stimfit manual
rev. 0.8.13.1

Christoph Schmidt-Hieber

December 8, 2008

Contents

Contents

Preface

Stimfit was originally written by Peter Jonas, University of Freiburg, in the early 1990s. It was primarily designed to analyse the kinetics of evoked excitatory postsynaptic currents (EPSCs; Jonas et al. (1993)). The name "Stimfit" was chosen because the program allowed to *fit* exponential functions to the decay of EPSCs evoked by extracellular *stim*ulation[1]. The program was written in Borland Pascal, running under DOS and entirely controlled using keyboard shortcuts (Fig. 1). 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 analyse 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.
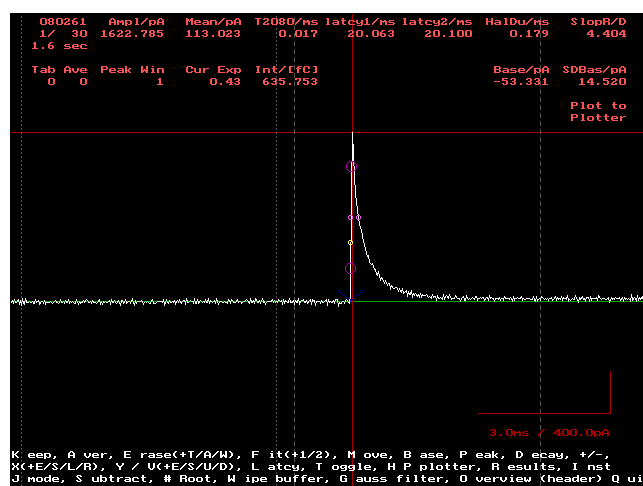
Fig. 1: The original Stimfit for DOS.

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

---

[1]Be reassured that I would choose a different name if I had to sell the program.

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[2]. A large amount of helpful comments and bug reports were filed by Emmanuel Eggermann and Daniel Doischer. The Levenberg-Marquardt algorithm used for curve fitting was implemented by Manolis Lourakis[3].

---

[2]`http://www.winltp.com`
[3]`http://www.ics.forth.gr/~lourakis/levmar/`

1 Getting started

1.1 File opening

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

- Download the Stimfit setup program[1] and install Stimfit on your computer.

- Download the sample data file[2].

- 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.

1.2 Trace scaling

- If you just see scale bars, but no trace is displayed, press $\boxed{F}$ or click the corresponding button (Fig. 2). This will fit the first trace of the active channel (plotted in black) to the screen.
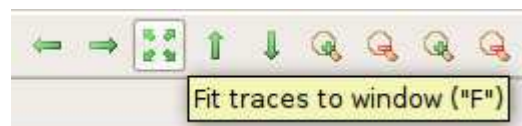
Fig. 2: Fit traces to the window size.

- If you prefer coordinates to scale bars, you can uncheck "View"→"Scale bars" in the menu (Fig. 3).

- Fit the inactive channel (plotted in red) to the screen as well: Press $\boxed{3}$. The buttons labelled "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. Press $\boxed{F}$ again. The inactive channel (red trace) will now be fitted to the screen as well. If you want to scale channels individually, press either $\boxed{1}$ or $\boxed{2}$.

---

[1]http://www.stimfit.org
[2]http://stimfit.org/tutorial/sample.dat

Fig. 3: Show coordinates rather than scale bars.
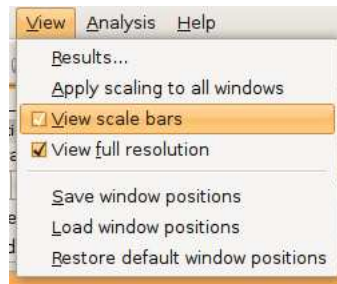

Fig. 4: Scaling applies to both channels.

- Enlarge the vertical scale: Press $\boxed{+}$. 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 $\boxed{-}$.

- Enlarge the time scale: Press $\boxed{\text{Ctrl}}$ and $\boxed{+}$ 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 $\boxed{\text{Ctrl}}$ and $\boxed{-}$ simultaneously.

- Shift the trace by pressing $\boxed{\text{Ctrl}}$ and one of the cursor (arrow) keys simultaneously.

- You can zoom into parts of the trace using a zoom window: Press $\boxed{\text{Z}}$. The zoom button (showing a magnifying glass) will be highlighted (Fig. 5).
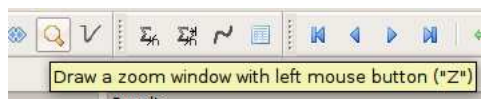

Fig. 5: Setting the mouse cursor to draw zoom windows.

- 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).
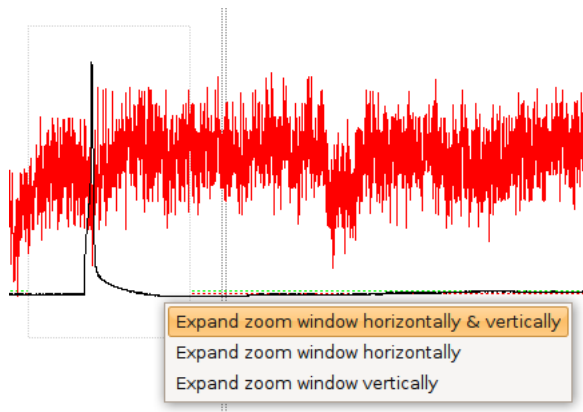
Fig. 6: Magnifying a region of interest.

