**Documentation of EMF\_RP**

Tao Gong, Jonathan Gordils, David Braze

Haskins Laboratories, New Haven, CT, USA

EMF\_RP (GitHub: github.com/gtojty/emf\_rp) is a general purpose, stand-alone Python software package designed to assist with the chores of implementing single- or multi-line text reading experiments and analyzing collected eye-movement (EM) data. The package consists of a set of functions to generate pixel-based bitmaps of single or multi-line texts for reading, to extract and visualize time-stamped EM data, saccades and fixations detected by eye trackers during reading, and to calculate widely-adopted EM measures to analyze individual reading behavior. Developed using Python 2.7, the package depends on some Python packages, including os, fnmatch, re, csv, codecs, pandas, numpy, PIL, turtle, matplotlib, and winsound. Many of the packages are pre-installed in popular Python distributions in Windows or Linux (WinPython, Anaconda, etc.).

Functions in the package include *user functions*, which are intended to be called directly by users, and *helper functions*, which are called by user functions to implement various tasks. Names of helper functions start with ‘\_’. We also provide batch versions of some user functions, which routinely process the data of multiple subjects in a batching fashion. Names of batching function end with ‘\_b’. The whole package consists of five sections: (1) functions to generate bitmaps of single- or multi-line texts and corresponding region files of word position information in texts; (2) functions to extract time-stamped raw data, detected saccades and fixations, and other relevant information from the ascii data files transformed from EyeLink edf files; (3) functions to classify saccades, fixations, and time-stamped EM data into text lines and word regions, and identify cross-line saccades, fixations, and time-stamped EM data; (4) functions to visualize saccades, fixations, and time-stamped EM data on bitmaps; and (5) functions to calculate region-based EM measures.

In the following sections, we introduce the arguments and outputs of the major user functions in different sections, briefly describe the involved algorithms, and show a list of examples of using those functions. We take the results of a paragraph reading experiment for example to illustrate how to use those functions in practice. The relevant stimuli and data files of the example are in the folders of GitHub repository.

**Paragraph reading experiment:** This experiment aims to analyze individual reading patterns during paragraph reading. It consists of three trials. In each trial, subjects were asked to read overtly a short story shown in one paragraph on the screen, and their EMs were recorded by EyeLink 1000 at a sampling rate of 250 Hz. Their oral reading were recorded by microphone. The stories used in the three trials are in the three text files in the folder ‘bitmap\_regfile’, story01.txt, story02.txt, story03.txt; line breaks specified in the text files are respected in created bitmaps. Commenting line (starting with ‘#’) in each text shows meta-data of the story (title and number of words). Passages.txt contains all the stories in one file. Using the functions in EMF\_RP, we can generate the bitmaps of texts, along with the csv region files. A region file specifies regions of interest (ROIs) within each bitmap; conventionally, one ROI for each word in the story. Bitmaps are used as stimulus items during EM data collection. Region files are used during post-data collection to create region-wise (e.g., word-wise) EM summaries. Additional functions facilitate visualization and classification of saccades and fixations recorded in each trial.

1. **Functions for generating bitmaps and region files**

There are two user functions: Praster is the main workhorse, and Gen\_Bitmap\_RegFile is a convenience wrapper around Praster.

**User functions:**

1. Praster**:** tocreatea single bitmap of single- or multi-line text according to parameters, along with attendant region file.

Arguments:

* *direct*: directory storing text files and generated bitmaps and region files.
* *fontpath*: fully qualified path to a font file. It must be a font accessible to Python.
* *langType*: language of shown texts. It can be ‘English’, or other alphabetic (‘German’, ‘French’, ‘Italian’, etc.) or non-alphabetic (‘Chinese’, ‘Korean’, or ‘Japanese’) languages.
* *stPos*: the way in which the text is displayed on the screen. The starting x position of the text is *lmargin* (see below). If *stPos* is ‘Center’, the text is displayed in the center of the screen. In this case, if the text is in alphabetic languages such as English, its starting baseline is *dim*[1]/2.0 (see below). If the text is in Chinese, Japanese or Korean, its starting baseline is *dim*[1]/2 + *fht*/2.0 (see below). If *stPos* is ‘TopLeft’, the text is displayed in the top left corner of the screen. If the text is in alphabetic languages such as English, its starting baseline is *tmargin* (see below). If the text is in Chinese, Japanese or Korean, its starting baseline is *tmargin* + fht/2.0. Finally, if *stPos* is ‘Auto’, the function will vertically center-align the text to be displayed, based on the number of lines in the text. **Note that this option has to be carefully used. In normal reading experiment, before displaying the text stimuli, a crossing is shown at the starting position of the text to catch reader’s attention. If the texts to be displayed in different trials are automatically vertically center-aligned, the positons of the crossing may be also distinct in different trials.**
* *codeMethod*: encoding method of texts; default is ‘utf\_8’. To display English, French, German, and other alphabetic languages, ‘utf\_8’ is an efficient coding method. To display Chinese, Korean, or Japanese, specific codeMethod is needed. For example, simsun.ttc can be used to display Chinese and Japanese, and batang.ttc to display Korean. Corresponding ttc font files are also put in the example folder.
* *text*: a list containing one or more lines of text for rasterizing, e.g., [u‘The quick brown fox jumps over the lazy dog.’, u‘The lazy tabby cat sleeps in the sun all afternoon.’]. Text is encoded in unicode.
* *dim*: (x, y) bitmap dimension; default is (1280,1024). It is the monitor resolution used to display bitmaps during data collection. Screen-based coordinate system to generate bitmaps and region files places the origin (0,0) in the upper left corner of the bitmap.
* *fg*: font color in rgb format; default is (0,0,0).
* *bg*: background color in rgb format; default is (232,232,232).
* *regfile*: whether to generate the region file together with the bitmap; default is True.
* *lmargin*: left margin for text in pixels; default is 86.
* *tmargin*: top margin for text in pixels; default is 86. *lmargin* and *tmargin* determine the starting position of a multi-line text. For a single-line text, the starting position is at the center of the bitmap.
* *linespace*: line spacing in pixels; default is 43. It can be adjusted if font size is set to a value other than its default value.
* *fht*: font height (maximum vertical distance between the highest and lowest painted pixel of each character shown in a particular font) in pixels; default is 18.
* *fwd*: font width in pixels, default is None. *fwd* takes precedence over *fht*.
* *bbox*: whether or not to draw a minimal bounding box, corresponding to a region of interest around each word in the bitmap; default is False. It is used for testing purposes. Coordinates of the bounding box are stored in region file.
* *bbox\_big*; whether or not to draw an expanded bounding box around each word in the bitmap. Such bounding boxes are expanded vertically per the argument *addspace* below; default is False (not drawing). It is used for testing purposes. Coordinates of the bounding box are also stored in region file.
* *ID*: unique identity for the stimulus item, used to set filenames of generated bitmap and region file; default is *test*, resulting in a bitmap test.png and a region file test.region.csv, both in specified folder (*direct*).
* *addspace*: the amount of vertical padding (in pixels) to be added above the top and below the bottom of each word when creating the ‘big’ ROIs; default is 18.
* *log*: whether or not to record intermediate results in a log file; default is False. It is used for testing purposes.

Outputs:

Praster creates a bitmap in png format (ID.png), which is used as stimulus during data collection and background for visualizing saccades and fixations. If *regfile* is True (default), it also creates a csv region file (ID.region.csv), which is used to generate regional summaries of gaze patterns. Region information of each word in the displayed text is stored as one line of the region file, which consists of 16 columns:

* + *Name*: identical to *ID*; conventionally, a unique ID for each trial or stimulus item.
  + *Language*: type of language.
  + *WordID*: numerical index of each word in the text, starting at 1.
  + *Word*: individual word in the text, including, if present, the space(s) before the word and punctuation behind.
  + *length*: length of a word in characters, including attendant space(s) before the word and punctuation behind.
  + *height*: height of a word in pixels; the maximum vertical distance between the highest and lowest painted pixel of each character shown in a particular font.
  + *baseline*: baseline position in pixels of each line of words, words on the same line have identical baseline values.
  + *line\_no*: numerical index of each line of words, starting at 1.
  + *x1\_pos*, *y1\_pos*: top left corner of the minimal bounding box containing a word (including attendant space(s) before the word and punctuation behind), in pixels.
  + *x2\_pos*, *y2\_pos*: bottom right corner of the minimal bounding box containing a word, in pixels.
  + *b\_x1*, *b\_y1*: top left corner of the expanded bounding box for a word (including attendant space(s) before the word and punctuation behind), in pixels.
  + *b\_x2*, *b\_y2*: bottom right corner of the expanded bounding box containing a word, in pixels. Among the position values, *b\_x1* and *b\_x2* of the expanded bounding box are identical to *x1\_pos* and *x2\_pos* of the minimal bounding box. However, unlike the minimal bounding box, the top and bottom values of the expanded bounding box (*b\_y1* and *b\_y2*) are aligned by lines; all words on the same line have the same *b\_y1* (minimum values of *y1\_pos* of all minimal bounding boxes of these words – *addspace*) and *b\_y2* (maximum values of *y2\_pos* of these minimal boxes + *addspace*). If *addspace* is 0 (not adding extra space above and below a word), *b\_y1* and *b\_y2* of the expanded bounding box correspond to the top and bottom boundaries of all words on the same line.

Algorithm:

Praster reads the single- or multi-line text in *text*. Spaces are associated with the word that they precede, and punctuation is associated with the word that it follows. Praster calculates the height and length of each word (including associated spaces and punctuation), and the maximum height for all words on the same line. Then, it sets up the baseline according to *tmargin*, *lmargin*, and *linespace*. Words on a given line have the same baseline. Bounding box (region) position values are calculated for each word and recorded in the region file. If font size or *tmargin* and *dim* values cause any words to overflow the bitmap boundaries, it throws a warning message to the screen and stops.

