## CONTENTS

1 Preface ...................................................... 3

2 Getting started .................................................. 7
   2.1 File opening ........................................... 7
   2.2 Trace scaling ........................................ 8
   2.3 Navigate within a file ................................ 10
   2.4 Analysis of individual events ............................. 10
   2.5 Analysis on selected traces .............................. 13
   2.6 Average calculation .................................. 14
   2.7 Fitting functions to data ............................... 14

3 The Python shell ................................................ 19
   3.1 Why use Python? ........................................ 19
   3.2 Before you start ....................................... 19
   3.3 The Python shell ...................................... 19
   3.4 Accessing data from the Python shell .................... 20
   3.5 Using NumPy with Stimfit ............................... 20
   3.6 Control Stimfit from the Python shell .................... 22
   3.7 Some recipes for commonly requested features .......... 25

4 Latency measurements ........................................... 27
   4.1 Measurement of synaptic delay .......................... 27
   4.2 Trace alignment ....................................... 27
   4.3 Setting the latency cursors ............................. 29

5 Event extraction by template matching ........................ 31
   5.1 Introduction ........................................... 31
   5.2 A practical guide to event detection ..................... 31

6 Bibliography .................................................. 37

7 GNU/Linux Build Guide ......................................... 39
   7.1 Building the Python module only ....................... 39
   7.2 Building Stimfit ...................................... 40

8 OS X Build Guide ................................................ 43
   8.1 Building Stimfit ...................................... 43

9 Windows Build Guide ............................................ 45
   9.1 Prerequisites ........................................ 45
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>Building Stimfit</td>
<td>47</td>
</tr>
<tr>
<td>10</td>
<td><strong>The Stimfit book of spells</strong></td>
<td>49</td>
</tr>
<tr>
<td>10.1</td>
<td>Resistance Calculation</td>
<td>49</td>
</tr>
<tr>
<td>10.2</td>
<td>Running mean</td>
<td>51</td>
</tr>
<tr>
<td>10.3</td>
<td>Calculations on selected traces</td>
<td>53</td>
</tr>
<tr>
<td>10.4</td>
<td>Cutting traces</td>
<td>56</td>
</tr>
<tr>
<td>10.5</td>
<td>Event counting</td>
<td>58</td>
</tr>
<tr>
<td>10.6</td>
<td>Object oriented programming with Stimfit</td>
<td>61</td>
</tr>
<tr>
<td>10.7</td>
<td>Calculating latencies</td>
<td>66</td>
</tr>
<tr>
<td>11</td>
<td><strong>stf</strong></td>
<td>73</td>
</tr>
<tr>
<td>12</td>
<td>References</td>
<td>87</td>
</tr>
<tr>
<td>13</td>
<td><strong>The stfio module</strong></td>
<td>89</td>
</tr>
<tr>
<td>13.1</td>
<td>Reading files</td>
<td>89</td>
</tr>
<tr>
<td>13.2</td>
<td>Constructing Recordings from scratch</td>
<td>90</td>
</tr>
<tr>
<td>13.3</td>
<td>Writing files</td>
<td>91</td>
</tr>
<tr>
<td>14</td>
<td>Links</td>
<td>93</td>
</tr>
<tr>
<td>14.1</td>
<td>Free electrophysiology software</td>
<td>93</td>
</tr>
<tr>
<td>14.2</td>
<td>Electrophysiology software sites</td>
<td>93</td>
</tr>
<tr>
<td>15</td>
<td>Indices and tables</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td><strong>Python Module Index</strong></td>
<td>99</td>
</tr>
</tbody>
</table>
Contents:

**Author** Christoph Schmidt-Hieber (christse at gmx.de)

**Date** March 21, 2014

Contents:
Stimfit was originally written by Peter Jonas, University of Freiburg, in the early 1990s. It was primarily designed to analyze the kinetics of evoked excitatory postsynaptic potentials (EPSCs; Jonas et al., 1993\(^1\)). The name Stimfit was chosen because the program allowed to fit exponential functions to the decay of EPSCs evoked by extracellular stimulation. The program was written in Borland Pascal, running under DOS and entirely controlled using keyboards shortcuts. The user interface was similar to a digital oscilloscope, with vertical cursors defining measurement windows for baseline calculation, peak detection and curve fitting. This allowed to analyze data with surprising efficiency once the keyboard shortcuts were mastered. However, the Borland Pascal compiler imposed some significant restrictions which became apparent with increasing data size and computing power: for instance, arrays were not allowed to be longer than $10^4$ elements, and faster processors had to be artificially slowed down to avoid runtime errors.

When I converted the original Pascal program to C/C++, I rewrote the code almost entirely from scratch. Only the algorithms to calculate latencies, rise times, half durations and slopes are direct translations of the original Pascal code. By contrast, I tried to preserve the user interface as far as possible. Therefore, the program only poorly adheres to common conventions for graphical user interfaces: for instance, clicking the right mouse button will usually set a cursor position rather than popping up a context menu.

A number of people have contributed to the program: First, I would like to thank Peter Jonas for the original Stimfit code. Josef Bischofberger has added some functions to the DOS version which I have adopted. Bill Anderson has made helpful suggestions concerning the user interface and provided some very large files that have been recorded with his free program WinLTP. A large amount of helpful comments and bug reports were filed by Emmanuel Eggermann and Daniel Boischer. The Levenberg-Marquardt-algorithm used for curve fitting was implemented by Manolis Lourakis.

---

Figure 1.1: The current platform-independent Stimfit.
Figure 1.2: **Fig. 1:** Above the current platform-independent Stimfit. Below, the original Stimfit for MS-DOS.
2.1 File opening

This tutorial will cover the basic program functionality from opening a file to fitting functions to data.

• Download Stimfit from here and install it on your computer.
• Download a sample file from here.
• Open the data file: you can either double-click it from an Explorer Window, or you can start Stimfit and choose “File”->”Open…” from the menu.
• The file will be opened in a new child window, and the first trace will be displayed.

2.1.1 Opening multiple files

Stimfit supports drag-and-drop file opening. Select a file or a list of files in your file browser and move it to the main Stimfit program window. A dialog window will appear (see below) asking you whether you want to merge all your selected files in a single file.
2.2 Trace scaling

- If you just see scale bars, but no trace is displayed, press F or click the corresponding button (Fig. 2). This will fit the first trace of the active channel (plotted in black) to the screen.

![Fig. 2: Fit traces to the window size.](image)

- If you prefer coordinates to scale bars, you can check “View”->”Scale bars” in the menu (Fig. 3).

- Fit the inactive channel (plotted in red) to the screen as well. Press 3. The buttons labeled 1 and 2 should now both be highlighted (Fig. 4). That means that any changes to the scaling will now be applied to both channels simultaneously.

![Fig. 3: Show coordinates rather than scale bars.](image)

- Press F again. The inactive channel (red trace) will now be fitted to the screen as well. If you want to scale channels individually, press either 1 or 2.

![Fig. 4: Scaling applies to both channels.](image)
• Enlarge the vertical scale: Press +. Depending on which channel(s) you selected, the vertical scale will be enlarged by a factor of 10%. Shrink the scale back to its original value by pressing -.

• Enlarge the time scale: Press Ctrl and + simultaneously. The time scale will be enlarged for both channels, regardless of which channel you have chosen, because Stimfit assumes that both channels have been sampled at the same time and frequency.

• Shrink the time scale back to its original value by pressing Ctrl and - simultaneously.

• Shift the trace by pressing Ctrl and one of the cursor (arrow) keys simultaneously.

• You can zoom into parts of the trace using a zoom window: Press Z. The zoom button (showing a magnifying glass) will be highlighted (Fig. 5).

• Drag a window over the region of interest holding down the left mouse button. Release the left mouse button once you are done. When you right-click on the window, a menu will pop up showing different zoom options. Select “Expand zoom window horizontally & vertically” (Fig. 6).

Figure 2.4: Fig. 5: Setting the mouse cursor to draw zoom windows.

Figure 2.5: Fig. 6: Magnifying a region of interest.
• If you cannot see any trace because you zoomed in or out too much, press F to fit the trace to the screen again.

2.3 Navigate within a file

• You can toggle through traces using the left and right keys (without pressing Ctrl at the same time). The current trace number will be displayed in the drop-down box labeled “Trace ... of ...”. You can directly select a trace from this box as well (Fig. 7).

• All measurements will be performed on the active channel plotted in black. You can swap channels by either selecting “View”->“Swap channels” from the menu, or setting the channels in the drop-down boxes (Fig. 8).

2.4 Analysis of individual events

An “event” can be anything from an EPSC to an action potential. In this case, we will analyze a large spontaneous EPSC in trace no. 12 of the second channel. Navigate to trace number 12, swap channels, and zoom into the large EPSC as described above. All results are displayed in the results table (Fig. 9). You can select which results to show in the bar by right-clicking on one of the column or row title labels, and then selecting or unselecting the corresponding items.
Stimfit uses cursors to define measurement windows. Cursors are represented by vertical dashed lines extending throughout the window, similar as on an oscilloscope. For example, the baseline can be calculated as the average of all sampling points between the two base window cursors (vertical two dashed lines) or the median of the points between these cursors. To move the cursors by clicking the left mouse button where you want the baseline calculation to start. Set the right cursor by clicking the right mouse button where you want the baseline calculation to end. Press Enter. The result of the baseline calculation is displayed in the results table, and the baseline is plotted as a horizontal green dashed line (Fig. 10).

![Results Table and Cursors](image)

**Figure 2.8: Fig. 9: Showing analysis results.**

**Figure 2.9: Fig. 10: Setting the baseline window cursors.**

**Note:** You have to press Enter after changing any cursor position to update all calculations. Otherwise, you will see the results of your previous cursors settings. Alternatively, you can call `stf.measure()` from the Python shell.

The peak value will be determined between the two peak window cursors (vertical red dashed lines). To move the cursors press P. The corresponding tool-bar button will be highlighted. Set the left cursor by clicking the left mouse button where you want the peak detection to start. Set the right cursor by clicking the right mouse button where you want the peak detection to end. Press Enter. The result of the peak calculation is displayed in the results bar. “Peak (from base)” is the difference between the peak value and the baseline, and “Peak (from 0)” is the “raw” value of the peak, measured from zero, without any subtraction. A horizontal red dashed line will indicate the peak value, and a vertical dashed line will indicate the point in time when this peak value has been detected (Fig. 11).
There are three ways the peak value can be calculated: As a default, it is calculated as the maximal absolute value measured from baseline; hence, both positive- or negative-going events may be detected, whichever is larger. If you want only positive-going events to be detected, select “Edit”->”Cursor settings” from the menu. A dialog will appear. Select the “Peak” tab, and then check “Up” radio button (Fig. 12). Click the “Apply” button to measure the peak using your new settings. If you only want negative-going events to be detected, select “Down” instead. Selecting “Both” resets the peak calculation to the default mode. If you want to set the peak direction from the Python shell, you can call `stf.set_peak_direction()`, where direction can be one of “up”, “down” or “both”. The Python shell will be explained in some more detail in chapter 2.

In case the event you want to analyze is noisy, it may be helpful to use the average of several neighboring sampling points for the peak calculation instead of a single sampling point. A moving average algorithm will then be used to calculate the peak value. The number of sampling points can either be set in the cursor settings dialog (Fig. 12) or from the Python shell using `stf.set_peak_mean()`, where pts is the number of sampling points.

Some other values describing the event can be found in the results table (Fig. 13):

- **RT(20-80%)** refers to the time required for the signal to change from 20% to 80% of the peak value (measured...
from the baseline), commonly called the “20-to-80%-rise time”. The points corresponding to 20 and 80% of the peak value are indicated by green circles. They are determined by linear interpolation between neighboring sampling points.

- $t_{1/2}$ refers to the full width of the signal at half-maximal amplitude (measured from the baseline), commonly called “half-duration”. The points where the signal reaches its half-maximal amplitude are indicated by blue circles. Again, this is determined by linear interpolation between neighboring sampling points.

- **Rise** and **Decay** refer to the maximal slope during the rising and the falling phase of the signal, respectively. The corresponding points are indicated by violet circles.

- **R/D** is the ratio of the maximal slopes during the rising and the falling phase of the signal

**Note:** From version 0.8.6 on, the rise time and the half duration is independent of the baseline and peak window cursor positions. In versions prior to 0.8.6, the baseline cursors had to precede the peak window cursors. However, the calculation of the maximal slopes of decay is still restricted to the peak window.

---

**2.5 Analysis on selected traces**

You can perform the analysis of a repeated event in a recording that contains several traces. This is important if you want to study the time evolution of an event. The Trace selection window informs you about the number of traces in your recording and the index of the current trace. Zero-based index (i.e. first trace is zero, and not one) can be selected if checked in the corresponding checkbox.

To perform the analysis on only some traces, you need first to select them. For that, we can move the current trace with the left/right arrow keys and buttons (as described above), or click up/down buttons in the Trace selection window. Press $S$ if you want to select the current trace, or click the selection button (Fig. 14). The number of traces that you have already selected will be indicated in the Trace selection window. You can click on the checkbox if you want to visualize them together with your current trace.
If you selected a trace accidentally, you can remove it from the selected traces list by pressing R or clicking the trash bin button to the right of the selection button (Fig. 14).

After selecting some traces, we can select Analysis->Batch Analysis to select the type on the selected traces.

Once the measurements are selected, a result table will appear, that contains a first column with the ordered of traces and several columns with the measured parameteres.

**Note:** This is a general concept for most analysis functions: you first select traces, and the analysis will then be performed on the selected traces.

### 2.6 Average calculation

First, you have to select the traces to average (Fig. 14). Once you are done, click the “Average” button to compute the average of all selected traces (Fig. 15). A new child window will pop up showing the average. In the original child window, the average is shown as blue trace.

**Note:** If you want to perform an average (or any other measurement) of all traces in a recording, simply select Edit->Select all, or type Ctrl+A

### 2.7 Fitting functions to data

- Navigate to trace number 12 which contains a large spontaneous EPSC. Swap channels as described above, then zoom into the large EPSC.
- Set the peak and baseline cursors appropriately; the peak and baseline values will be used as initial values for the fit. Do not forget to press Enter
- The function will be fitted to the data between the two fit window cursors (gray vertical dashed lines). To move the cursors, press D (historically “D” stands for “decay”). The corresponding button will be highlighted. Set the left cursor by clicking the left mouse button where you want the fit to start. Set the right cursor by clicking the right mouse button where you want the fit to end. Press Enter to confirm the cursor settings.
Figure 2.14: **Fig. 15:** Average calculation.
• Select “Analysis”->”Fit”->”Non-linear regression” from the menu. Select a bi exponential function (Fig. 16).

![Non-linear regression settings](image)

Figure 2.15: Fig. 16: Non-linear regression settings.

• The fitted function will be displayed as a thick gray line, and a table showing the best-fit parameters and the sum of squared errors (SSE) will pop up (Fig. 17).

`stf.leastsq()` can be called from the Python shell to fit the function with index `fselect` to the data. `fselect` refers to the number that you can find in front of the function in the fit settings dialog (see Fig. 16). If `refresh=False`, the trace will not be re-drawn, which can be useful to avoid flicker when performing a series of fits.

You can use `stf.get_fit()` to get the function resulted from the fitting. This returns the evaluated function as a two dimensional NumPy array. The first dimension is the x-axis (e.g time), and the second dimension is the function evaluation results.
Figure 2.16: **Fig. 17:** Results of a non-linear regression using a bi-exponential function.
3.1 Why use Python?