- If you can't see any trace because you zoomed in or out too much, press $\boxed{\text{F}}$ to fit the trace to the screen again.

Note: Although all commands mentioned above can be accessed with the mouse, I strongly recommend using the keyboard shortcuts; once you get used to it, the shortcuts are much faster (see p. 31 for a list).

1.3 Navigate within a file

- You can toggle through traces using the $\boxed{\leftarrow}$ or $\boxed{\rightarrow}$ keys (*without* pressing $\boxed{\text{Ctrl}}$ at the same time). The current trace number will be displayed in the drop-down box labelled "Trace ... of ...". You can directly select a trace from this box as well (Fig. 7). *traces*

Fig. 7: Selecting a trace.

channels

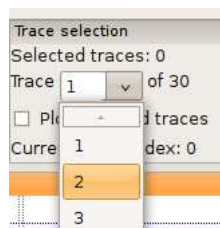- All measurements will be performed on the active channel plotted in black. You can swap channels by either selecting "View"→"Swap channels" from the menu, or setting the channels in the drop-down boxes (Fig. 8).
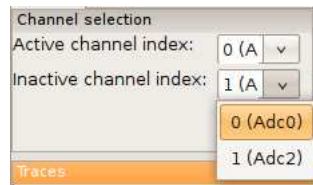
Fig. 8: Setting the active and inactive channel.

1.4 Analyse individual events

An "event" can be anything from an EPSC to an action potential. In this case, we will analyse 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 table by right-clicking on one of the column or row title labels, and then selecting or unselecting the corresponding items.

results table

Fig. 9: Showing analysis results.

cursors

baseline

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 is calculated as the average of all sampling points between the two base window

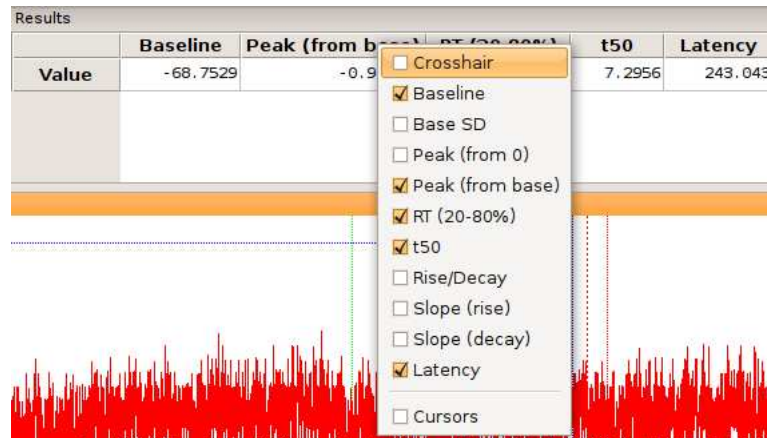cursors (vertical green dashed lines). To move the cursors, press $\boxed{\text{B}}$. The corresponding toolbar button will be highlighted. Set the left cursor by clicking the left mouse button where you want the baseline calculation to start. Set the right cursor by clicking the right mouse button where you want the baseline calculation to end. Press $\boxed{\text{Enter}}$. 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).
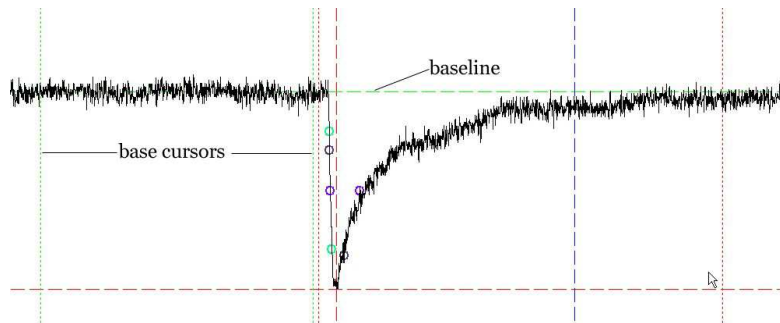

Fig. 10: Setting the baseline window cursors.

> **Note:** You have to press $\boxed{\text{Enter}}$ after changing any cursor position to update all calculations. Otherwise, you will see the results of your previous cursor settings. Alternatively, you can call `measure()` from the Python shell (see p. 15).

peak calculation The peak value will be determined between the two peak window cursors (vertical red dashed lines). To move the cursors, press $\boxed{\text{P}}$. The corresponding toolbar 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 $\boxed{\text{Enter}}$. 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).