The function can generate single- or multi-line text of English, German, French, Dutch, Italian, Spanish, Greek, as well as languages in East Asia, such as Chinese, Korean, and Japanese. Displaying the latter three languages require specific font files as input, and these languages have different print styles from those of the other Latin like languages. For example, in English, words are regularly separated by space. However, in Chinese, there is no space between words; in Japanese or Korean, it is optional to have a space between words, but there is no space between a Chinese-style punctuation (e.g., ‘，’, ‘。’, or ‘、’) and the following word. In addition, the baseline for Chinese, Japanese or Korean is different from that of English and other alphabetic languages. Via argument *stPos*, the function can adjust the baseline to make sure that single- or multi-line text is displayed either in the center of the screen, in the top left corner of the screen, or vertically center-aligned. **Note that due to mismatch between Windows and Linux versions of the Image package in Python, generated region files in the two systems slightly differ in *y1\_pos* and *y2\_pos*, but *x1\_pos* and *x2\_pos* are the same.**

1. Gen\_Bitmap\_RegFile**:** aconvenience wrapper around Praster, primarily to simplify the task of generating many bitmaps and region files at once. It calls Praster to work.

Arguments:

* *direct*: directory of text files for generating bitmaps and region files, the generated files are also stored there.
* *fontName*: name of a chosen font accessible to Python, e.g., ‘LiberationMono’, a mono-space font. Mono-space font is commonly used in text reading experiments, due to the ease of handling word positions.
* *textFileNameList*: list of names of the text files for generating bitmaps and region files. Based on *genmethod* below, the list can contain no, one, or several text files.
* *genmethod*: method to generate bitmaps and region files; default is 2. If it is 0, the function is used for testing purposes; if it is 1, the function reads one text file in *textFileNameList*, which contains one or more single/multi-line texts; if it is 2, the function reads one or more text files in *textFileNameList*, each containing a single- or multi-line text. Here, *textFileNameList* can be empty; in this case, it reads all text files in *direct*.
* Additional arguments passed to Praster(*stPos*, *langType*, *codeMethod*, *dim*, *fg*, *bg*, *lmargin*, *tmargin*, *linespace*, *fht*, *fwd*, *bbox*, *bbox\_big*, *addspace*, and *log*): see Praster.

Outputs:

Gen\_Bitmap\_RegFile generates a bitmap and a region file for each text file in *textFileNameList*.

Algorithm:

Gen\_Bitmap\_RegFileprovides two ways of processing text files. If *genmethods* is 1, it reads one text file in *textFileNameList*. The text file can contain one or more single/multi-line texts, and it generates a bitmap and a region file for each text, whose names are story01.png, story01.region.csv, story02.png, story02.region.csv, etc. If *genmethods* is 2, it reads one or more text files in *textFileNameList*, each containing one single/multi-line text, and generates corresponding bitmaps and region files, whose names are determined by the names of the text files. If some text files are missing in the specified directory (*direct*), it throws a warning message and stops. Here, if *textFileNameList* is empty, it reads all text files in direct, each containing one single/multi-line text, and generates corresponding bitmaps and region files, whose names are determined by the names of the text files.

In a text file, users can separate the text into different lines using the delimiter ‘\r\n’; in a text file of many single- or multi-line texts, users can use the delimiter ‘\r\n\r\n’ (an empty line) to separate different texts. Users can insert comments (starting with ‘#’) to record meta-data of the text, which are squeezed out by the function.

For English and other alphabetic languages texts, words are separated by spaces. For Chinese, there are no spaces between words. To let the region files to correctly record each word’s boundary, users can use ‘|’ to mark word boundaries in the texts, and such separator will be removed in generated bitmaps. For Korean and Japanese, spaces between words are optional, so users can freely insert spaces between words.

**Examples:**

1. Praster(‘.’, fontpath, ‘Center’, ‘English’, text=[u‘The quick brown fox jumps over the lazy dog.’]): generate a bitmap (test.png) displaying the sentence in the middle of it and a region file (test.region.csv) recording the position information of each word in the sentence (*fontpath* is set according to the system used, see examples of Gen\_Bitmap\_RegFile below). The results are in the current folder (‘.’).
2. Praster(‘./example’, fontpath, ‘TopLeft’, ‘English’, text=[u‘The quick brown fox jumps over the lazy dog.’, u‘The lazy tabby cat sleeps in the sun all afternoon.’]): generate a bitmap showing the two sentences, line by line, starting at the top left corner of the screen (according to *lmargin* and *tmargin*, here, using default), and a region file storing the position information of each word in the two sentences. The results are in the pre-existent folder ‘example’.
3. Praster(‘./bitmap\_regfile’, ‘simsun.ttc’, ‘Center’, ‘Chinese’, text = [u’‘我们|爱|你！’]): generate a bitmap showing the Chinese sentence, starting in the center of the screen, and a region file storing the position information of each word (separated by ‘|’) in the sentence. simsun.ttc has to be in the same folder. The results are stored in the folder ‘bitmap\_regfile’.
4. Gen\_Bitmap\_RegFile(‘./bitmap\_regfile’, ‘LiberationMono’, ‘TopLeft’, ‘English’, [‘Passages.txt’], 1): read Passages.txt in the folder ‘bitmap\_regfile’. The text file contains three multi-line texts. It generates three bitmaps (story01.png, story02.png, story03.png) and three region files (story01.region.csv, story02.region.csv, story03.region.csv) in the same folder. On the screen, it also summaries the number of lines in each text displayed in the bitmaps.
5. Gen\_Bitmap\_RegFile(‘./bitmap\_regfile’, ‘LiberationMono’, ‘TopLeft’, ‘English’, [‘story01.txt’, ‘story02.txt’, ‘story03.txt’], 2): read three text files (story01.txt, story02.txt, story03.txt) in the folder ‘bitmap\_regfile’, and generates the same bitmaps and region files as in the previous example. If *textFileNameList* is empty ([ ]), it reads all text files in the folder, and generates the same bitmaps and region files.
6. Gen\_Bitmap\_RegFile(‘./bitmap\_regfile’, ‘simsun.ttc’, ‘TopLeft’, ‘Chinese’, [‘ChineseStory.txt’], 2): read the text file ChineseStory.txt in the folder ‘bitmap\_regfile’, and generates the corresponding bitmap and region file as in the previous example. Here, the encoding method has to be specified using the particular font file simsun.ttc, which has to be in the same folder with the text file. If *textFileNameList* is empty, it reads all text files in the folder, and generates the same bitmaps and region files.
7. Gen\_Bitmap\_RegFile(‘./bitmap\_regfile’, ‘simsun.ttc’, ‘TopLeft’, ‘Japanese’, [‘JapaneseStory.txt’], 2): read the Japanese text file and generates the corresponding bitmap and region file.
8. Gen\_Bitmap\_RegFile(‘./bitmap\_regfile’, ‘batang.ttc’, ‘TopLeft’, ‘Korean’, [‘KoreanStory.txt’], 2): read the Korean text file and generates the corresponding bitmap and region file.

1. **Functions for extracting time-stamped EM data, detected saccades and fixations, and other information in data files**

Functions in this section read eye-tracker generated data files to recover EM events (blinks, saccades, fixations, etc.) as well as other session and trial information, and store all this information in useful data structures for subsequent processing.

The functions require the data file generated by eye-tracker. The data files used in the EMF\_RP package are the ascii data files converted from EyeLink edf files by way of SR Research’s utility edf2asc.exe. To be parsed properly by the functions, the ascii data files must contain certain flags or events. For example, *DATE* indicates the date of data collection, *TRIALID* indicates the ID of different trials, *EBLINK* for a blink event, *EFIX* for a fixation, *ESACC* for a saccade, *START* for the starting time of each trial, *END* for the ending time of each trial, *ARECSTART* for the starting time of recording, *ARECSTOP* for the ending time of recording, etc. Please refer to the EyeLink manual for details of edf/ascii file content; users focusing only on extracted results do not need to understand such detail. We provide helper functions to extract this information from data files and store it in a Python Numpy data frame for subsequent processing.

In the folder ‘./example’, we put three ascii data files (1950138.asc, 1950149.asc, and 1950168.asc) collected in the paragraph reading experiment respectively in three folders having the same names as the corresponding subject IDs. Each file records the EM data of both eyes of a subject (the file name is subject ID) reading the stories shown in the three bitmaps. Apart from the ascii data files, we also put in the corresponding folders the audio files recording the subjects’ reading of those paragraphs in the three trials. Generated csv reports and png figures based on the ascii and audio files are also stored in the corresponding folders.

The functions extract data in ascii data files (e.g., basic information of eye-tracker setting, time-stamped physical locations recorded by the eye-tracker, and physical locations and durations of detected saccades and fixations) and put them into user-friendly data frames. For fixation extraction, we provide the option of lumping (merging) fixations shorter than 50 ms. We also provide functions to store data frames into csv files as reports of stamped data, saccade and fixation. Users can write their own functions to manipulate the data frames and reports. For example, in the paragraph reading experiment, before reading the paragraph line by line, some subjects may browse the whole paragraph, and saccades and fixations during the browsing stage need to be excluded; without clearly grasping the instructions, some subjects may stop in the middle of reading and interact with the experimenter, and saccades and fixations during such interruptions need to be removed. Moreover, considering that there could be many asci data files for extraction, we provide batch functions to routinely process many data files at once.

**User functions:**

1. read\_SRRasc: to read an ascii data file and extract saccades and fixations in data frames.

Arguments:

* *direct*, *subjID*, *ExpType*: same as those in read\_TimeStamp.
* *rec\_lastFix*: whether or not to treat the last fixation in each trial as a valid fixation; default is False. The last fixation in each trial is not conventionally treated as a valid fixation.
* *lump\_Fix*: whether or not to conduct lumping operation (see Algorithm) on extracted fixations; default is True (do lumping). If it is True, the following arguments will be used.
* *ln*: threshold duration in milliseconds for a fixation to be lumped; default is 50. Fixations shorter than *ln* are subject to lumping with adjacent and near enough (determined by *zn*) fixations.
* *zn*: threshold distance in pixels between two fixations for lumping; default is 50, roughly 1.5 character. Fixations closer than *zn* are subject to the lumping operation.
* *mn*: minimum legal fixation duration; default is 50. Any fixation (with or without lumping) shorter than *mn* is marked as an invalid fixation.

Outputs:

read\_SRRasccreates data frames (SacDF and FixDF) respectively storing saccades and fixations from the ascii file. Information of saccade and fixation is stored in different columns of the data frames.

