optimizer Documentation

Release 0.0.1

Szabolcs Káli, Péter Friedrich, Mike Vella

Apr 27, 2017

Contents

Optimizer is a GUI-based application for the optimization of conductance based neuron models.

CHAPTER 1

Installation

Get a read only copy of optimizer

Install *git* and type:

> git clone https://github.com/KaliLab/optimizer.git

More information on this here: http://rogerdudler.github.com/git-guide/

Dependencies

The following python libraries are required:

- python
- numpy
- scipy
- matplotlib
- wxPython
- inspyred
- pyelectro
- eFEL

The following libraries are recommended:

- neuron

You can get *numpy*, *scipy* and *inspyred* with *easy_install* or *pip* with the following command (for numpy):

> pip install numpy

or

easy_install install numpy

You can get eFEL with *pip*:

pip install efel

You can get *matplotlib* with the following command:

Debian / Ubuntu : sudo apt-get install python-matplotlib

Fedora / Redhat : sudo yum install python-matplotlib

If you encounter any difficulties you can find a more detailed description at:

http://matplotlib.org/users/installing.html

You can get *wxpython* with the following command:

apt-get install python-wxgtk2.8 python-wxtools wx2.8-i18n

This command might not work if your OS has an earlier version in the standard software repository. If so, please follow the instructions at:

http://wiki.wxpython.org/InstallingOnUbuntuOrDebian

Currently, Optimizer works properly only with version 2.8 of wxpython. Some distributions (e.g., Ubuntu 16.04) may install version 3.0 instead (or in addition to) 2.8. In this case, please remove version 3.0 and make sure to install only version 2.8. Support for later versions of wxpython will be added to Optimizer in the future.

You can get *pyelectro* from:

https://github.com/vellamike/pyelectro.git

After cloning the repository you can install it by the standard installation method:

python setup.py install

Installing Neuron as a python package is optional since Optimizer can use any executable to run the simulations. WARNING: Many features of Optimizer are designed to work with Neuron, so we recommend that you install it! Installing Neuron as a python package is beyond the scope of this tutorial as it is somewhat complicated, but you can find a step-by-step guide at:

http://andrewdavison.info/notes/installation-neuron-python/

Install optimizer

Use the standard install method for Python packages:

sudo python setup.py install

Run optimizer

You can run Optimizer (with a GUI) directly from its installation folder with:

python optimizer.py -g

Or for the command line version (you must specify a configuration file as well):

python optimizer.py -c example.xml

Build documentation

If you should require a local copy of the Optimizer documentation, you need a working install of Sphinx, then run the command:

> sphinx-build ./doc <local build directory>

from the top-level optimizer directory where <local build directory> should be replaced with a custom filepath.

Test Platforms

The package was tested on the following systems:

1. Mandriva 2010.2 (kernel 2.6.33, glibc 2.11)

- python 2.6.5
- numpy 1.6.2
- scipy 0.7.2
- matplotlib 1.1.1
- wxPython 2.8.10.1
- inspyred 1.0
- pyelectro
- neuron 7.3

2. CentOS 6.4 (kernel 2.6.32, glibc 2.12)

- python 2.6.6
- numpy 1.6.1
- scipy 0.10.1
- matplotlib 1.3.1
- wxPython 2.8.12.0
- inspyred 1.0
- pyelectro
- neuron 7.2

3. Ubuntu 12.04.3 LTS (kernel 3.2.0-54-generic, glibc 2.15)

- Python 2.7.3
- numpy 1.7.0
- scipy 0.11.0
- matplotlib 1.1.1rc
- wxPython 2.8.12.1
- inspyred 1.0
- pyelectro
- neuron 7.2

4. Ubuntu 14.04.4 LTS

- Python 2.7.6
- numpy 1.8.2
- scipy 0.13.3
- matplotlib 1.3.1
- wxPython 2.8.12.1
- inspyred 1.0
- pyelectro 0.1.6
- neuron 7.4

Notes

- Since Neuron with the python interpreter is not working perfectly on Windows, we recommend to use Ubuntu (installing and setting up a virtual os is not hard).
- inspyred requires a feature which is only included in python 2.7, but ther is a workaround for this problem: https://groups.google.com/forum/#!topic/inspyred/YwJb3ABVtL8

Developers

Project Leader:

- **Szabolcs Káli:** kali@koki.hu

Lead Developer:

- **Peter Friedrich:** p.friedrich.m@gmail.com
- **Sára Sáray** saraysari@gmail.com

Contributors:

- Mike Vella

CHAPTER 2

Optimizer tutorial

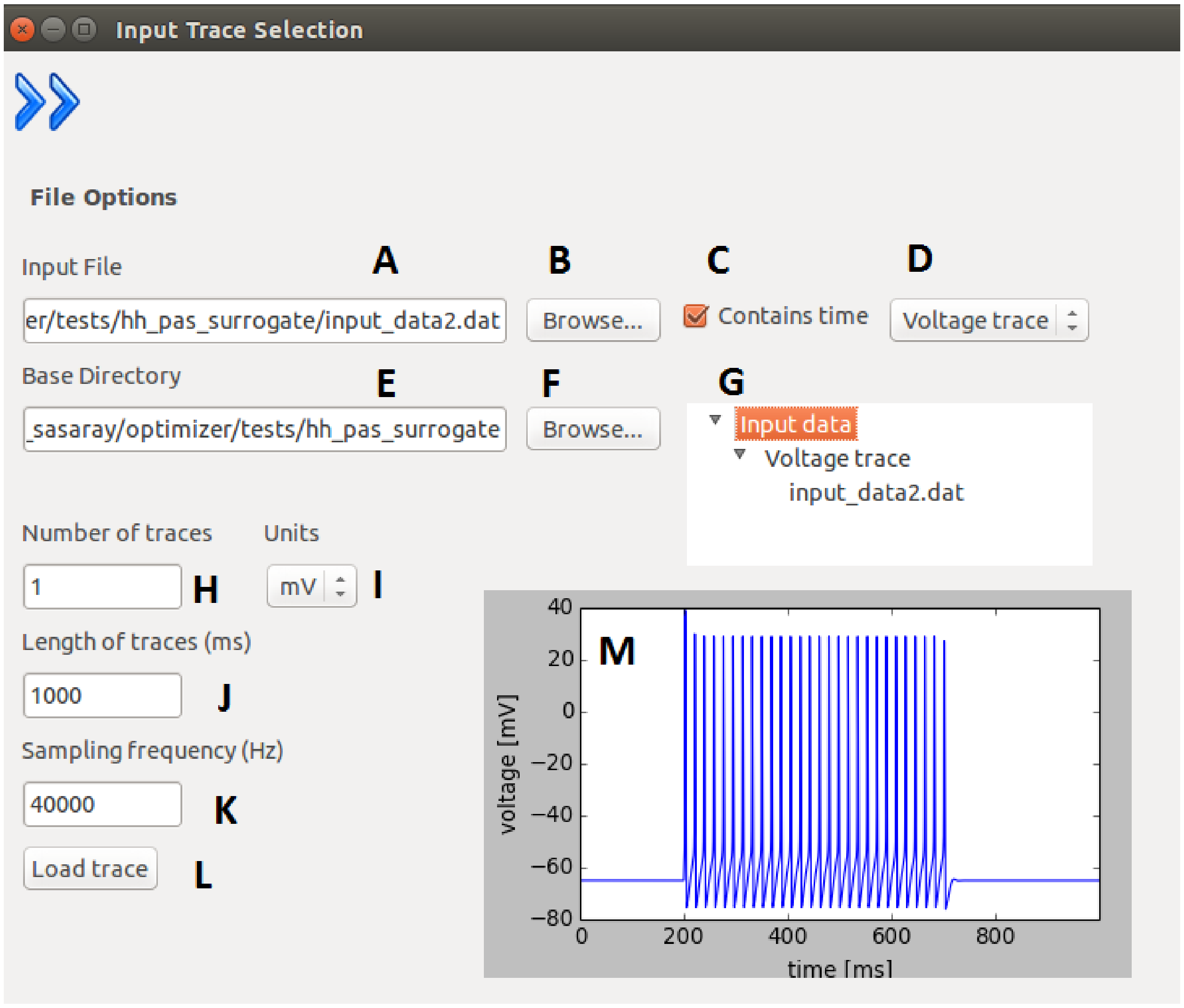
Layer 1

After the program started, the first layer will appear where the user can select the file containing the input trace(s). The user must specify the path to this file, and the working directory (base directory) where the outputs will be written. Apart from these the user must input the requested parameters of the trace set he/she wants to use. After loading the selected file, the user can check if the traces were loaded properly with the help of a plot which displays all the traces concatenated and with the help of a tree display. The concatenation only performed for displaying purposes, the traces are otherwise handled separately. The program only handles one input file (loading a new file will overwrite the existing one), but with arbitrary number of traces.