peak direction 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 the "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 `set_peak_direction(direction)`, `set_peak_direction` where `direction` can be one of `"up"`, `"down"` or `"both"`. The Python shell will be explained
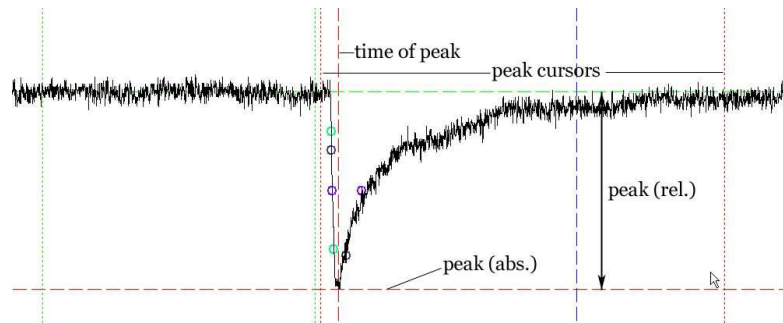
5

Fig. 11: Setting the peak window cursors.

in some more detail in chapter 2.
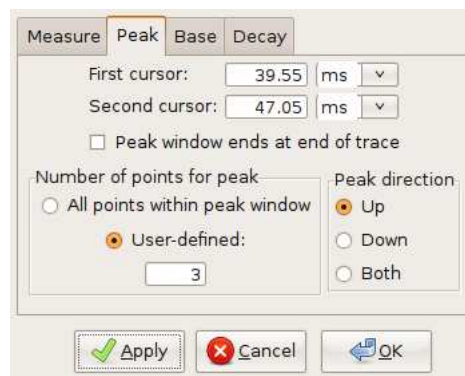
Fig. 12: Setting the peak calculation properties.

moving average   In case the event you want to analyse is noisy, it may be helpful to use the average of several neighbouring 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 `set_peak_mean(pts)`, where `pts` is the number of sampling points.

`set_peak_mean`

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 baseline), commonly called the "20-to-80%-risetime". The points corresponding to 20 and 80% of the peak value are indicated by green circles. They are determined by linear interpolation between neighbouring sampling points.

- "t1/2" refers to the full width of the signal at half-maximal amplitude (measured from

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 neighbouring 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 half duration calculation 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 rise and decay is still restricted to the peak window.
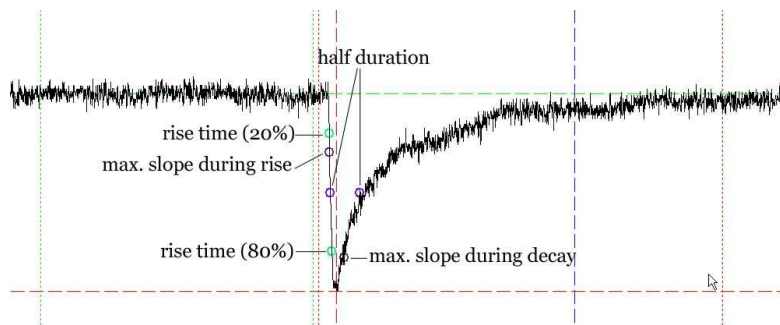

Fig. 13: Analysis of individual events.

1.5 Average calculation

First, you have to select the traces that you want to average: navigate through the file with the ⬅ and ➡ keys (as described above), and press Ⓢ if you want to select a trace, or click the selection button. The number of traces that you have already selected will be shown just above the trace selection drop-down box (Fig. 14). If you selected a trace accidentally, you can remove it from the selected traces list by pressing Ⓡ or clicking the trash bin button to the right of the selection button (Fig. 14). *average calculation*

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 a blue trace.

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.
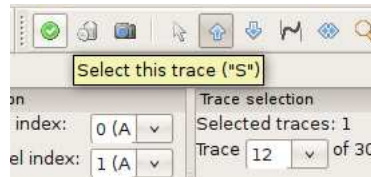
Fig. 14: Trace selection.

Fig. 15: Average calculation.

1.6 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. Don't forget to press Enter.

- The function will be fitted to the data between the two fit window cursors (grey 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.

- Select "Analysis"→"Fit"→"Non-linear regression" from the menu. Select a bi-exponential function (Fig. 16).