SacDF has 24 columns:

* + *subj*: subject ID.
  + *trial\_id*: numerical index of trials, starting from 0. It is the unique identifier for each trial. In ascii data file, the trial index starting from 0, corresponding to trial 1.
  + *trial\_type*: trial type, a unique label for the trial condition.
  + *sampfreq*: sampling frequency of the eye-tracker, e.g., 250, 500, or 1000, dependent on the model and configuration of the eye-tracker.
  + *script*: name of script used in the experiment.
  + *sessdate*: date of the experiment.
  + *srcfile*: edf file used for generating the ascii data file.
  + *trialstart*: starting time of each trial.
  + *trialend*: ending time of each trial.
  + *tdur*: total duration of each trial, equal to *trialend* – *trialstart*.
  + *recstart*: starting time of recording. It is important for aligning the recorded sound file and the recorded EMs (see section 4.2 for generating animations of showing time-stamped data or fixations and playing sounds).
  + *recend*: ending time of recording.
  + *blinks*: total number of blinks detected in each trial.
  + *eye*: which eye the EM data are recorded from, e.g., ‘L’ (left eye), ‘R’ (right eye), or ‘LR’ (both eyes).
  + *start\_time*: starting time of the saccade in milliseconds.
  + *end\_time*: ending time of the saccade in milliseconds.
  + *duration*: duration of the saccade, equal to *end\_time* – *start\_time* + 1/*sampfreq*.
  + *x1\_pos*, *y1\_pos*: starting position (in pixels) of the saccade.
  + *x2\_pos*, *y2\_pos*: ending position (in pixels) of the saccade.
  + *ampl*: amplitude in degrees of the saccade.
  + *pk*: peak velocity of the saccade.
  + *line\_no*: on which text line this saccade lies.

FixDF contains 23 columns:

* + *subj*, *trial\_id*, *trial\_type*, *sampfreq*, *script*, *sessdate*, *srcfile*, *trialstart*, *trialend*, *tdur*, *recstart*, *recend*, *blinks*, *eye*: same as those in SacDF.
  + *start\_time*, *end\_time*: starting and ending times of the fixation.
  + *duration*: duration of the fixation, equal to *end\_time* – *start\_time* + 1/*sampfreq*.
  + *x\_pos*, *y\_pos*: position (in pixels) of the fixation.
  + *pup\_size*: the radius (or area) of the pupil recorded during the fixation.
  + *valid*: whether it is a valid (‘yes’) or invalid (‘no’) fixation.
  + *line\_no*: same as *line\_no* in SacDF.
  + *region\_no*: on which region this fixation lies.

Algorithm:

read\_SRRascfirst checks whether the ascii data file exists in the specified directory (*direct*). If not, it throws a warning message and stops. If the data file exists,read\_SRRascreads it, extracts relevant information in it, and creates two data frames respectively storing the saccades and fixations for all trials found in it. As for the data file storing saccades and fixations of both eyes, in each trial, read\_SRRasc first extracts and stores saccades and fixations of the left eye, and then the right eye.

An optional operation during extraction of fixations is lumping (merging) fixations. Users need to set *lump\_Fix* as True to apply this operation. Generally speaking, a fixation shorter than 50 ms (marked by the argument *ln*) is deemed short and unreliable for follow-up analysis.

The lumping operation proceeds as follows. First, for each short fixation (whose duration is smaller than the value of *ln*), the function checks its previous and next fixations. If those fixations are also short, the function merges them into one fixation. The starting and ending times of the merged fixation are the earliest and latest times of the original fixations, its duration is the sum of the durations of the original fixations, and its position is the centroid of the original fixations. Second, if the previous and next fixations are not short or the duration of the merged fixation is still short, the function keeps checking the distance (in terms of *x\_pos*) between this fixation and any others physically close to it. If the distance between this fixation and one of its neighbors is smaller than the value of *zn* (say, 50 pixels) the two fixations are merged. If the distances between it and more than one of its neighbor are smaller than the value of *zn*, the function merges it with its closest neighbor. Finally, if the distances between it and all its neighbors are bigger than the value of *zn*, no operation is done. As for the data file storing fixations of both eyes, the lumping operation is applied to fixations of the same eye.

No matter whether the lumping operation is conducted, before storing extracted fixations, the function marks any fixation shorter than the value of *mn* (say, 50 ms) as invalid fixations. If *rec\_lastFix* is False, the last fixation in each trial is also marked as invalid.

1. write\_Sac\_Report: to store the data frame SacDF into a csv file as the generated saccade report. The csv file has the name \*\_Sac.csv, ‘\*’ is subject ID. It has three arguments: *direct*, directory of the generated csv file; *subjID*, subject ID; and *SacDF*, data frame of extracted saccades.
2. write\_Fix\_Report: store the data frame FixDF into a csv file as the generated fixation report. The csv file has the name \*\_Fix.csv, ‘\*’ is subject ID. It has three arguments: *direct*, directory of the generated csv file; *subjID*, subject ID; and *FixDF*, data frame of extracted fixations.
3. read\_write\_SRRasc: to read a particular ascii data file, extract saccades and fixations in it, and store corresponding data frames into saccade and fixation reports. It first checks whether the data file exists in the specified folder; if not, it throws a warning message and stops. It has the same arguments as in read\_write\_TimeStamp, plus the additional arguments (*rec\_lastFix*, *lump\_Fix*, *ln*, *zn*, *mn*) used by read\_SSRasc. It calls read\_SSRasc, write\_Sac\_Report, and write\_Fix\_Report to work.
4. read\_write\_SRRasc\_b: the batch version of read\_write\_SRRasc. It reads one or more ascii data files of the same type (*ExpType*) in the specified folder (*direct*), extracts saccades and fixations in those files, and stores corresponding data frames into saccade and fixation reports. If there is no such file in the specified folder, it throws a warning message and stops. It has no argument *subjID*, the other arguments are the same as those in read\_write\_SRRasc.
5. read\_TimeStamp: to read an ascii data file and extract time-stamped data into a data frame.

Arguments:

* *direct*: directory of the ascii file.
* *subjID*: subject ID, name of the ascii file.
* *ExpType*: type of experiment, it can be ‘RP’, meaning ‘reading paragraph’, in the paragraph reading experiment.

Outputs:

read\_TimeStampcreates a data frame, StampDF, storing time-stamped EM data recorded in the ascii data file. Each data point is stored in one line with many columns.

StampDF has 26 columns:

* + *subj*, *trial\_id*, *trial\_type*, *sampfreq*, *script*, *sessdate*, *srcfile*, *trialstart*, *trialend*, *tdur*, *recstart*, *recend*, *blinks*, *eye*: same as those in SacDF.
  + *time*: time tag.
  + *x\_pos1*, *y\_pos1*: position (in pixels) of the left/right eye at the time tag.
  + *pup\_size1*: the radius (or area) of the pupil recorded.
  + *x\_pos2*, *y\_pos2*: if it is double eye recording, position (in pixels) of the right eye at the time tag; if it is single eye recording, they are NaN (missing value).
  + *pup\_size2*: if it is double eye recording, the radius (or area) of the right pupil recorded; if it is single eye recording, it is NaN.
  + *line\_no*: on which line of the text this EM lies; initial value is NaN.
  + *gaze\_region\_no*: on which region this EM lies; initial value is NaN.
  + *label*: word of the region on which this EM lies; initial value is NaN.
  + *error\_free*: whether this EM record is free of error (1) or not (0); default is 1.
  + *Fix\_Sac*: whether this EM belongs to a fixation (‘Fix’) or a saccade (‘Sac’); initial value is NaN. *line\_no*, *gaze\_region\_no*, *label*, and *Fix\_Sac* are assigned by functions in Section 3.

Algorithm:

read\_TimeStampfirst checks whether the ascii data file exists in the specified directory (*direct*). If not, it throws a warning message and stops. If the file exists,read\_TimeStampreads it, extracts relevant information in it, and creates the data frame for all trials found in it.

1. write\_TimStamp\_Report: to store the data frame StampDF into a csv file as the generated time-stamped data report. The csv file has the name \*\_Stamp.csv, ‘\*’ is subject ID. It has three arguments: *direct*, directory of the generated csv file; *subjID*, subject ID; and *StampDF*, data frame of extracted time-stamped data.
2. read\_write\_TimeStamp: to read an ascii data file and extract time-stamped data into a data frame, and write it into a csv file as the generated time-stamped data report. It calls read\_TimeStamp and write\_TimeStamp\_Report to work. It has the same arguments as those in read\_TimeStamp.
3. read\_write\_TimeStamp\_b: the batch version of read\_write\_TimeStamp. It reads one or more ascii data files of the same type in the specified folder, extracts time-stamped data in those files, and stores corresponding data frames into time-stamped data reports. If there is no data file in the specified folder, it throws a warning message and stops. It has the same arguments of read\_write\_TimeStamp, except *subjID*.

**Examples:**

1. SacDF, FixDF = read\_SRRasc(‘./example’, ‘1950138’, ‘RP’): read 1950138.asc in the folder ‘./example/1950138’, and extract saccades and fixations into SacDF and FixDF. The lumping operation is done based on default parameters.
2. write\_Sac\_Report(‘./example’, ‘1950138’, SacDF): generate a saccade report based on SacDF (1950138\_Sac.csv), the report is stored in the folder ‘./example/1950138’.
3. write\_Fix\_Report(‘./example’, ‘1950138’, FixDF): generate a saccade report based on FixDF (1950138\_Fix.csv), the report is stored in the folder ‘./example/1950138’.
4. read\_write\_SRRasc(‘./example’, ‘1950138’, ‘RP’): read 1950138.asc in the folder ‘./example/1950138’, extract saccades and fixations therein, and generate reports.
5. read\_write\_SRRasc\_b(‘./example’, ‘RP’): read all ascii files in the folder ‘./example’, extract saccades and fixations therein, and generate reports.
6. StampDF = read\_TimeStamp(‘./example’, ‘1950138’, ‘RP’): read 1950138.asc in the folder ‘./example/1950138’, and extract time-stamped data into StampDF.
7. write\_TimeStamp\_Report(‘./example’, ‘1950138’, StampDF): generate a time-stamped data report based on StampDF (1950138\_Stamp.csv). The report is in the folder ‘./example’.
8. read\_write\_TimeStamp(‘./example’, ‘1950138’, ‘RP’): read 1950138.asc in the folder ‘./example/1950138’, extract time-stamped data and store the report (1950138\_Stamp.csv) in the same folder.
9. read\_write\_TimeStamp\_b(‘./example’, ‘RP’): read all ascii data files in the folder ‘./example’, extract time-stamped data therein, and generate reports.
10. **Functions for classifying time-stamped data, saccades and fixations into different lines of text and word regions, and identifying cross-line saccades and fixations**

These functions address the occasional messiness encountered in EM data over print, especially with regard to vertical positions of fixations, to classify saccades and fixations as corresponding to different lines of a multi-line text stimulus, and to classify fixations as corresponding to different word regions of a multi-line text stimulus. Classifications conducted by these functions can be used to facilitate calculation of regional summaries of gaze data.