Why would you want to use Python (or more specifically [SciPy]) to analyse neuroscientific data? Here are a couple of reasons:

- Widely used, general-purpose programming language
- Predicted to become the major programming language in neurosciences
- About to replace hoc as the standard NEURON interpreter, allowing to analyse the output from NEURON simulations in a single, integrated development environment
- Favoured by the German Neuroinformatics Node as the standard neural data analysis language
- Has a reputation of having a cleaner syntax than most other scientific programming languages
- Free software

3.2 Before you start

If you are new to Python, I suggest that you first have a look at the [Python-tutorial]. If that is not enough, abundant documentation is freely available on the [Python-website]. If you are new to Stimfit, I recommend going through the tutorial in chapter 1 of this manual first.

3.3 The Python shell

When you start up Stimfit, you will find an embedded Python shell in the lower part of the program window. From this shell, you have full access to the Python interpreter. For instance, you could type:

```python
>>> stf.
```

which will pop up a window showing all the available functions from the Stimfit module (abbreviated stf). For example, you could now check whether a file is open by selecting the `stf.check_doc()` function from that list:
The function documentation will pop up when you type in the opening bracket. The function returns the boolean False because you have not opened any file yet. Since the stf module is imported in the namespace, you can omit the initial ‘stf.’ when calling functions. Thus, you could get just the same result by simply typing

```python
>>> check_doc()
False
```

If you press `Ctrl+UP-arrow` at the same time, you can go through all the commands that you have previously typed in. This can be very useful when you want to call a function several times in a row.

### 3.4 Accessing data from the Python shell

- **get_trace(trace=-1, channel=-1)**

  The `stf.get_trace()` function returns the currently displayed trace as one-dimension [NumPy] array when called without arguments:

  ```python
  >>> a = get_trace()
  ```

  You can now access individual sampling points using squared brackets to specify the index. For example:

  ```python
  >>> print a[123]
  -26.3671875
  >>> print a[0]
  -21.2249755859
  ```

  Python will check for indices that are out of range. For example,

  ```python
  >>> print a[1e9]
  Traceback (most recent call last):
  File "<input>", line 1, in <module>
  IndexError: index out of bounds
  ```

  You can use the `stf.get_trace()` function to return any trace within a file. The default values of `trace = -1` and `channel = -1` will return the currently displayed trace of the active channel. By passing a value of 1 as the first argument, you could access the second trace within your file (assuming it contains more than one trace of course). Remember that indices are zero-based!

  ```python
  >>> b = get_trace(1)
  >>> print b[234]
  >>> -23.7731933594
  ```

### 3.5 Using NumPy with Stimfit

[NumPy] allows you to efficiently perform array computations from the Python shell. For example, you can multiply an array with a scalar:
>>> a = get_trace()
>>> print a[234]
-27.0385742188
>>> b = a*2
>>> print b[234]
-54.0771484375

Or multiply two arrays:

>>> a = get_trace()
>>> b = get_trace(1)
>>> c = a*b
>>> print a[234], "*", b[234], "=", c[234]
-27.0385742188 * -23.7731933594 = 642.793253064

- new_window()

You can now display the results of the operation in a new window by passing a 1D-NumPy array to the stf.new_window() function:

>>> new_window(c)

The sampling rate and units will be copied from the window of origin. A short way of doing all of the above within a single line would have been:

>>> new_window(get_trace() * get_trace(1))

- new_window_matrix()

You can pass a 2D-NumPy array to stf.new_window_matrix(). The first dimension will be translated into individual traces, the second dimension into sampling points. This example will put the current trace and its square root into subsequent traces of a new window:

>>> numpy_matrix = np.empty( (2, get_size_trace()) )
>>> numpy_matrix[0] = get_trace()
>>> numpy_matrix[1] = np.sqrt( np.abs(get_trace()) )
>>> new_window_matrix(numpy_matrix)

In this example, np is the NumPy namespace. Typing np. at the command prompt will show you all available NumPy functions. stf.get_size_trace() will be explained later on.

- new_window_list()

Although using a 2D-NumPy array is very efficient, there are a few drawbacks: the size of the array has to be known at construction time, and all traces have to be of equal lengths. Both problems can be avoided using stf.new_window_list(), albeit at the price of a significant performance loss. stf.new_window_list() takes a Python list of 1D-NumPy arrays as an argument:

>>> python_list = [get_trace,]
>>> python_list.append( np.concatenate( (get_trace(), get_trace()) ) )
>>> new_window_list(python_list)

Note that items in Python list are written between squared braces, and that a comma is required at the end of single-item lists.

The [SciPy] library, which is build on top of [NumPy], provides a huge amount of numerical tools, such as special functions, integration, ordinary differential equation solvers, gradient optimization, genetic algorithms or parallel programming tools. Due to its size, it is not packaged with Stimfit by default, but I highly recommend installing it for more advanced numerical analysis.

3.5. Using NumPy with Stimfit
3.6 Control Stimfit from the Python shell

3.6.1 Cursors

Cursors can be positioned from the Python shell using one of the `set_[xy]_start` or `set_[xy]_end` functions, where `[xy]` stands for one of peak, base or fit, depending on which cursor you want to set. Correspondingly, the `get_[xy]_start` or `get_[xy]_end` functions can be used to retrieve the current cursor positions.

- `set_[xy]_start(pos, is_time = False)` and `set_[xy]_end(pos, is_time = False)` take one or two arguments. `pos` specifies the new cursor position. `is_time` indicates whether `pos` is an index, i.e. in units of sampling points (False, default), or in units of time (True), with the trace starting at t=0 ms. If there was an error, such as an out-of-bounds-index, these functions will return False.

- `get_[xy]_start(pos, is_time = False)` and `get_[xy]_end(pos, is_time = False)` optionally take a single argument that indicates whether the return value should be in units of sampling points (is_time = False, default) or in units of time (is_time = True). Again, traces start at t=0 ms. These functions will return -1 if no file is opened at the time of the function call. Indices can be converted into time values by multiplying with `stf.get_sampling_interval()`. For example:

```plaintext
>>> print "Peak start cursor index:", get_peak_start()
Peak start cursor index: 254
>>> print "corresponds to t =", get_peak_start(True), "ms"
corresponds to t = 2.54 ms
>>> print "=", get_peak_start() * stf.get_sampling_interval(), "ms"
= 2.54 ms
>>> set_peak_start(10, True)
True
>>> print "new cursor position:", get_peak_start()
new cursor position: 1000.0
>>> print "at t =", get_peak_start(True), "ms"
at t = 10 ms
```

The peak, baseline and latency values will not be updated until you either select a new trace, press Enter in the main window or call `stf.measure()` from the Python shell.

3.6.2 Trace selection and navigation

- `select_trace(trace = -1)`

You can select any trace within a file by passing its zero-based index to `stf.select_trace()`. The function will return False if there was an error. The default value of -1 will select the currently displayed trace as if you had pressed S. If you wanted to select every fifth trace, starting with an index of 0 and ending with an index of 9 (corresponding to numbers 1 to 10 in the drop-down box), you could do:

```plaintext
>>> for n in range(0, 10, 5): select_trace(n)
... True
```

Note that the Python range function omits the end point.

- `unselect_all() select_all() get_selected_traces() new_window_selected_this()`

The list of selected traces can be cleared using `stf.unselect_all()`, and conversely, all traces can be selected using `stf.select_all()`. `stf.get_selected_indices()` returns the indices of all selected traces as a Python tuple. Finally, the selected traces within a file can be shown in a new window using `stf.new_window_selected_this()`.
• get_size_trace(trace=-1, channel=-1) and get_size_channel(channel=-1)

Return the number of sampling points in a trace the number of traces in a channel, respectively. trace and channel have the same meaning as in stf.get_trace(). These functions can be used to iterate over an entire file or to check ranges:

```python
>>> unselect_all(0)
>>> for n in range(0, get_size_channel(), 5): select_trace(n)
True
>>> print get_selected_indices()
(0, 5)
>>> for n in get_selected_indices():
...   print "Length of trace", n, ":", get_size_trace(n)
...Length of trace 0 : 13050
Length of trace 1 : 13050
```

• set_trace(trace)

sets the currently displayed trace to the specified zero-based index and returns False if there was an error. This will update the peak, base and latency values, so there is need to call stf.measure() directly after this function.

• get_trace_index()

Correspondingly, stf.get_trace_index() allows you to retrieve the zero-based index of the currently displayed trace. There is a slight inconsistency in function naming here: do not confound this function with stf.get_trace().

3.6.3 File I/O

• file_open(filename) and file_save(filename)

will open or save a file specified by filename. On windows, use double backslashes (\) between directories to avoid conversion to special characters, such as `\t` or `\n`; for example:

```python
>>> file_save("C:\data\datafile.dat")
in Windows or
>>> file_save("/home/cs/data/datafile.dat")
in GNU/Linux.
```

• close_this()

stf.close_this() will close the currently displayed file, whereas

• close_all()

stf.close_all() closes all open files.

3.6.4 Define your own functions

By defining your own functions, you can apply identical complex analysis to different traces and files. The following steps are required to make use of your own Python files:

1. Create a Python file in a directory that the Python interpreter will find. If you do not know where that is, use the Stimfit program directory (typically, this will be C:\Program Files\Stimfit in Windows or
2. Import the Stimfit module in your file:

```python
>>> import stf
```

3. Start Stimfit and import your file in the embedded Python shell. Assuming that your file is called `myFile.py`, you would do:

```python
>>> import myFile
```

4. If you have applied changes to your file, there is no need to restart Stimfit. Just do:

```python
>>> reload(myFile)
```

To give you an example, this program shows a function that returns the sum of the squared amplitude values across all selected traces of a file.

```python
# import the Stimfit core module:
import stf

def get_amp():
    """ Returns the amplitude (peak-base)""
    return stf.get_peak()-stf.get_base()

def sqr_amp():
    """ Returns the sum of squared amplitudes of all selected traces, or -1 if there was an error. Uses the current settings for the peak direction and cursor positions.""

    # store the current trace index:
    old_index = stf.get_trace_index()

    sum_sqr = 0
    for n in stf.get_selected_indices():
        # setting a trace will update all measurements so there is no need to call measure()
        if not stf.set_trace(n):
            return -1
        sum_sqr += get_amp()**2

    # restore the displayed trace:
    stf.set_trace(old_index)

    return sum_sqr
```

To import and use this file, you would do:

```python
>>> import myFile
>>> myFile.sqr_amp()
497.70163353882447
```

**Note:** You can import a file from any directory by selecting File->Import Python module or simply by pressing Ctrl+I. The file will be automatically loaded in the embedded Python shell and ready to be used (it is not necessary to type `import(myFile)`). To reload the file, you have simply to select File->Import Python module or Ctrl+I again.
3.6.5 Add a Python function to the Stimfit menu

It is possible to create a submenu with your own Python functions in the Extensions menu of Stimfit. To include your own function as a submenu you have to edit the file extensions.py. This file is located in the Stimfit program folder. extensions.py contains a list called extensionsList that contains a list of submenus. For example, in the following example, only one submenu can be found, called myAPCounter:

```python
extensionList = [myAPCounter,]
```

To create a submenu we have first to define it as Extension. In extensions.py we should add:

```python
myAPCounter = Extension("Count APs", spells.count_aps, "Counts APs in selected traces", True)
```

this command creates the extension called myAPCounter, which has the name “Counts APs” and executes the function spells.count_aps. If we add the extension myAPCounter to the extensionList, we will see now that a submenu appear within the Extension menu.

In general, an extension requires four arguments:

1. **Function name**: this is the name that will appear in the submenu (in our example this is “Counts APs”).
2. **Python function**: the custom Python function to be executed when clicking on the submenu. We used here spells.count_aps
3. **Description**: a more elaborate description of what the function is doing. We wrote “Counts APs in selected traces”.
4. **Does your function require a file?**: If your function needs a file to be open type ‘True’, otherwise ‘False’. Because our Python function operate on files, we typed ‘True’.

**Note**: Use a boolean return type when using your own Python functions in the extensions Menu. If you write a function that returns False upon failure, Stimfit will show a warning indicating that your custom function did not work properly.

3.7 Some recipes for commonly requested features

Some often-requested features could not be integrated into the program easily without cluttering up the user interface. The following sections will show how the Python shell can be used to solve these problems.

3.7.1 Cutting traces to arbitrary lengths

Cutting traces is best done using the squared braked operators ([]) to slice a [NumPy] array. For example, if you wanted to cut a trace at the 100th sampling point, you could do:

```python
>>> a = get_trace()
>>> new_window(a[:100])
>>> new_window(a[100:]):
```

In this example, a[:100] refers to a sliced NumPy array that comprises all sampling points from index 0 to index 99, and a[100:] refers to an array from index 100 to the last sampling point.

- **cut_traces(pt)** and **cut_traces_multi(pt_list)**

These functions cut all selected traces at a single sampling point (pt) or at multiple sampling points (pt_list). The cut traces will be shown in a new window. Both functions are included in the stf namespace from version 0.8.11 on. The code for stf.cut_traces() is listed here.
import stf
import numpy as np

def cut_traces( pt):
    """Cuts the selected traces at the sampling point pt, and shows the cut traces in a new window.
    Returns True upon success, False upon failure."""

    # Check whether anything has been selected:
    if not stf.get_selected_indices():
        print("Trace is not selected!")
        return False

    new_list = list()

    for n in stf.get_selected_indices():
        if not stf.set_trace(n):
            return False

    # Check for out of range:
    if pt < stf.get_size_trace():
        new_list.append( stf.get_trace()[:pt] )
        new_list.append( stf.get_trace()[pt:] )
    else:
        print("Cutting point", pt, "is out of range"

    # Don’t create a new window if everything was out of range
    if len(new_list) > 0:
        return stf.new_window_list( new_list )
    else:
        return False

For example:

>>> dt = stf.get_sampling_interval()
>>> cutPoints = [int(100/dt), int(900/dt)]
>>> cut_traces_multi(cutPoints)

will cut all selected traces at time 100 and 900 (in units of the x axis) and show the cut traces in a new window. Note that you can pass a list or a tuple as argument.

>>> cut_traces_multi(range(100,2000,100)) # cut at 100 sampling points not ms!

will cut the selected traces at every 100th sampling point, starting with the 100th and ending with the 1900th.
4.1 Measurement of synaptic delay

Stimfit is frequently used to measure the delay between a synaptic signal and a post-synaptic response. Classically, this synaptic delay or latency is defined as “the time interval between the peak of the inward current through the synaptic membrane and commencement of inward current through the postsynaptic membrane” (Katz and Miledi, 1965). Neglecting cable properties of neurons for a while, the maximal inward current during an action potential is expected to flow at the time of maximal slope during the rising phase (Jack et al., 1983), since

\[ I_m = I_{\text{cap}} + I_{\text{ionic}} = C_m \frac{dV_m}{dt} + I_{\text{ionic}} = 0, \text{ and hence} \]

\[ I_{\text{ionic}} = -I_{\text{cap}} = -C_m \frac{dV_m}{dt} \]

The commencement (sometimes called “foot”) of the postsynaptic current can robustly be estimated from the extrapolated intersection of the baseline with a line through the two points of time when the current is 20 and 80% of the peak current (Jonas et al., 1993, Bartos et al., 2001).

Although the method described above yields reliable results when both the pre- and the postsynaptic whole-cell recording show little noise and few artifacts, it may sometimes be favorable to use other estimates for the pre- and postsynaptic signals, for example, when extracellular stimulation was used or when there are a lot of failures in the postsynaptic response. The following sections will explain how this is done in practice.

4.2 Trace alignment

It may sometimes be useful to align traces before measuring the latency, either for visualization purposes or to create an average without temporal jitter. Although an aligned average can be created using a tool-bar button, the recommended way to align traces is to use the Python shell.

• align_selected(alignment, active=False)

stf.align_selected() aligns the selected traces to a point that is determined by the user-supplied function alignment and then shows the aligned traces in a new window. The alignment function is applied to the active channel if active=True or to the inactive channel if active=False. The alignment function has to return an index within a traces, and it should adhere to the general form index(active), where active is a boolean indicating whether the active or the inactive channel should be used. The most common alignment functions are built into the program:

• maxrise_index(active)

stf.maxrise_index() returns the zero-based index of the maximal slope of the rise in units of sampling points (see Fig. 13), interpolated between adjacent sampling points, or a negative value upon failure.

