PCSIM : A *P*arallel Neural *C*ircuit *SIM*ulator

Version 0.5.0

User Manual

April 19, 2008

The PCSIM group
Institute for Theoretical Computer Science
Graz University of Technology
Inffeldgasse 16/b, A-8010 Graz, AUSTRIA
www.igi.tugraz.at/pcsim

Contents

Chapter 1

Preliminaries

1.1 What is PCSIM?

PCSIM is a tool for simulating heterogeneous networks composed of different model neurons and synapses. This simulator is written in C++ with a primary interface to the programming language python . It is intended to simulate networks containing up to millions of neurons and on the order of billions of synapses. This is achieved by distributing the network over different nodes of a compute cluster by using MPI.

1.2 About this Manual

THIS MANUAL IS UNDER CONSTRUCTION!

This manual is intended to describe how to use PCSIM from the (python) users point of view. It *does not* try to explain (or give an introduction to) the type of models which can be simulated with PCSIM . Regarding neural modeling we refer the reader to [Dayan and Abbott, 2001] and [Gerstner and Kistler, 2002]. Furthermore python programming knowledge is assumed.

This manual is also available in HTML format.

1.3 Features of the current version

Easy to use python interface Since PCSIM is incorporated into python it is not necessary to learn any other script laguage to set up the simulation. This is all done with python scripts. Furthermore the results of a simulation are directly returned as python arrays and hence any plotting and analysis tools available in python (via the matplotlib package) can easily be applied.

Distributed Simulation Via MPI

Different levels of modeling Different neuron models: leaky-integrate-and-fire neurons, compartmental based neurons, sigmoidal neurons. Different synapse models: static synapses and a certain model of dynamic synapses are available for spiking as well as for sigmoidal neurons. Spike time dependent synaptic plasticity is also implemented.

Object oriented design We adopted an object oriented design for PCSIM which is similar to the approaches taken in GENESIS and NEURON. That is there are objects (e.g. a `LifNeuron` object implements the standard leaky-integrate-and-fire model) which are interconnected by means of some signal channels. The creation of objects, the connection of objects and the setting of parameters of the objects is controlled at the level of python scipts whereas the actual simulation is done in the fast C++ core.

Fast C++ core Since PCSIM is implemented in C++ and is not yet as general as GENESIS or NEURON simulations are performed quite fast. We also implemented some ideas from event driven simulators like SpikeNet which result on an average speedup of 3 (assuming an average firing rate of the neurons of 20 Hz and short synaptic time constants) compared to a standard fixed time step simulation scheme.

GPL Licensed PCSIM is distributed under the GNU General Public License

Chapter 2

Installing PCSIM

2.1 Recipe for the impatient

Theoretically the following commands are sufficient to install PCSIM globally on your Linux box if you have the neccessary write permissions and all the required software packages (Sec. 2.2) are already properly installed:

```
wget 'http://downloads.sourceforge.net/pcsim/pypcsim-0.5.0.tar.gz'
tar xvzf pypcsim-0.5.5.tar.gz
cd pypcsim-0.5.5
python setup.py install
python setup.py test
```

2.2 Dependencies

PCSIM depends on several third party software packages. However, most modern Linux distributions have most of the necessary software packages in their package repositories, or for some there are binary packages (rpm or dpkg) on the home page of the software. If possible, for the sake of simplicity of the installation process, the user should prefer installing these binary packages, instead of compiling them from the source release. For example for openSUSE 10.2 and 10.3 with the exception of GCCXML and the pure python packages pygccxml and Py++ for any of the following packages there is a binary RPM distribution available which can conviniently be installed using Yast or easiliy be found via `http://packages.opensuse-community.org/`.

In the following we list all the packages needed in order to compile PCSIM from its source distribution. However only packages marked with ** are strictly required after PCSIM is compiled (e.g. when using a pre-compiled PCSIM library). Packages marked with * are neccessary only if PCSIM is linked dynamically against them (which is preferable and hance the default).

1. *Python ($\geq$2.4)*** The primary interface for using PCSIM is based on the modern scriptin language python. This languae is used to setup and control PCSIM simulations.

Home: http://www.python.org, Download: http://www.python.org/download/

2. *CMake (≥2.4-patch8)* CMake is the build tool which is used to compile the C++ part of PCSIM .