In multi-line text reading experiments, analysis of reading patterns is usually based on region-wise (e.g., word-wise) EM summaries, and how to reliably classify EM data, and especially fixations, belonging to specific words in different lines of text is critical for analysis. This chore is difficult in the context of multi-line text presentation, given that in practice it is challenging to accurately measure vertical positions of fixations. In addition, gaze data of some subjects during multi-line text reading may not be well-aligned with the baselines of displayed text. Furthermore, some subjects tend to jump forward and backward during multi-line text reading, especially at the beginning or end of a line, and such long EM may not be achieved by a single saccade, which adds difficulty in correctly assigning fixations into different lines of text. Noting these, extending from line-by-line presentation to multi-line presentation is by no means trivial, but requires technical and psychological considerations. Yet, reading multi-line texts is common in everyday life, and in the experimental context, such behavior may afford the opportunity for interesting research questions concerning text comprehension in a way that is more nature than what can be done with line-by-line presentation of multi-line texts.

The functions in this section aim to overcome these difficulties and enable an efficient, automatic, and reliable way of classifying EM data into different lines of text, assigning fixations into different word regions, and capturing forward and backward cross-line jumps during reading. We design our algorithms with the following four assumptions, all of which are derived from our experiences of conducting multi-line text reading experiments.

First, we rely on long cross-line saccade or fixation transitions from one line of text to the next (or previous) line to locate the lines of text that subjects are currently reading. A saccade is *a cross-line saccade*, if it starts at the current line of text and stops at the next (forward cross-line saccade) or the previous (backward cross-line saccade) line. A fixation is *a cross-line fixation*, if it occurs at the current line of text and the immediately previous fixation occurs at the previous (forward cross-line fixation) or the next (backward cross line fixation) line. Such transition usually occurs at the beginning or end of a line, but due to parafoveal reading and reader choices of optimal viewing position within a word, it may not necessarily start at the first or last word of a line.

Second, we assume that during a normal reading of multi-line text, subjects would not typically jump, forward or backward, more than one line of text. In addition, a backward cross-line EM is usually achieved in a single saccade, whereas a forward cross-line EM may be achieved in one or a series of EMs. This is based on observations of many subjects in multi-line text reading experiments.

Third, typical regressive EMs within a line of text are usually not big in terms of horizontal movement (jumping only a few words), whereas a cross-line saccade tends to jump a large horizontal distance (more than a half of total words in the same line). Therefore, a large, forward jump in terms of horizontal position can indicate that subjects start to read a new line, whereas a large, backward jump can indicate that subjects go back to the previous line.

Finally, at the end of reading, subjects may rescan some parts of the whole paragraph. Saccades and fixations detected during such wrap-up behavior are often not useful for analysis, and need to be removed. Large change in vertical position can indicate such wrap-up behavior. Therefore, vertical position, though not accurate enough to classify fixations across lines, is still informative at the end of reading.

These assumptions are implemented in relevant helper functions. They attempt to identify which line of text subjects are reading based on saccades with large (and mostly) horizontal displacement and/or temporally-concatenated fixations inducing large horizontal distance, and detect the end of reading based on saccades with largely vertical displacement and/or temporally-concatenated fixations inducing large vertical distance. Considering that some of the assumptions may not hold in extreme cases and the classified results therein could be questionable, the functions can generate a log file to record information of questionable data for additional check.

User functions in this section first read the time-stamped data, saccade and fixation reports (\*\_Stamp.csv, \*\_Sac.csv and \*\_Fix.csv). Then, they call relevant helper functions to classify the EM data into different lines of text and word regions. Finally, they create data frames respectively storing classified EM data and identified cross-line saccades and fixations. We also provide user functions to generate reports of classified EM data and cross-line saccades and fixations. To ease processing, we provide batching functions to routinely generate such reports for many subjects at once. Apart from these functions, we provide two user functions unifying all the processing from reading ascii data files to generating csv reports.

**User functions:**

1. cal\_crlSacFix: toread saccade and fixation reports of a subject, classify the EM data into different lines of text (for saccades) and different word regions (for fixations), and return data frames recording classified and cross-line saccades and fixations.

Arguments:

* *direct*: directory of saccade and fixation reports (\*\_Sac.csv and \*\_Fix.csv) and region file.
* *subj*: subject ID.
* *regfileNameList*: list of region file names. Region files are sorted according to the experimental trials. If it is empty ([ ]), the function automatically reads all \*.region.csv files in *direct* as region files.
* *ExpType*: type of experiment.
* *classify\_method*: method to classify fixations into different lines of text; default is ‘DIFF’. If it is ‘DIFF’, fixations are classified based on identified cross-line fixations; if it is ‘SAC’, fixations are classified based on identified cross-line saccades. See more description in Algorithm. Since the logics of identifying cross-line fixations and saccades appear to be similar, the classification results using either method are usually identical; in cases where some saccades are not captured by eye-tracker (due to blinking or other reason), the classification results based on ‘DIFF’ are more reliable than those based on ‘SAC’.
* *recStatus*: whether or not record information of questionable EM data into a log file; default is True (recording). See more description in Algorithm.
* *diff\_ratio*: ratio of the maximum horizontal distance between the center of the last word and that of the first word in a line of text; default is 0.6. It determines the threshold for identifying a cross-line saccade and fixation. See more description in Algorithm.
* *frontrange\_ratio*: ratio of the maximum horizontal distance between the center of the last word and that of the first word in a line of text; default is 0.2. It is used to identify backward cross-line saccades and fixations. See more description in Algorithm.
* *y\_range*: threshold of vertical position change (in pixels); default is 60. Saccade or fixation inducing a vertical position change greater than this threshold can indicate the end of reading. See more description in Algorithm.
* *addCharSp*: number of single character space added to region boundaries (left side of the first word or right side of the last word in a text line) to capture overshoot fixations; default is 1.

Outputs:

cal\_crlSacFix reads the saccade and fixation reports into data frames SacDF and FixDF, identifies cross-line saccades and fixations among the EM data, stores identified cross-line saccades and fixations in crlSacDF and crlFixDF, respectively, and updates column *line\_no* in SacDF and FixDF and column *region\_no* in FixDF accordingly. The function returns the above four data frames. For saccade and fixation reports of both eyes, the function first classifies the EM data of the left eye, and then the right eye.

After update, column *line\_no* in SacDF is changed from NaN to a number corresponding to the index (starting from 1) of a line in text, or a combination of two line indices linked by ‘\_’, e.g., ‘2\_3’, which indicates that the saccade is a cross-line saccade from line 2 to line 3 of text. Column *line\_no* of saccades after the end of reading remains NaN. Column *line\_no* in FixDF is updated from NaN to a line index, indicating the line of text to which that fixation belongs. Column *line\_no* of fixations occurring after the end of reading remains NaN. Column *region\_no* in FixDF is changed from NaN to a word region index according to the region file. Column *region\_no* of fixations occurring after the end of reading remains NaN.

crlSacDF records the information of identified cross-line saccades. It has 15 columns:

* + *subj*, *trial\_id*, *eye*: same as those in SacDF.
  + *startline*, *endline*: starting and ending lines of cross-line saccades.
  + *Saclineindex*: numerical index of saccade in SacDF, starting from the first one of the trial.
  + *start\_time*, *end\_time*, *duration*, *x1\_pos*, *y1\_pos*, *x2\_pos*, *y2\_pos*, *ampl*, *pk*: identical to those in SacDF.

crlFixDF stores the information of identified cross-line fixations, it has 13 columns:

* + *subj*, *trial\_id*, *eye*: identical to those in FixDF.
  + *startline*, *endline*: starting and ending lines of cross-line fixations. Current fixation is at *endline*, and immediately previous fixation is at *startline*.
  + *Fixlineindex*: numerical index of fixation in FixDF, starting from the first one of the trial.
  + *start\_time*, *end\_time*, *duration*, *x\_pos*, *y\_pos*, *pup\_size*, *valid*: same as those in FixDF.

If *recStatus* is True, cal\_crlSacFix may generate a text file (log.txt) recording information (subject ID, trial ID, saccade or fixation) of data that could be questionable. The format of such text file could be: ‘Subj: 1950138 Trial 1 crlFix start/end need check!’ or ‘Subj: 1950138 Trial 2 crlSac start/end need check!’ The messages indicate that the data of a particular subject (1950138) in a particular trial (1 or 2) need additional check. In general, within a trial, if the very first cross-line saccade or fixation identified is a backward one (indicating that some data are missing or the first part of the data are messy), or the last cross-line saccade or fixation identified does not cover the last text line (indicating that the data are incomplete or some cross-line saccades or fixations are incorrectly identified), the data in this trial will be marked as questionable, and the relevant information will be passed to log.txt.

Algorithm:

cal\_crlSacFix first checks whether the relevant EM reports and region files exist in the specified folder (*direct*); if not, it throws a warning message and stops. If all required files exist, it reads the reports, calls relevant helper functions to classify saccades and fixations, and stores the results in data frames.

Identification of cross-line saccades and fixations proceeds as follows. Here, we only describe the identification of cross-line fixations; the same logic applies to identification of cross-line saccades.