• peak_index(active)

stf.peak_index() returns the zero-based index of the peak value in units of sampling points (see Fig. 13) or a negative value upon failure. The return value may be interpolated if a moving average is used for the peak calculation.

• foot_index(active)

stf.foot_index() returns the zero-based index of the foot of an event, as described in Fig. 18, or a negative value upon failure.

• t50left_index(active)

stf.t50left_index() returns the zero-based index of the left half-maximal amplitude in units of sampling points (see Fig. 13), or a negative value upon failure. The return value will be interpolated between sampling points.

• t50right_index(active)

stf.t50right_index() returns the zero-based index of the right half-maximal amplitude in units of sampling points (see Fig. 13), or a negative value upon failure. The return value will be interpolated between sampling points.

The following code can be used to align all traces within a file to the maximal slope of rise in the inactive channel.

```python
# import the Stimfit core module:
import stf

def align_maxrise():
    """Aligns all traces to the maximal slope of rise \n```

Figure 4.1: Fig. 18: Foot of an EPSC (red circle), estimated from the extrapolated intersection of the baseline with a line through the two points of time when the current is 20 and 80% of the peak current (black open circles).
of the inactive channel. Baseline and peak cursors have to be set appropriately before using this function. Return value: True upon success. False otherwise."

```
stf.select_all()

# check whether there is an inactive channel at all:
if ( stf.maxrise_index( False ) < 0 ):
    print "File not open, or no second channel; aborting now"
    return False

stf.align_selected( stf.maxrise_index, False )

return True
```

4.3 Setting the latency cursors

The latency cursors (plotted as dotted vertical blue lines) can either be set automatically to some predefined points within a trace, or manually using the mouse buttons. The predefined points can be chosen from the menu: “Edit”->”Measure latency from...” and “Edit”->”Measure latency to...”. The “beginning” of an event refers to the foot as explained above (Fig. 18). If “manually” is selected, the left and right mouse buttons can be used to set the first and second latency cursors while the latency mode is activated. To switch to the latency mode, you can either click the corresponding button in the toolbar (Fig 19) or press L.

![Activate latency mode.](image)

Figure 4.2: Fig. 19: Activate latency mode.

To confirm your latency cursor settings and measure latencies, you can either press Enter or call `stf.measure()` from the shell. The latency, i.e. the time interval between the first and the second latency cursor, will be shown in the results table as long as you activated this value. The latency will be indicated as double-headed arrow connecting the two latency cursors (Fig. 20).
Figure 4.3: **Fig. 20:** The latency between maximal slope of rise of an action potential (red) and the foot of an EPSC (black) is indicated by a horizontal double-headed arrow.
5.1 Introduction

To isolate individual events such as EPSCs or EPSPs from recorded data, Stimfit uses a template matching algorithm as described by Jonas et al. (1993)\(^1\), with some implementation details adopted from Clemens and Bekkers (1997)\(^2\). The template consists of a waveform \(p(t)\) with a length of \(n\) sampling points that represents the time course of a typical event. The template is slid over the trace of recorded values \(r(t)\), and at each sampling point with index \(s\), it is multiplied by a scaling factor \(m\) and an offset \(c\) is added or subtracted so that the sum of squared errors \(\chi^2(t_s)\) between the trace and the template is minimized:

\[
\chi^2(t_s) = \sum_{k=0}^{n-1} [r(t_s+k) - (m \cdot p(t_k) + c)]^2
\]

As can be seen from this equation, this amounts to the fairly simple operation of fitting a straight line that relates \(p(t)\) and \(r(t)\) at every sampling point.

Finally, some detection criterion has to be applied to decide whether an event has occurred at a sampling point. Two options are available in Stimfit: Jonas et al. (1993)\(^1\) suggest to use the linear correlation coefficient between the optimally scaled template and the data, whereas Clements and Bekkers (1997)\(^2\) compare the scaling factor with the noise standard deviation.

5.2 A practical guide to event detection

In practice, the following steps need to be performed to extract events with Stimfit:

1. Create a preliminary template by fitting a function to a single, large and isolated event.
2. Use this preliminary template to extract some more exemplary large and isolated events using a high detection threshold.
3. Create a final template by fitting a function to the average of the exemplary events.

---


4. Extract all events with the final template using a low detection criterion threshold.
5. Eliminate false-positive, add false-negative events.

This procedure will be explained in some more detail in the following sections.

### 5.2.1 Create a preliminary template

In general, the template waveform \( p(t) \) can be of arbitrary shape. A typical way of creating such a template is to fit a function with a time course matching the event kinetics to some exemplary events. For example, EPSCs can typically be modeled with the sum or the product of two exponential functions. In practice, a robust estimate for a template can be obtained using an iterative approach, which will be illustrated here using a recording of miniature EPSCs that you can download here.

![Figure 5.1](image)

**Figure 5.1:** Creation of a “bait” template.

First, we fit a function to a single large and isolated event to create a preliminary “bait” template. In this case, we will use the EPSC that can be found roughly between \( t = 20990 \) ms and \( t = 21050 \) ms. Then, we fit the sum of two exponential functions with a delay to this EPSC. To obtain the same template as in the example, you can call the function `preliminary` from the `minidemo` module that comes bundled with Stimfit.

```python
>>> import minidemo
>>> minidemo.preliminary()
```

This will take care of the appropriate cursor positions and the biexponential fit. If you prefer, you can use the fit settings dialog, as described in chapter 1 (Fig. 16).

### 5.2.2 Extract exemplary events

We now use the bait example to fish some more large and isolate events. Choose “Analysis”->”Event detection”->”Template matching...” from the menu.

In the dialog that will pop up (Fig. 22), you can set the threshold for the detection criterion. Since we want to extract some large and isolated events during this first pass, we set this to a high number, say 10, using the template scaling.

---

3 Note that the product of two exponentials \( f(t) = a(1 - e^{-\frac{t}{\tau_1}})e^{-\frac{t}{\tau_2}} \) can equivalently be expressed as the sum of two exponentials: \( f(t) = a(e^{-\frac{t}{\tau_3}} - e^{-\frac{t}{\tau_2}}) \), with \( \tau_3 = \frac{\tau_2\tau_1}{\tau_2 - \tau_1} \).
factor (Clemens and Bekkers, 1997). Click “OK” to start the event detection. When finished, press F to fit the whole trace to the window. The detected events will be marked by blue arrows in the upper part of the window, and blue circles will indicate the peak values of the detected events (Fig 23).

To view the isolated events in a new window, you have to switch to the event editing mode, either by pressing E or by activating the corresponding button in the toolbar (Fig. 24). When you now click on the trace with the right mouse button, a menu will show up. Select “Extract selected events” from this menu. this will put the exemplary EPSCs into a new window.
5.2.3 Create the final template

We now create the average of all extracted events, as explained in chapter 1. Then, we fit a biexponential function to the average, as explained above for the single EPSC. Remember to set the baseline, peak and fit window cursors appropriately before performing the fit, and to update all calculations. Again, you can make use of a function from the minidemo module to set the cursors and perform the fit:

```python
>>> import minidemo  # if you have not imported it already
>>> minidemo.final()
```

The final template should look similar as shown in Fig. 25.

![Figure 25: Creating a final template.](image)

5.2.4 Extract all events

Go back to the original file (minis.dat). Extracting all events with the final template is done in nearly the same way as described above for the preliminary template. However, you have to choose the correct template in the event dialog: The final template in this case is the second on the list (Fig. 26). For this final run, we will lower the detection threshold to a value of 3, as suggested by Clements and Bekkers (1997).
5.2.5 Edit detected events

Usually, the detected events have to be screened visually to remove false-positives and add false-negatives. Removing false-positives is done by unselected the checkbox next to the arrow indicating an event (Fig. 23). To add false-negatives, you have to switch to the event-editing mode (Fig. 24) and then right-click on the trace at the position where the event starts. From the context menu that will pop up, select “Add an event that starts here” (Fig. 27). To efficiently screen the whole trace, it is convenient to use Shift and left arrow at the same time. This will move the trace left by the width of one window. Once you are done with editing, choose “Extract selected events” from the context menu.
5.2.6 Analyze extracted events

If you used the same settings as suggested above, 97 events will be extracted. You will find a table on the left of the traces: This will show you the time of onset of the events and the inter-event intervals. Usually, you will want to apply some further analysis to the extracted events. To do so, you first have to adjust the baseline, peak and fit cursors. Again, there is a function in the minidemo module taking care of that:

>>> minidemo.batch_cursors()

To analyze all traces efficiently, you can now perform a “batch analysis” on all traces at once: First, select all traces, either using stf.select_all() from the shell, or “Edit”->”Select all traces” from the menu or pressing Ctrl + A. Then choose “Analysis”->”Batch analysis” from the menu.

Figure 5.8: Fig. 28: Batch analysis settings.

From the dialog (Fig 28) choose the analysis functions that you want to apply to your data. Click “OK” once your are done. A new table will appear to the left of the traces. You can copy and paste values from the tables to spreadsheet programs for further analysis.

5.2.7 Adjusting event detection settings

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Too many false-positive events have been detected.</td>
<td>1. Increase the detection threshold</td>
</tr>
<tr>
<td>2. Too many events have been missed (false-negatives)</td>
<td>2. Decrease the detection threshold</td>
</tr>
<tr>
<td>3. One and the same event is detected multiple times at short time intervals</td>
<td>3. Increase the number of sampling points between events</td>
</tr>
<tr>
<td>4. Closely spaced events are not detected separately</td>
<td>4. Decrease the number of sampling points between events</td>
</tr>
</tbody>
</table>
7.1 Building the Python module only

Author Christoph Schmidt-Hieber
Date March 21, 2014

Building only the standalone Python file i/o module is fairly straightforward. First, you need a couple of libraries:

```bash
$ sudo apt-get install build-essential git libboost-dev python-dev python-numpy libhdf5-serial-dev swig
```

Then, you need the Stimfit source code:

```bash
$ cd $HOME
$ git clone https://code.google.com/p/stimfit stfio
```

This will download the code to a directory called `stfio`.

Next, you need to generate the build system:

```bash
$ cd $HOME/stfio
$ ./autogen.sh
```

Now you can configure. I strongly recommend building in a separate directory.

```bash
$ cd $HOME/stfio
$ mkdir build
$ cd build
$ mkdir module
$ cd module
$ ../..configure --enable-module
```

Remember to add the argument `--with-biosig` to the configure script if you want to have extra biomedical file formats for `stfio`.

Then, build and install:
$ make -j 4 # where 4 refers to the number of parallel build processes
$ sudo make install

Finally, run python to test the module, as described in *The stfio module.*

## 7.2 Building Stimfit

**Author** Jose Guzman, Christoph Schmidt-Hieber  
**Date** March 21, 2014

This document describes how to install Stimfit 0.13.17 under GNU/Linux. The installation was tested on a GNU/Debian testing/unstable system, with a 2.6-based kernel and with support for Python 2.5. and Python 2.6. It should work on other Debian-based systems (e.g Ubuntu) as well. I assume that you have the GNU C compiler (gcc) and the GNU C++ compiler (g++) already installed in your system. Please, check that both versions match. For our installation we will use gcc-4.2.4 and the same version of g++.

### 7.2.1 What we need before we start

For the impatient, here are all Stimfit build dependencies:

```
$ sudo apt-get install libboost-dev 
    python-dev 
    python-numpy 
    python-matplotlib 
    libhdf5-serial-dev 
    swig 
    python-wxgtk2.8 
    libwxgtk2.8-dev 
    fftw3-dev 
    liblapack-dev 
    chrpath 
    git 
    automake 
    autoconf 
    libtool 
    libgtest-dev
```

This will get you, amongst others:

- **[boost]**: C++ library that is mainly used for its shared pointers.
- **[Lapack]**: A linear algebra library.
- **[fftw]**: Library for computing Fourier transformations.
- **[NumPy]**: To handle multidimensional arrays and perform more complex numerical computations with Python.
- **[HDF5]**: This is the hierarchical Data Format 5 (HDF5) to manage large amount of data.

In addition, you can install doxygen, python-sphinx and graphviz if you want to build yourself the documentation.

### 7.2.2 Optional: wxWidgets 2.9

- **[wxWidgets]** and **[wxPython]** 2.9 (unstable): If you’d like to live on the bleeding edge and get decent printing support through gtk-print, you can build against wxWidgets 2.9, which in turn needs to be built from source. To
get the build dependencies (which are the same as for 2.8), do:

$ sudo apt-get build-dep wxwidgets2.8

Get the source for both wxWidgets and wxPython in a single tarball:

$ wget http://downloads.sourceforge.net/wxpython/wxPython-src-2.9.1.1.tar.bz2
$ tar xvfj wxPython-src-2.9.1.1.tar.bz2

Check http://www.wxpython.org/download.php#unstable for updates.

From there, follow the build instructions found here

7.2.3 Optional: PyEMF

[PyEMF] is needed to export figures to the windows meta file format (WMF/EMF). EMF is a vector graphics format and can be imported in different Office software including LibreOffice. In order to install it, do:

$ wget http://sourceforge.net/projects/pyemf/files/pyemf/2.0.0/pyemf-2.0.0.tar.gz/download -O pyemf-2.0.0.tar.gz
$ tar xvf pyemf-2.0.0.tar.gz && cd pyemf-2.0.0
$ sudo python setup.py install

7.2.4 Download the Stimfit source code

You can download the latest development code for Stimfit from the Github code repository. For that, simply type from your current $HOME directory:

$ git clone https://github.com/neurodroid/stimfit.git

This will grab all the required files into $HOME/stimfit. If you’d like to update at a later point, do:

$ cd $HOME/stimfit
$ git pull

7.2.5 Build Stimfit

Go to the stimfit directory (in our example $HOME/stimfit) and type:

$ ./autogen.sh

to generate the configure script. Remember that we need Autoconf, Automake and LibTool to use autogen. After that, you can call it with

$ ./configure --enable-python

The \texttt{--enable-python} option is absolutely necessary to install Stimfit since some of the functionality depends on Python. The configure script has some additional options. For example, we may want to use IPython instead of the default embedded python shell with the option \texttt{--enable-ipython} (note that the IPython shell is only available under GNU/Linux and it is still very experimental).

Finally, after running configure, you can type

$ make -j[N]

where [N] is the number of parallel builds you want to start. And finally:
$ sudo make install
$ sudo /sbin/ldconfig

Note: If you want to install Stimfit as local user (e.g. in ~/.local) with a local version of Python (e.g. ~/.local/lib/python2.6) you have to add the following argument to configure script:

$ ./configure --prefix= $HOME/.local PYTHON = $HOME/.local/lib/python2.6 --enable-python

and after that simply call make and make install as normal user. The Stimfit executable will be now in $HOME/.local

### 7.2.6 Building Stimfit with BioSig import filter

It is recommended to build Stimfit with the BioSig import the file filters to read HEKA files or to have the possibility import some other file formats used biomedical signal processing. To do it, follow this instructions:

1. It is first recommended to install libsuitesparse and libz libraries:

   ```bash
   sudo apt-get install libsuitesparse-dev libz-dev gawk
   ```

2. Download BioSig sources: you can obtain the latest BioSig version in BioSig downloads. Choose BioSig for C/C++, libbiosig (v1.5.6 or higher is recommended). Alternatively, you can obtain the latest developmental version from the git repository:

   ```bash
   git clone git://git.code.sf.net/p/biosig/code biosig-code
   ```

3. Compile and install the sources: enter the directory biosig4c++ and type:

   ```bash
   sudo make install_libbiosig
   ```

   After that you can enter the option --with-biosig in the configure script of Stimfit and compile as usual.

### 7.2.7 Building documentation

The manual of Stimfit including the documentation is accessible on-line in http://www.stimfit.org/doc/sphinx/. To have your own local copy of the documentation, you will need to install sphinx:

```bash
sudo apt-get install python-sphinx
```

It is possible to build a local copy of the documentation there by simply calling:

```bash
sphinx-build $HOME/Stimfit/doc/sphinx/ <destinyFolder>
```

The html documentation will be located in <destinyFolder>/index.html

Additionally, the source code is documented with [Doxygen] and is also accessible on-line in http://www.stimfit.org/doc/doxygen/html/. If you want to have a local copy of the documentation, you will need to install the doxygen and gravphvix:

```bash
sudo apt-get install doxygen gravphvix
```

Enter a directory called doc inside Stimfit (e.g. $HOME/stimfit/doc) and type:

```bash
doxygen DoxyFile
```

The local documentation of the source code will be in $HOME/stimfit/doc/doxygen/html
Building Stimfit from scratch requires you to install Xcode, MacPorts and a couple of libraries. Note that this may take several hours.

Contents:

8.1 Building Stimfit

8.1.1 Installing with MacPorts

Download and install MacPorts from here, and then run the following command to install git

```
sudo port install git-core
```

Get the Stimfit source from the git repository

```
git clone https://code.google.com/p/stimfit/
```

Edit the MacPorts sources configuration file (/opt/local/etc/macports/sources.conf) and place the following line before the one that reads

```
rsync://rsync.macports.org/release/tarballs/ports.tar [default]
```

(change the path to the Stimfit directory accordingly).