   Home: http://www.cmake.org, Download: http://www.cmake.org/HTML/Download.html

3. *Doxygen (≥1.5.3)* Doxygen is used to generate source documentation and to parse C++ source code to generate certain kind of wrapper code needet to allow for convienient access of properties of simulated objects.
   Home: http://www.doxygen.org, Download: http://www.stack.nl/~dimitri/doxygen/download.html

4. *elementtree* This python module is used to parse the XML output of doxygen to generate the above mentioned wrapper code. As of python 2.5 this is contained in the python distribution.
   Home: http://www.effbot.org/zone/element-index.htm, Download: http://www.effbot.org/downloads/#elementtree

5. *Boost C++ (≥1.33.1)** These libraries can not be avoided for any serious C++ project.
   Home: http://www.boost.org, Download: http://sourceforge.net/project/showfiles.php?group_id=7586

6. *MPI*** The message passsing interface standard is used for distributed simulations. We are developing PCSIM with MPICH2 (1.0.6p1)
   Home: http://www.mcs.anl.gov/research/projects/mpich2, Download: http://www.mcs.anl.gov/research/projects/mpich2/downloads/index.php?s=downloads
   Note: On openSUSE 10.2 we compiled this from the sources since for some reasons the available RPM packages did not work.

7. *cppunit (≥1.12.1)* This unit test framework for C++ is used for testing the C++ core of PCSIM.
   Home: http://sourceforge.net/projects/cppunit, Download: https://sourceforge.net/project/showfiles.php?group_id=11795

8. *GSL - GNU Scientific Library (≥1.9)** This library provides many useful methods for scientific computing. For example a library of methods for numeric integration.
   Home: http://www.gnu.org/software/gsl, Download: ftp://ftp.gnu.org/gnu/gsl/gsl-1.10.tar.gz

9. *GCCXML (≥0.9)* This code parser generates an XML description of C++ code. Such a description is used py `Py++` (see below) to generate the python interface code.
   Home: http://www.gccxml.org, Download: GCCXML can only be downloaded from the development CVS repository `:pserver:anoncvs@www.gccxml.org:/cvsroot/GCC_XML`

10. pygccxml and Py++ (≥0.9.5): These two powerful tools are used to generate the code which interfaces the PCSIM C++ library with python.
    Home: http://www.language-binding.net, Download: http://sourceforge.net/project/showfiles.php?group_id=118209

The source distribution of PCSIM also contains the file HowTo-Install-PCSIM-Dependencies-Linux.txt which describes how to install the above mentioned packages form their source distribution.

IMPORTANT NOTE: Paths to all installed libraries (boost, mpich2, gsl, cppunit) should be added to `LD_LIBRARY_PATH` if they are installed in non-standard locations, so that the PCSIM build system can find them!

2.3 Installing from the source distribution

1. Downloaded the source distribution from `http://www.igi.tugraz.at/pcsim` or directly from the project site at sourceforge.

   ```
   wget 'http://downloads.sourceforge.net/pcsim/pypcsim-0.5.0.tar.gz'
   ```

2. Unpack the downloaded compress archive file. E.g.

   ```
   tar xvzf pypcsim-0.5.5.tar.gz
   ```

3. Change to the newly created directory. We will refer to this as `${PCSIM_ROOT_DIR}` in the following.

   ```
   cd pypcsim-0.5.5
   ```

4. Run the python setup script in the PCSIM-root directory:

   ```
   python setup.py install --prefix=<pcsim install dir>
   ```

   Without the prefix, PCSIM is installed in the standard install location (usually `/usr`) which depends on your python installation, i.e. a *global* installtion is done and you need the neccessary write permissions. For other available options and commands of the setup script run `python setup.py --help`.

   The command `python setup.py install` invokes several `cmake` instances which generate the python interface wrapper code, compile the C++ code, create the binaries and install them. The installation creates the following files locally in the PCSIM-root directory:

   - `${PCSIM_ROOT_DIR}/bin/pcsim_test` Executable which runs all PCSIM unit tests
   - `${PCSIM_ROOT_DIR}/lib/libpcsim.so` PCSIM shared library object. A dynamic library wich contains the core functionality of PCSIM .
   - `${PCSIM_ROOT_DIR}/lib/pypcsim.so` PyPCSIM python extension module. A dynamic library which contains the python interface to PCSIM an can be loaded via the python import command `import pypcsim`.