A	Path to input data.
B	Browse the location of the input data.
C	Input contains time or not.
D	Type of input (voltage or current trace).
E	Path to base directory (resulting files will be stored here).
F	Browse the base location.
G	Input data browser (the loaded file is displayed under it's type).
H	Number of traces in file (trace set).
I	Units of the data.
J	Length of trace(s). (in case of multiple traces, they must have the same length)
K	Sampling frequency (in case of multiple traces, they must have the same).
L	Loads the trace(s) from the given file.
M	Displays the loaded trace (if given file contains more, the trace will be concatenated for displaying).

Layer 2

On the second layer the user can specify the simulator which can be Neuron or external. If the user decided to use Neuron as the simulator then the model file must contain only the necessary structure and mechanisms. The model can be loaded simply after selecting the model file and the special folder where the necessary .mod files are located (optional). As Neuron can not load it's .dlls after startup, if the special files were not found, the software must be



Input File: er/tests/hh_pas_surrogate/input_data2.dat

Contains time: ✓

Voltage trace

Base Directory: _sasaray/optimizer/tests/hh_pas_surrogate

Input data
Voltage trace
input_data2.dat

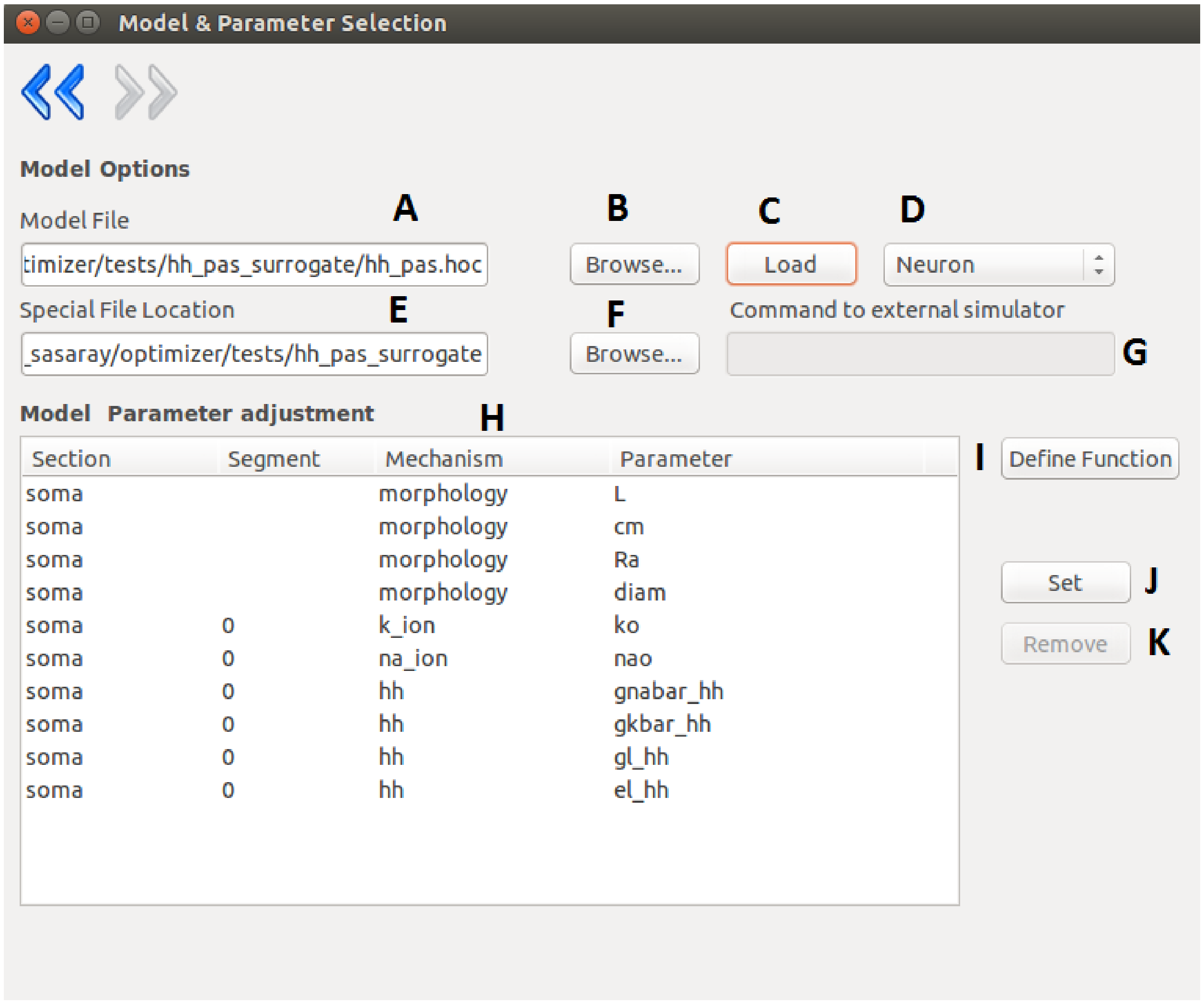
Number of traces: 1

Units: mV

Length of traces (ms): 1000

Sampling frequency (Hz): 40000

restarted. Once the model is loaded successfully, the content of the model will be displayed, and the user can select the parameters by picking them int the list and pressing the “set” button. Removing a parameter is done in a similar fashion. At this point the user can load or define a special function which carries out different tasks during the optimization.



timizer/tests/hh_pas_surrogate/hh_pas.hoc

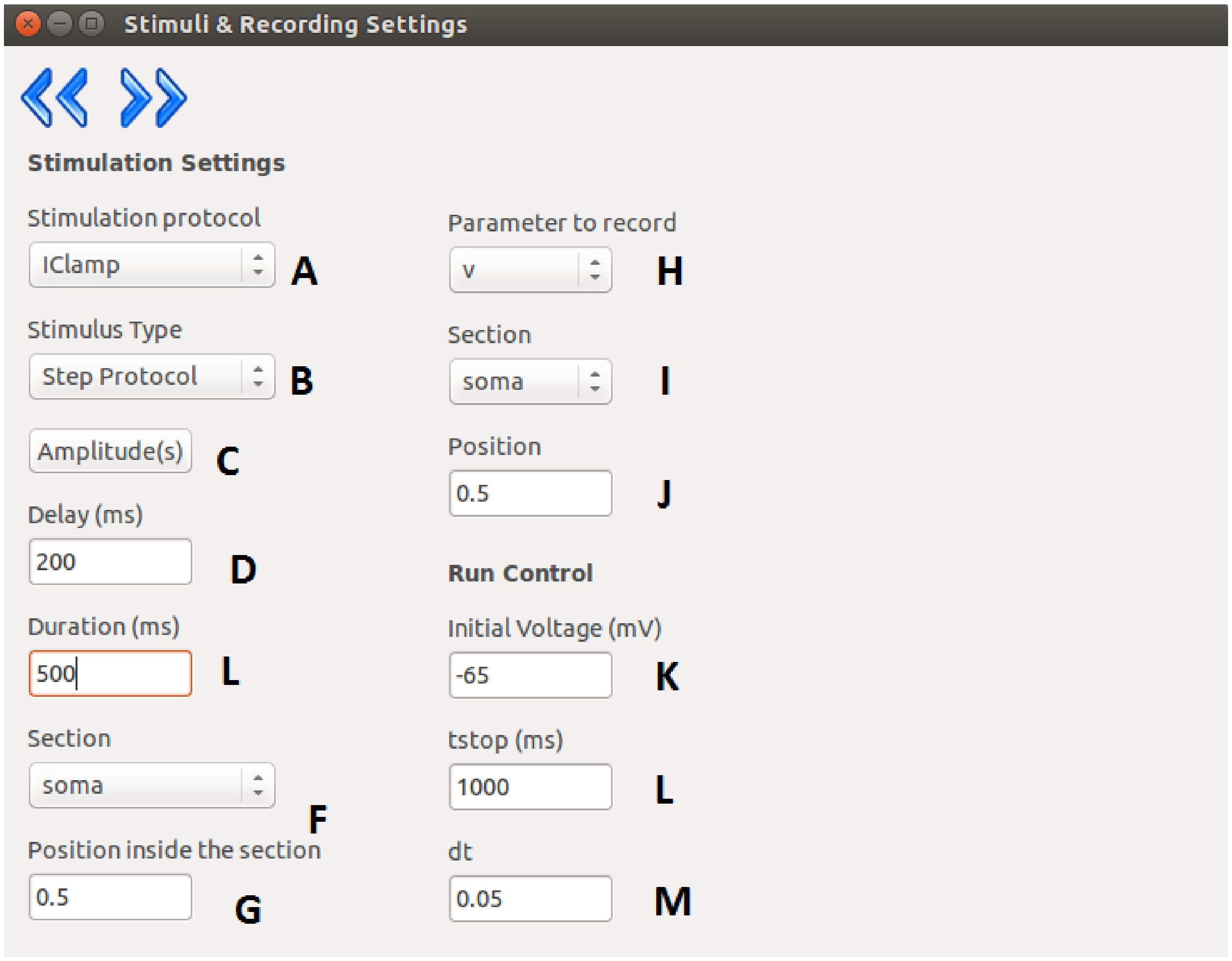
_sasaray/optimizer/tests/hh_pas_surrogate