```
file:///${STIMFITDIR}/stimfit/macosx/macports
```

Note: using the root of your account as opposed to a subdirectory (ie, Documents or Downloads folders) may prevent permissions access errors when building.

Next, go to the Stimfit macports directory
cd ${STIMFITDIR}/macosx/macports

Add the local ports file to MacPorts by running the following command at this location

    sudo portindex

When finished, you can now build Stimfit in MacPorts

    sudo port install stimfit

MacPorts will download and install various dependencies, and then attempt to build Stimfit from source.
This document describes how to build Stimfit version 0.13.17 on Windows. I strongly recommend sticking to the suggested directory names. If for any reason you’d like to use different target directories, you’ll have to update all the property sheets (Config.vsprops) in the Visual Studio solution.

Getting all the prerequisites takes about an hour, but only needs to be completed once. Building the full solution takes about 3 minutes.

9.1.1 Visual C++ Express 2008

The official Windows version of Python 2.7 was built with Visual Studio 2008. We therefore have to use Visual C++ 2008 so that we link against the same C runtime library. Luckily, there’s a free version called Visual C++ 2008 Express that you can get directly from here.

9.1.2 Libraries

HDF5

Get the HDF5 libraries from here. Use HDF5189-win32-vs9-shared.zip. There are no VS2008-prebuilt binaries for HDF5 >= 1.8.10 unfortunately. Extract the zip file, and then install to a folder called “hdf5” in your home directory (e.g. C:\Users\username) using the extracted executable.
Boost

Get the Boost C++ libraries. Move the extracted folder to your home directory and rename to “boost”. If you used the zip file, you might have to move the first folder (called something like boost_1_54_0, or whatever the current version is) one directory up and rename it to “boost”. At any rate, you should check that you have the boost header files (*.hpp) sitting in C:\Users\username\boost\boost\*.hpp.

Python

Download and install Python 2.7. Make sure to install it for all users so that it ends up in C:\Python27

PyEMF

Get PyEMF 2.0.0 from here. Install to your home directory(_not_ to C:\Python27\*) to pyemf_2.0.0. Rename the folder to “pyemf-2.0.0”.

NumPy

Install NumPy from here. Use the win32 “superpack” for Python 2.7.

Matplotlib

Install Matplotlib from here. Use the win32 version for Python 2.7.

wxWidgets

Get the prebuilt wxWidgets 2.9 libraries from the wxPython site here. Choose the “32-bit binaries for MSVC 9” in the “Windows Development Files” section. Install to your home directory. Rename the folder to “wx”.

wxPython

Get wxPython 2.9 from here. Choose the 32-bit version for Python 2.7 in the “Windows Binaries” section. Install to your home directory(_not_ to C:\Python27\*!). Rename the folder to “wxPython”.

FFTW

Get the latest 32-bit fftw dlls from here. Extract to a folder called “fftw” in your home directory. As instructed on the fftw install page, open the Visual Studio Command Prompt that is in the Visual Studio Tools section of the Program menu. cd into the fftw folder and type:

C:\Users\username\fftw> lib /def:libfftw3-3.def
C:\Users\username\fftw> lib /def:libfftw3f-3.def
C:\Users\username\fftw> lib /def:libfftw3l-3.def
libbiosig

Get the latest 32-bit biosig dlls from here. Extract to a folder called “biosig” in your home directory. Similar to fftw, open the Visual Studio Command Prompt that is in the Visual Studio Tools section of the Program menu. cd into the biosig folder and type:

C:\Users\username\biosig> lib /def:libbiosig2.def

9.1.3 Build Tools

SWIG

Cygwin has SWIG in its repositories. Otherwise, you can download it from here. At any rate, make sure that the binary is located in C:\cygwin\bin\swig.exe.

git

Cygwin has git in its repositories. Otherwise, you can download it from here.

nsis

Get nsis from here.

9.2 Building Stimfit

Author  Christoph Schmidt-Hieber
Date  March 21, 2014

9.2.1 Get the source code

Clone the latest source code into your home directory (on cygwin, this will be something like /cygdrive/c/Users/username/)

$ cd /cygdrive/c/Users/username
$ git clone https://code.google.com/p/stimfit/

9.2.2 Build Stimfit

Open the solution in stimfit/windows/VS2008/Stimfit/Stimfit.sln with Visual C++ Express 2008. Build the solution by clicking “Build” -> “Build Solution”. Alternatively, open the Visual Studio Command Prompt:

C:\> cd C:\Users\username\stimfit\windows\VS2008\Stimfit
C:\Users\username\stimfit\windows\VS2008\Stimfit> msbuild Stimfit.sln /p:Configuration=Release

9.2.3 Create an installer

Use nsis to compile the installer script in stimfit/nsis/installer.nsi.
This document collects answers to some questions like “How do I make ... in Stimfit with python?”. Though much of the material can be easily found in the Stimfit manual, the examples provided here are a good way for the casual user to start using Python in Stimfit.

It assumes a basic knowledge of the embedded Python shell of Stimfit. Some Python knowledge and a substantial proficiency in Stimfit are recommendable. Please note that this is not a Python manual, but a way to use Python for some basic analysis tasks provided with Stimfit. For a detailed Python manual, we encourage the user to visit the official Python documentation on the [Python-website] and to read carefully the Stimfit manual.

The functions described along this document are available in your current Stimfit version. To make use of them you have simply to type the following line in the Stimfit embedded Python shell:

```python
>>> import spells
```

After that, functions can be called with the dot notation (i.e just typing `spells` before the function) For example, if we want to call the function rmean() we would simply do it in this way:

```python
>>> spells.rmean(10)
```

Finally, the contents of this document are organized with increased level of complexity, assuming some of the last chapters concepts and topics described in the first chapters. Thus, we encourage the newcomer to follow the order exposed below or to visit the section code commented in previous chapters.

Contents:

10.1 Resistance Calculation

The resistance can be simply calculated using Ohm’s law. Currents passing through the pipette will be proportional to the applied voltage difference. This proportional factor is the resistance.

\[ R = \frac{\Delta V}{I} \]

In voltage clamp, resistance can be calculated by recording the current once the voltage difference is known. This Python routine should calculate the current difference, and resistance will be calculated given the voltage amplitude.
10.1.1 The resistance function

Note: You can find different routines to calculate the resistance in the file charlie.py. This file can be found in the Stimfit program directory (C:\Program Files\Stimfit in Windows or /usr/lib/python2.5/site-packages/Stimfit in GNU/Linux, assuming python2.5 is your current python environment). The name charlie.py is in acknowledgment to Charlie, for her contribution to the development of the given routines.

Note that this function assumes that current is recorded in pA. It sets the stf cursors (peak and baseline) to calculate the current deviation in response to the voltage difference. Finally, the voltage amplitude should be entered in mV.

```python
import numpy as np

# stimfit python module:
import stf

def resistance( base_start, base_end, peak_start, peak_end, amplitude):
    ""
    Calculates the resistance from a series of voltage clamp traces.
    ""
    Keyword arguments:
    base_start -- Starting index (zero-based) of the baseline cursors.
    base_end -- End index (zero-based) of the baseline cursors.
    peak_start -- Starting index (zero-based) of the peak cursors.
    peak_end -- End index (zero-based) of the peak cursors.
    amplitude -- Amplitude of the voltage command.
    
    Returns:
    The resistance.
    ""
    if not stf.check_doc():
        print "Couldn’t find an open file; aborting now."
        return 0

    #A temporary array to calculate the average:
    set = np.empty( (stf.get_size_channel(), stf.get_size_trace()) )
    for n in range( 0, stf.get_size_channel() ):
        # Add this trace to set:
        set[n] = stf.get_trace( n )

    # calculate average and create a new section from it:
    stf.new_window( np.average(set,0) )

    # set peak cursors:
    if not stf.set_peak_mean(-1): return 0 # -1 means all points within peak window.
    if not stf.set_peak_start(peak_start): return 0
    if not stf.set_peak_end(peak_end): return 0

    # set base cursors:
    if not stf.set_base_start(base_start): return 0
    if not stf.set_base_end(base_end): return 0

    # measure everything:
    stf.measure()

    # calculate r_seal and return:
    return amplitude / (stf.get_peak()-stf.get_base())
```

50 Chapter 10. The Stimfit book of spells
10.1.2 Code commented

The stf functions stf.set_base_start() and stf.set_base_end() set the corresponding baseline cursors. The functions stf.set_peak_start() and stf.set_peak_end() set the corresponding peak cursors. These functions returns the Boolean True if the cursor was properly set in the desired position, and False if the cursors cannot be set (for example, they are out of the limits of the trace). This was designed to provide a control of the cursor positioning.

It is a good practice to test the correct position of the cursors with the conditional sentence if.

```python
>>> if not stf.set_base_start(base_start): return 0
```

- if the stf function returns False the if condition will be True (not False means True). Our defined function will be finished with return and give the value 0.
- In contrary, if the stf function returns True the if condition will be False (not True means False). In our function, the if condition will not be executed (does not read the return 0) and continue the operations.

We can enter directly the x-value (e.g ms) as argument, in stead of using the zero-based index of the sampling points. By adding the argument is_time==True to the function we set the cursor in the give time position:

```python
>>> if not stf.set_base_start(base_start,True) : return 0
```

Now base_starts should be given in units of x (i.e ms). This is more intuitive if you are using the stf interface.

10.1.3 Usage

Now, you can use this function for different purposes. For example, you may want to test the value of the series resistance in response to a 5 mV hyperpolarizing pulse. First, let’s assume that your recording has the current peak between the 10700 and 10999 sampling points. You should set the baseline (for example between 0 and 999) and then peak between 10700 and 10999. After that, and given that 5 mV is the voltage difference, you simply type:

```python
>>> spells.resistance(0,999,10700,1999,-5)
```

Note that charlie.py has a routine called r_in(amplitude=-5) that does exactly this.

In the same way, if you wanted to calculate the value of the seal resistance (assuming this is the smallest resistance in the circuit, so no current will flow through any other resistance), you could test it with a larger voltage pulse.

```python
>>> spells.resistance(0,199,1050,1199,50)
```

Again, the file charlie.py has a routine called r_seal(50) to calculate the seal resistance. Just change the parameters (baseline and peaks) to adapt it for your recordings.

10.2 Running mean

Author Jose Guzman

Date March 21, 2014

The running mean (or running average) is simple way to smooth the data. Given a certain set of points, a running average will create a new set of data points which will be computed by adding a series of averages of different subsets of the full data set.
Given for example a sequence $X$ of $n$ points, we can create a new set of data points $S$ of length $n$ by simply taking the average of a subset of $w$ points from the original data set for every point $S_i$ within the set:

$$S_i = \frac{1}{w} \sum_{j=i}^{w+i} X_j$$

### 10.2.1 The running mean function

The following Python function calculates the running mean of the current channel. Both trace and channel can be selected as zero-based indices. The width of the running average (referred to here as binwidth) can be selected. The resulting average will appear in a new Stimfit window.

```python
# load main Stimfit module
import stf

# load NumPy for numerical analysis
import numpy as np

def rmean(binwidth, trace=-1, channel=-1):
    
    # loads the current trace of the channel in a 1D Numpy Array
    sweep = stf.get_trace(trace, channel)

    # creates a destination python list to append the data
    dsweep = np.empty((len(sweep)))

    # running mean algorithm
    for i in range(len(sweep)):
        if (len(sweep)-i) > binwidth:
            # append to list the running mean of 'binwidth' values
            # np.mean(sweep) calculates the mean of list
            # sweep[p0:p10] takes the values in the vector between p0 and p10 (zero-based)
            dsweep[i] = np.mean( sweep[i:(binwidth+i)] )
        else:
            # use all remaining points for the average:
```
Stimfit Documentation, Release 0.13.17

dsweep[i] = np.mean( sweep[i:] )

```
stf.new_window(dsweep)
```

### 10.2.2 Code commented

Stimfit commonly uses the value -1 to set the current trace/Channel. In this function the default argument values are -1 (see the function arguments `trace=-1` and `channel=-1`).

```python
>>> sweep = stf.get_trace(trace,channel)
```

`stf.get_trace()` simply imports the trace of the channel into a 1D-Numpy array that we called sweep. The default values provided by the function are -1. This means that by default, the current trace/channel will be imported.

We create a new stf window with the following

```python
>>> stf.new_window(dsweep)
```

where dsweep is the 1D-NumPy array obtained after performing the running average.

### 10.2.3 Usage

To perform the running average of 10 sampling points of the current trace, simply type:

```python
>>> spells.rmean(10)
```

A new window with the running mean will appear.

### 10.3 Calculations on selected traces

**Author** Jose Guzman  
**Date** March 21, 2014

A widely used feature of Stimfit is the selection of some traces of interest within a file to make some calculations on them (average, peaks, amplitudes, etc.). The batch-analysis of Stimfit does precisely that. However, in some cases we can enhance its possibilities writing our custom functions in Python for the selected traces.

In Stimfit, the indices of selected traces can be easily retrieved using `stf.get_selected_indices()`. This function returns a tuple with the selected indices.

```python
>>> stf.get_selected_indices()
```

```python
>>> (1,2,3) # traces 2,3 and 4 were selected
```

In this case, we selected the 2nd, 3rd and 4th trace in the file (note the zero-based index!).

The routine described below performs a simple algorithm only on the traces selected previously (either with the menu bar or with typing S). I’ve chosen a very simple calculation (amplitude of the signal) for didactic purposes, but a more complex function can be written.
10.3.1 Using selected traces

In the following function we calculate the amplitude of the signal of the selected traces. One of the arguments of the function (\texttt{trace=None}) will select the trace that we want to use to make the calculation. Note that this is an optional argument; by default it will accept the current trace (or sweep) of the file, but if not, you can enter the zero-based index of the traces in the channel.

The amplitude function will be calculated based on the traces selected by \texttt{trace}. Here is the function:

```python
# stimfit python module
import stf

def get_amplitude(base, peak, delta, trace=None):
    """ Calculates the amplitude deviation (peak-base) in units of the Y-axis

    Arguments:
    base -- Starting point (in ms) of the baseline cursor.
    peak -- Starting point (in ms) of the peak cursor.
    delta -- Time interval to calculate baseline/find the peak.
    trace -- Zero-based index of the trace to be processed, if None then current
              trace is computed.