   Additionaly, the following files are copied to the chosen installation locations:

   - `libpcsim.so` PCSIM shared library object (copied to `<prefix-dir>/lib/libpcsim.so`)

- `pypcsim.so` PyPCSIM python extension module (copied to `<prefix-dir>/lib/python2.x/site-packages/pypcsim.so`)

  To do a *local* installation and skip the copying of the libraries to the install directories, execute:

  ```
  python setup.py install -l
  ```

  In this case one has to set the path environment variables to the localy created library files, as explained in step 6.

5. Run the tests

   ```
   python setup.py test
   ```

6. For the installation to work one has to make sure that the environment variables `LD_LIBRARY_PATH` and `PYTHONPATH` contain the installation location you have choosen during installation. For a global installtion this should be the case anyway. However, for a local installation it is likely that you have to add the paths to `libpcsim.so` and `pypcsim.so` to the `LD_LIBRARY_PATH` and `PYTHONPATH` environment variables manually:
   For bash:

   ```
   export LD_LIBRARY_PATH ${PCSIM_ROOT_DIR}/lib:${LD_LIBRARY_PATH}
   export PYTHONPATH ${PCSIM_ROOT_DIR}/lib:${PYTHONPATH}
   ```

   For csh/tcsh:

   ```
   setenv LD_LIBRARY_PATH ${PCSIM_ROOT_DIR}/lib:${LD_LIBRARY_PATH}
   setenv PYTHONPATH=${PCSIM_ROOT_DIR}/lib:${PYTHONPATH}
   ```

   Put the previous lines in your `.profile` or `.tcshrc` depending on the shell you are using.

7. If you also want to install the python packages accompanying PCSIM (currently only `pypcsimplus`):

   ```
   python setup_pkg.py install
   ```

   For other available options and commands of the setup script run `python setup_pkg.py --help`. Additionally you need to add `${PCSIM_ROOT_DIR}` to `PYTHONPATH`.
   For bash:

   ```
   export PYTHONPATH=${PCSIM_ROOT_DIR}:${PYTHONPATH}
   ```

   For csh/tcsh:

   ```
   setenv PYTHONPATH ${PCSIM_ROOT_DIR}:${PYTHONPATH}
   ```

   Put the previous line in your `.profile` or `.tcshrc` depending on the shell you are using.

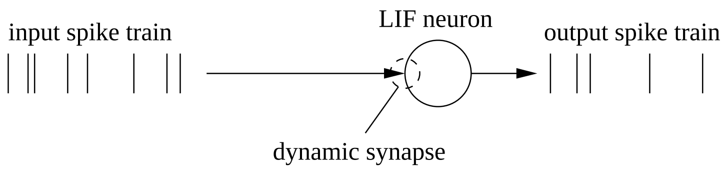
8. If your reached this point with all tests passed, you made it!

(usage: `pcsim_test  <AutoBuild|Nightly>` or `mpirun -np 4 pcsim_test <AutoBuild|Nightly>`)

Chapter 3

Getting Started

In this section we will introduce PCSIM by means of a simple example. We will use PCSIM to simulate a model where a leaky-integrate-and-fire (Sec. ??) neuron (LIF neuron) is driven by a (hand made) spike train which is transmitted by a dynamic synapse (Sec. ??).



The full code of the following example is contained as the file `first_model.py` in the example directory of the PCSIM package.

3.1 Starting PCSIM

Since PCSIM is a package which extends python we first have to start python. E.g. on the command prompt just enter python . After this you should see something like:

```
Python 2.5 (r25:51908, Nov 27 2006, 19:14:46)
[GCC 4.1.2 20061115 (prerelease) (SUSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

After the python interpreter has been started one has to load the PCSIM python module `pypcsim`

```
   >>> from pypcsim import *
```

which makes all the functionality of PCSIM available in the python environment.

3.2 The basis: Creating a network

The most important concept of a PCSIM simulation is the *network*. A network consists of different kinds of objects wich are linked together by certain *message channels*. Hence the construction of each PCSIM simulation starts out with the construction of an (initially empty) network.