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

leastsq `leastsq(fselect, refresh=True)`

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 functions in the fit settings dialog (see Fig. 16). If `refresh=False`, the trace will not be re-drawn, wich can be useful to avoid flicker when performing a series of fits.
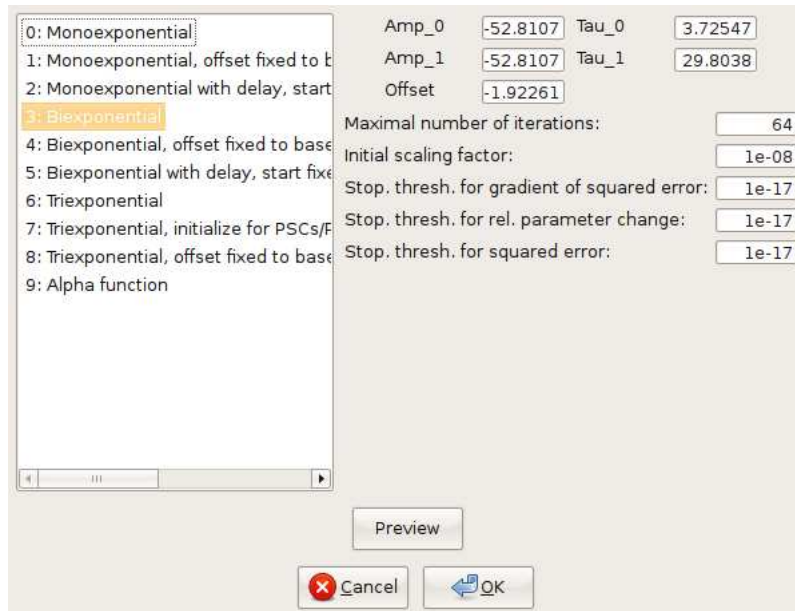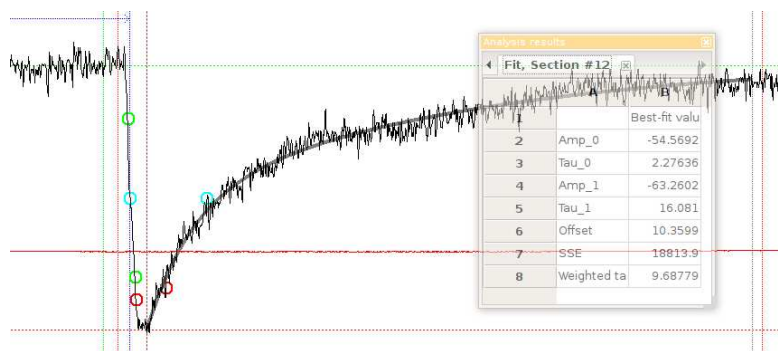
Fig. 16: Non-linear regression settings.

Fig. 17: Results of a non-linear regression using a bi-exponential function.

1 Getting started

2 The Python shell

2.1 Before you start

If you're new to Python, I suggest that you first have a look at the Python tutorial[1]. If that's not enough, abundant documentation is freely available on the Python web site[2]. If you're new to stimfit, I recommend going through the tutorial in chapter 1 of this manual first.

2.2 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:

```
>>> 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 `check_doc` function from that list:

check_doc

```
>>> stf.check_doc()
False
```

The function documentation will pop up when you type in the opening bracket. The function returns the boolean value `False` because you haven't 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 the same result by simply typing

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

If you press $\boxed{\text{Ctrl}}$ and $\boxed{\uparrow}$ 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.

2.3 Accessing data from the Python shell

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

get_trace

---

[1]http://docs.python.org/tut/
[2]http://www.python.org/doc/

The `get_trace` function returns the currently displayed trace as a one-dimensional NumPy[3] array when called without any arguments:

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

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

```
>>> print a[123]
-26.3671875
```

prints out the y-value of the sampling point with index 123. Note that indices in Python are *zero-based*, i.e. the first sampling point has the index 0:

```
>>> print a[0]
-21.2249755859
```

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

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

You can use the `get_trace(trace=-1, channel=-1)` 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!

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

2.4 Using NumPy with stimfit

NumPy[4] 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
```

---

[3]http://numpy.scipy.org/
[4]http://numpy.scipy.org/

```
>>> print b[234]
-54.0771484375
```

Or you can 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 operations in a new window by passing a 1D-NumPy array to the `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 `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 = N.empty( (2, get_size_trace()) )
>>> numpy_matrix[0] = get_trace()
>>> numpy_matrix[1] = N.sqrt( N.abs(get_trace()) )
>>> new_window_matrix(numpy_matrix)
```

In this example, `N` is the NumPy namespace. Typing `N.` at the command prompt will show you all available NumPy functions. `get_size_trace` will be explained later on (see p. 15).

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 `new_window_list`, albeit at the price of a significant performance loss. `new_window_list` takes a Python list of 1D-NumPy arrays as an argument:

```
>>> python_list = [get_trace(),]
>>> python_list.append( \
...     N.concatenate((get_trace(), get_trace())) )
>>> new_window_list(python_list)
```

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

The SciPy[5] library, which is built 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 analyses.

2.5 Control stimfit from the Python shell

2.5.1 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_peak_start
set_base_start
set_fit_start
set_peak_end
set_base_end
set_fit_end

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_peak_start
get_base_start
get_fit_start
get_peak_end
get_base_end
get_fit_end

get_[xy]_start(is_time = False) and
get_[xy]_end(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 func-

get_sampling_interval

tion call. Indices can be converted into time values by multiplying with get_sampling_interval(). For example:

```
>>> 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()*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
```

---

[5]http://www.scipy.org/

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 `measure()` from the Python shell.

2.5.2 Trace selection and navigation

`select_trace(trace = -1)`

You can select any trace within a file by passing its zero-based index to `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:

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

Note that the Python `range` function omits the end point.

`unselect_all()`

`select_all()`

`get_selected_indices()`

`new_window_selected_this()`

The list of selected traces can be cleared using `unselect_all()`, and conversely, all traces can be selected using `select_all()`. `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 `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 and the number of traces in a channel, respectively. `trace` and `channel` have the same meaning as in `get_trace` (see p. 12). These functions can be used to iterate over an entire file or to check ranges:

```
>>> unselect_all()
>>> for n in range(0, get_size_channel(), 5): select_trace(n)
...
True
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 5 : 13050
```

Use backspace to remove the indentation after you have finished the second for-loop in line 10.

set_trace **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's no need to call `measure()` directly after this function.

get_trace_index **get_trace_index()**
Correspondingly, `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: don't confound this function with `get_trace()` .

2.5.3 File I/O

file_open **file_open(filename)**
file_save **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:

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

in Windows or

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

in GNU/Linux.

close_this **close_this()**
will close the currently displayed file, whereas

close_all **close_all()**
closes all open files.

2.5.4 Define your own functions

By defining your own functions, you can apply identical complex analyses 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 don't know where that is, use the stimfit program directory (typically, this will be C:\Program Files\Stimfit in Windows or /usr/lib/python-2.5/site-packages/stimfit in Linux). You will find some example files in that directory that you can use as a template, but you shouldn't touch stf.py which is the core stimfit module.

2. Import the stimfit module in your file:

   ```
   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:

   ```
   >>> import myFile
   ```

4. If you have applied changes to your file, there's no need to restart stimfit. Just do:

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

   from the embedded Python shell.

To give you an example, listing 1 shows a function that returns the sum of the squared amplitude values across all selected traces of a file.

   To import and use this file from stimfit, you would do:

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

2.6 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.

2.6.1 Cutting traces to arbitrary lengths

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

```
>>> 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)`   `cut_traces`
`cut_traces_multi(pt_list)`   `cut_traces_multi`
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 `cut_traces` is shown in listing 2. For example,

```
>>> cut_traces_multi([100,900])
```

will cut all selected traces at sampling points 100 and 900 and show the cut traces in a new window. Note that you can pass a list or a tuple as an argument.

```
>>> cut_traces_multi(range(100,2000,100))
```

will cut the selected traces at every 100th sampling point, starting with the 100th and ending with the 1900th (see p. 15 for the syntax of the `range` function).

```
1  # import the stimfit core module:
2  import stf
3
4  def get_amp():
5      """Returns the amplitude (peak-base)"""
6      return stf.get_peak()-stf.get_base()
7
8  def sqr_amp():
9      """Returns the sum of squared amplitudes of all
10     selected traces, or -1 if there was an error. Uses
11     the current settings for the peak direction and
12     cursor positions."""
13
14     # Store the current trace index:
15     old_index = stf.get_trace_index()
16
17     sum_sqr = 0
18     for n in stf.get_selected_indices():
19         # Setting a trace will update all measurements,
20         # so there's no need to call measure()
21         if ( not(stf.set_trace(n)) ):
22             return -1
23         sum_sqr += get_amp()**2
24
25     # Restore the displayed trace:
26     stf.set_trace(old_index)
27
28     return sum_sqr
```

Listing 1: myFile.py

```python
1  import stf
2  import numpy as N
3
4  def cut_traces( pt ):
5      """Cuts the selected traces at the sampling point pt,
6      and shows the cut traces in a new window.
7      Returns True upon success, False upon failure."""
8
9      # Check whether anything has been selected:
10     if not stf.get_selected_indices():
11         return False
12     new_list = list()
13     for n in stf.get_selected_indices():
14         if not stf.set_trace(n): return False
15
16         # Check for out of range:
17         if pt < stf.get_size_trace():
18             new_list.append( stf.get_trace()[:pt] )
19             new_list.append( stf.get_trace()[pt:] )
20         else:
21             print "Cutting point", pt, "is out of range"
22
23     # Don't create a new window if everything was out of
24     # range
25     if len(new_list) > 0: stf.new_window_list( new_list )
26
27     return True
```

Listing 2: `cut_traces`

3 Latency measurements

3.1 Measurement of synaptic delay

Stimfit is frequently used to measure the delay between a presynaptic signal and a postsy- *synaptic delay*
naptic response. Classically, this synaptic delay or latency is defined as "the time interval
between peak of inward current through the presynaptic membrane and commencement of
inward current through the postsynaptic membrane" (Katz and Miledi, 1965). Neglecting ca-
ble 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

$$I_\mathrm{m} = I_\mathrm{cap} + I_\mathrm{ionic} = C_\mathrm{m}\frac{\mathrm{d}V_\mathrm{m}}{\mathrm{d}t} + I_\mathrm{ionic} = 0, \text{ and hence}$$

$$I_\mathrm{ionic} = -I_\mathrm{cap} = -C_\mathrm{m}\frac{\mathrm{d}V_\mathrm{m}}{\mathrm{d}t}$$

The commencement (sometimes called "foot") of the postsynaptic current can robustly be esti- *foot*
mated 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);
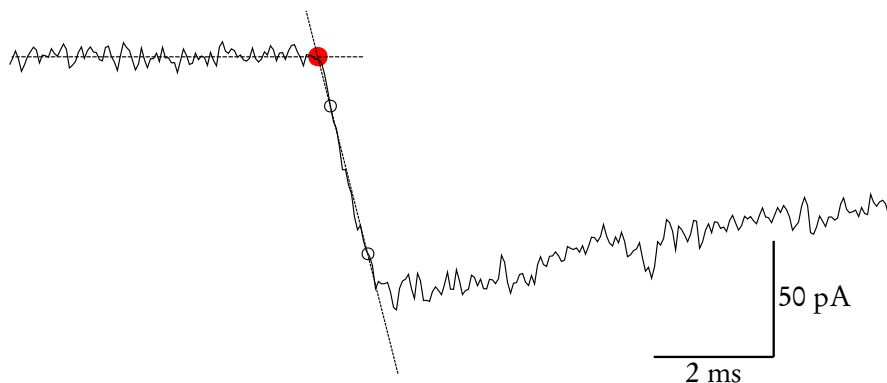Fig. 18).

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).

Although the method described above yields reliable results when both the pre- and the postsynaptic whole-cell recording show little noise and few artefacts, it may sometimes be favourable 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.

3.2 Trace alignment

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

align_selected **`align_selected(alignment, active=False)`**
This function 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 trace, 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 **`maxrise_index(active)`**
returns the zero-based index of the maximal slope of rise in units of sampling points (see Fig. 13), interpolated between adjacent sampling points, or a negative value upon failure.

peak_index **`peak_index(active)`**
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 (see p. 6).

foot_index **`foot_index(active)`**
returns the zero-based index of the foot of an event, as described on p. 21 and in Fig. 18, or a negative value upon failure.

t50left_index **`t50left_index(active)`**
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 **`t50right_index(active)`**
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.

Listing 3 shows a function that 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
    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
```