    Returns:
    A float with the variation of the amplitude. False if

    Example:
    get_amplitude(980,1005,10,i) returns the variation of the Y unit of the trace i between
    peak value (10050+10) msec and baseline (980+10) msec
    """

    # sets the current trace or the one given in trace
    if trace is None:
        sweep = stf.get_trace_index()
    else:
        if typ(trace) != int:
            print "trace argument admits only intergers"
            return False
        sweep = trace

    # set base cursors:
    if not(stf.set_base_start(base,True)): return False # out-of range
    if not(stf.set_base_end(base+delta,True)): return False

    # set peak cursors:
    if not(stf.set_peak_start(peak,True)): return False # out-of range
    if not(stf.set_peak_end(peak+delta,True)): return False

    # update measurements
    stf.set_trace(sweep)

    amplitude = stf.get_peak()-stf.get_base()

    return amplitude
```

54 Chapter 10. The Stimfit book of spells
10.3.2 Code commented

None is a Python built-in constant. It is used in to represent the absence of a value. Therefore, in our example, when the argument trace is empty (its value is None) we will simply select the current trace with stf.get_trace_index() and store it in the variable sweep. If not, the variable sweep will take the value taken by trace. This is controlled by the following if-block within the function:

```python
if trace is None:
    sweep = stf.get_trace_index()
else:
    if type(trace) != int:
        print "trace argument admits only integers"
        return False
    sweep = trace
```

An additional if block inside the else instruction allows us to control that trace will be an integer.

```python
>>> if type(trace) != int:
```

If the argument traces is not an integer, the function will be cancelled and returns False.

Note that after setting the stf cursors, we update the measurements in the trace whose index is given by the local variable sweep with stf.set_trace().

10.3.3 Usage

The function accepts an optional trace argument. That means, that we do not need to declare it when using the function. In that case, the function will work on the current trace. For example, if we want to calculate the amplitude between a baseline between (500+10) msec and a peak between 750 and 760 msec on the current trace, we simply enter:

```python
>>> spells.get_amplitude(500, 750, 10)
```

To calculate the same amplitude in the trace number 10 (zero-based index is 9) we can type:

```python
>>> spells.get_amplitude(500, 750, 10, 9)
```

More interesting is to get the amplitude in the selected traces, we can pass the tuple of selected traces to the trace argument and thereby calculate the amplitude on our selected traces:

```python
>>> amplitudes_list = [spells.get_amplitude(500, 750, 10, i) for i in stf.get_selected_indices()]
```

In this way the tuple of selected indices is passed by the for loop to the function. Next, everything is wrapped in a Python list called amplitudes_list.

For further analysis in spreadsheet programs (Calc, Gnumeric, Excel or similar), the values can be printed into a table that allows to copy and paste the contents. stf.show_table() takes a dictionary as its first argument. The dictionary has to be composed of strings as keys and numbers as values. You could use it as follows:

```python
>>> mytable = dict()
>>> for i in stf.get_selected_indices(): mytable["Trace %.3d" % i] = amplitudes_list[i]
>>> stf.show_table(mytable)
```

Note that the dictionary will be sorted alphabetically according to its keys. Therefore, using “%.3d” is used to keep the table in the same order as the traces. If you wanted to print out more than one value for each trace, you could use stf.show_table_dictlist() that uses a similar syntax, but requires a list of numbers as the values of the dictionary.
10.4 Cutting traces

Author Jose Guzman

Date March 21, 2014

As described in The Python shell chapter of the Stimfit manual, a very often requested feature of Stimfit is to cut an original trace to show it in a presentation or publication. This feature, however, has been only integrated into the stf module, and not in the Stimfit main menu bar. With this, Stimfit preserves its user interface as clear and user-friendly as possible.

We can use the built-in stf function stf.new_window() to show a new stf window with the current trace within an interval. For example,

```python
>>> stf.new_window(stf.get_trace()[1600:3200])
```

presents a new window with the current trace between the sampling points 1600 and 3200. Remember that stf.new_window() takes a 1D-NumPy array as argument. To cut the trace within the desired limits, we have to slice it before with

```python
>>> stf.get_trace()[1600:3200]
```

Note that the index $i$ of a sampling point and the corresponding time $t$, measured from the start of the trace, are related as follows:

$$i = \frac{t}{\Delta t}$$

where the sampling interval $\Delta t$ can be obtained with the following function:

```python
>>> dt = stf.get_sampling_interval()
```

Then, if our sampling interval (dt) is 0.05 ms, the points selected correspond to 80 and 160 ms respectively. Alternatively, one could have thought about this command:

```python
>>> stf.get_trace()[80/dt:160/dt]
```

However this will not work. Slicing requires integers as argument and not floats (both 80/dt and 160/dt are floats). So we have to transform this ratios to integers with int(80/dt) and int(160/dt). Besides that, the float precision of python will play against us here. If we make dt = stf.get_sampling_interval and get a value dt = 0.05000000074505806 (rather than 0.05) then the corresponding values for 80/dt will be 1599.99. The function int() will take only the value without decimal and will take erroneously 1599 in stead of 1600. We have to use the python function round(float,ndigits) to round up this value.

```python
>>> pstart = int( round(80/dt) )  # now round(80/dt) = 1600.0 instead of 1599.99
>>> pend = int( round(160/dt) )  # now round(160/dt) = 3200.0 instead of 3199.99
>>> stf.get_trace()[pstart:pend]  # now the slicing within the integer values
```

Note: You can round up dt with round(dt,ndigits) or the ratio 80/dt with round(80/dt,ndigits) (with ndigits=2 for example). In any case, do it always before int() takes the integer part of the real number.

10.4.1 The cutting traces function

In the chapter devoted to Python (The Python shell) in Stimfit manual you can find a function to cut a given trace within the sampling points. This function is slightly different. As described above, we would take times and not sampling points as argument. After that, we will take list of traces and not a single trace to cut. This function will use stf.new_window_list() which takes a list of 1D-Numpy arrays to present a new stf window.
# load main Stimfit module
import stf

def cut_sweeps(start, delta, sequence=None):
    
    Cuts a sequence of traces and present them in a new window.

    Arguments:

    start -- starting point (in ms) to cut.
delta -- time interval (in ms) to cut
sequence -- list of indices to be cut. If None, every trace in the channel will be cut.

    Returns:
A new window with the traced cut.

    Examples:
cut_sweeps(200,300) cut the traces between t=200 ms and t=500 ms within the whole channel.
cut_sweeps(200,300,range(30,60)) the same as above, but only between traces 30 and 60.
cut_sweeps(200,300,stf.get_selected_indices()) cut between 200 ms and 500 ms only in the selected traces.

    ""

    # select every trace in the channel if not selection is given in sequence
if sequence is None:
    sequence = range(stf.get_size_channel())

    # transform time into sampling points
dt = stf.get_sampling_interval()
pstart = int(round(start/dt))
pdelta = int(round(delta/dt))

    # creates a destination python list
dlist = [ stf.get_trace(i)[pstart:(pstart+pdelta)] for i in sequence ]

    return stf.new_window_list(dlist)

10.4.2 Code commented

We provide some flexibility with the argument sequence. If we do not give any argument to sequence, we will select every trace in the channel with the function stf.get_size_channel(), which returns the number of traces in the channel.

if sequence is None:
    sequence = range(stf.get_size_channel())

Finally we add to the list the 1D-NumPy arrays whose index is described in the sequence.
dlist = [ stf.get_trace(i)[pstart:(pstart+pdelta)] for i in sequence ]

and slice the 1D-NumPy array as described above.
10.4.3 Usage

In any case, a new stf window with the traces cut will appear

```python
>>> spells.cut_sweeps(200,300)
```

will create a new window with all the traces of the channel cut between t=200 ms and t=500 ms.

```python
>>> spells.cut_sweeps(200,300,range(30,60))
```

will create a new window with the same selection, but only between the traces 30 and 60.

```python
>>> spells.cut_sweeps(200,300,stf.get_selected_indices())
```

will create a new window with the cut traces only if they were previously selected.

10.5 Event counting

**Author** Jose Guzman

**Date** March 21, 2014

Counting the number of events (for example action potentials) within a time window is a very common task in electrophysiology. In its simplest form, the user would like to know how many spikes occur following the onset of a stimulus (i.e. current injection). We can write a simple Python function which automatically performs this calculation with a simple event detection routine.

**Note:** Stimfit has built-in functions to count the number of events (i.e action potentials). From the menu, select Analysis -> event detection-> threshold crossing... or alternatively with the Analysis-> Batch analysis->threshold crossing. However, this Python script allows for more flexibility while counting events, such as detecting positive- or negative-going events.

10.5.1 The counter event function

The following function counts the number of events (e.g action potentials) by detecting upward (up=True) or downward (up=False) threshold-crossings.

```python
# load main Stimfit module
import stf
def count_events(start, delta, threshold=0, up=True, trace=None, mark=True):
    """
    Counts the number of events (e.g action potentials (AP)) in the current trace.
    Arguments:
    start        -- starting time (in ms) to look for events.
    delta        -- time interval (in ms) to look for events.
    threshold    -- (optional) detection threshold (default = 0).
    up           -- (optional) True (default) will look for upward events, False downwards.
    trace        -- (optional) zero-based index of the trace in the current channel, 
                    if None, the current trace is selected.
    mark         -- (optional) if True (default), set a mark at the point of threshold crossing
    Returns:
    An integer with the number of events.
    """
```

58 Chapter 10. The Stimfit book of spells
Examples:

- `count_events(500,1000)` returns the number of events found between \(t=500\) ms and \(t=1500\) ms above 0 in the current trace and shows a stf marker.
- `count_events(500,1000,0,False,-10,i)` returns the number of events found below -10 in the trace \(i\) and shows the corresponding stf markers.

```
# sets the current trace or the one given in trace.
if trace is None:
    sweep = stf.get_trace_index()
else:
    if type(trace) != int:
        print "trace argument admits only integers"
        return False
    sweep = trace

# set the trace described in sweep
stf.set_trace(sweep)

# transform time into sampling points
dt = stf.get_sampling_interval()

pstart = int( round(start/dt) )
pdelta = int( round(delta/dt) )

# select the section of interest within the trace
selection = stf.get_trace()[pstart:(pstart+pdelta)]

# algorithm to detect events
EventCounter,i = 0,0  # set counter and index to zero

# choose comparator according to direction:
if up:
    comp = lambda a, b: a > b
else:
    comp = lambda a, b: a < b

# run the loop
while i<len(selection):
    if comp(selection[i],threshold):
        EventCounter += 1
        if mark:
            stf.set_marker(pstart+i, selection[i])
        while i<len(selection) and comp(selection[i],threshold):
            i+=1  # skip values if index in bounds AND until the value is below/above threshold again
    else:
        i+=1

return EventCounter
```

### 10.5.2 Code commented

The traces are selected by the optional argument `trace`, as explained in *Calculations on selected traces*. The algorithm to detect action potentials requires some familiarity with Python iterations but it is easy to understand.
while i<len(selection):
    if comp(selection[i], threshold):
        EventCounter +=1
        while i<len(selection) and comp(selection[i], threshold):
            i+=1  # skip if index in bounds AND values until the value is below or above threshold again
    else:
        i+=1

The while loop allows us to move within the indices of the array called selection. We insert an if-block inside to test whether the threshold is crossed at [i]. In this case we will add 1 to the counter (EventCounter +=1) and move to the second while loop.

while i<len(selection) and comp(selection[i], threshold):
    i+=1  # skip if index in bounds AND values until the value is below or above threshold again

This second loop is very important. The index moves within the array until the value crosses the threshold again in the other direction. We have to skip every value until the threshold is crossed again. If we do do this while here, the if condition will be True for all values after the threshold crossing, and the counter would give us the number of sampling points between threshold crossings (and not the number of events). Finally, it is important to do this loop whenever the index is inside the limits of the selection.

Finally, if the condition is not true, the else statement will move the index one position next in the array. The main while loop (while i<len(selection)) will evaluate for every point if the threshold is achieved. Note that preserving the Python indentation is extremely important here.

**Note:** Do not try to write while loops in the embedded python console of Stimfit unless you are very familiar with while loops in Python or in any other language. While loops, if written incorrectly, may run infinite iterations and block the Python terminal of Stimfit. For that reason, it is a good idea to explore while loops in an independent python terminal before using them in Stimfit.

### 10.5.3 Usage

As in *Calculations on selected traces* we can use the function in different ways:

```python
>>> spells.count_events(start=500,delta=1000)
```

will return the number of events above 0 mV in the current trace/channel between t=500 ms and t=1500 ms, and shows a blue stf marker.

```python
>>> spells.count_events(start=500,delta=1000,threshold=-40,up=False,trace=10,mark=False)
```

this will look for events below the value -40 but not in the current trace, only in the trace 11 (zero-based index is 10) in the downwards direction. Here a blue marker around the point found bellow the threshold will be shown too. Note that functions with a large number of arguments are difficult to remember. You can always change the order of the arguments if you describe the arguments in the function. For example, the following sentence has the same effect as the one above, but shows a different argument order:

```python
>>> spells.count_events(threshold=-40,start=500,up=False,delta=1000,mark=False,trace=10)
```

If you want to create a list of events with the events found in a selection of traces, you can simply type:

```python
>>> spikes_list= [spells.count_events(500,1000,0,True,i,False) for i in stf.get_selected_indices()]
```

this will create a Python list with the number of events (e.g spikes) found between t=500ms and t=1500ms above 0 in the selected traces and no marker will be shown. In the same way as described previously in , you can create a table to copy the results.
this creates a table with 2 columns with the trace number a number of spikes found previously.

Obviously, the function could be extended to return the time points of threshold crossings so that the interspike intervals can be calculated. This is left as an exercise to the reader.

---

**Note:** Use the `stf.erase_markers()` to clean the blue markers on the main stf window. If not, every time that you call the routine in the given trace, a series of blue markers showing the crossing points of the different threshold will overlap with each other. Alternatively, you can add `stf.erase_markers()` in the beginning of `count_events()` to delete any marker presented previously:

---

### 10.6 Object oriented programming with Stimfit

**Author** Jose Guzman  
**Date** March 21, 2014

Object-oriented programming (OOP) is a software philosophy where the problems are solved with objects rather than with simple functions. These objects behave similarly to objects in the physical world. For example, image you may want to travel from Freiburg (Germany) to London (UK). For that, you will need to use a transport (e.g car, airplane, etc...) which certain properties (e.g airplanes are much faster than cars, whereas cars are more flexible in terms of schedule). The key concept here is that an object has distinct attributes (associated variables) and methods (associated functions). Interestingly, object attributes are extremely dynamic. They may change as they are involved in different tasks. Because object have an state they provide much more versatility to solve problems than static function alone. Thus, in OOP the design of the code is near to the question, and much more far away from the machine and the hardware details.

**Note:** moving from the procedural/functional programming paradigm to the object oriented programming paradigm requires some mind re-wiring. In principle, everything what you can do in OOP can be done in functional programming. However, large programs would benefit from the OOP approach as their code is more reusable. Abstraction level is higher, because we will work with concepts rather than with complex software algorithms.

There are some key concepts in OOP.

- **Class** is the blueprint used to generate objects. It contains the instructions to generate an object. Although a class describes how to create an object, it may not describe the particular properties of an object.

- **Object** is the practical application of the class. It combines state (i.e variables) and behavior (i.e functions, algorithms).

- **Encapsulation** because objects are exposed to the user, attributes and functions may be easily modified without permission. In order to prevent accidental overwriting, some attributes and methods may be hidden to the user, and this is called encapsulation.

- **Inheritance** a common mistake when creating classes is to define a class for every object that we want to use. To avoid extreme redundancy, classes may inherit properties from other classes, providing thereby a way of creating more complex objects without having to re-write all the known instructions of a class inside another class.
10.6.1 Classes and objects

We will start with a basic example to start using objects in the embedded Python shell. We will use an object to collect the sampling interval and trace index in our recording. For that, we will create a class, which defines an object that collects this information at once for us. To create that class we can use this code:

```python
import stf
class Trace(object):
    ""
    A class to generate a Trace object with contains attributes of the current trace.
    ""
    def __init__(self, owner):
        ""
        create instance with dt and trace index and as attributes.
        Arguments:
        type -- string containing the name of the user
        ""
        self.dt = stf.get_sampling_interval()
        self.trace = stf.get_trace_index()
        self.owner = owner
```

We can save this class in a file called test.py and import into our Python session. After importing the file, nothing will happen. This is because we simply loaded the class (i.e instructions of how to create the object), but not the object itself. Now, we can create an object called `myTrace` with the instructions described in that class as follows:

```python
>>> myTrace= test.Trace('root') # test.py contains the class Trace()
```

`myTrace` is now my particular object. It was created with the instructions given in the class `Trace`. This is commonly refereed as `myTrace is an instance of the class Trace`.

10.6.2 Object attributes

Object attributes can be accessed with the dot notation. To test the attributes of “myTrace” we simply type:

```python
>>> myTrace.dt
0.05000000074505806
>>> myTrace.trace
7
>>> myTrace.owner
'root'
```

This tells us that the trace 8 has a sampling rate of 0.05 msec. The owner was set at construction, and it is a user called root.

10.6.3 Encapsulation

As you can see bellow, nothing would prevent us to assign a new value to any of the current object attributes. For example, if we now type:

```python
>>> myTrace.dt = 3
```

This potentially very dangerous (imagine the consequences of setting the sampling rate to 3 in further calculations). For that reason, it is a very good programming practice to hide some object attributes to the user. This is called
encapsulation. To hide the attributes of “myTrace”, we have just to insert a single underscore before the attribute in
the class. These objects are **private** which simply means, “look, but do not touch!”

**Note:** Python strongly relies on convention rather than on enforcement. For example, encapsulated attributes are not
really private (i.e user can overwrite them if necessary), but the underscore notation is used to indicate internal use
only. If you find a good reason to overwrite them, Python is not going to stop you. However, it is a good programming
practice to keep the Python conventions if you want to share your programs with other users.

Additionally, we could give the user the opportunity to retrieve these values without the dot notation by simply creating
some functions available to this object. These would be the object methods. For example, we can create 2 functions
called `get_sampling_interval()` and `get_trace_index()` inside the class. These are the methods of the object.