Section	Segment	Mechanism	Parameter
soma		morphology	L
soma		morphology	cm
soma		morphology	Ra
soma		morphology	diam
soma	0	k_ion	ko
soma	0	na_ion	nao
soma	0	hh	gnabar_hh
soma	0	hh	gkbar_hh
soma	0	hh	gl_hh
soma	0	hh	el_hh

A	Path to model file.
B	Browser for model file.
C	Loads the specified model.
D	Simulator type selection (Neuron or external)
E	Path to special files (the compiled mod files for Neuron), this should point to a folder, which contains the folder of the compiled files (e.g.: to a folder which has an x86-64 directory)
F	Browser for special file location.
G	Here you can give the command which invokes the external simulator. The given command must consists of the following: - the command that calls the simulator - the name of the model - options to the simulator (optional) - as the last parameter, the number of parameters subject to optimization
H	Displays the recognized parameters. These can be selected for optimization. If the parameters you need, are missing, you can create a user defined function.
I	Opens the window to define/load your own function for the optimization.
J	Adds the currently selected parameter to the list of parameters subject to optimization.
K	Removes the parameter from the aforementioned list.

Layer 3

On the second layer the user can specify the simulator which can be Neuron or external. If the user decided to use Neuron as the simulator then the model file must contain only the necessary structure and mechanisms. The model can be loaded simply after selecting the model file and the special folder where the necessary .mod files are located (optional). As Neuron can not load it's .dlls after startup, if the special files were not found, the software must be restarted. Once the model is loaded successfully, the content of the model will be displayed, and the user can select the parameters by picking them int the list and pressing the "set" button. Removing a parameter is done in a similar fashion. As mentioned earlier the functionality of the GUI can be extended by the usage of external files. At this point the user can load or define a special function which carries out different tasks during the optimization. On the next layer the settings regarding stimulation and simulation can be made. The user can select the stimulation protocol which can be either current clamp or voltage clamp (the voltage clamp is implemented as a SEClamp from Neuron). The stimulus type also can be selected, either step protocol or custom waveform. If the step protocol is selected the properties of the step can be specified. In case of multiple stimuli, only the amplitude of the stimuli can vary, no other parameter (position of stimulus, duration, delay, etc) can be changed. Via the GUI the user can specify up to ten stimuli amplitude. The user can make use of external files here as well by selecting the custom waveform as stimulus type. After the stimulation parameters are selected, the user must chose a section and a position inside that section to stimulate the model. In the second column the parameters regarding the simulation and the recording process can be given. The user must give an initial voltage parameter, the length of the simulation and the integration step used for calculations (variable time step methods are not supported yet). After these settings are done, the user can select the parameter to be measured (either current or voltage), the section and the position where the measurement takes place.



Stimulation Settings

Stimulation protocol: IClamp
Stimulus Type: Step Protocol
Delay (ms): 200
Duration (ms): 500
Section: soma
Position inside the section: 0.5

Parameter to record: v
Section: soma
Position: 0.5

Run Control

Initial Voltage (mV): -65
tstop (ms): 1000
dt: 0.05

A	Stimulation protocol (Vclamp or Iclamp).
B	Type of the stimulus (Step protocol or Custom Waveform).
C	Opens the window for specifying step amplitudes or loading custom waveform (depending on the
p	revious options).
D	Delay of stimulus onset.
E	Duration of stimulus.
F	Section which receives stimulus.
G	Point of stimulation inside the section.
H	The parameter to be recorded.
I	The section where the recordings takes place.
J	Position inside the recording section.
K	Initial membrane potential.
L	Length of the recording.
M	Integration step size.

Layer 4

On the next layer the combination of fitness functions can be selected with the desired weights. Optimizer offers weight normalization with the press of a button, but not normalized values are acceptable as well. The user can fine tune the behavior of the functions by giving parameters to them (the value of the same parameter should be the same across the functions).

A	List of available fitness function.
B	Weight assigned to the selected function.
C	Parameters passed to the fitness functions.
D	Normalizes the weights (not necessary).

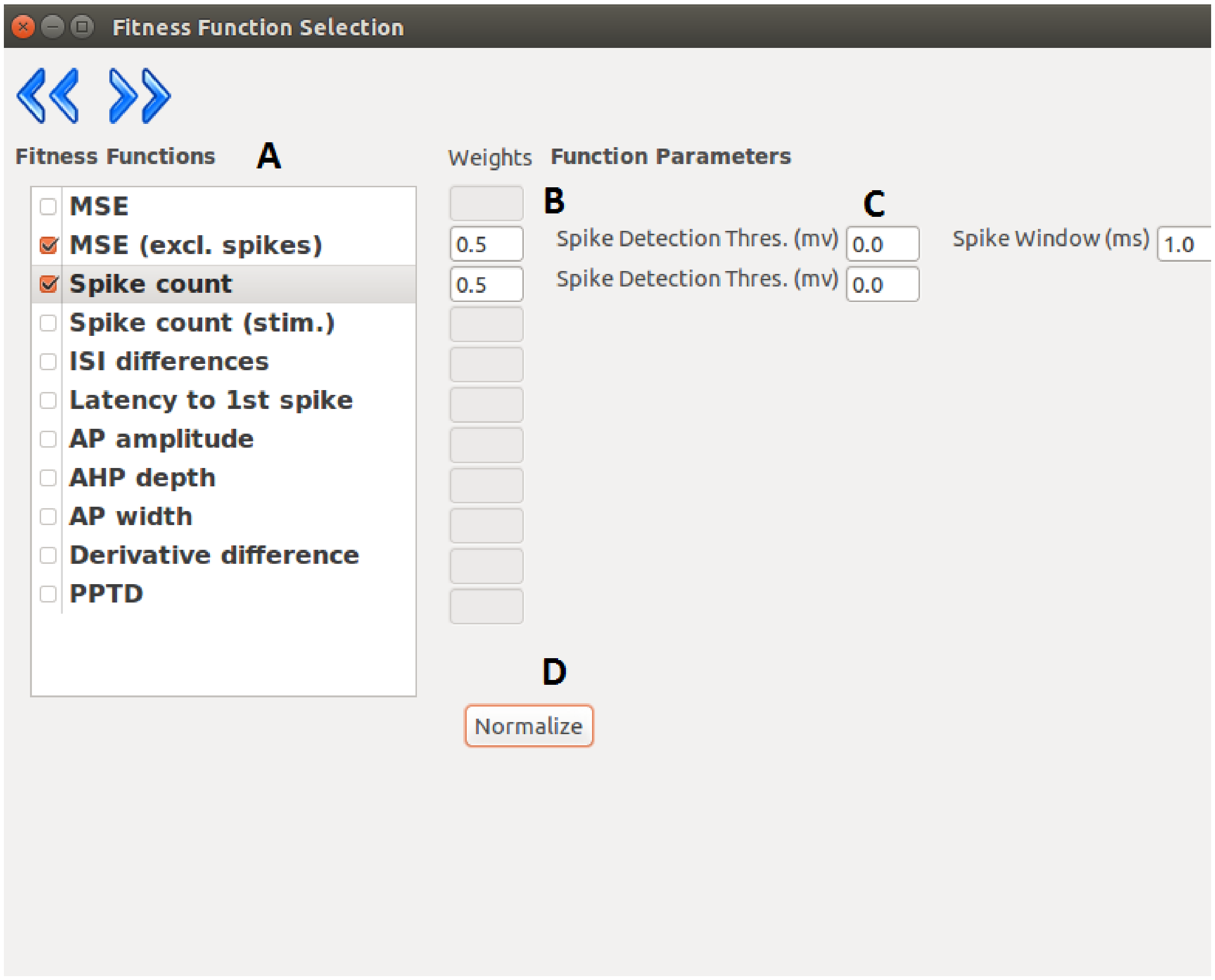
Layer 5

On the next layer, the user can select the desired algorithm from the current list and tune the parameters of it. Since optimizing neuron models is a bounded optimization problem the program requires boundaries for the parameters. The user can give a set of values as starting points to the algorithm which will be interpreted differently, depending on the used algorithm. In the case of the global algorithms the given set of values will be included in the initial set of parameters. In the case of the local algorithms the algorithm will start form the point specified by the parameters.