Listing 3: `align_maxrise`

3.3 Setting the latency cursors

The latency cursors (plottes as dotted vertical blue lines) can either be set automatically to   *latency cursors*
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].

To confirm your latency cursor settings and measure latencies, you can either press [Enter]
or call `measure()` from the shell (see p. 15). 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
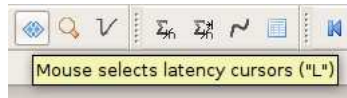
Fig. 19: Activate latency mode.

this value (see p. 5). The latency will be indicated as a double-headed arrow connecting the two latency cursors (Fig. 20).
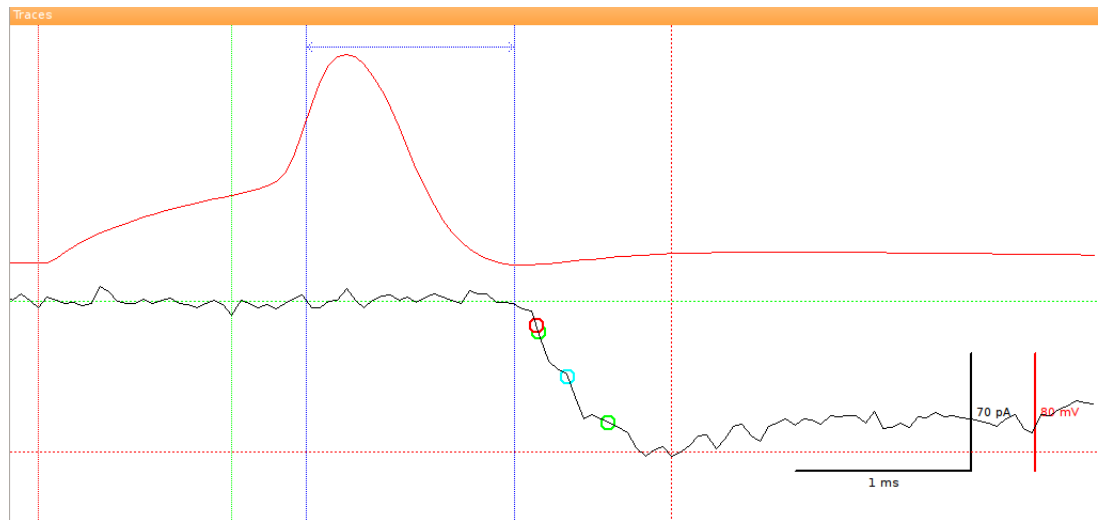

Fig. 20: The latency between the maximal slope of rise of an action potential (red) and the foot of an EPSC (black) is indicated by a horizontal double-headed arrow.

4 Event extraction by template matching

4.1 Introduction

To isolate individual events such as EPSCs or EPSPs from recorded data, Stimfit uses a template matching algorithm as decribed by Jonas et al. (1993), with some implementation details adopted from Clements and Bekkers (1997). 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 minimised:

template matching

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

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) suggest to use the linear correlation coefficient between the optimally scaled template and the data, whereas Clements and Bekkers (1997) compare the scaling factor with the noise standard deviation.

detection criterion

4.2 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 examplary large and isolated events using a high detection criterion threshold.

3. Create the 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.

4.2.1 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 modelled with the sum or the product of two exponential functions [1]. 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[2].