Based on the region file, we first measure the horizontal distance between the center of the first word and that of the last word in each line of text; such distance can be viewed as the maximum distance for that line of text. Based on these distances, the cross-line jumping thresholds (*diff\_ratio* × maximum distance) and the frontal regions (*frontrange\_ratio* × maximum distance) for each line of text are set up.

Then, we group concatenated fixations all moving leftward (*a leftward moving fixation* is a fixation whose *x\_pos* is smaller than that of its previous neighbor). For rightward moving fixations (*a rightward moving fixation* is a fixation whose *x\_pos* is bigger than that of its previous neighbor), each of them forms a group. We use ‘leftward’ or ‘rightward’, instead of ‘regressive’, because in the context of multi-line text reading, a leftward moving fixation is not necessarily regressive. For example, a typical ‘retrace’ saccade from the end of one line to the beginning of the next, although its direction is leftward, actually moves forward to the word on the next line.

After grouping the fixations, we examine each group in sequence. For each rightward moving fixation, if its previous fixation occurs in the frontal region of the line and the horizontal distance between it and its previous fixation is greater than the cross-line jumping threshold of the line, the fixation is identified as a backward cross-line fixation.

For each group of leftward moving fixation, there are two situations. First, the group contains only one fixation. In this case, if the horizontal distance between the fixation and its previous neighbor is greater than the cross-line jumping threshold of the line, the fixation is identified as a forward cross-line fixation. Second, the group contains more than one fixation. In this case, if the horizontal distance between the last fixation and the first one in the group is greater than the cross-line jumping threshold, the group of fixations is deemed to collectively achieve a forward cross-line EM.

In the second case, we need to further identify which fixation in the group is a cross-line fixation. If there exists a fixation in the group such that the horizontal distance between it and its previous neighbor (in the same group of the previous group) is greater than the cross-line jumping threshold of the line, that fixation is identified as the forward cross-line fixation. If there is no such fixation, we look for the fixation(s) having the greatest change in either horizontal (difference in *x\_pos*) or vertical (difference in *y\_pos*) position compared with its previous neighbor. If the fixation having the greatest change in horizontal position is identical to the one having the greatest change in vertical position, that fixation is identified as the forward cross-line fixation. If the two fixations are distinct, the one having a smaller index (occurring earlier in time) is identified as the forward cross-line fixation.

Once a cross-line fixation is identified, we update the cross-line jumping threshold and the frontal region by the values of the next or previous line of the text, dependent on whether the cross-line fixation is forward or backward. After the threshold and the frontal region of the last line of text are used to identify cross-line fixations, if there remain some fixations unclassified, they are classified into the last line of text. Among them, if there exists one fixation such that the vertical distance between it and its previous neighbor is greater than *y\_range*, the reading is assumed to end right before that fixation, and column *line\_no* of that fixation and those afterward are set to NaN. However, if all fixations are classified before the threshold and the frontal region of the last line of text are used for identification, the data of that trial are flagged as questionable, and the relevant information is passed to log.txt.

Finally, after cross-line fixations are identified and stored in crlFix, other fixations lying between two cross-line fixations are classified into the corresponding line. For fixations of both eyes, we first classify cross-line fixations of the left eye, and then the right eye.

If *classify\_method* is ‘SAC’, we use the above logic to identify cross-line saccades. Then, we classify fixations based on the starting and ending times of cross-line saccades; fixations whose starting and ending times are between the starting (or ending) times of two concatenated cross-line saccades in crlSac are classified into the corresponding line, and column *line\_no* of the fixations occurring after the starting time of the saccade indicating the end of reading (inducing a vertical position change greater than *y\_range*) are set to NaN.

After classifying saccades and fixations into different text lines, the function assigns fixations into word regions of the text. For regions at the beginning or the end of a line of text, fixations may overshoot and lie slightly out of the word regions. To capture such overshoot fixations around the boundary words of each line, we extend the left boundaries of the starting words of each line and the right boundaries of the ending words of each line by *addcharSp*. For regions in the middle of a line, no boundary adjustment is needed, fixations are assigned to regions according to the bounding boxes around the words and the line indices of fixations.

1. write\_Sac\_crlSac: to store the data frames of classified saccades (SacDF) and cross-line saccades (crlSac) into csv files (\*\_Sac.csv and \*\_crlSac.csv), ‘\*’ is the subject ID. It has four arguments: *direct*, directory of the generated csv file; *subjID*, subject ID; *SacDF* and *crlSac*, data frames of classified saccades and cross-line saccades.
2. write\_Fix\_crlFix: to store the data frames of classified fixations (FixDF) and cross-line fixations (crlFix) into csv files (\*\_Fix.csv and \*\_crlFix.csv), ‘\*’ is the subject ID. It has four arguments: *direct*, directory of the generated csv file; *subjID*, subject ID; *FixDF* and *crlFix*, data frames of classified fixations and cross-line fixations.
3. cal\_write\_SacFix\_crlSacFix: to read the saccade and fixation reports from a particular subject and relevant region files, classify these saccades and fixations, and store classified and cross-line saccades and fixations into csv files. It may generate a log file recording information of questionable data. It first checks whether the reports of the subject and relevant region files exist in the specified folder; if not, it throws a warning message and stops. It calls cal\_crlSacFix, write\_Sac\_crlSac and write\_Fix\_crlFix to work. It has the same arguments as cal\_crlSacFix.
4. cal\_write\_SacFix\_crlSacFix\_b: the batch version of cal\_write\_SacFix\_crlSacFix. It reads the saccade and fixation reports from one or more subjects and relevant region files, classifies these saccades and fixations, and stores classified and cross-line saccades and fixations into csv files. It may generate a log file recording information of questionable data. It first collects the reports and relevant region files in the specified folder; if there is no such file, it throws a warning message and stops. It has no argument *subjID*, and the other arguments are the same as those in cal\_write\_SacFix\_crlSacFix.
5. read\_cal\_SRRasc: to read ascii data file of a subject, extract saccades and fixations therein, classify them, and return data frames respectively storing classified saccades (SacDF), cross-line saccades (crlSac), classified fixations (FixDF), and cross-line fixations (crlFix). Its arguments*direct, subjID, regfileNameList*, *ExpType*, *classify\_method*, *recStatus*, *diff\_ratio*, *frontrange\_ratio*, *y\_range*, *addCharSp* are the same as those in cal\_crlSacFix, and its arguments *rec\_lastFix*, *lump\_Fix*, *ln*, *zn*, *mn* are the same as those in read\_SRRasc.
6. read\_cal\_write\_SRRasc: to read the ascii data file of a particular subject, extract saccades and fixations therein, classify them, and generate csv reports of classified and cross-line saccades and fixations. It first checks whether the ascii data file and relevant region files exist in the specified folder; if not, it throws a warning message and stops. It calls read\_cal\_SRRasc, write\_Sac\_crlSac and write\_Fix\_crlFix to work. It has the same arguments as those in read\_cal\_SRRasc.
7. read\_cal\_write\_SRRasc\_b: the batch version of read\_cal\_write\_SRRasc. It reads ascii data files of one or more subjects, extracts saccades and fixations therein, classify them, and generate csv reports of classified and cross-line saccades and fixations. It first collects the ascii data files and relevant region files exist in the specified folder; if there is no such file, it throws a warning message and stops. It has no argument *subjID*, and the other arguments are the same as those in read\_cal\_write\_SRRasc.
8. cal\_TimeStamp**:** to read time-stamped data report of a subject, classify the EM data into different lines of text and word regions, and return a data frame with classified time-stamped data.

Arguments:

* *direct*, *subj*, *regfileNameList*, *ExpType*: same as those in cal\_crlSacFix.
* *align\_method*: method to classify time-stamped data into different lines of text. There are two options here. If *align\_method* is ‘FixRep’, classification of time-stamped data are based on the fixation report generated by DataViewer and already classified by hand (the file is “subj-FixReportLines.txt”). If it is ‘Fix\_Sac’, classification of time-stamped data are based on FixDF and SacDF generated by cal\_crlSacFix.
* *addCharSp*: same as those in cal\_crlSacFix.

Outputs:

cal\_TimeStamp reads the time-stamped data into data frame StampDF, and relevant fix report data into FixRepDF, or relevant saccade and fixation reports into data frames SacDF and FixDF. Then, it classifies each time-stamped data into the corresponding text line (updating the column *line\_no*) and word region (updating the column *gaze\_region\_no*), based on either FixRepDF or SacDF and FixDF. It also classifies whether a time-stamped data point belongs to a fixation or a saccade. If the starting and ending times of a time-stamped data point fall into the starting and ending time of a particular saccade, that data point is classified as a saccade: its column *Fix\_Sac* is set to ‘Sac’ and its *line\_no* is set to the *line\_no* of that saccade. If the starting and ending times of a time-stamped data point fall into the starting and ending time of a particular fixation, that data point is classified as a fixation: its column *Fix\_Sac* is set to ‘Fix’ and its *line\_no* is set to the *line\_no* of that fixation. In addition, based on the region file, the *gaze\_region\_no* of the data point is also updated as the corresponding word region index. After these, the function returns the data frame NewStampDF with updated columns of *line\_no*, *gaze\_region\_no*, and *Fix\_Sac*. For time-stamped data of both eyes, the function first classifies the EM data of the left eye, and then the right eye.

1. read\_cal\_TimeStamp: to read ascii data file of a particular subject, extract the time stamped data in it, classify the time-stamped data into different text lines and word regions, and return a data frame StampDF storing classified time-stamped data. It has the same arguments as those in cal\_TimeStamp.
2. cal\_write\_TimeStamp: to read time-stamped data report of a subject, classify the EM data into different lines of text and word regions, and update the result data frame into the same time-stamped data report. It calls cal\_TimeStamp and write\_TimeStamp\_Report to work. It has the same arguments as those in cal\_TimeStamp.
3. cal\_write\_TimeStamp\_b: the batch version of cal\_write\_TimeStamp. It reads all time-stamped data reports, classify the EM data into different lines of text and word regions, and update the result data frames into the same time-stamped data reports. It has no argument *subjID*, and the other arguments are the same as those in cal\_write\_TimeStamp.
4. read\_cal\_write\_TimeStamp: to read ascii data file of a particular subject, extract the time stamped data in it, classify the time-stamped data into different text lines and word regions, and write the classified time-stamped data into a csv file as the time-stamped report. It calls read\_cal\_TimeStamp and write\_TimeStamp\_Report to work. It has the same arguments as those in read\_cal\_TimeStamp.
5. read\_cal\_write\_TimeStamp\_b: the batch version of read\_cal\_write\_TimeStamp. It reads the ascii data file of all subjects in a particular directory, extracts the time stamped data in them, classify the time-stamped data into different text lines and word regions, and write the classified time-stamped data into csv files as the time-stamped reports. It has no argument *subjID*, and the other arguments are the same as those in read\_cal\_write\_TimeStamp.