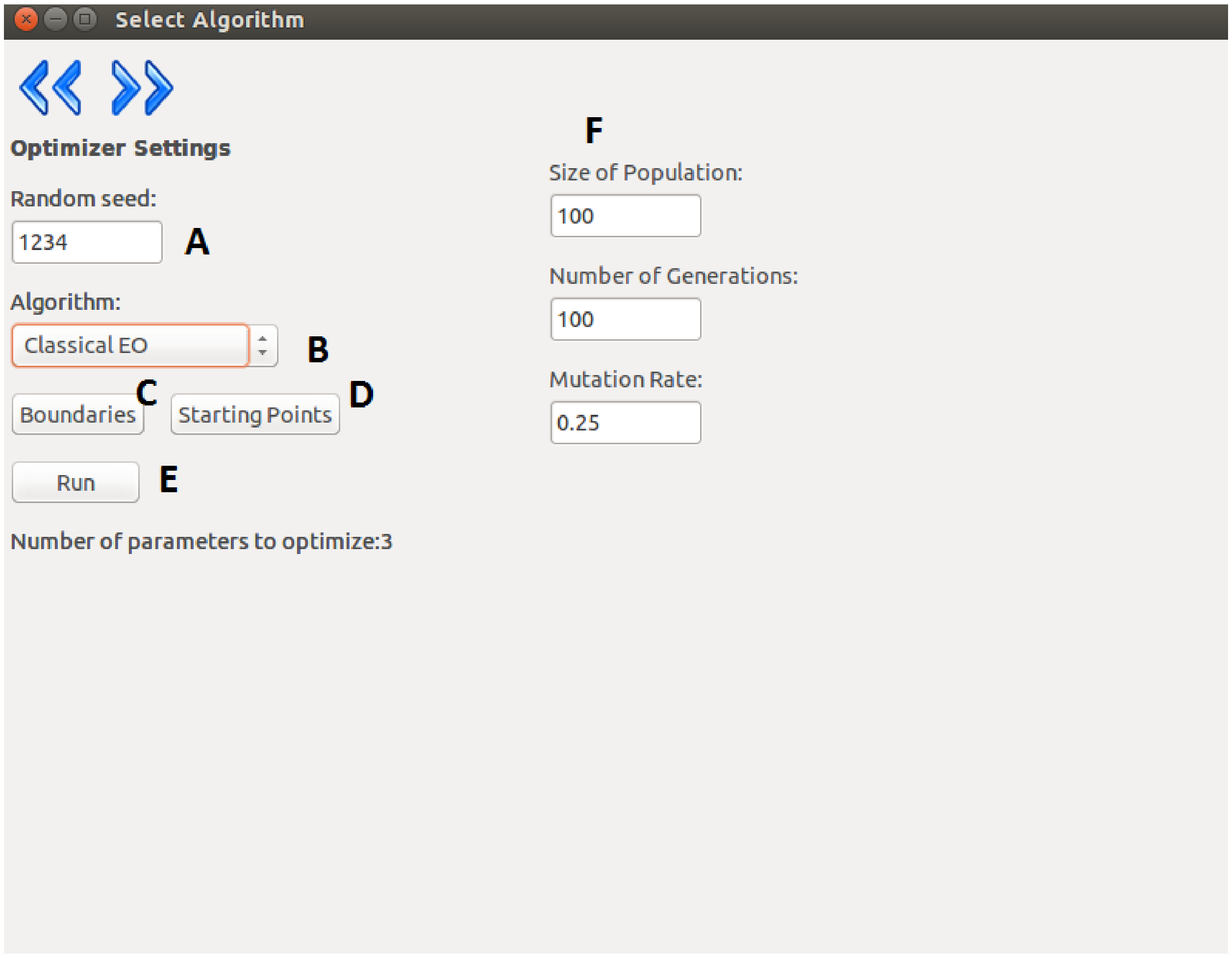
A	Seed for the random generator.
B	Selected algorithm.
C	Boundaries of the parameters subject to optimization.
D	Starting points
E	Run the optimization.
F	Depending on the selected algorithm, different settings will appear here.

Layer 6

A	The resulting parameters.
B	The trace(s) obtained with the resulting parameters.



- [ ] MSE
- [✓] MSE (excl. spikes) — Weight: 0.5 — Spike Detection Thres. (mv) 0.0 — Spike Window (ms) 1.0
- [✓] Spike count — Weight: 0.5 — Spike Detection Thres. (mv) 0.0
- [ ] Spike count (stim.)
- [ ] ISI differences
- [ ] Latency to 1st spike
- [ ] AP amplitude
- [ ] AHP depth
- [ ] AP width
- [ ] Derivative difference
- [ ] PPTD