```
>>> net = SingleThreadNetwork()
```

When we will discuss how to run a distributed simulation (Sec. **??**) we will see that there are different types of networks available. For the moment we will use `SingleThreadNetwork` which is the simplest one.

3.3 Adding objects to the network

Each element/entity of the simple model we will implement, will be simulated by a corresponding *object* in PCSIM . The following table shows the correspondence between the elements of the model and the *class* of the object used to simulate the element

element of model	**class of PCSIM object**
input spike train	`SpikingInputNeuron`
dynamic synapse	`DynamicSpikingSynapse`
LIF neuron	`LifNeuron`

Hence, for the simulation to run one must create *models* of each entity we want to simulate by creating an instance of the corresponding class:

```
>>> input   = SpikingInputNeuron()
>>> synapse = DynamicSpikingSynapse())
>>> neuron  = LifNeuron()
```

The code above generates models with the detault parameters. For simple models like the `LifNeuron` it is convinient to set its parameter directly when defining the model.[1]

```
>>> inp_model = SpikingInputNeuron( [ 0.01, 0.02, 0.03, 0.05, 0.1, 0.15 ] )
>>> nrn_model = LifNeuron( Cm=2e-10, Rm=1e8, Vthresh=-50e-3, Inoise=0.8e-9 )
>>> syn_model = DynamicSpikingSynapse( W=2e-8, tau=5e-3, delay=1e-3 )
```

It is important to note that the three created instances do not have any relation to the network `net` at the moment. In order to create object instances within the network we have to call `net.create`:

```
>>> inp_handle = net.create( inp_model )
>>> nrn_handle = net.create( nrn_model )
```

[1]See the PCSIM class reference for detailed information about the parameters of individual models.

The effect of the `create` method is that a *object instance* are created from the *object model*[2]. passed as arument to `create`. The value returned from `create` is a *handle* or *id* to the actual object instance which is managed by the network.

3.4 Connecting objects

Now that we have the created the input neuron and the leaky-intagrate and fire neuron we want to connect them by the synapse. This is as easy as:

```
>>> syn_handle = net.connect( inp_handle, nrn_handle, syn_model )
```

The effect of the above command is that an *object instance* based on `syn_model` is created and a handle to it is returned.

3.5 Simulating the model

From the point of the model we want to simulate we are already done and we could issue the following command to simulate the network for a time span of 200ms.

```
>>> net.simulate(0.2)
```

3.6 Analysing the simulation

After the `simulate` command one wants to examin what was going on during the simulation. However we have not yet told to the simulator what we are interested in. In PCSIM this is done by connecting so called *recorders*.

In order to record the spikes which are emitted by the neuron we have to create a `SpikeTimeRecorder` and connect the spiking output of the neuron to it:

```
>>> st_rec_handle = net.record( nrn_handle, SpikeTimeRecorder() )
```

To record the membrane potetial we have to create a `AnalogRecorder` and connect the membrane potential to it.

```
>>> vm_rec_handle = net.record( nrn_handle, "Vm", AnalogRecorder() )
```

And similarly for the postsynaptic current of the synapse:

```
>>> ps_rec_handle = net.record( syn_handle, "psr", AnalogRecorder() )
```

Let us do the simulation again, starting at $t = 0$:

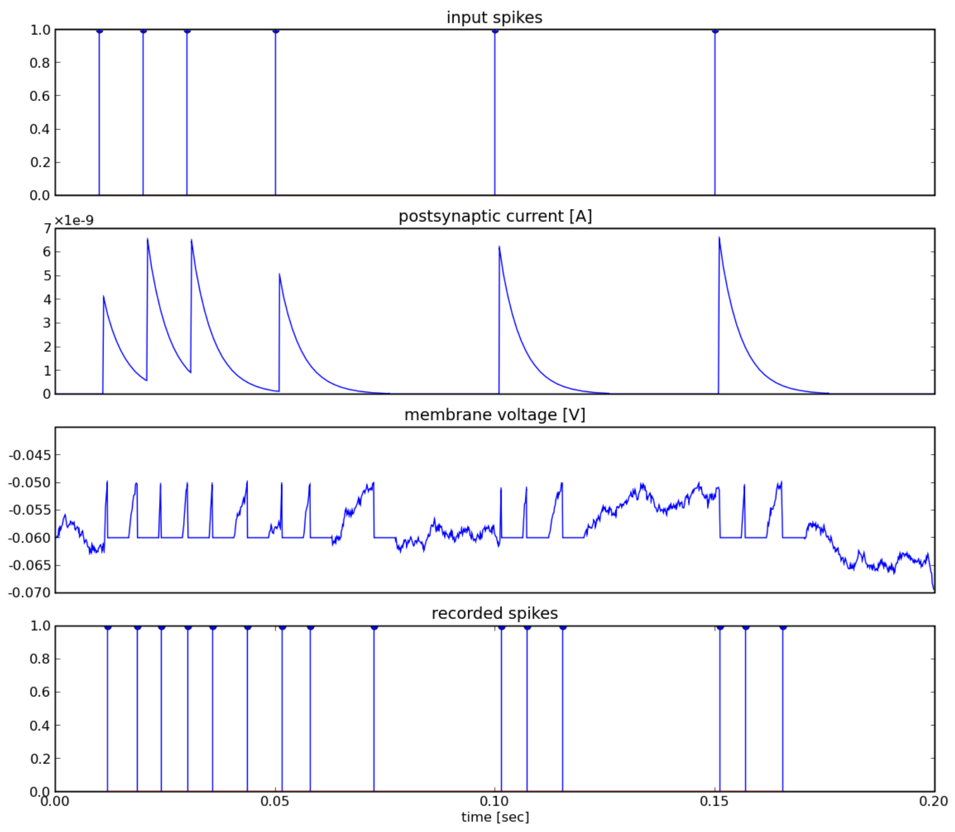
[2] Actually these models are object factories from which the actual simulated object instances will be created

```
>>> net.reset()
>>> net.simulate(0.2)
```

Now the recordes have the stored the parameters of interest. These can now be analysed and plotted. The following code shows how to generate a figure using matplotlib and numpy which is actually a must have when doing serious numeric compuations with python.

```
>>> from pylab import *
>>> from numpy import *

>>> clf()
>>> dt = net.simParameter().dt.in_sec()
>>> t  = arange( 0, 0.2-dt, dt )

>>> subplot(4,1,1)
>>> stem( input_spike_times, ones_like(input_spike_times) )
>>> xlim( 0, 0.2 ); xticks( [] )
>>> title('input spikes')

>>> subplot(4,1,2)
>>> psc = net.object( ps_rec_handle ).getRecordedValues()
>>> plot( t, psc )
>>> xlim( 0, 0.2 ); xticks( [] )
>>> title('postsynaptic current [A]')

>>> subplot(4,1,3)
>>> vm = net.object( vm_rec_handle ).getRecordedValues()
>>> plot(t,vm)
>>> xlim( 0, 0.2 ); ylim( -0.07, -0.04 ); xticks( [] )
>>> title('membrane voltage [V]')

>>> subplot(4,1,4)
>>> recorded_spike_times = net.object( st_rec_handle ).getSpikeTimes()
>>> stem( recorded_spike_times, ones_like(recorded_spike_times) )
>>> xlim( 0, 0.2 )
>>> xlabel( 'time [sec]' )
>>> title('recorded spikes')

>>> show()
```

The code above generates the following figure:



Remarks

- The statement `net.object( h )` is neccessary to get the actual instance of an object with handle `h` from the network. In the code above these have been recorder. But we can as well access for examples the synapse or the neuron:

  ```
  >>> print net.object( nrn_handle ).getVm()
  -0.0620427952915
  >>> print net.object( syn_handle ).psr
  0.0
  ```

- The command `dt = net.simParameter().dt.in_sec()` returns the currently used simulation time step.

Chapter 4

A more realistic examples

In this section we will describe the usage of PCSIM by using a more realistic example. We will implement the model defined as “Benchmark 3: Conductance based HH network” in Simulation of networks of spiking neurons: A review of tools and strategies. This network is composed of Hudgkin and Huxly type neurons coupled by conductance based synapses. Furthermore we will learn how to perform a *distributed simulation* with PCSIM .

The full source code of this example is available as examples/example1.py