```python
import stf
class Trace(object):
    """
    A class to generate a Trace object which contains attributes
    of the current trace.
    """
    def __init__(self, owner):
        """
        create instance with dt and trace as attributes.
        Arguments:
        type -- string containing the name of the user
        """

        # please, note that underscore attributes are private
        self._dt = stf.get_sampling_interval()
        self._trace = stf.get_trace_index()
        self.owner = owner

    def get_sampling_interval(self):
        """ get sampling interval ""
        return self._dt

    def get_trace_index(self):
        """ get trace index"
        return self._trace
```

Now we can import/reload test.py and create a new object.

```python
>>> myTrace2 = test.Trace('user')
```

and test its attributes as follows:

```python
>>> myTrace2.get_sampling_interval()
0.05000000074505806
>>> myTrace2.get_trace_index()
7
>>> myTrace2.owner
>>> 'user'
```

**Note:** do not confuse methods/attributes that start and end with two underscores with those which only start with a
single underscores. The firsts are special methods and customize the standard python behavior (like `__init__`), whereas
the lasts are encapsulated methods.

---

10.6. Object oriented programming with Stimfit
10.6.4 Dynamic nature of objects

As soon as we move through the recording, the trace index may change. However, if we call the methods get_trace_index() or get_sampling_interval() of the object they will return the object attributes in the old status. We need a new method to update the object attributes every time that we change the trace. This is where the dynamic nature of the objects come handy.

```python
import stf
class Trace(object):
    ""
    A class to generate a Trace object which contains attributes
    of the current trace.
    ""
    def __init__(self, owner):
        ""
        create instance with dt and trace as attributes.
        Arguments:
        type -- string containing the name of the user
        ""
        self.owner = owner
        self.update()
    def update(self):
        ""
        update dt and trace according to the current position
        ""
        self._trace = stf.get_trace_index()
        self._dt = stf.get_sampling_interval()
    def get_sampling_interval(self):
        ""
        get sampling interval
        ""
        return self._dt
    def get_trace_index(self):
        ""
        get trace index
        ""
        return self._trace
```

After reloading this class, and creating “myTrace” we can use the update() method. This simply collects the current trace index and sampling interval. If we change the trace or even the window, we have to call update() again to retrieve the current index and sampling interval.

```python
>>> myTrace3 = test.myTrace('user')
>>> myTrace3.get_trace_index()
0
>>> stf.set_trace_index(3)
>>> myTrace3.get_trace_index() # this returns the old state!!!
0
>>> myTrace3.update() # update attributes
>>> myTrace3.get_trace_index() # this returns the updated state!!!
3
```

10.6.5 Class inheritance

Object-oriented languages like Python support class inheritance. This means that we can inherit attributes and methods from a pre-existing class. Thus, we do not need to rewrite again this code. We can simply inherit from another class
(called mother class). To inherit code from another class, we have to add the name of the mother class in the class headline. For example:

```python
>>> class Channel(Trace):
```

The class `Channel` will automatically inherit the code from the class `Trace`. We say that the class `Channel` is a subclass of the superclass `Trace`. If we want to extend the functionality of our new class, we can add new methods and/or attributes, or even overwrite the existing inherited methods. We can create a new `Channel` class in the same file like this:

```python
class Channel(Trace):
    
    A class derived from Trace class
    
    def __init__(self, owner):
        Trace.__init__(self, owner)  # let Trace to get owner
        self._channel = stf.get_channel_index()

    def update(self):
        
        Trace.update(self)  # update dt and trace
        self._channel = stf.get_channel_index()

    def get_channel_index(self):
        
        return self._channel
```

From this example we can see that the class `Channel` not only inherits, but extends its functionality to the current channel. We have not only functions to calculate the sampling rate (`get_sampling_rate()` and `trace.get_trace_index()` but also a new function called `get_channel_index()`). A new attribute is also added (`self._channel`). The `update()` function that we used to update the sampling interval and the trace in the `Trace` class, is now extended to include the updated channel number. We can now test it:

```python
>>> stf.set_trace(3), stf.set_channel(1)

>>> True, True  # remember, True if successful

>>> myChannel = test.Channel('user')  # create a instance of Channel

>>> myChannel.get_trace_index()  # this methods is inherited from Trace

>>> 3

>>> myChannel.get_sampling_interval()  # inherited from Trace

>>> 0.05000000074505806

>>> myChannel.get_channel_index()  # this is only for Channel

>>> 1

We can change trace and channel to test the update function

```python
>>> stf.set_trace(5), stf.set_channel(0)

>>> True

>>> myChannel.get_trace_index()

>>> 5  # this value was updated!

>>> myChannel.get_sampling_interval()  # inherited from Trace

>>> 0.05000000074505806

>>> myChannel.get_channel_index()  # this is only for Channel

>>> 0  # this is the updated value!
```

Finally, we can check if an object belongs to certain class with the function `isinstance(object, class)`. For example

```python
>>> isinstance(myChannel, test.Channel)

>>> True

>>> isinstance(myChannel, test.Trace)

>>> True  # This is because Channel inherits from Trace
```
>>> isinstance(myTrace, test.Channel)
>>> False

or we can use the \_\_class\_\_ method included in every instance to check the type of the object:

>>> myChannel.__class__
>>> <class 'test.Channel'>

we can get this class definition as string with:

>>> myChannel.__class__.__name__
>>> 'Channel'

### 10.7 Calculating latencies

**Author** Jose Guzman  
**Date** March 21, 2014

Stimfit was originally used to calculate synaptic latencies (Katz and Miledi, 1965 \(^1\)) but now can be used to calculate synaptic latencies and latencies between events or action potentials in the same or between different channels (see Latency measurements in the Stimfit manual). Stimfit also provides a very useful collection of Python functions which allow us to easily adapt the latency calculation for our particular conditions. We will use these functions to calculate the latency between two signals in two different channels (e.g. one corresponding to the soma, and another to the dendrite). We will use the object oriented programming paradigm (OOP) to solve this problem and applied it in the embedded Python shell of Stimfit.

#### 10.7.1 The Spike class

We will create an object to calculate basic action potential (AP) kinetics in the current/active channel. AP peak and half-width will be calculated from a threshold (in mV/ms) defined by the user, as described in Stuart et al. (1997) \(^2\). In principle, this can be easily adjusted in the Stimfit menu toolbar (see Edit->Cursor settings and select Peak tab). However, as the number of traces to analyze increase, the manipulation of the menu becomes unnecessarily repetitive and prone to errors. We will use the object to access different AP parameters (i.e baseline, peak, half-width and maximum rise-time) all them calculated from the threshold value. Note these values are accessible in Stimfit result table (see Fig. 9 in the Stimfit manual), but we will access them within Python.

**Note:** Once the threshold is set, it can be accessed in terms of time with \(\text{stf.get_threshold_time}()\) or voltage with \(\text{stf.get_threshold_value}()\).

Additionally, some other methods will be necessary to calculate the AP latencies. For example, we may want to calculate onset latency (i.e time difference between the beginning of the action potential in two different recordings) or peak latency (i.e difference in time between the peak of two APs in different recordings). More interestingly, we can calculate the half-width latency according to Schmidt-Hieber et al., (2008) \(^3\). In this last case, the AP latency is calculated by the time different between the times of the AP at its half-maximal amplitudes.

import stf
from math import ceil, floor

class Spike(object):
    
    A collection of methods to calculate AP properties from threshold (see Stuart et al., 1997). Note that all calculations are performed in the active/current channel!!!

        def __init__(self, threshold):
            Create a Spike instance with sampling rate and threshold measurements are performed in the current/active channel!!!
            Arguments:
            threshold -- slope threshold to measure AP kinetics
            
            self._thr = threshold
            # set all the necessary AP parameters at construction
            self._updateattributes()

        def _updateattributes(self):
            update base, peak, t50, max_rise and tamplitude
            
            self.base = self.get_base() # in Stimfit is baseline
            self.peak = self.get_peak() # in Stimfit peak (from threshold)
            self.t50 = self.get_t50()  # in Stimfit t50
            self.max_rise = self.get_max_rise() # in Stimfit Slope (rise)
            self.thr = self.get_threshold_value() # in Stimit Threshold

            # attributes necessary to calculate latencies
            self.tonset = self.get_threshold_time()
            self.tpeak = self.get_tamplitude()
            self.t50_left = self.get_t50left()

        def update(self):
            update current trace sampling rate, cursors position and measurements (peak, baseline & AP kinetics) according to the threshold value set at construction or when the object is called with a threshold argument.
            
            # set slope
            stf.set_slope(self._thr) # on stf v0.93 or above

            # update sampling rate
            self._dt = stf.get_sampling_interval()

            # update cursors and AP kinetics (peak and half-width)
            stf.measure()

        def __call__(self, threshold=None):
            update AP kinetic parameters to a new threshold in the current trace/channel
            threshold (optional) -- the new threshold value
Examples:

```python
dend = Spike(40)  # set the spike threshold at 40mV/ms
dend(20)  # now we set the spike threshold at 20mV/ms
```

The AP parameters will be thereby updated in the current trace/channel. This method allow us to use the same object to calculate AP latencies in different traces.

```python
if threshold is not None:
    self._thr = threshold  # set a new threshold

self.update()  # update dt and sampling rate
self._updateattributes()
```

def get_base(self):
    
    Get baseline according to cursor position in the given current channel/trace

    
    
    self.update()

    return stf.get_trace(trace = -1 ,channel = -1)[stf.get_base_start():stf.get_base_end()+1].mean()

def get_peak(self):
    
    calculate peak measured from threshold in the current trace, (see Stuart et al (1997))

    stf.set_peak_mean(1)  # a single point for the peak value
    stf.set_peak_direction("up")  # peak direction up

    self.update()

    peak = stf.get_peak()-stf.get_threshold_value()
    return peak

def get_t50(self):
    
    calculates the half-width in ms in the current trace

    self.update()

    # current t50’s difference to calculate half-width (t50)
    return (stf.t50right_index()-stf.t50left_index())*self._dt

def get_max_rise(self):
    
    maximum rate of rise (dV/dt) of AP in the current trace, which depends on the available Na+ conductance, see Mainen et al, 1995, Schmidt-Hieber et al, 2008

    self.update()
pmaxrise = stf.maxrise_index() # in active channel

trace = stf.get_trace(trace = -1, channel = -1) # current trace

dV = trace[int(ceil(pmaxrise))]-trace[int(floor(pmaxrise))]

return dV/self._dt

def get_tamplitude(self):
    """ return the time at the peak in the current trace"
    
    # stf.peak_index() does not update cursors!!!
    self.update()

    return stf.peak_index()*self._dt

def get_t50left(self):
    """ return the time at the half-width ""
    self.update()

    return stf.t50left_index()*self._dt

def show_threshold(self):
    """ return the threshold value (in mV/ms) set at construction or when the object was called"
    return self._thrr

def get_threshold_value(self):
    """ return the value (in y-units) at the threshold ""
    self.update() # stf.get_threshold_value does not update
    return stf.get_threshold_value()

def get_threshold_time(self):
    """ return the value (in x-units) at the threshold ""
    self.update()
    return stf.get_threshold_time('True')

10.7.2 Code commented

Note that all methods but show_threshold() are preceded by self.update(). This is to update the sampling rate of the current trace (necessary to transform index points into time) and the position of the cursors. In this way, we are sure that every function will return the values according to the current trace and the update position of the cursors.

Because we want to group all the AP parameters (i.e baseline, peak, half-width and max rise) of a single trace together, we set the object attributes to the following values:

def _updateattributes(self):
    self.base = self.get_base() # in Stimfit is baseline
    self.peak = self.get_peak() # in Stimfit peak (from threshold)
    self.t50 = self.get_t50() # in Stimfit t50
    self.max_rise = self.get_max_rise() # in Stimfit slope (rise)

    # attributes necessary to calculate latencies
    self.tpeak = self.get_threshold_time()
    self.tamplitude = self.get_tamplitude()

These values refer to the trace present in the current Stimfit window when the object was created. This will allow us
to store them for future calculations.

Note for example, that if we create an object with threshold 40

```python
>>> myspike = Spike(40)
```

and move after that to another trace, we could calculate the difference between the peaks of the previous and present trace as follows:

```python
>>> myspike.peak - myspike.get_peak()
```

the former will give the peak value when in the trace where the object was created, and the later will return the peak in the current trace.

Additionally, we can decide to change the threshold value of the AP in a trace. For that, we can simply type:

```python
>>> myspike(20)
```

And now the Spike attributes will be updated with the new threshold in the current trace. The function `__call__` simply allows to call the object with a given argument, and we used it to set a different threshold and update the object attributes.

### 10.7.3 Usage

To use this class we have to create an object in the current trace with a threshold value as argument. Do not forget to set both baseline and peak cursors before creating the object.

```python
>>> soma = spells.Spike(50)
```

Now we can calculate the parameters with the methods available to this object. Note that these values change as we change the trace (i.e., we do not need to type `update()` or use `stf.measure()`). This means that the method `soma.get_base()` will return different values if we call it in different traces or move the cursors. Compare the values obtained with the functions with the corresponding values in the result table of Stimfit.

```python
>>> soma.get_base() # correspond to baseline in the results table
>>> soma.get_peak() # correspond to Peak (from threshold) in the results table
>>> soma.get_t50() # correspond to t50 in the results table
>>> soma.get_max_rise() # correspond to slope (rise) in the results table
>>> soma.get_threshold_value() # correspond to Threshold in the results table
```

Additionally, we have methods like `get_tamplitude()`, `get_threshold()` and `get_threshold_time()` to calculate latencies with different methods. For example, if we have two different Spike objects, one corresponding to the soma and the other corresponding to the dendrite, we could calculate calculate the latencies with the 3 following methods.