Random seed: 1234

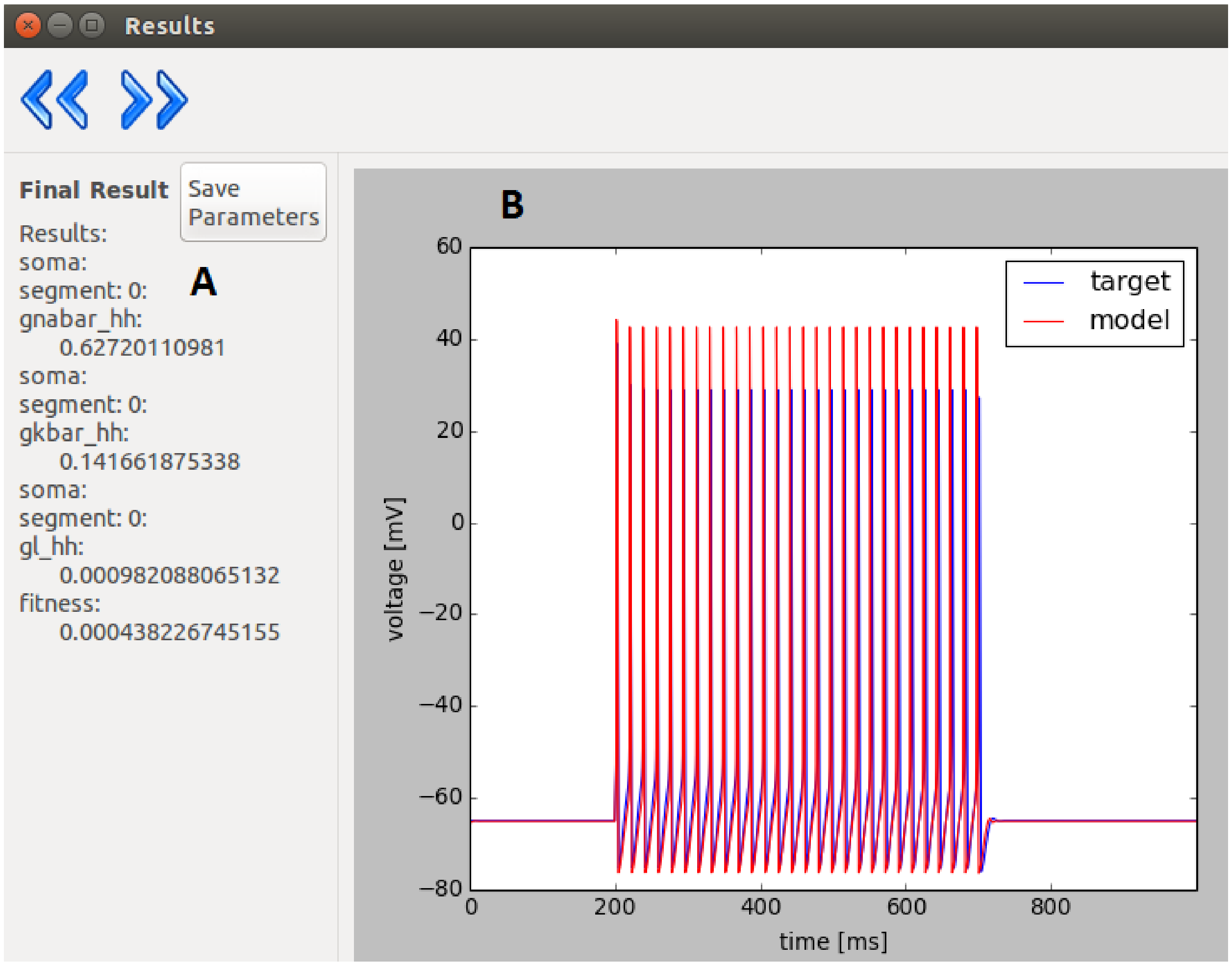
Algorithm: Classical EO

Number of parameters to optimize:3

Size of Population: 100

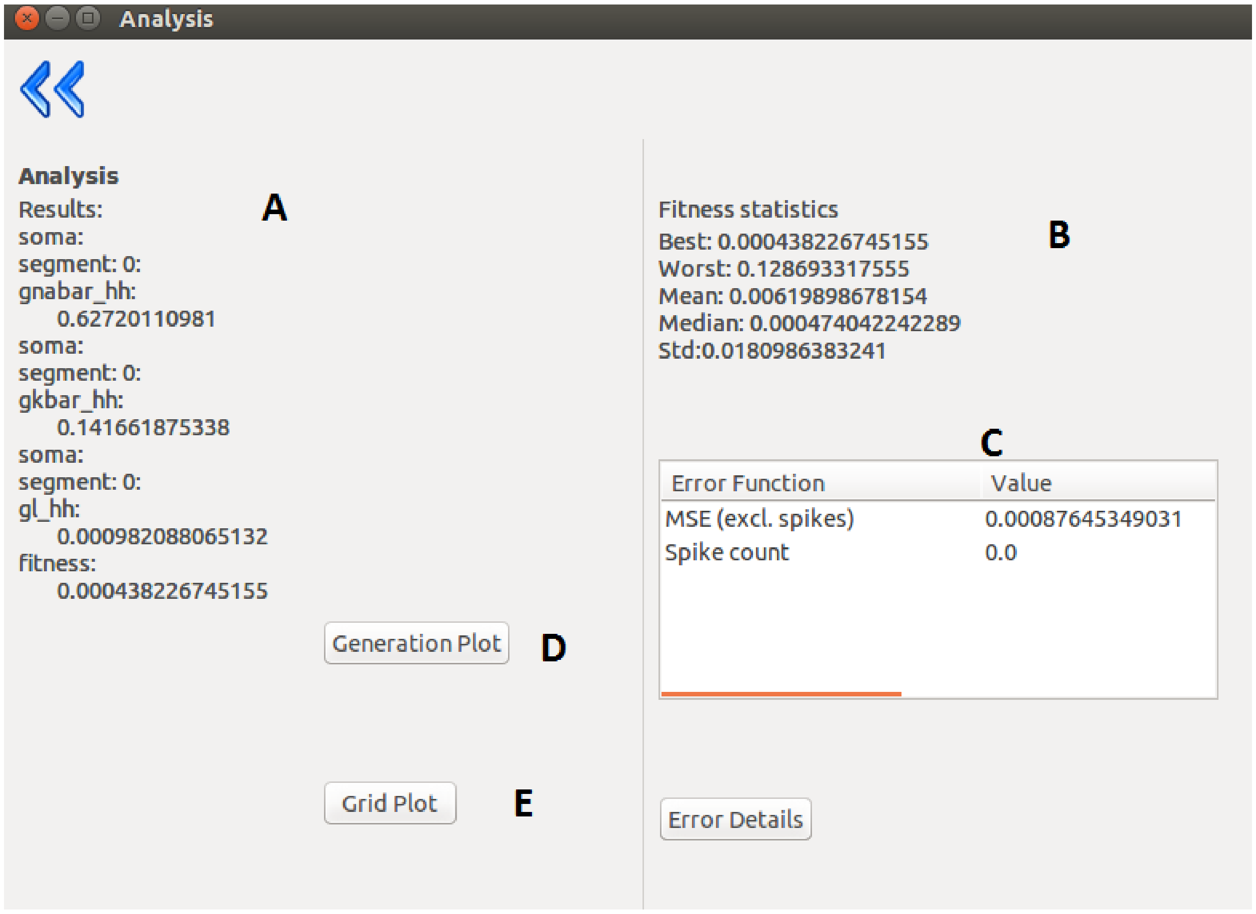
Number of Generations: 100

Mutation Rate: 0.25



Results:
soma:
segment: 0:
gnabar_hh:
    0.62720110981
soma:
segment: 0:
gkbar_hh:
    0.141661875338
soma:
segment: 0:
gl_hh:
    0.000982088065132
fitness:
    0.000438226745155

Layer 7



Analysis
Results:
soma:
segment: 0:
gnabar_hh:
    0.62720110981
soma:
segment: 0:
gkbar_hh:
    0.141661875338
soma:
segment: 0:
gl_hh:
    0.000982088065132
fitness:
    0.000438226745155

Fitness statistics
Best: 0.000438226745155
Worst: 0.128693317555
Mean: 0.00619898678154
Median: 0.000474042242289
Std:0.0180986383241

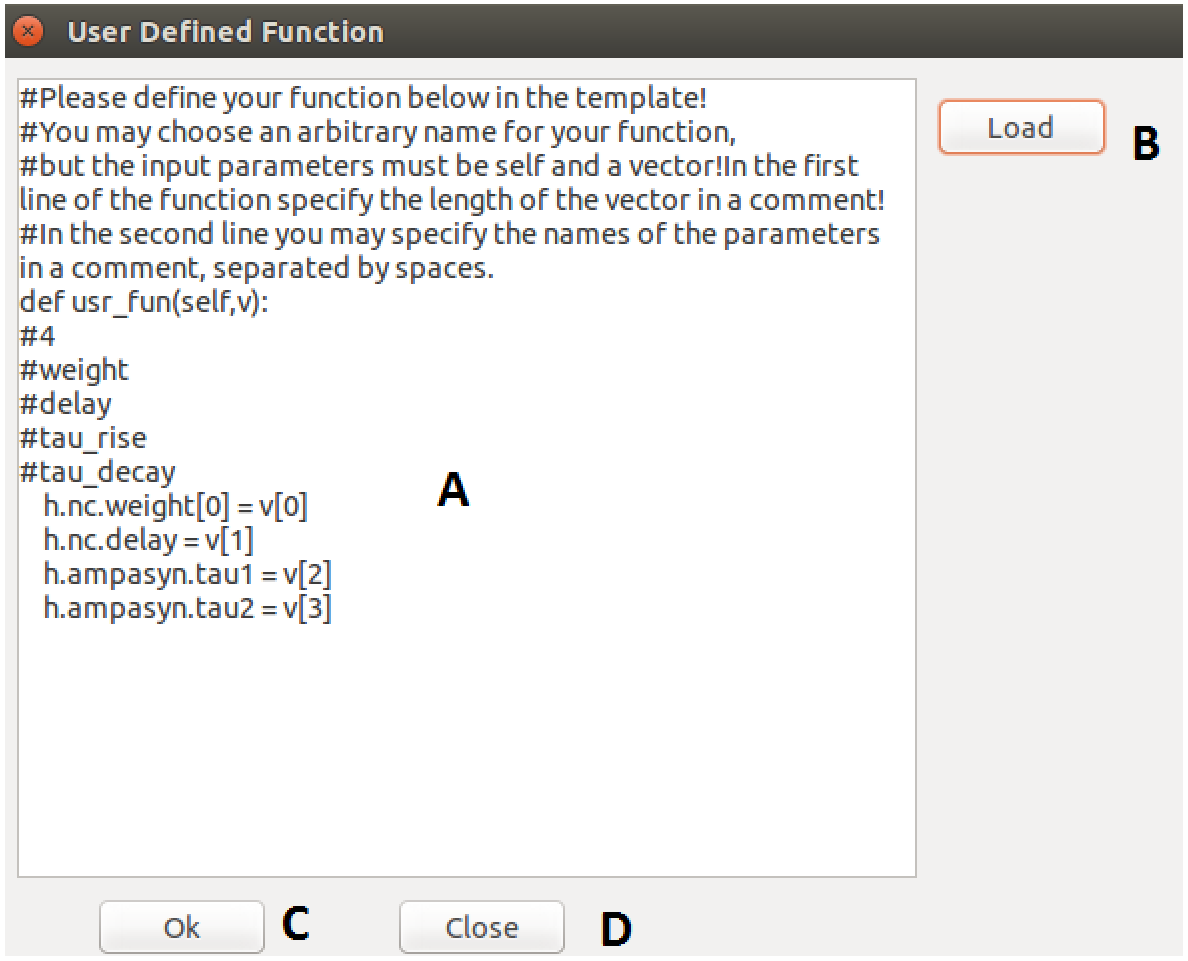
Error Function	Value
MSE (excl. spikes)	0.00087645349031
Spike count	0.0