**Examples:**

1. SacDF, crlSac, FixDF, crlFix = cal\_crlSacFix(‘./example’, ‘1950138’, [‘story01.region.csv’, ‘story02.region.csv’, ‘story03.region.csv’], ‘RP’): read the saccade and fixation reports (1950138\_Sac.csv and 1940138\_Fix.csv) in the folder ‘./example/1950138’, classify those saccades and fixations into different lines, and return data frames respectively storing classified and cross-line saccades and fixations. *regfileNameList* can be empty; in this case, all \*.region.csv files in the folder are recognized as region files.
2. write\_Sac\_crlSac(‘./example’, ‘1950138’, SacDF, crlSac): generate two csv reports for classified saccades (1950138\_Sac.csv) and cross-line saccades (1950138\_crlSac.csv) of subject 1950138 in the folder ‘./example/1950138’.
3. write\_Fix\_crlFix(‘./example’, ‘1950138’, FixDF, crlFix): generate two csv reports for classified fixations (1950138\_Fix.csv) and cross-line fixations (1950138\_crlFix.csv) of subject 1950138 in the folder ‘./example/1950138’.
4. cal\_write\_SacFix\_crlSacFix(‘./example’, ‘1950138’, [], ‘RP’): read the saccade and fixation reports of the subject 1950138 in the folder ‘./example/1950138’, extract saccades and fixations therein, classify those saccades and fixations, and generate csv reports storing classified and cross-line saccades and fixations. It may also generate a log file storing the information of some questionable data. *regfileNameList* is empty in this case, all \*.region.csv files in the folder are recognized as region files.
5. cal\_write\_SacFix\_crlSacFix\_b(‘./example’, [], ‘RP’): automatically read the saccade and fixation reports of the three subjects in the folder ‘./example’, extract saccades and fixations therein, classify those saccades and fixations, and generate csv reports storing classified and cross-line saccades and fixations. It may also generate a log file storing the information of some questionable data.
6. SacDF, crlSac, FixDF, crlFix = read\_cal\_SRRasc(‘./example’, ‘1950138’, [‘story01.region.csv’, ‘story02.region.csv’, ‘story03.region.csv’], ‘RP’): read 1950138.asc in the folder ‘./example/1950138’ to extract saccades and fixations, classify them, and return four data frames storing classified and cross-line saccades and fixations. *regfileNameList* can be empty; in this case, all \*.region.csv files in the folder are recognized as region files.
7. read\_cal\_write\_SRRasc(‘./example’, ‘1950138’, [], ‘RP’): read 1950138.asc in the folder ‘./example/1950138’, extract saccades and fixations therein, classify them into different lines of the texts, and generate csv reports for classified and cross-line saccades and fixations.A log file is also generated to store information about questionable data. *regfileNameList* is empty in this case, all \*.region.csv files in the folder are recognized as region files.
8. read\_cal\_write\_SRRasc\_b(‘./example’, [], ‘RP’): automatically read all ascii data files of the three subjects in the folder ‘./example’, extract saccades and fixations therein, classify them into different lines of the texts, and generate csv reports for classified and cross-line saccades and fixations of different subjects.A log file is also generated to store information about questionable data.
9. NewStampDF = cal\_TimeStamp(‘./example’, ‘1950138’, [‘story01.region.csv’, ‘story02.region.csv’, ‘story03.region.csv’], ‘RP’, ‘Fix\_Sac’): read the time-stamped data file ‘1950138\_Stamp.csv’ in the folder ‘./example/1950138’, region file, and relevant fixation and saccade reports (\*\_Fix.csv and \*\_Sac.csv, \* is subject ID), classifies the time-stamped data of all trials into corresponding text lines and word regions, and returns the data frame NewStampDF recording classified time-stamped data. *regfileNameList* can be empty; in this case, all \*.region.csv files in the folder are recognized as region files.
10. cal\_write\_TimeStamp(‘./example’, ‘1950138’, [], ‘RP’, ‘Fix\_Sac’): read the time-stamped data file ‘1950138\_Stamp.csv’ in the folder ‘./example/1950138’, region file, and relevant fixation and saccade reports (\*\_Fix.csv and \*\_Sac.csv, \* is subject ID, fixations and saccades therein are already classified), classifies the time-stamped data of all trials into corresponding text lines and word regions, and write the time-stamped report into a csv file.
11. NewStampDF = read\_cal\_TimeStamp(‘./example’, ‘1950138’, [], ‘RP’, ‘Fix\_Sac’): read 1950138.asc in the folder ‘./example/1950138’, the region file, and the relevant fixation and saccade reports (\*\_Fix.csv and \*\_Sac.csv, \* is subject ID), classifies the time-stamped data into corresponding text lines and word regions, and return the data as a data frame.
12. read\_cal\_write\_TimeStamp(‘./example’, ‘1950138’, [], ‘RP’, ‘Fix\_Sac’): read 1950138.asc in the folder ‘./example/1950138’, for each trial, extract time-stamped data in it, classify them into different text lines and word regions, and generate a csv report for classified time-stamped data.
13. read\_cal\_write\_TimeStamp\_b(‘./example’, [], ‘RP’, ‘Fix\_Sac’): read all ascii files in the folder ‘./example’, for each trial, extract time-stamped data in it, classify them into different text lines and word regions, and generate csv reports for classified time-stamped data of different subjects.*regfileNameList* is empty in this example, all \*.region.csv files in the folder are recognized as region files.
14. **Functions for visualizing saccades and fixations on bitmaps**

To better visualize the experimental data, we provide two sets of functions in this section to draw saccades and fixations upon actual bitmaps that subjects read in the experiment. The first set of functions generate bitmaps with saccades and fixations. The second set of functions generate animations that dynamically illustrate fixations or time-stamped data along with oral reading sounds. All the functions are for demonstration purpose.

* 1. ***Functions for showing saccades and fixations on bitmaps***

To use these functions, users need to put the bitmaps, region files, and csv reports (classified saccades and fixations) generated by the functions in the other sections into the same folder. As for the reports containing saccades and fixations of both eyes, the functions first show saccades and fixations of the left eye, and then the right eye. In the figures created by the functions, the single/multi-line text is shown as background. Saccades are denoted by solid lines, and fixations by solid circles with radiuses scaled to fixation durations. A leftward moving saccade is shown in blue, and a rightward moving saccade in red. A left eye fixation is shown in a green circle, and a right eye fixation in a red circle. The functions have the options of showing the actual durations of fixations near the circles in blue texts and showing all saccades and fixations or only those before the end of reading.

**User functions:**

1. draw\_SacFix: todisplay classified and/or cross-line saccades and fixations of a particular subject on corresponding bitmaps.

Arguments:

* *direct*: directory of the bitmaps, region files, and csv reports of a subject, the generated figures are also stored there.
* *subjID*: subject ID.
* *regfileNameList*: list of names of region files. If it is empty, the function automatically reads all \*.region.csv files in ‘direct’ as region files.
* *bitmapNameList*: list of names of bitmaps used in each trial of the experiment. Names of region files in *regfileNameList* and names of bitmaps in *bitmapNameList* are sorted by the trials of the experiment. If *PNGopt* (see below) is 0, *bitmapNameList* can be empty; in this case, the function automatically reads all \*.png files in *direct* as bitmap files.
* *drawType*: type of drawing. If it is ‘ALL’, the function draws two figures (\*\_FixSac\_trial#.png and \*\_crlFixSac\_trial#.png, ‘\*’ is subject ID, and ‘#’ is trial ID), one showing all the saccades and fixations during reading (the fixations and saccades occurring after the end of reading are not shown), and the other showing all the identified cross-line saccades and fixations; if it is ‘Sac’, the function draws two figures (\*\_Sac\_trial#.png and \*\_Fix\_trial#.png) respectively showing all the saccades during reading and all the identified cross-line saccades; and if it is ‘Fix’, the function draws two figures (\*\_Fix\_trial#.png and \*\_crlFix\_trial#.png) respectively showing all the fixations during reading and all the identified cross-line fixations.
* *max\_FixRadius*: maximum radius of fixation circles in pixel; default is 30.
* *drawFinal*: whether or not to draw saccades and fixations after the end of reading; default is False (only drawing the saccades and fixations during normal reading).
* *showFixDur*: whether or not to show actual durations of fixations; default is False (not showing durations).
* *PNGopt*: options of using PNG bitmap. If it is 0, saccades and fixations are drawn upon existing bitmaps; if it is 1, the function regenerates bitmaps based on region files, and then, draw saccades and fixations on these newly-created bitmaps. Due to different monitor setting, regenerated bitmaps may be slightly different from the ones used in the experiment.

1. draw\_SacFix\_b: the batch version of draw\_SacFix. It calls draw\_SacFix to draw relevant figures for the data files of each subject. It first collects the relevant files exist in specified folder; if there is no such file, it throws a warning message and stops. It has no argument *subjID*, and the other arguments are identical to those of draw\_SacFix.