**1. Onset latency:** this is the latency between the beginning of 2 APs. We can calculate it as follows:

```python
>>> soma = spells.Spike(50) # threshold of somatic AP is 50mV/ms
>>> stf.set_channel(1) # move to channel 2
>>> dend = spells.Spike(20) # threshold for dendritic AP is 20mV/ms
>>> latency = dend.tonset - soma.tonset
```

**2. Peak latency:** this is the latency between the peaks of 2 APs. Similarly to the previous calculate, we can use:

```python
>>> latency = dend.tpeak - soma.tpeak
```

**3. T50 latency:** this method is included in the Edit option of the Stimfit menu toolbar. However, this menu assumes that both thresholds are the same. If we want to set different latencies for the calculation of the t50 latency, we can the Spike property called `t50left`: 

```python
latency = dend.t50_left - soma.t50_left

You can find the class Spike described above in your current Stimfit version. To use it, you can simply import it from the spells module with the following command:

```python
>>> from spells import Spike
>>> soma = Spike(50)
>>> dend = Spike(20)  # in a different trace/window
```

Additionally, the spells module contains a function which creates a result table (see Figure bellow) with all the AP kinetic parameters described previously, and the latency calculated with the 3 methods described here. Once the soma and dend objects are created with the class Spike, we can use the latency function:

```python
>>> from spells import latency
>>> latency(soma, dend)  # both soma and dend are Spike objects
```

note that this function assumes that you set the cursors property in your trace, and that the dendritic and somatic AP are already initialized and contains the AP attributes of some trace.

<table>
<thead>
<tr>
<th></th>
<th>Threshold (mV/ms)</th>
<th>Onset (ms)</th>
<th>Onset (mV)</th>
<th>Baseline (mV)</th>
<th>AP Peak (mV)</th>
<th>AP Peak (ms)</th>
<th>Half-width (ms)</th>
<th>Vmax (mV/ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Soma</strong></td>
<td>50.0000</td>
<td>512.4250</td>
<td>-44.7693</td>
<td>-60.0942</td>
<td>66.8396</td>
<td>513.2000</td>
<td>1.6108</td>
<td>158.2031</td>
</tr>
<tr>
<td><strong>Dend</strong></td>
<td>20.0000</td>
<td>512.6750</td>
<td>-37.2498</td>
<td>-60.0545</td>
<td>29.7383</td>
<td>513.7000</td>
<td>2.1776</td>
<td>54.6875</td>
</tr>
<tr>
<td>latency</td>
<td>0.2500</td>
<td>0.5000</td>
<td></td>
<td>0.5000</td>
<td></td>
<td></td>
<td>5.3033</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.1: Result table returned by the AP.calc() function.

**Note:** In the figure, the cell highlighted represent the latency calculated as the difference between the times at the half-width of the AP (as we did previously), and NOT the difference between the half-widths!!!
The `stf` module defines the following functions:

- **`stf.align_selected(alignment, active=False)`**
  Aligns the selected traces to the index that is returned by the alignment function, and then creates a new window showing the aligned traces.

  **Arguments:**
  
  - `alignment` – The alignment function to be used. Accepts any function returning a valid index within a trace. These are some predefined possibilities: `maxrise_index` (default; maximal slope during rising phase), `peak_index` (Peak of an event), `foot_index` (Beginning of an event), `t50left_index`, `t50right_index` (Left/right half-maximal amplitude)
  
  - `active` – If True, the alignment function will be applied to the active channel. If False (default), it will be applied to the inactive channel.
  
  - `zeropad` – Not yet implemented: If True, missing parts at the beginning or end of a trace will be padded with zeros after the alignment. If False (default), traces will be cropped so that all traces have equal sizes.

- **`stf.check_doc(*args)`**
  Checks whether a file is open.

  **Returns:**
  
  True if a file is open, False otherwise

- **`stf.close_all(*args)`**
  Closes all open files.

  **Returns:**
  
  True if all files could be closed.

- **`stf.close_this(*args)`**
  Closes the currently active file.

  **Returns:**
  
  True if the file could be closed.

- **`stf.cut_traces(pt)`**
  Cuts the selected traces at the sampling points in `pt_list` and shows the cut traces in a new window.
Returns:
True upon success, False upon failure.

`stf.cut_traces_multi(pt_list)`
Cuts the selected traces at the sampling points in `pt_list` and show the cut traces in a new window.

Returns:
True upon success, False upon failure.

`stf.erase_markers()`
Delete the marks created with `set_marker()`

`stf.file_open(*args)`
Opens a file.

Arguments:
`filename` – The file to be opened. On Windows, use double backslashes ("\") between directories to avoid conversion to special characters such as “t” or “n”. Example usage in Windows:

```python
>>> file_open("C:\\data\\datafile.dat").
```

Example usage in Linux:

```python
>>> file_open("/home/cs/data/datafile.dat").
```

This is surprisingly slow when called from python. Haven’t figured out the reason yet.

Returns:
True is the file could be opened, False otherwise.

`stf.file_save(*args)`
Saves a file.

Arguments:
`filename` – The file to be saved. On Windows, use double backslashes ("\") between directories to avoid conversion to special characters such as “t” or “n”.

Example usage in Windows:

```python
>>> file_save("C:\\data\\datafile.dat")
```

Example usage in Linux:

```python
>>> file_save("/home/cs/data/datafile.dat")
```

This is surprisingly slow when called from python. Haven’t figured out the reason yet.

Returns:
True if the file could be saved, False otherwise.

`stf.foot_index(active=True)`
Returns the zero-based index of the foot of an event in the active channel. The foot is the intersection of an interpolated line through the points of 20 and 80% rise with the baseline. Uses the currently measured values, i.e. does not update measurements if the peak or base window cursors have changed.

Arguments:
`active` – If True, returns the current index of the foot within the active channel. Only implemented for the active channel at this time. Will return a negative value and show an error message if `active` == False.
Returns:
The zero-based index of the foot of an event in units of sampling points. Interpolates between sam-
pling points. Returns a negative value upon failure.

```
stf.get_base(*args)
```

Returns the current baseline value. Uses the currently measured values, i.e. does not update measurements if
the peak or base window cursors have changed.

Returns:
The current baseline.

```
stf.get_baseline_method()
```

Gets the method used to compute the baseline.

Returns:
A string specifying the method to compute the baseline. Can be one of “mean” or “median”

```
stf.get_base_end(is_time=False)
```

Returns the zero-based index or the time point of the base end cursor.

Arguments:

```
is_time – If False (default), returns the zero-based index. If True, returns the time from the begin-
ing of the trace to the cursor position.
```

```
stf.get_base_start(is_time=False)
```

Returns the zero-based index or the time point of the base start cursor.

Arguments:

```
is_time – If False (default), returns the zero-based index. If True, returns the time from the begin-
ing of the trace to the cursor position.
```

```
stf.get_channel_index(active=True)
```

Returns the ZERO-BASED index of the specified channel.

Arguments:

```
active – If True, returns the index of the active (black) channel. If False, returns the index of the
inactive (red) channel.
```

```
stf.get_channel_name(index=-1)
```

Returns the name of the channel with the specified index.

Arguments:

```
index – The zero-based index of the channel of interest. If < 0, the name of the active channel will be
returned.
```

Returns:
the name of the channel with the specified index.

```
stf.get_filename(*args)
```

Returns the name of the current file.

```
stf.get_fit(trace=-1, channel=-1)
```

Get the waveform resulted from the fitting, if available.

Arguments:

```
trace – The zero-based index of the trace of interest. If negative, the name of the active trace will be
returned.
```
channel – The zero-based index of the trace of interest. If negative, the active channel will be used.

Returns:
A two dimensional NumPy array with the x-values of the fit in the first dimension and the y-values in the second dimension. None if no fit is available.

```
stf.get_fit_end(is_time=False)
```
Returns the zero-based index or the time point of the fit end cursor.

Arguments:

is_time – If False (default), returns the zero-based index. If True, returns the time from the beginning of the trace to the cursor position.

```
stf.get_fit_start(is_time=False)
```
Returns the zero-based index or the time point of the fit start cursor.

Arguments:

is_time – If False (default), returns the zero-based index. If True, returns the time from the beginning of the trace to the cursor position.

```
stf.get_halfwidth(active=True)
```
Returns the half-maximal amplitude of an event in the specified channel. Uses the currently measured values, i.e. does not update measurements if the peak or base window cursors have changed. Only implemented for the active channel.

Arguments:

active – If True, returns the current half-maximal amplitude within the active channel.

Returns:
The half-maximal amplitude in units of x-units. Returns a negative value upon failure.

```
stf.get_latency(*args)
```
Returns the latency value (in x-units) determined by the latency cursors set in the cursors settings menu. Call measure() or hit enter to update the cursors.

```
stf.get_latency_end_mode(*args)
```
Returns a string specifying the latency end mode. Can be one of “manual”, “peak”, “rise”, “foot” or “half”.

```
stf.get_latency_start_mode(*args)
```
Returns a string specifying the latency start mode. Can be one of “manual”, “peak”, “rise”, or “half”.

```
stf.get_maxdecay(*args)
```
Returns the the maximal slope of the decay between the peak cursors. Returns -1.0 upon error. Call measure() or hit enter to update the value.

```
stf.get_maxrise(*args)
```
Returns the the maximal slope of the rise between the peak cursors. Returns -1.0 upon error. Call measure() or hit enter to update the value.

```
stf.get_risetime(*args)
```
Returns the 20-80% rise time (in x-units) by calculation of the interpolated adjacent sampling points at 20% and 80% of the peak amplitude. Returns -1.0 upon failure. Call measure() or hit enter to update the value.

```
stf.get_risetime_factor(*args)
```
Returns the lower proportion factor used to calculate the rise time (e.g 0.2 if we calculate the 20–80% rise time).

```
stf.get_slope(*args)
```
Returns the slope value using the cursors described in the cursors setting dialog.
Returns:
The slope value

```
stf.get_peak(*args)
```
Returns the current peak value, measured from zero (!). Uses the currently measured values, i.e. does not update measurements if the peak or base window cursors have changed.

Returns:
The current peak value, measured from zero (again: !).

```
stf.get_peak_end(is_time=False)
```
Returns the zero-based index or the time point of the peak end cursor.

Arguments:

`is_time` – If False (default), returns the zero-based index. If True, returns the time from the beginning of the trace to the cursor position.

```
stf.get_peak_start(is_time=False)
```
Returns the zero-based index or the time point of the peak start cursor.

Arguments:

`is_time` – If False (default), returns the zero-based index. If True, returns the time from the beginning of the trace to the cursor position.

```
stf.get_recording_comment(*args)
```
Returns a comment about the recording.

```
stf.get_recording_date(*args)
```
Returns the date at which the recording was started as a string.

```
stf.get_recording_time(*args)
```
Returns the time at which the recording was started as a string.

```
stf.get_sampling_interval(*args)
```
Returns the sampling interval.

```
stf.get_selected_indices(...)
```
Returns a tuple with the indices (ZERO-BASED) of the selected traces.

```
stf.get_size_channel(channel=-1)
```
Retrieves the number of traces in a channel. Note that at present, stimfit only supports equal-sized channels, i.e. all channels within a file need to have the same number of traces. The channel argument is only for future extensions.

Arguments:

`channel` – ZERO-BASED index of the channel. Default value of -1 will use the current channel.

Returns:
The number traces in a channel.

```
stf.get_size_recording(*args)
```
Retrieves the number of channels in a recording.

Returns:
The number of channels in a recording.

```
stf.get_size_trace(trace=-1, channel=-1)
```
Retrieves the number of sample points of a trace.
Arguments:

*trace* – ZERO-BASED index of the trace. Default value of -1 will use the currently displayed trace. Note that this is one less than what is displayed in the drop-down list.

*channel* – ZERO-BASED index of the channel. Default value of -1 will use the current channel.

Returns:

The number of sample points.

```python
stf.get_trace(trace=-1, channel=-1)
```

Returns a trace as a 1-dimensional NumPy array.

Arguments:

*trace* – ZERO-BASED index of the trace within the channel. Note that this is one less than what is shown in the drop-down box. The default value of -1 returns the currently displayed trace.

*channel* – ZERO-BASED index of the channel. This is independent of whether a channel is active or not. The default value of -1 returns the currently active channel.

Returns:

The trace as a 1D NumPy array.

```python
stf.get_trace_index(...)
```

Returns the ZERO-BASED index of the currently displayed trace (this is one less than what is shown in the combo box).

```python
stf.get_trace_name(trace=-1, channel=-1)
```

Returns the name of the trace with the specified index.

Arguments:

*trace* – The zero-based index of the trace of interest. If < 0, the name of the active trace will be returned.

*channel* – The zero-based index of the channel of interest. If < 0, the active channel will be used.

Returns:

the name of the trace with the specified index.

```python
stf.get_threshold_time(is_time=False)
```

Returns the crossing value of the threshold slope. Note that this value is not update after changing the AP threshold. Call `measure()` or hit enter in the main window to update the cursors.

Arguments:

*is_time* – If false (default), returns the zero-based index at which the threshold slope is crossed. If True, returns the time at which the threshold slope is crossed (e.g. in units of the y-axis). A negative number is returned upon failure.

Returns:

False upon failure (such as out-of-range).

```python
stf.get_threshold_value()
```

Returns value found at the threshold slope. Note that this value is not update after changing the AP threshold. Call `measure()` or hit enter in the main window to update the cursors.

Returns:

False upon failure (such as out-of-range).
**Stimfit Documentation, Release 0.13.17**

**stf.get_xunits(trace=-1, channel=-1)**

Returns the x units of the specified section. X units are not allowed to change between sections at present, and they are hard-coded to “ms”. This function is for future extension.

**Arguments:**

_**trace**_ – The zero-based index of the trace of interest. If < 0, the name of the active trace will be returned.

_**channel**_ – The zero-based index of the channel of interest. If < 0, the active channel will be used.

**Returns:**

The x units as a string.

**stf.get_yunits(trace=-1, channel=-1)**

Returns the y units of the specified trace. Y units are not allowed to change between traces at present.

**Arguments:**

_**trace**_ – The zero-based index of the trace of interest. If < 0, the name of the active trace will be returned.

_**channel**_ – The zero-based index of the channel of interest. If < 0, the active channel will be used.

**Returns:**

The x units as a string.

**stf.leastsq(fselect, refresh=True)**

Fits a function to the data between the current fit cursors.

**Arguments:**

_**fselect**_ – Zero-based index of the function as it appears in the fit selection dialog.

_**refresh**_ – To avoid flicker during batch analysis, this may be set to False so that the fitted function will not immediately be drawn.

**Returns:**

A dictionary with the best-fit parameters and the least-squared error, or a null pointer upon failure.

**stf.leastsq_param_size(fselect)**

Retrieves the number of parameters for a function.

**Arguments:**

_**fselect**_ – Zero-based index of the function as it appears in the fit selection dialog.

**Returns:**

The number of parameters for the function with index fselect, or a negative value upon failure.

**stf.maxrise_index(active=True)**

Returns the zero-based index of the maximal slope of rise in the specified channel. Uses the currently measured values, i.e. does not update measurements if the peak window cursors have changed.

**Arguments:**

_**active**_ – If True, returns the current index of the maximal slope of rise within the active channel. Otherwise, returns the current index of the maximal slope of rise within the inactive channel.

**Returns:**

The zero-based index of the maximal slope of rise in units of sampling points interpolated between adjacent sampling points. Returns a negative value upon failure.
stf.maxdecay_index()

Returns the zero-based index of the maximal slope of decay in the current channel. Uses the currently measured values, i.e. does not update measurements if the peak window cursors have changed. Note that in contrast to maxrise_index(), this function only works on the active channel.

Returns:

The zero-based index of the maximal slope of decay in units of sampling points interpolated between adjacent sampling points. Returns a negative value upon failure.

stf.measure()

Updates all measurements (e.g. peak, baseline, latency) according to the current cursor settings. As if you had pressed Enter in the main window.

Returns:

False upon failure, True otherwise.

stf.new_window(*args)

Creates a new window showing a 1D NumPy array.

Arguments:

arg – The NumPy array to be shown.

stf.new_window_list(array_list)

Creates a new window showing a sequence of 1D NumPy arrays, or a sequence of a sequence of 1D NumPy arrays. As opposed to new_window_matrix(), this has the advantage that the arrays need not have equal sizes.

Arguments:

array_list – A sequence (e.g. list or tuple) of numpy arrays, or a sequence of a sequence of numpy arrays.

stf.new_window_matrix(*args)

Creates a new window showing a 2D NumPy array.

Arguments:

arg – The NumPy array to be shown. First dimension are the traces, second dimension the sampling points within the traces.

stf.new_window_selected_all(*args)

Creates a new window showing the selected traces of all open files.

Returns:

True if successful.

stf.new_window_selected_this(*args)

Creates a new window showing the selected traces of the current file.

Returns:

True if successful.

stf.peak_index(active=True)

Returns the zero-based index of the current peak position in the specified channel. Uses the currently measured values, i.e. does not update measurements if the peak window cursors have changed.

Arguments:

active – If True, returns the current peak index of the active channel. Otherwise, returns the current peak index of the inactive channel.
Returns:
The zero-based index in units of sampling points. May be interpolated if more than one point is used for the peak calculation. Returns a negative value upon failure.

