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 Ctrl and ↑ 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.

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

2.3 Accessing data from the Python shell

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

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:

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

```
>>> a = get_trace()
>>> print a[234]
-27.0385742188
>>> b = a*2
>>> 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() 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() 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() 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 function call. Indices can be converted into time values by multiplying with `get_sampling_interval()`. For example:

get_sampling_interval

```
>>> 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"
```

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

```
= 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 `measure()` from the Python shell. measure

2.5.2 Trace selection and navigation

`select_trace(trace = -1)` select_trace
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. range

`unselect_all()` unselect_all
`select_all()` select_all
`get_selected_indices()` get_selected_indices
`new_window_selected_this()` 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_trace
`get_size_channel(channel=-1)` get_size_channel
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()` (see p. 12).

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 array. For example, if you wanted to cut a trace at the 100th sampling point, you could do: *trace cutting*

```
>>> 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 **`cut_traces(pt)`**

cut_traces_multi **`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 `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).

```
# 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's 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
```

Listing 1: myFile.py

```
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