression filter will be used, regardless of which events are highlighted. Similarly, if you click "Use trial var filter", the event variable filter will be used, again regardless of what's highlighted.

For example, if you use the regular expression filter to select all events containing the word `Start`, and then you manually deselect the event `Start of recording`, and then you click "Use regexp filter", `Start of recording` will still end up being selected.

References

[1] Julie Brisson, Marc Mainville, Dominique Mailloux, Christelle Beaulieu, Josette Serres, and Sylvain Sirois. Pupil diameter measurement errors as a function of gaze direction in corneal reflection eyetrackers. *Behavior research methods*, 45(4):1322–1331, 2013.

[2] Benjamin Gagl, Stefan Hawelka, and Florian Hutzler. Systematic influence of gaze position on pupil size measurement: analysis and correction. *Behavior research methods*, 43(4):1171–1181, 2011.

[3] Krzysztof J Gorgolewski, Tibor Auer, Vince D Calhoun, R Cameron Craddock, Samir Das, Eugene P Duff, Guillaume Flandin, Satrajit S Ghosh, Tristan Glatard, Yaroslav O Halchenko, et al. The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. *Scientific data*, 3(1):1–9, 2016.

[4] Taylor R Hayes and Alexander A Petrov. Mapping and correcting the influence of gaze position on pupil size measurements. *Behavior Research Methods*, 48(2):510–527, 2016.

[5] Ronen Hershman, Avishai Henik, and Noga Cohen. A novel blink detection method based on pupillometry noise. *Behavior research methods*, 50(1):107–114, 2018.

[6] Marieke Jepma and Sander Nieuwenhuis. Pupil diameter predicts changes in the exploration–exploitation trade-off: Evidence for the adaptive gain theory. *Journal of cognitive neuroscience*, 23(7):1587–1596, 2011.

[7] Mariska E Kret and Elio E Sjak-Shie. Preprocessing pupil size data: Guidelines and code. *Behavior research methods*, 51(3):1336–1342, 2019.

[8] Anaïs Lemercier, Geneviève Guillot, Philippe Courcoux, Claire Garrel, Thierry Baccino, and Pascal Schlich. Pupillometry of taste: Methodological guide–from acquisition to data processing-and toolbox for matlab. 2014.

[9] Sebastiaan Mathôt et al. A simple way to reconstruct pupil size during eye blinks. *Retrieved from*, 10:m9, 2013.

[10] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[11] Dario D Salvucci and Joseph H Goldberg. Identifying fixations and saccades in eye-tracking protocols. In *Proceedings of the 2000 symposium on Eye tracking research & applications*, pages 71–78, 2000.

[12] Henrik Singmann, Ben Bolker, Jake Westfall, and Frederik Aust. *afex: Analysis of Factorial Experiments*, 2019. R package version 0.23-0.

[13] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[14] Matthew B Winn, Dorothea Wendt, Thomas Koelewijn, and Stefanie E Kuchinsky. Best practices and advice for using pupillometry to measure listening effort: An introduction for those who want to get started. *Trends in hearing*, 22:2331216518800869, 2018.

[15] Sijia Zhao, Gabriela Bury, Alice Milne, and Maria Chait. Pupillometry as an objective measure of sustained attention in young and older listeners. *Trends in hearing*, 23:2331216519887815, 2019.