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
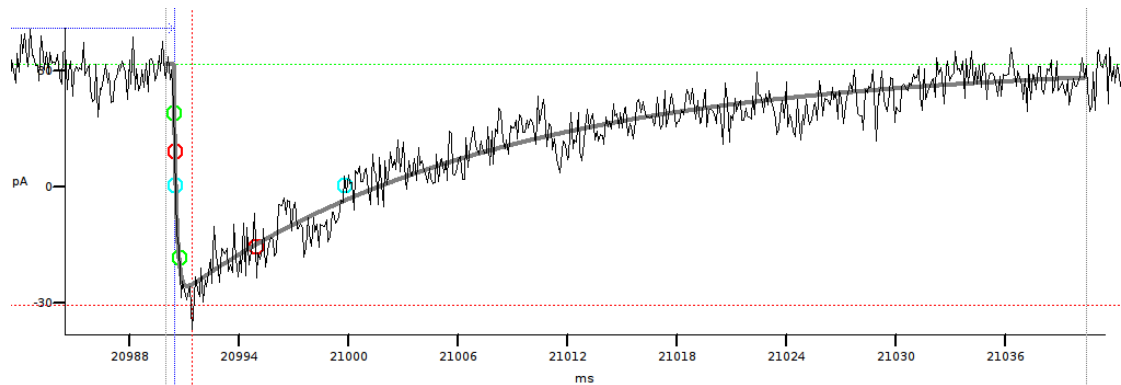
Fig. 21: Creation of a "bait" template.

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:

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

This will take care of the appropriate cursor positions and the biexponential fit (see p. 14 for setting the cursors from Python, and p. 9 for the `leastsq` function). If you prefer, you can use the fit settings dialog as described in chapter 1 (Fig. 16, p. 9).

---

[1]Note that the product of two exponentials $f(t) = a \left(1 - e^{-\frac{t}{\tau_1}} \right) e^{-\frac{t}{\tau_2}}$ can equivalently be expressed as the sum of two exponentials: $f(t) = a \left(e^{-\frac{t}{\tau_2}} - e^{-\frac{t}{\tau_3}} \right)$, with $\tau_3 = \frac{\tau_1 \tau_2}{\tau_2 - \tau_1}$.

[2]http://stimfit.org/tutorial/minis.dat

4.2.2 Extract exemplary events

We now use the bait template to fish some more large and isolated events. Choose "Analysis"→"Event detection" →"Template matching..." from the menu. In the dialog that will pop up (Fig. 22),
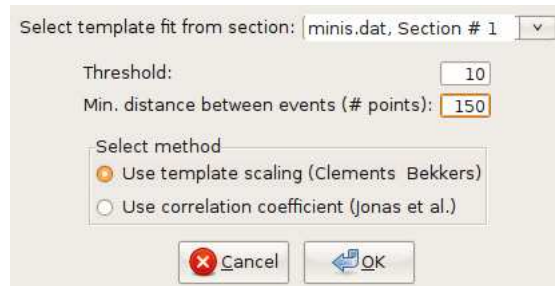

Fig. 22: Event detection settings.

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 factor (Clements and Bekkers, 1997). Click "OK" to start the event detection. When finished, press $\boxed{\text{F}}$ 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
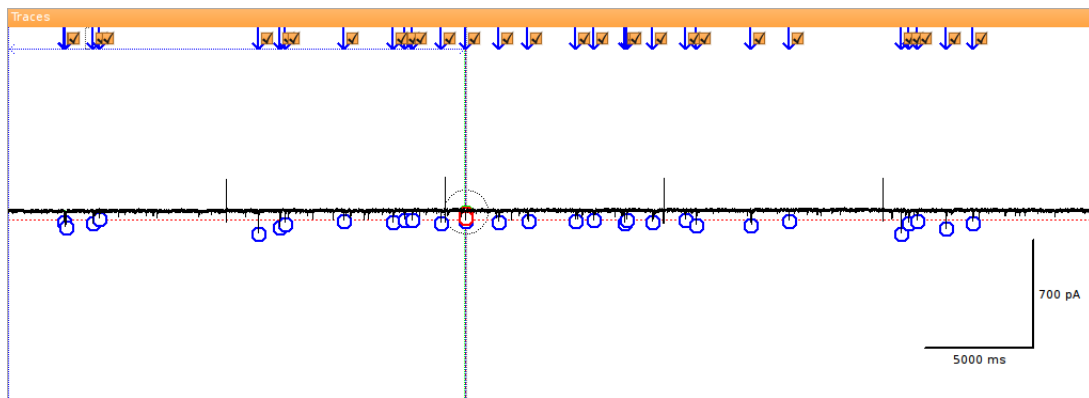
Fig. 23: Detected events.

to the event editing mode, either by pressing $\boxed{\text{E}}$ 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.

Fig. 24: Switching to event editing mode.

4.2.3 Create the final template

We now create the average of all extracted events, as explained in chapter 1 (p. 7). Then, we fit a biexponential function to the average, as explained above for the single EPSC (see p. 26). 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:

```
>>> import minidemo # if you haven't imported it already
>>> minidemo.final()
```

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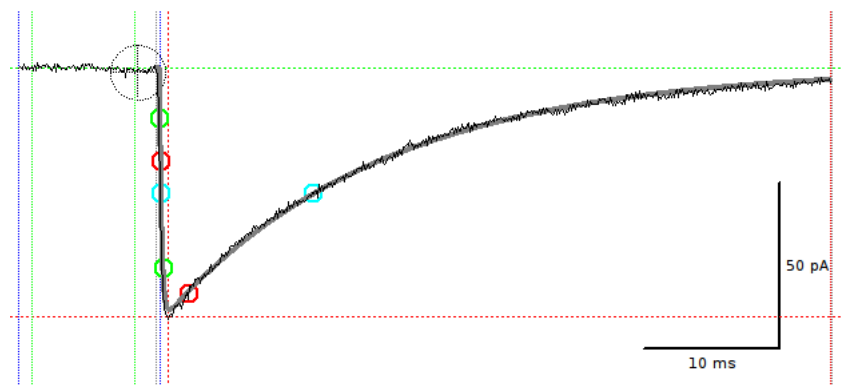
Fig. 25: Creating a final template.