**Examples:**

1. draw\_SacFix(‘./example’, ‘1950138’, [‘story01.region.csv’, ‘story02.region.csv’, ‘story03.region.csv’], [‘story01.png’, ‘story02.png’, ‘story03.png’], ‘ALL’): based on the csv reports of subject 1950138 (1950138\_Sac.csv, 1950138\_crlSac.csv, 1950138\_Fix.csv, 1950138\_crlFix.csv), draw figures showing all saccades and fixations during reading (1950138\_FixSac\_trial0.png, 1950138\_FixSac\_trial1.png, 1950138\_FixSac\_trial2.png) and figures showing only cross-line saccades and fixations (1950138\_crlFixSac\_trial0.png, 1950138\_crlFixSac\_trial1.png, 1950138\_crlFixSac\_trial2.png). To properly use the function, users need to move the bitmaps (story01.png, story02.png, story03.png) and region files (story01.region.csv, story02.region.csv, story03.region.sv) in the folder ‘./bitmap\_regfile’ to the folder ‘./example’. *regfileNameList* can be empty; in this case, all \*.region.csv files in the folder are recognized as region files. If *PNGopt* is 0 (default), *bitmapNameList* can be empty; in this case, all \*.png files in the folder are recognized as bitmap files.
2. draw\_SacFix\_b(‘./example’, [], [], ‘ALL’):for the csv reports of all three subjects,generate relevant figures showing classified and cross-line saccades and fixations.
   1. ***Functions for animation of fixations or time-stamped data on bitmaps***

To use these functions, users need to put the bitmaps, sound files (in the folder ‘sounds’), region files, and csv reports (raw or classified fixations or time-stamped data) generated by the functions in the other sections into the same folder. Due to restrictions of the turtle package in Python, bitmaps as animation background have to be gif files. We provide a helper function, changePNG2GIF, to transform png bitmaps into gif bitmaps (also stored in the folder ‘bitmap\_regfile’). The sound files are wav files, whose filenames contain the subject ID and trail index (starting from 1). **Note that the animation function requires winsound package, which is available only in Python distributions in Windows, so the animation function cannot be properly run in Python distributions in Linux.**

**User functions:**

1. changePNG2GIF: to transform png bitmaps into gif bitmaps for the purpose of animation. The gif bitmaps have the same names of the png bitmaps. It has one argument, *direct*, which shows the directory of the png bitmaps. The generated gif bitmaps are also stored here.
2. animate: to generate an animation showing fixations and oral reading in a particular trial of a particular subject. It first checks whether the relevant files (including the sound files, gif bitmaps, csv fixation reports) exist in specified folder; if not, it throws a warning message and stops. Then, it generates a screen to display the bitmaps and the fixations along reading. The title of the screen shows the subject ID, trial number, and which eye’s fixations are shown. Fixations are denoted by green circles, whose radiuses are scaled to the durations of fixations. As for the reports containing fixations of both eyes, fixations of the left eye are shown in green circles, and those of the right eye are shown in red circles. Both types of fixations are updated simultaneously during the animation. Along with the animation, it plays the sound files in the background.

Arguments:

* *direct*: directory of the png bitmaps.
* *subjID*: subject ID.
* *trialID*: trial ID (starting from 0, corresponding to trial 1).

Algorithm:

animate first checks whether required files are in the specified folder *direct*; if not, it throws a warning and stops. If all required files are there, based on *trialID*, it extracts the fixations in that particular trial from the csv file, and checks whether the data file contains fixations of a single eye or both eyes. Then, it creates a turtle screen using the bitmap as background. If the data file contains fixations of a single eye, it generates one turtle player (green circle for left eye fixations and red circle for right eye fixations); if the data file has fixations of both eyes, it generates two turtle players respectively for fixations of each eye. After that, it listens to key input to start or stop the animation. If the key ‘s’ is pressed, based on winsound.PlaySound(), the function plays the sound file in background, and displays each fixation in the extracted data file in time. The logic of displaying fixations is as follows. First, it calculates the time difference between the starting times of two consecutive fixations (or between the starting of the sound file and the starting time of the first fixation), uses time.sleep() to wait for that amount of time, and then, displays the corresponding fixation. If the key ‘e’ is pressed, the function stops the animation. If the key ‘s’ is pressed again, the function restarts the animation. If one clicks on the animation screen, the function terminates the animation and quits.

The function adds another timer to compensate the time lags due to imprecision of time.sleep() function and running times of Python codes for moving and displaying turtle players, making sure a largely aligned display of fixations along with the sound. However, for a long time paragraph reading, the accumulated errors may still cause mismatch between the sounds and fixations at the later part of the animation. In addition, due to restrictions of the turtle and winsound packages, the animation and the sound played cannot be paused or resumed, and the whole animation cannot be stored as a separate file.

1. animate\_TimeStamp: to generate an animation showing time-stamped data and oral reading in a particular trial of a particular subject. It first checks whether the relevant files (including the sound files, gif bitmaps, csv time-stamped reports) exist in specified folder; if not, it throws a warning message and stops. Then, it generates a screen to display the bitmaps and the time-stamped data along reading. The title of the screen shows the subject ID, trial number, and which eye’s time-stamped data are shown. Time-stamped data are denoted by green circles with fixed radiuses (3 pixels). As for the reports containing time-stamped data of both eyes, those of the left eye are shown in green circles, and those of the right eye are shown in red circles. Both types of time-stamped data are updated simultaneously during the animation. Along with the animation, it plays the sound files in the background. This function has the same arguments as those in animate. **Note that if the sampling rate of time-stamped data is much higher than the refresh rate of the computer monitor, this function cannot accurately animate the time-stamped data.**

**Examples:**

1. changePNG2GIF(‘./example’): transform png bitmaps in the folder ‘./example’ into gif bitmaps.
2. animate(‘./example’, ‘1950138’, 0): generate an animation showing the fixations and oral reading of subject 1950138 in the first trial of the experiment. The fixation data are stored in 1950138\_Fix.csv. Once it is called, a screen is generated, whose background is the story bitmap of that trial. Due to the resolution of the monitor, the whole screen may not cover the whole bitmap. One can manually adjust the screen size and scroll down or up the screen, without affecting displayed fixations. One can start or restart the animation by pressing the key ‘s’, stop the animation by pressing the key ‘e’, and quit the animation by clicking on the animation screen.
3. animate\_TimeStamp(‘./example’, ‘1950138’, 0): generate an animation showing the time-stamped data and oral reading of subject 1950138 in the first trial of the experiment. The time-stamped data are stored in 1950138\_Stamp.csv. Once it is called, a screen is generated, whose background is the story bitmap of that trial. Due to the resolution of the monitor, the whole screen may not cover the whole bitmap. One can manually adjust the screen size and scroll down or up the screen, without affecting displayed time-stamped data. One can start or restart the animation by pressing the key ‘s’, stop the animation by pressing the key ‘e’, and quit the animation by clicking on the animation screen.
4. **Functions for calculating EM measures**

Based on region-wised summaries, we provide functions in this section to calculate widely-adopted EM measures. The functions require the region files storing position information of words in the single/multi-line text, and the csv reports recording classified fixations and saccades in different text lines and word regions. All these files can be generated by the functions in the other sections.

The function (cal\_write\_EM) calculate three groups of EM measures, including eight first-pass fixation and regression measures, four regression path measures, and two second-pass fixation measures. Apart from these region-based measures, it also calculates four whole-text-based measures (definitions of all these measures are shown below in Outputs). All measures are calculated by appropriate helper functions, and their values are stored in the corresponding columns of the csv reports.

**User functions:**

1. cal\_write\_EM: to calculate the EM measures based on the saccade and fixation reports of a subject, and store the results in csv files. If some reports are missing in the specified folder, it throws a warning message and stops. As for the reports containing saccades and fixations of two eyes, the function first calculates the EM measures based on the left eye data and stores the result in the csv files, and then the right eye.

Arguments:

* *direct*: directory of the saccade and fixation reports (\*\_Sac.csv and \*\_Fix.csv, ‘\*’ is the subject ID) of a subject and relevant region files, generated csv files are also stored there.
* *subjID*: subject ID.
* *regfileNameList*: list of names of the region files relative to the data files. If it is empty, it reads all \*.region.csv files in ‘direct’ as region files.
* *addCharSp*: number of single character space added to capture overshoot fixations; default is 1.

Outputs:

Based on the number of trials,cal\_write\_EM generates a number of csv files (\*\_EM\_trial#.csv, ‘\*’ is the subject ID and ‘#’ is the trail index starting at 0) respectively storing the values of the EM measures based on the csv reports in each trial. If the saccade and fixation reports contain data of both eyes, it generates two sets of csv files: \*\_EM\_trial#\_L.csv records the values of the EM measures based on the left eye data, and \*\_EM\_trial#\_R.csv records the values of the EM measures based on the right eye data.

\*\_EM\_trial#.csv, \*\_EM\_trial#\_L.csv, and \*\_EM\_trial#\_R.csv have 32 columns. Some columns record the general information of the reading data (extracted from the saccade and fixation reports), some record word information of the single- or multi-line text (extracted from the region files), and others record the values of EM measures (calculated by the relevant helper functions):