A	The obtained parameters.
B	Fitness statistics (only in case of inspyred algorithms: Classical EO, SA)
C	Fitness components: name of fitness function; fitness value;calculated by the function; weight assigned to the function; the weighted fitness value; the resulting cumulated fitness value.
D	Displays the “state” of the population during the evolution. (only for inspyred algorithms)
E	Displays the given proximity of the optimum.

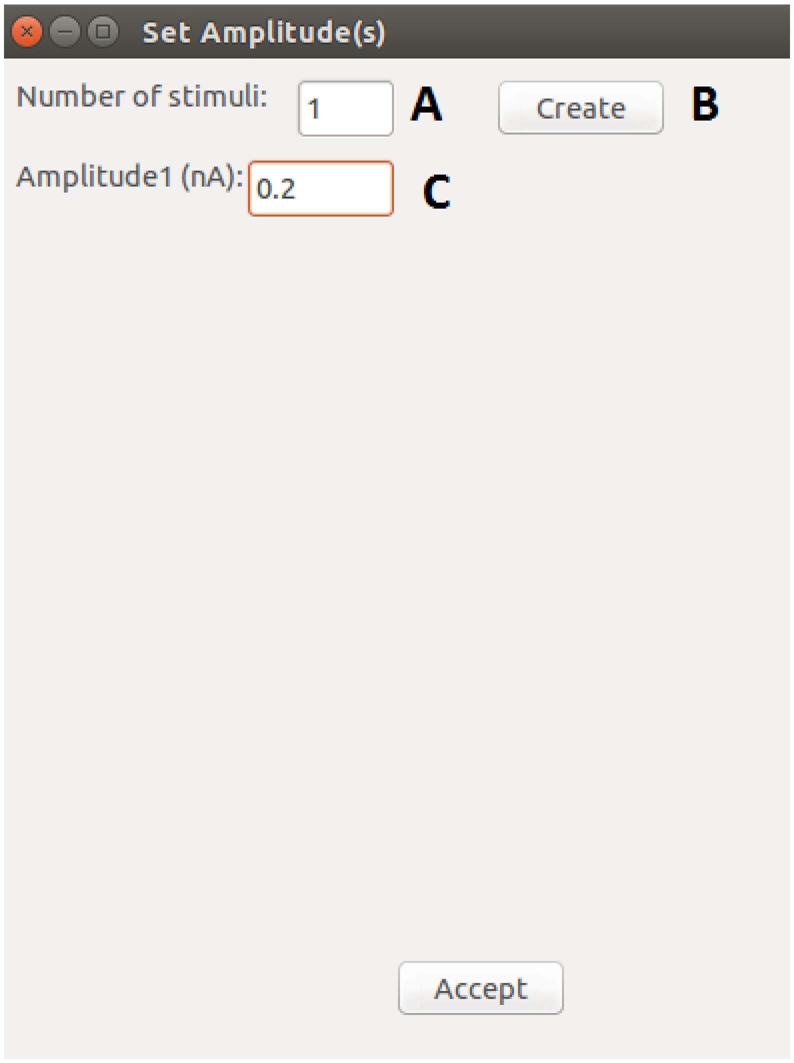
Other windows and layers

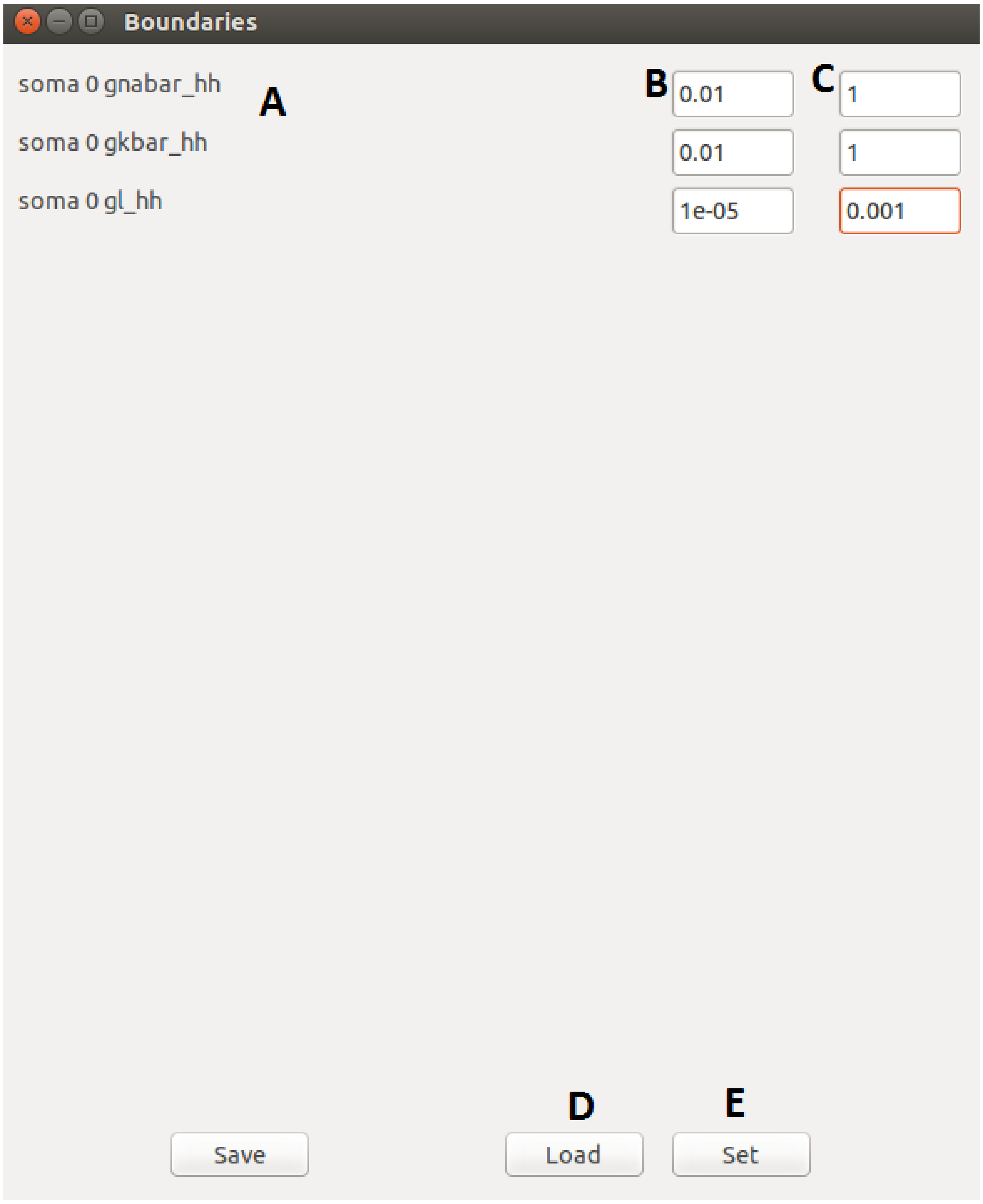
A	Entry field for function definition.
B	Load a previously defined function from a txt.
C	Done editing, save function and continue.
D	Discard function and go back.