4.2.4 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).
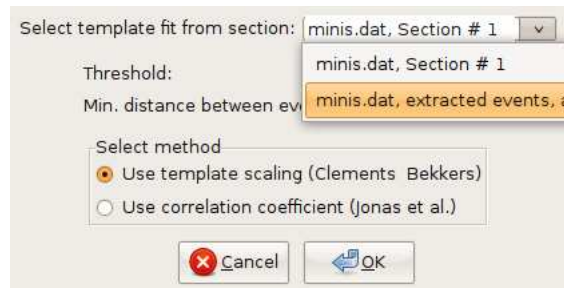
Fig. 26: Selecting the final template.

4.2.5 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 unselecting 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 $\boxed{\text{Shift}}$ and $\boxed{\Longleftarrow}$ 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.
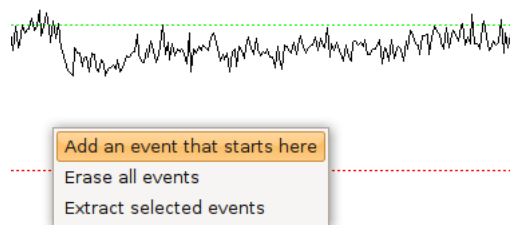
Fig. 27: Adding a false-negative event.

4.2.6 Analyse 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()
```

batch analysis To analyse all traces efficiently, you can now perform a "batch analysis" on all traces at once: First, select all traces, either using `select_all()` from the shell (see p. 15) or "Edit"→"Select all traces" from the menu or pressing Ctrl and A at the same time. Then, choose "Analysis"→"Batch analysis" from the menu. From the dialog (Fig. 28), choose the analysis functions that you
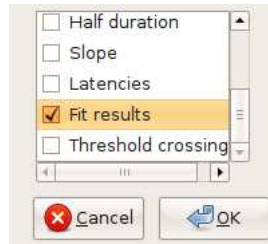

Fig. 28: Batch analysis settings.

want to apply to your data. Click "OK" once you 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.

4.2.7 Adjusting event detection settings

Problem	Solution
Too many false-positive events have been detected	Increase the detection threshold
Too many events have been missed (false-negatives)	Decrease the detection threshold
One and the same event is detected multiple times at short time intervals	Increase the number of sampling points between events
Closely spaced events are not detected separately	Decrease the number of sampling points between events

Keyboard shortcuts

Key	Action	
B	*followed by*	
	left mouse button	Sets left cursor (lower limit) of baseline window
	right mouse button	Sets right cursor (upper limit) of baseline window
D	*followed by*	
	left mouse button	Sets left cursor (lower limit) of fit window
	right mouse button	Sets right cursor (upper limit) of fit window
E	*followed by*	
	right mouse button	Pops up a context menu for event editing
L	*followed by*	
	left mouse button	Sets the left latency cursor
	right mouse button	Sets the right latency cursor
P	*followed by*	
	left mouse button	Sets left cursor (lower limit) of peak window
	right mouse button	Sets right cursor (upper limit) of peak window
Z	*followed by*	
	left mouse button	Drag a window around the region you want to zoom in
	right mouse button	Choose action from context menu
Enter	Measure peak, baseline and latency, update results table	
1	Scaling operations apply to the active (black) channel	
2	Scaling operations apply to the inactive (red) channel	
3	Scaling operations apply to both channels	
↑	Shift traces up	
↓	Shift traces down	
⇐	Previous trace	
⇒	Next trace	
+	Enlarge traces vertically	
−	Shrink traces vertically	
S	Select the currently displayed trace	
R	Remove the currently displayed trace from the stack of selected traces	
Ctrl	*together with*	
	+	Enlarge traces horizontally
	−	Shrink traces horizontally
	⇐	Shift traces left
	⇒	Shift traces right
	A	Select all traces
Shift	*together with*	
	⇐	Shift traces left by one window width
	⇒	Shift traces right by one window width

Keyboard shortcuts

Bibliography

Bartos M, Vida I, Frotscher M, Geiger JRP, Jonas P (2001) Rapid signaling at inhibitory synapses in a dentate gyrus interneuron network. *J Neurosci* 21:2687–2698.

Clements JD, Bekkers JM (1997) Detection of spontaneous synaptic events with an optimally scaled template. *Biophys J* 73:220–229.

Jack JJB, Noble D, Tsien RW (1983) *Electric current flow in excitable cells.* Oxford University Press, Oxford, UK.

Jonas P, Major G, Sakmann B (1993) Quantal components of unitary EPSCs at the mossy fibre synapse on CA3 pyramidal cells of rat hippocampus. *J Physiol* 472:615–663.

Katz B, Miledi R (1965) The measurement of synaptic delay, and the time course of acetyl-choline release at the neuromuscular junction. *Proc R Soc Lond B Biol Sci* 161:483–495.

Bibliography

Index

Index