```
stf.select_all(*args)
```
Selects all traces in the current file. Stores the baseline along with the trace index.

```
stf.select_trace(trace=-1)
```
Selects a trace. Checks for out-of-range indices and stores the baseline along with the trace index.

**Arguments:**

- `trace` – ZERO-BASED index of the trace. Default value of -1 will select the currently displayed trace. Note that this is one less than what is displayed in the drop-down list.

**Returns:**

True if the trace could be selected, False otherwise.

```
stf.set_baseline_method(method)
```
Sets the method to compute the baseline.

**Arguments:**

- `method` – A string specifying the method to calculate the baseline. Can be one of “mean” or “median”

**Returns:**

False upon failure.

```
stf.set_base_end(pos, is_time=False)
```
Sets the base end cursor to a new position. This will NOT update the baseline calculation. You have to either call `measure()` or hit enter in the main window to achieve that.

**Arguments:**

- `pos` – The new cursor position, either in units of sampling points if `is_time` == False (default) or in units of time if `is_time` == True.
- `is_time` – see above.

**Returns:**

False upon failure (such as out-of-range).

```
stf.set_base_start(pos, is_time=False)
```
Sets the base start cursor to a new position. This will NOT update the baseline calculation. You have to either call `measure()` or hit enter in the main window to achieve that.

**Arguments:**

- `pos` – The new cursor position, either in units of sampling points if `is_time` == False (default) or in units of time if `is_time` == True.
- `is_time` – see above.

**Returns:**

False upon failure (such as out-of-range).

```
stf.set_channel(channel)
```
Sets the currently displayed channel to a new index. Subsequently updates all measurements (e.g. peak, base, latency, i.e. you do not have to call `measure()` yourself.)
Arguments:

channel – The zero-based index of the new trace to be displayed.

Returns:

True upon sucess, false otherwise (such as out-of-range).

\texttt{stf.set\_channel\_name(name, index=-1)}

Sets the name of the channel with the specified index.

Arguments:

name – The new name of the channel.

index – The zero-based index of the channel of interest. If < 0, the active channel will be used.

Returns:

True upon success.

\texttt{stf.set\_latency\_end\_mode(mode)}

Sets the mode of the latency end cursor

Arguments:

mode – A string specifying the mode for the latency start cursor. Can be one of “manual”, “peak”, “rise”, “foot” or “half”.

Returns:

False upon failure

\texttt{stf.set\_latency\_start\_mode(mode)}

Sets the mode of the latency start cursor

Arguments:

mode – A string specifying the mode for the latency start cursor. Can be one of “manual”, “peak”, “rise” or “half”.

Returns:

False upon failure

\texttt{stf.set\_fit\_end(pos, is\_time=False)}

Sets the fit end cursor to a new position.

Arguments:

pos – The new cursor position, either in units of sampling points if \texttt{is\_time == False} (default) or in units of time if \texttt{is\_time == True}.

is\_time – see above.

Returns:

False upon failure (such as out-of-range).

\texttt{stf.set\_fit\_start(pos, is\_time=False)}

Sets the fit start cursor to a new position.

Arguments:

pos – The new cursor position, either in units of sampling points if \texttt{is\_time == False} (default) or in units of time if \texttt{is\_time == True}.

is\_time – see above.


Returns:
False upon failure (such as out-of-range).

\texttt{stf.set_marker}(x, y)
Sets a marker to the specified position in the current trace.

Arguments:
\begin{itemize}
\item \texttt{x} – The horizontal marker position in units of sampling points.
\item \texttt{y} – The vertical marker position in measurement units (e.g. mV).
\end{itemize}

Returns:
False upon failure (such as out-of-range).

\texttt{stf.set_peak_direction}(\texttt{direction})
Sets the direction of the peak detection.

Arguments:
\texttt{direction} – A string specifying the peak direction. Can be one of: “up”, “down” or “both”

Returns:
False upon failure.

\texttt{stf.set_peak_end}(pos, \texttt{is_time}=False)
Sets the peak end cursor to a new position. This will NOT update the peak calculation. You have to either call \texttt{measure()} or hit enter in the main window to achieve that.

Arguments:
\texttt{pos} – The new cursor position, either in units of sampling points if \texttt{is_time} == False (default) or in units of time if \texttt{is_time} == True. \texttt{is_time} – see above.

Returns:
False upon failure (such as out-of-range).

\texttt{stf.set_peak_mean}(pts)
Sets the number of points used for the peak calculation.

Arguments:
\texttt{pts} – A moving average (aka sliding, boxcar or running average) is used to determine the peak value. Pts specifies the number of sampling points used for the moving window. Passing a value of -1 will calculate the average of all sampling points within the peak window.

Returns:
False upon failure (such as out-of-range).

\texttt{stf.get_peak_direction}()
Gets the direction of the peak detection.

Returns:
A string specifying the peak direction. Can be one of: “up”, “down”, or “both”.

\texttt{stf.get_peak_mean}()
Returns the number of sampling points used for peak calculation.

Returns:
0 upon failure (i.e no file opened). -1 means average of all sampling points.
**Stimfit Documentation, Release 0.13.17**

**stf.set_peak_start**(pos, is_time=False)
Sets the peak start cursor to a new position. This will NOT update the peak calculation. You have to either call *measure()* or hit enter in the main window to achieve that.

**Arguments:**

- pos – The new cursor position, either in units of sampling points if is_time == False (default) or in units of time if is_time == True. is_time – see above.

**Returns:**

False upon failure (such as out-of-range).

**stf.set_recording_comment**(comment)
Sets a comment about the recording.

**Argument:**

- comment – A comment string.

**Returns:**

True upon successful completion.

**stf.set_recording_date**(date)
Sets a date about the recording.

**Argument:**

- date – A date string.

**Returns:**

True upon successful completion.

**stf.set_recording_time**(time)
Sets a time about the recording.

**Argument:**

- time – A time string.

**Returns:**

True upon successful completion.

**stf.set_risetime_factor**(factor)
Sets the lower proportion factor to calculate the rise time (e.g. 0.2 if we want to calculate the 20–80% rise time). It will update the risetime measurement.

**Arguments:**

- factor – the low proportion factor to calculate the rise time

**Returns:**

False upon failure (such a factor lower than 0.05 or larger than 0.45).

**stf.set_sampling_interval**(si)
Sets a new sampling interval.

**Argument:**

- si – The new sampling interval.

**Returns:**

False upon failure.
**stf.set_slope(slope)**

Sets the AP threshold to the value given by the slope and takes it as reference for AP kinetic measurements. Note that you have to either call `measure()` or hit enter to update calculations.

**Argument:**

- `slope` – Slope value in mV/ms

**Returns:**

False upon failure (such as out-of-range)

**stf.set_trace(trace)**

Sets the currently displayed trace to a new index. Subsequently updates all measurements (e.g. peak, base, latency, i.e. you don’t need to call `measure()` yourself.)

**Arguments:**

- `trace` – The zero-based index of the new trace to be displayed.

**Returns:**

True upon success, false otherwise (such as out-of-range).

**stf.set_xunits(units, trace=-1, channel=-1)**

Sets the x unit string of the specified section. X units are not allowed to change between sections at present, and they are hard-coded to “ms”. This function is for future extension.

**Arguments:**

- `units` – The new x unit string.
- `trace` – The zero-based index of the trace of interest. If < 0, the name of the active trace will be returned.
- `channel` – The zero-based index of the channel of interest. If < 0, the active channel will be used.

**Returns:**

True if successful.

**stf.set_yunits(units, trace=-1, channel=-1)**

Sets the y unit string of the specified trace. Y units are not allowed to change between traces at present.

**Arguments:**

- `units` – The new y unit string.
- `trace` – The zero-based index of the trace of interest. If < 0, the name of the active trace will be returned.
- `channel` – The zero-based index of the channel of interest. If < 0, the active channel will be used.

**Returns:**

True if successful.

**stf.show_table(dict, caption="Python table")**

Shows a python dictionary in a results table. The dictionary has to have the form “string”.

**Arguments:**

- `dict` – A dictionary with strings as key values and floating point numbers as values.
- `caption` – An optional caption for the table.

**Returns:**
True if successful.

```
stf.show_table_dictlist(dict, caption="Python table", reverse=True)
```

Shows a python dictionary in a results table. The dictionary has to have the form “string” : list.

**Arguments:**
- `dict` – A dictionary with strings as key values and lists of floating point numbers as values.
- `caption` – An optional caption for the table.
- `reverse` – If True, The table will be filled in column-major order, i.e. dictionary keys will become column titles. Setting it to False has not been implemented yet.

**Returns:**
True if successful.

```
stf.subtract_base(*args)
```

Subtracts the baseline from the selected traces of the current file, then displays the subtracted traces in a new window.

**Returns:**
True if the subtraction was successful, False otherwise.

```
stf.t50left_index(active=True)
```

Returns the zero-based index of the left half-maximal amplitude of an event in the specified channel. Uses the currently measured values, i.e. does not update measurements if the peak or base window cursors have changed.

**Arguments:**
- `active` – If True, returns the current index of the left half-maximal amplitude within the active channel. If False, returns the current index of the left half-maximal amplitude within the inactive channel.

**Returns:**
The zero-based index of the left half-maximal amplitude in units of sampling points. Interpolates between sampling points. Returns a negative value upon failure.

```
stf.t50right_index(active=True)
```

Returns the zero-based index of the right half-maximal amplitude of an event in the active channel. Uses the currently measured values, i.e. does not update measurements if the peak or base window cursors have changed.

**Arguments:**
- `active` – If True, returns the current index of the right half maximal amplitude within the active channel. Only implemented for the active channel at this time. Will return a negative value and show an error message if `active == False`.

**Returns:**
The zero-based index of the right half-maximal amplitude in units of sampling points. Interpolates between sampling points. Returns a negative value upon failure.

```
stf.unselect_all(*args)
```

Unselects all previously selected traces in the current file.
A list of recent publications that have used Stimfit for data analysis. Please let me know if something is missing here.


- Turesson HK, Rodriguez-Sierra OE, Pare D (2013). Intrinsic connections in the anterior part of the bed nucleus of the stria terminalis. *J Neurophysiol* February 27, 2013, doi: 10.1152/jn.00004.2013


The stfio Python module allows to read and write data in common electrophysiology formats without running Stimfit. Build instructions for GNU/Linux can be found in Building the Python module only.

The central object in the stfio module is called a Recording. There are two ways to construct a Recording: You can either read it in from a file, or you can build it up from scratch using NumPy arrays.

## 13.1 Reading files

Files can be opened using the read function that returns a Recording object:

```python
>>> import stfio

>>> rec = stfio.read("/home/cs/data/test.abf")
```

read takes a filename and optionally a file type (as a string) as an argument. At present, the following types are supported:

<table>
<thead>
<tr>
<th>ftype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cfs&quot;</td>
<td>CED filing system</td>
</tr>
<tr>
<td>&quot;hdf5&quot;</td>
<td>HDF5</td>
</tr>
<tr>
<td>&quot;abf&quot;</td>
<td>Axon binary file</td>
</tr>
<tr>
<td>&quot;atf&quot;</td>
<td>Axon text file</td>
</tr>
<tr>
<td>&quot;axg&quot;</td>
<td>Axograph X binary file</td>
</tr>
<tr>
<td>&quot;heka&quot;</td>
<td>HEKA binary file</td>
</tr>
</tbody>
</table>

If the file type is None (default), it will be guessed from the file name extension.

A Recording has a number of attributes that describe the recording:

```python
>>> print(rec.comment)  
Created with Clampex

>>> print(rec.date)  
2008/1/18

>>> print(rec.dt)  
# sampling interval
0.1

>>> print(rec.file_description)  
# no file description in this case

>>> print(rec.time)  
15:08:20
```
A `Recording` consists of one or more `Channels`, which in turn are composed of one or more `Sections`. They can be accessed using indexing operators (\([\])\).

```python
>>> print(rec.xunits)
ms
```

The time series in a `Section` can be accessed as a NumPy array:

```python
>>> arr = rec[0][0].asarray()
>>> type(arr)
<type 'numpy.ndarray'>
>>> arr.shape
(146450,)
```

Note that the `Section` itself is not a NumPy array and therefore needs to be converted as described above before you can do fancy arithmetics:

```python
>>> type(rec[0][0])
<class 'stfio.Section'>
>>> res = rec[0][0] + 2.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Section' and 'float'
>>> res = rec[0][0].asarray() + 2.0
```

### 13.2 Constructing Recordings from scratch

`Recordings` can be assembled from NumPy arrays. Here’s a particularly stupid example:

```python
import stfio
import numpy as np

arr = np.arange(0,500,0.1)

# construct Sections from arrays:
seclist = [stfio.Section(arr), stfio.Section(arr)]

# construct Channels from lists of Sections
chlist = [stfio.Channel(seclist), stfio.Channel(seclist)]

# Set channel units
chlist[0].yunits = "pA"
chlist[1].yunits = "mV"
```
### 13.3 Writing files

*Recordings* can be stored to files using the *write* method:

```python
>>> import stfio
>>> rec = stfio.read("/home/cs/data/test.abf")
>>> rec.write("/home/cs/data/out.h5")
```

At present, *write* only supports hdf5 files.
14.1 Free electrophysiology software

**RELACS**  Software for data acquisition, analysis, and stimulus generation. Runs under GNU/Linux.

**WinLTP**  Stimulation and data acquisition program for Windows, specialised for LTP/LTD experiments.

**NClamp**  Plugins for IgorPro for data acquisition. Igor is available for Mac and Windows.

**Neuromatic**  Plugins for IgorPro for data analysis. Igor is available for Mac and Windows.

**Strathclyde Electrophysiology Software**  Suite of programs for recording and analysing signals from intracellular electrophysiology experiments. Runs under Windows.

14.2 Electrophysiology software sites

**Neuronal Networks Lab, University of Nottingham**  Quite a number of useful links to electrophysiology software sites.
INDICES AND TABLES

Author Christoph Schmidt-Hieber (christsc at gmx.de) and Jose Guzman
Date March 21, 2014
• genindex
• modindex
• search
(wxWidgets) http://www.wxwidgets.org
(wxPython) http://www.wxpython.org
(boost) http://www.boost.org
s
    stf, 73