A	Number of stimuli.
B	Create the specified number of stimuli.
C	Specify the amplitude of the stimuli.



```
#Please define your function below in the template!
#You may choose an arbitrary name for your function,
#but the input parameters must be self and a vector!In the first
line of the function specify the length of the vector in a comment!
#In the second line you may specify the names of the parameters
in a comment, separated by spaces.
def usr_fun(self,v):
#4
#weight
#delay
#tau_rise
#tau_decay
  h.nc.weight[0] = v[0]
  h.nc.delay = v[1]
  h.ampasyn.tau1 = v[2]
  h.ampasyn.tau2 = v[3]
```


Set Amplitude(s)
Number of stimuli:
1
A
Create
B
Amplitude1 (nA):
0.2
C
Accept



soma 0 gnabar_hh	0.01	1
soma 0 gkbar_hh	0.01	1
soma 0 gl_hh	1e-05	0.001

A	The list of selected parameters.
B	Lower bounds.
C	Upper bounds.
D	Load boundaries from file.
E	Boundaries are set, continue.

CHAPTER 3

optimizer Package

Core Module

cmd_line Module

fitnessFunctions Module

graphic Module

modelHandler Module

class optimizer.modelHandler.**externalHandler**(*command*)

Handles models which are using a simulator other than Neuron. :param command: the command string which should be executed

Note:

The command must consist of the following parts:

- the command to execute
- the model file
- options (optional)
- number of parameters to optimize

CreateStimuli(*s*)

GetExec()

Creates the command that runs the simulator with the model and with the appropriate options. :return: a `list` of strings ready for execution

GetParameters()

SetNParams(*o*)

Sets the number of parameters in the given object by calling it's `SetObjTOOpt` method.

Parameters **o** – the object whose method will be called

Note: This is necessary because the other parts expects that the option handler objects knows the parameters subjects to optimization. Since this is not true in the case of an external simulator, this workaround is needed.

SetStimuli(*p*, *e*)

class `optimizer.modelHandler.`**modelHandlerNeuron**(*model_path*, *special_path*, *base='/home/docs/checkouts/readthedocs.org/user_builds/optimize*

Imports the necessary modules to handle Neuron models and loads the model as well as the additional mechanisms. Creates containers for the sections and the channels for easier handling.

Parameters

- **model_path** – the path to the model file
- **special_path** – the path to the special file (.mod files)
- **base** – the base working directory

CreateStimuli(*stims*)

Creates a Neuron pointprocess which is responsible for the stimulation of the model.

Note: The type of the point process is either an `IClamp` or a `SEClamp`.

Parameters **stims** – a `list` with the following values:

- stimulation type as `string`
- position inside section
- name of the section

GetParameters()

Collects every member of every section object and filters out those that are not parameters of the model. The function will collect:

- every parameter of the the mechanisms
- every mechanism
- some default parameters that are always included in a model, and pointprocesses that are not some sort of Clamp

Returns the filtered content of the model in a string matrix

Recordings(*vector*)

Converts the hoc vector obtained from the simulation and converts it into a `Trace` object.

Parameters **vector** – a hoc vector

Returns the data trace from the created object

RunControll (*settings*)

Sets up the recording procedure and the simulation, then runs it.

Parameters **settings** – the settings of the recording and the parameters of the simulation:

- length of simulation
- integration step size
- parameter to record
- section to record from
- position inside the section
- initial voltage

SetChannelParameters (*section*, *segment*, *channel*, *params*, *values*)

Sets the given channel's parameter to the given value. If the section is not known that indicates a serious internal error and the program will abort.

Parameters

- **section** – the selected section's name as `string`
- **channel** – the selected channel's name as `string`
- **params** – the selected channel parameter's name as `string`
- **values** – the value to be set

SetCustStimuli (*params*)

Uses the vector.play method from Neuron to create a time varying stimulus. The stimulus is read from the given file.

Parameters **params** – `list` with the name of the file containing the stimulus as first element

Note: The delay value must be set to zero and the duration must be set to 1e9, but these are not the actual parameters of the stimulus. This is necessary for Neuron in order to work.

SetMorphParameters (*section*, *params*, *values*)

Sets the given morphological parameter to the given value. If the section is not known that indicates a serious internal error and the program will abort. If the section has no parameter with the given name then it is interpreted as a parameter of a pointprocess and the function will set the parameter assuming the pointprocess exists in the middle (0.5) at the given section and there is only one other pointprocess in the section.

Note: This workaround is implemented because some mechanisms are implemented as pointprocesses.

param section the name of the section as `string`

param params the name of the parameter as `string`

param values the value to set

SetStimuli (*params*, *extra_params*)

Sets the parameters of the stimulating object. The parameters are the following:

- amplitude
- delay
- duration

or

- amplitude1
- amplitude2
- amplitude3
- duration1
- duration2
- duration3

Parameters

- **`params`** – the `list` of parameters containing the first 3 values from the above list
- **`extra_params`** – `list` of parameters containing additional values to set up the `SEClamp`

Note: The rs parameter of the `SEClamp` is set to 0.01

`contains`(*string*, *ss*)

Checks if substring is in the given `list` and creates a string which contains only the matching elements separated by spaces.

Parameters

- **`string`** – `list` of strings
- **`ss`** – the substring to be matched

Returns a string which contains only the matching elements separated by spaces

`optimizer` Module

`optimizer.optimizer.`**`main`**(*parameters*)

The main function, which starts to software according to the given command line arguments.

Parameters **`parameters`** – the command line parameters: * -h help * -c command line * -g graphic interface

`optimizerHandler` Module

`optionHandler` Module

class `optimizer.optionHandler.`**`optionHandler`**

Bases: `object`

Object to store the settings required by the optimization work flow.

GetFileOption()

Returns the current working directory (referred as base in modelHandler, used in traceReader)

GetFitnessParam()

Gets the parameters required by the fitness functions:

- **`list` consisting of:**
  - a `dictionary` containing the spike detection threshold and the spike window
  - a `list` of fitness function names
- `list` of weights to combine the fitness functions

Returns a `list` containing the structures described above

GetInputOptions()

Gets the input related settings:

- input file
- number of traces in file
- unit of input
- length of the individual traces (see traceHandler)
- sampling frequency of the trace(s)
- flag indicating if file included time scale or not (will be removed, see traceHandler)
- the type of the trace(s)

Returns the parameters listed above in a `list`

GetModelOptions()

Gets the model related options:

- path to the model
- path to the directory containing the special files (see modelHanlder)

Returns the parameters listed above in a `list`

GetModelRun()

Gets the parameters corresponding to the simulation:

- length of simulation
- integration step
- parameter to record
- section name
- position inside the section
- initial voltage

Returns the parameters above in a `list`

GetModelStim()

Gets the parameters regarding the stimulus type:

- type of the stimulus
- position of stimulus
- name of the stimulated section

Returns the parameters listed above in a `list`

GetModelStimParam()

Gets the parameters of the stimulus:

- amplitude
- delay
- duration

Returns the parameters listed above in a `list`

GetObjTOOpt()

Gets the parameters selected to optimization.

Returns a `list` of `strings`

GetOptParam()

Not in use! Gets the list of parameter values corresponding to the parameters subject to optimization.

Returns `list` of real values

GetOptimizerOptions()

Gets the parameters regarding the optimization process:

- seed: random seed
- evo_strat: name of evolution algorithm
- Size of Population: size of population
- Number of Generations: number of generations
- Mutation Rate: mutation rate (0-1)
- Cooling Rate: cooling rate (0-1)
- Mean of Gaussian: mean value of gaussian
- Std. Deviation of Gaussian: standard deviation of gaussian
- Cooling Schedule: index of cooling schedule
- Initial Temperature: initial temperature
- Final Temperature: final temperature
- Accuracy: accuracy
- Dwell: number of evaluation on the given temperature level
- Error Tolerance for x: error tolerance for input values
- Error Tolerance for f: error tolerance for fitness values

- num_params: number of input parameters
- boundaries: bounds of the parameters
- starting_points: initial values to the algorithm

Returns a `dictionary` containing the parameters above

GetSimParam()

Gets the simulator related parameters:

- the name of the simulator
- the command which should be executed to run the model (see modelHandler)

Returns the parameters listed above in a `list`

GetUFunString()

Gets the user defined function.

Returns the function as a `string`

SetFileOptions(*options*)

Sets the current working directory

Parameters **`options`** – the path of the directory

SetFitnesParam(*options*)

Sets the parameters required by the fitness functions.

Parameters **`options`** – the required values in the structure described in `GetFitnessParam`

SetInputOptions(*options*)

Sets the options related to the input to the given values.