```
from pypcsim import *
import random
import numpy as N

nNeurons        = 4000;   # number of neurons
minDelay        = 1e-3;   # minimum synapse delay [sec]
ConnP           = 0.02;   # connectivity probability
Frac_EXC        = 0.8;    # fraction of excitatory neurons
Tsim            = 0.1;    # duration of the simulation [sec]
DTsim           = 1e-4;   # simulation time step [sec]
nRecordNeurons  = 25;     # number of neurons to plot the spikes from
Tinp            = 50e-3;  # length of the initial stimulus [sec]
nInputNeurons   = 10 ;    # number of neurons which provide initial input (for a time span of
inpConnP        = 0.01 ;  # connectivity from input neurons to network neurons
inputFiringRate = 80;     # firing rate of the input neurons during the initial input [spikes
```

SimParameter, DistributedSingleThreadNetwork

```
sp = SimParameter( dt=Time.sec( DTsim ) , minDelay = Time.sec(minDelay), simulationRNGSeed =
net = DistributedSingleThreadNetwork( sp )
```

SimObjectVariationFactory HHNeuronTraubMiles91 UniformDistribution

```
exz_nrn_model = SimObjectVariationFactory( HHNeuronTraubMiles91( ) )
```

```
exz_nrn_model.set( "Vresting", UniformDistribution( -50e-3, -48e-3) )
exz_nrn_model.set( "Cm", UniformDistribution( 1.5e-10, 2.5e-10 ) )

inh_nrn_model = SimObjectVariationFactory( HHNeuronTraubMiles91( ) );
inh_nrn_model.set( "Vresting", UniformDistribution( -55e-3, -50e-3) )
inh_nrn_model.set( "Cm", UniformDistribution( 2.2e-10, 2.7e-10 ) )

num_exz = int( nNeurons *  Frac_EXC )
num_inh = nNeurons - num_exz

exz_nrn = net.create( exz_nrn_model, num_exz );
inh_nrn = net.create( inh_nrn_model, num_inh );

all_nrn = list(exz_nrn) + list(inh_nrn);
```

StaticCondExpSynapse

```
exz_syn = SimObjectVariationFactory( StaticCondExpSynapse( W=2e-9, tau= 5e-3, delay=1e-3, Ere
exz_syn.set( "tau", UniformDistribution( 4e-3, 6e-3 ) )

inh_syn = SimObjectVariationFactory( StaticCondExpSynapse( W=33e-9, tau=10e-3, delay=1e-3, Er
inh_syn.set( "tau", UniformDistribution( 9e-3, 13e-3 ) )
```

RandomConnections

```
n_exz_syn = net.connect( exz_nrn, all_nrn, exz_syn, RandomConnections( conn_prob = ConnP ) )[
n_inh_syn = net.connect( inh_nrn, all_nrn, inh_syn, RandomConnections( conn_prob = ConnP ) )[
```

Create input neurons for the initial stimulus and connect them to random neurons in circuit

SpikingInputNeuron StaticCondExpSynapse

```
inp_nrn = [ net.add( SpikingInputNeuron( [ random.uniform(0,Tinp) for x in range( int(inputFi

inp_syn = StaticCondExpSynapse( W=6e-9, tau=5e-3, delay=1e-3, Erev = 0 )
net.connect( inp_nrn, all_nrn, inp_syn, RandomConnections( conn_prob = inpConnP ) )
```

SpikeTimeRecorder AnalogRecorder

```
spike_rec = range( len(all_nrn)  )
for i in range( len(all_nrn) ):
    spike_rec[i] = net.create( SpikeTimeRecorder(), SimEngine.ID(0,0) )
    net.connect( all_nrn[i], spike_rec[i] , Time.ms(1) )
```

```
rec_nrn = random.sample( all_nrn, nRecordNeurons );
vm_rec  = range( nRecordNeurons )
for i in range( nRecordNeurons ):
    vm_rec[i] = net.create( AnalogRecorder(), SimEngine.ID(0,0) )
    net.connect( rec_nrn[i], 'Vm', vm_rec[i], 0, Time.ms(1) )

net.reset();
net.simulate( Tsim )
```

Chapter 5

Adding user defined models

In the file HowTo-Extend-PCSIM.txt it is described how to add your own models written in C++ to PCSIM .

A more detailed description will be geiven here in the near future.

Bibliography

[Dayan and Abbott, 2001] Dayan, P. and Abbott, L. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press. See also http://neurotheory.columbia.edu/~larry/book/.

[Gerstner and Kistler, 2002] Gerstner, W. and Kistler, W. (2002). *Spiking Neuron Models*. Cambridge University Press. See also http://diwww.epfl.ch/~gerstner/BUCH.html.