* + *subj*, *trial\_id*, *trial\_type*, *trialstart*, *trialend*, *tdur*, *recstart*, *recend*, *blinks*, *eye*: same as the corresponding columns in \*\_Sac.csv or \*\_Fix.csv.
  + *tffixos*: total offset of the first-pass fixation of each word from the beginning of the first sentence of the text (whole-text EM measure).
  + *tffixurt*: total duration of the first pass fixation of each word in the text (whole-text EM measure).
  + *tffixcnt*: total number of valid fixations in the trial (whole-text EM measure).
  + *tregrcnt*: total number of *regressive saccades* (a saccade is regressive if it starts at one word region in the text and ends at an earlier word region) in the trial (whole-text EM measure).
  + *region*: same as the column *WordID* in the region file.
  + *reglen*: same as the column *length* in the region file.
  + *word*: same as the column *Word* in the region file.
  + *line\_no*: same as the column *line\_no* in the region file.
  + x1\_pos, x2\_pos: same as the columns *x1\_pos* and *x2\_pos* in the region file.
  + *mod\_x1*, *mod\_x2*: for words in the middle of a line, *mod\_x1* and *mod\_x2* are the same as *x1\_pos* and *x2\_pos*; for words at the beginning of a line, *mod\_x1* = *x1\_pos* – *addCharSp* (see Arguments); for words at the end of a line, *mod\_x2* = *x2\_pos* + *addCharSp*. *mod\_x1* and *mod\_x2* are used to capture overshoot fixations.
  + *fpurt*: first-pass fixation time. It is the sum of the durations of one or more first-pass fixations falling into the word region. By default, only fixations of 50 ms or longer are recorded; shorter fixations are subject to lumping. If there is no first-pass fixation in the word region, *fpurt* = NaN (missing value) (first-pass fixation measure).
  + *fpcount*: number of first-pass fixations falling into the word region. If there is no first-pass fixation in the word region, *fpcount* = NaN (first-pass fixation measure).
  + *fpregres*: whether there is a first-pass regression starting from the current word region; if so, *fpregres* = 1, otherwise, *fpregres* = 0. If there is no first-pass fixation in the word region, *fpregres* = NaN (first-pass regression measure).
  + *fpregreg*: word region where the first-pass regression ends. If there is no first-pass regression (*fpregres* = 0), *fpregreg* = 0. If there is no first-pass fixation in the word region, *fpregreg* = NaN (first-pass regression measure).
  + *fpregchr*: offset in characters in the word region where the first-pass regression ends. If there is no first-pass regression (*fpregres* = 0), *fpregchr* is set to a value large enough to be out of boundaries of any possible string (here, it is set to the total number of characters of the whole text). If there is no first-pass fixation in the word region, *fpregchr* = NaN (first-pass regression measure).
  + *ffos*: offset in characters of the first first-pass fixation in the word region from the first character of the region. If there is no first-pass fixation in the word region, *ffos* = NaN (first-pass fixation measure).
  + *ffixurt*: duration of the first first-pass fixation in the word region. If there is no first-pass fixation in the word region, *ffixurt* = NaN (first-pass fixation measure).
  + *spilover*: duration of the first fixation falling beyond (either left or right) the word region. If there is no first-pass fixation in the word region, *spilover* = NaN (first-pass fixation measure).
  + *rpurt*: sum of durations of all fixations in the regression path. A regression path starts from the first fixation falling into the current word region and ends at the first fixation falling into the immediately next word region. If there is a first-pass regression (*fpregres* = 1), the regression path includes the fixations in the current region and those outside the current word region but falling into only the word regions before the current region. If there is no first-pass regression (*fpregres* = 0), *rpurt* = *fpurt*. If there is no first-pass fixation in the word region, *rpurt* = NaN (regression path measure).
  + *rpcount*: number of fixations in the regression path. If there is no first-pass fixation in the word region, *rpcount* = NaN (regression path measure).
  + *rpregreg*: the smallest index of the word region visited by the regression path. If there is no regression path (*fpregres* = 0), *rpregreg* = 0. If there is no first-pass fixation in the word region, *rpregreg* = NaN (regression path measure).
  + *rpregchr*: offset in characters in the smallest word region visited by the regression path. If there is no first-pass regression (*fpregres* = NaN), *rpregchr* is set to a value large enough to be out of boundaries of any possible string (here, it is set as the total number of characters of the whole text). If there is no first-pass fixation in the word region, *rpregreg* = NaN (regression path measure).
  + *spurt*: second-pass fixation time. It is the sum of durations of all fixations falling again into the current word region after the first-pass reading. If there is no second-pass fixation, *spurt* = NaN (second-pass fixation measure).
  + *spcount*: number of second-pass fixations. If there is no second-pass fixation, *spcount* = NA (second-pass fixation measure).

Algorithms:

cal\_write\_EM first checks whether the relevant csv reports (\*\_Sac.csv, \*\_Fix.csv, and the region files) exist in the specified folder; if not, it throws a warning message and stops. If the files exist, it reads them, assigns fixations to particular words in the text based on the position information of the words in the region files. Then, for each region, it calls relevant helper functions to calculate the first-pass fixation and regression measures, regression path measures, and second-pass fixation measures. After calculation, it sums up the whole-text-based measures, and stores the results into csv files.

Compared with single-line text, there are several difficulties in calculating the EM measures in multi-line text reading. For example, to check which word region a fixation falls into, we need to refer to *x\_pos* of the fixation, *mod\_x1* and *mod\_x2* of the word region, and *line\_no* in which the fixation falls. For words at the beginning or end of a line, we need to use slightly bigger word region to capture overshoot fixations. In addition, physical position of a word region is insufficient to indicate whether this region is before or after the target word region. We have to use *region*, a numerical index of each word in the text, to clarify regression probability and regression path.

1. cal\_write\_EM\_b: the batch version of cal\_write\_EM. For each set of saccade and fixation reports of a subject, it calls cal\_write\_EM to work. It has no argument *subjID*, and the other arguments are the same as those in cal\_write\_EM.

**Examples:**

1. cal\_write\_EM(‘./example’, ‘1950138’, [‘story01.region.csv’, ‘story02.region.csv’, ‘story03.region.csv’]): calculate the EM measures of subject 1950138 based on 1950138\_Fix.csv and 1950138\_Sac.csv, and store the results in 1950138\_EM\_trial0\_L.csv, 1950138\_EM\_trial1\_L.csv, 1950138\_EM\_trial2\_L.csv, as well as 1950138\_EM\_trial0\_R.csv, 1950138\_EM\_trial1\_R.csv, 1950138\_EM\_trial2\_R.csv. *regfileNameList* can be empty; in this case, all \*.region.csv files in the folder are recognized as region files.
2. cal\_write\_EM\_b(‘./example’, []): calculate the EM measures based on the csv reports of many subjects in the folder ‘./example’, and store the results in corresponding csv files.
3. **Summary and future directions**

Our package provides useful functions covering the major steps in designing and analyzing an EM reading experiment, including generating and displaying stimuli, collecting EM data, and calculating and storing the EM measures for subsequent analysis. Many of the functions are flexible and easy to extend to other EM experiments.

The package designs flexible ways to generate bitmaps showing single- or multi-line text and region files recording position information of words in the bitmaps. Apart from normal paragraph, the package can also generate gridded texts used in other reading tasks, such as rapid naming task. Apart from English characters, the package can also display characters in many Latin family languages, including French, German, Dutch, Spanish, Italian, and Greek, as well as other popular languages in the World, such as Chinese, Korean, and Japanese. This characteristics enriches the applications of the package, and makes it suitable for multi-language studies.

The package implements a series of efficient algorithms to lump short fixations, identify cross-line saccades and fixations, and classify saccades and fixations into different text lines. Identifying cross-line saccades and fixations are the most challenging task in analyzing EM data of multi-line text reading. To our knowledge, our distance-based algorithms for classifying cross-line saccades and fixations have not been used in previous data analysis of EM reading experiments. These algorithms recapitulate an online reading. They appear to be simpler and more efficient than some offline algorithms, which either rely primarily on detecting specific trajectories of saccades resembling ideal regressions from the end (or the beginning) of one line of text to the beginning (or the end) of another line, or use some global optimization method to fit saccades or fixations into different lines. In reality, individual reading patterns are quite distinct, so the ideal saccade trajectories may not exist. In addition, the fitted lines of fixations or saccades in each text line may not be well aligned with the corresponding text lines or with each other, so the global optimization method may not be able to classify these Ems into appropriate lines. By contrast, our algorithms make use of realistic assumptions (during cross-line EM, the change in x position of the relevant saccade or concatenate fixations tend to be huge) to detect cross-line EM, and these algorithms have been proved to be reliable for capturing not only forward and backward cross-line EMs but also a mixture of both types of cross-line EMs during paragraph reading.

Apart from saccades and fixations, the package also provides useful functions to process time-stamped raw EM data. This is useful for reading experiments that process the temporal relation between EM and other cognitive behavior such as oral reading. For example, classified time-stamped data allow careful calculation of the latency between eye movements along the words and oral reading of those words. Such latency could be used as a useful indicator of reading fluency. In addition, the package provides a number of useful functions to illustrate the reading patterns, including not only still figures showing locations of saccades and fixations, but also animations vividly illustrating the online reading progress.

Finally, the package calculates so far the most complete set of EM measures covering first-pass fixation, regression, and second-pass fixation. Many of these measures have been widely used in previous EM literature. This rich set of measures allows researchers to systematically analyze individual reading patterns at various stages of lexical or sentence processing, including the very early stage, the early stage, and the late stage.

All the above features make the package necessary and very useful to the current reading research based on EMs. Nonetheless, some aspects in the package still deserve improvement.

On the one hand, there are several theoretical issues in the current package that need to be resolved. For example, due to individual difference, the saccades and fixations of some subjects still cannot be accurately classified by the current functions. With more experimental studies in multi-line text reading, we can further update the available algorithms to automatically and efficiently capture various reading patterns and classify relevant saccades and fixations into different text lines. In addition, when a cross-line reading is achieved not by a single jump, but by a group of saccades and fixations, the current function estimates the cross-line fixation or saccade as the fixation or saccade that induces the greatest change in either horizontal or vertical position. This criterion may not always hold, and we have not clearly understood what cognitive process is behind such slow cross-line reading. This phenomenon does not occur in line-by-line presentation of multi-line text. It may serve as an entry point to better understand reading behavior and its cognitive bases.

On the other hand, there are some practical issues that can be improved by means of advanced techniques. For example, the current functions in the package only generate bitmaps displaying major world’s languages. We can further modify the functions to display texts of other academically interesting languages, such as Arabic or Hebrew, and to show other types of print style, such as displaying texts from right to left, as in Arabic, or displaying texts vertically, as in traditional ways of Chinese writing. Following the same logic, we can further extend the functions to capture cross-line or cross-column saccades or fixations during reading texts displayed in such formats. In addition, the current functions rely on special information to classify saccades and fixations into different text lines. Apart from the special information, online reading is also rich in temporal information, which can also be used for the purpose of classification. On this aspect, a unification of both special and temporal cues may lead to more accurate classification of saccades and fixations. Furthermore, considering individual differences in reading patterns, the same set of parameter values may not be useful to different individuals. Machine learning techniques may be adopted to implement automatic and heuristic classification. Finally, the total number of words in multi-line text (usually over 100 and more) is much bigger than that of single-line text (usually less than 20). Accordingly, processing time-stamped data and calculating EM measures in multi-line text are much slower than those in single-line text. For the current package, it takes a few seconds to extract and classify one subject’s data, and one or two minutes to calculate EM measures of one subject, but much longer time to handle many subjects’ data. There is a necessity to design more efficient algorithms for such calculation.

All the above-mentioned issues pave the future directions of the current package.