Parameters **`options`** – a `list` of values (order of parameter should be the same as listed in `GetInputOptions`)

SetModelOptions(*options*)

Sets the model related options.

Parameters **`options`** – a `list` of values

SetModelRun(*options*)

Sets the parameters regarding the simulation to the given values.

Parameters **`options`** – `list` of parameters

SetModelStim(*options*)

Sets the parameters regarding the stimulus type to the given values.

Parameters **`options`** – `list` of values

SetModelStimParam(*options*)

Sets the parameters of the stimulus to the given values.

Parameters **`options`** – `list` of values

Note: Only the parameters of the IClamp are stored this way since the parameters of the SEClamp are obtained by combining the values here and the values regarding the simulation.

`SetObjTOOpt`(*options*)
Adds the given parameter to the list of parameters selected for optimization.

Parameters **`options`** – a `string` containing the section, a channel name and a channel parameter name, or a morphological parameter separated by spaces

Note: If a given parameter is already stored then it will not added to the list.

`SetOptParam`(*options*)
Not in use! Adds the given value to the list of parameter values corresponding to the parameters subject to optimization.

Parameters **`options`** – a real value

`SetOptimizerOptions`(*options*)
Sets the parameters regarding the optimization process.

Parameters **`options`** – a `dictionary` containing the parameters

`SetSimParam`(*options*)
Sets the simulator related parameters.

Parameters **`options`** – a `list` of values

`SetUFunString`(*s*)
Sets the user defined function.

Parameters **`s`** – the function as a `string`

`dump`(*f_mapper*)
Dumps the content of the class into a string.

Parameters **`f_mapper`** – a `dictionary` that maps the fitness function objects to their names (used in the GUI)

Returns the content of the class as `string`

`read_all`(*root*)
Reads settings from an xml tree and converts them to the necessary type.

Parameters **`root`** – the root of the xml tree

Note: If there is an element in the tree whose tag is not a valid option name, then `AttributeError` is raised.

Note: The program does not verify if every parameter which are needed to the current process is present. We strongly recommend that you use the GUI to create a configuration file, which will contain the needed values, instead of writing the xml file by hand.

`optimizer.optionHandler.`**`prettify`**(*e*)
Converts the given xml tree object to human readable form.

Parameters **`e`** – the xml tree element

Returns the reformatted content of the xml tree as `string`

`traceHandler` Module

class `optimizer.traceHandler.`**`DATA`**

The main data container class. :attr: data: holds the `Trace` object which contains the trace set

Note: This class will be able to hold multiple data sets with multiple types.

`PyNNReader`(*path*, *no_traces*, *scale*, *t_length*, *freq*, *trace_type*)

Reads a default recording result from PyNN with 9 line of headers (the parameters are the same as in the `Read` function)

Returns a `trace` object holding the content of the file

Note: If the given file is not accessible the program will abort.

Note: If the number of traces in the file and the corresponding parameter is not equal, a `sizeError` is raised.

`Read`(*path=['/home/docs/checkouts/readthedocs.org/user_builds/optimizer/checkouts/latest/doc/inputTrace.txt'], no_traces=1, scale='mV', t_length=1000, freq=1000, trace_type='voltage'*)

The main reader function. This calls the recognition function `detect_format` and uses the obtained reader function to read the data.

Parameters **`path`** – list of data path(s) (currently only one file is handled)

(see the explanation of the other parameters in the description of `Trace`)

Note: The function uses the 5. line of the file for recognition.

`abstractDataReader`(*path*)

`convert`(*dict_to_conv*)

`detect_format`(*line*)

Automatically detects the format of the file and returns a reader functions which can process it correctly. The recognized formats are the following:

- simple text file with data columns separated by "\t"
- simple text file, containing time trace as well and data columns separated by "\t"
- default recording result from PyNN with 9 line of headers
- spike timing file from PyNN with 9 line of headers (not used)

Parameters **`line`** – one line from the file, which the recognition is based on

Returns a reader function

`get_type`()

Gets the type of the trace set.

Returns type

`number_of_traces`()
Gets the number of traces held by the object.

Returns number of traces

`spikeTimeReader`(*path*, *no_traces*, *scale*, *t_length*, *freq*, *trace_type*)
Not available yet! Reads a spike timing file from PyNN with 9 line of headers (the parameters are the same as in the `Read` function)

Returns a `dictionary` object holding the content of the file

Note: If the given file is not accessible the program will abort.

Note: If the number of traces in the file and the corresponding parameter is not equal, a `sizeError` is raised.

Note: In the future, it will return a `SpikeTimes` object instead.

`traceReader`(*path*, *no_traces*, *scale*, *t_length*, *freq*, *trace_type*)
Reads a simple text file with data columns separated by " ". (the parameters are the same as in the `Read` function)

Returns a `trace` object holding the content of the file

Note: If the given file is not accessible the program will abort.

Note: If the number of traces in the file and the corresponding parameter is not equal, a `sizeError` is raised.

`traceReaderTime`(*path*, *no_traces*, *scale*, *t_length*, *freq*, *trace_type*)
Reads a simple text file, containing time trace as well and data columns separated by "t". (the parameters are the same as in the `Read` function)

Returns a `trace` object holding the content of the file

Note: If the given file is not accessible the program will abort.

Note: If the number of traces in the file and the corresponding parameter is not equal, a `sizeError` is raised.

class `optimizer.traceHandler.`**`SpikeTimes`**(*dictionary*, *trace_type='spikes'*)
Not in use! Stores spike times in `dictionary` indexed by the cell's id.

Parameters

- **`dictionary`** – the `dictionary` which contains the data
- **`trace_type`** – type of the trace (should be fixed to "spikes")

class `optimizer.traceHandler.`**`Trace`**(*no_traces*, *scale='milli'*, *t_length=1000*, *freq=100*, *trace_type=None*)

Trace set object. Stores a trace set of a given type with every relevant data.

Parameters

- **`no_traces`** – number of traces held by the object
- **`scale`** – the unit of the data (required for conversions)
- **`t_length`** – length of the trace(s)
- **`freq`** – sampling frequency
- **`trace_type`** – type of the trace set

Attr data: the traces are contained in the attribute named `data` in the order they were in the input file

Note: The sampling rate should be uniform in the set.

Note: The length of the traces should be the same, or the length of the shortest one should be considered.

Note: If the type of the trace is not recognized, the program will abort. The recognized types are "voltage", "current" and "other" ("spike" is not available yet).

`Convert`(*hoc_obj*)

Converts a hoc vector into a python `list` and stores it in the container.

Parameters **`hoc_obj`** – a hoc vector object

`GetTrace`(*index*)

Returns the trace having the index `index` form the container. Always use this function to get a given trace as they are stored in a non intuitive way and direct access would probably cause errors.

Parameters **`index`** – the index of the trace to get

Note: If the given index is out of range, a `sizeError` is raised.

Returns the required trace

`Print`()

Prints the contained data. Created for debugging purpose.

`SetTrace`(*d*)

Adds the given trace to the container.

Parameters **`d`** – `list` containing the trace

`reScale`(*value*)

Re-scales the given value based on the scale of the `Trace` object.

Parameters **`value`** – the value to be rescaled

Returns the rescaled value

`optimizer.traceHandler.`**`real_range`**(*start*, *step*, *end*)

Not in use! Generates real values from the given range.

Parameters

- **`start`** – begin of range
- **`step`** – step between the values
- **`end`** – the end of the range

Returns `list` of real values

exception `optimizer.traceHandler.`**`sizeError`**(*message*)

Bases: `exceptions.Exception`

Exception class used by the trace handling related objects.

Parameters **`message`** – error message to be displayed

class `optimizer.traceHandler.`**`traceWriter`**(*tr_object*, *full_path*, *comment=''*, *flag_write=1*, *sep='n'*, *flag_multi=0*)

Bases: *`optimizer.traceHandler.Trace`*

Not used! Writes the content of the given trace object to the given file(s).

Parameters

- **`tr_object`** – trace object which must have `data` attribute
- **`full_path`** – the path of the output file
- **`comment`** – some header information
- **`flag_write`** – indicates if the properties of the trace is written or not
- **`sep`** – data separator `string`
- **`flag_multi`** – indicates if the separate traces should be written into separate files

`Write`()

Performs the writing.

Note: If `flag_multi` was set to true, then the traces will be written to multiple files.

CHAPTER 4

optimizer

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index