LCG
command line electrophysiology
A user manual

Daniele Linaro
João Couto
Michele Giugliano

3rd January 2015

Contents

Chapter 1

Introduction

This manual describes the functionalities of LCG, a suite of programs – called *commands* – that can be used to perform electrophysiological experiments.
LCG was developed by Daniele Linaro and João Couto while at the Theoretical Neurobiology and Neuroengineering Laboratory of the University of Antwerp. Michele Giugliano developed the concept and meta-description of stimulus files and implemented the first version of the related source code.

The main features of LCG are the following:

- dynamic clamp capabilities, with built-in active electrode compensation Brette et al. (2008);
- straightforward design and implementation of closed loop and hybrid experiments;
- command-line operability;
- ease of automation by scripting;
- compact description of stimulation waveforms;
- simple implementation of non real time protocols (e.g., current and voltage clamp);
- support for multiple real time engines;
- simple installation and operation procedures.

1.1 What is LCG?

LCG is a software toolbox that can be used to perform electrophysiological experiments using simple stimulation protocols but also more complex paradigms such as dynamic clamp and/or hybrid networks. LCG is made up of two parts: the first one is a C++ library that implements the low-level objects responsible, among other things, of analog input/output, saving data to disk and interfacing with the real-time scheduling system of the OS. The second part is a

Python interface that handles simpler tasks, mostly related to the high-level description of specific electrophysiological protocols and simple data analysis and plotting.

LCG consists of a set of *commands*, each of which performs a specific task. The entry point of all LCG commands is the program called `lcg`. Its purpose is merely to parse its arguments and call the appropriate protocol with the arguments with which it was invoked. To clarify this concept, let's look at the following example:

```
lcg steps -a -200,100,50 -d 2
```

This command instructs `lcg` to look for a program called `lcg-steps` in the directories where executable files are located,[1] and invoke it with the subsequent arguments unchanged. The previous call is therefore equivalent to

```
lcg-steps -a -200,100,50 -d 2
```

This approach is particularly suited to adding extensions to LCG: to do so, it is sufficient to follow the naming scheme just described and place additional scripts or programs where `lcg` can find them, i.e., anywhere in the path.

LCG comes with a `help` command, that can be used to display general information about LCG and a list of the most commonly used protocols, by simply invoking

```
lcg help
```

Alternatively, `lcg-help` accepts one single argument, which must be the name of a recognized command and invokes it with the `-h` option. For example, the following four commands produce the same result:

```
lcg help steps
lcg-help steps
lcg steps -h
lcg-steps -h
```

For this to work, all commands must accept the `-h` option and interpret it as a request for help. All commands that come with LCG adhere to this convention, and additional ones should do the same.

This manual is organized as follows: Chap. 2 covers in detail the installation of a real-time operating system and of LCG and its dependencies. Chapter 3 explains how to start using LCG. Chapter 4 describes the most commonly used commands that come with LCG and the related electrophysiological protocols, while Chap. 5 explains how stimulus files can be used to apply arbitrary stimulation protocols, in traditional voltage and current clamp experiments. Chapter 6 covers the internal organization of the data files saved by LCG and contains a few examples that show how to load LCG data files using both Matlab and Python. Chapters 7 through 8 explain how configuration files can be used to describe custom experimental protocols, and which basic building blocks – called *entities* or *streams* – are available in LCG. Finally, Chap. 9 covers some additional features of LCG.

[1] These are usually stored in the `$PATH` environment variable.

In the simplest scenario, in which LCG is already installed on a machine and the experimentalist only wants to perform conventional voltage and/or current clamp experiments, without the need for real time control over the experiment, Chapters 3, 4 and 5 with some parts of Chap. 6 are sufficient to get going.

Chapter 2

Installation

The primary application of LCG is to perform electrophysiology experiments. Albeit not necessary, as it will be described in the following, it is however recommended to install LCG on a Linux machine with a data acquisition card and a working installation of the Comedi library. A real-time kernel is required only to perform closed-loop or dynamic clamp experiments.

It is also possible to install LCG on any UNIX compatible operating system (Linux or Mac OS X, for instance) without Comedi and real-time capabilities. This approach can be useful to familiarize with the program and to develop new experiments. A detailed description of how to do this is given in Sec. 2.4.

In the following section, we describe how to set up a Linux machine with a real-time kernel and Comedi and how to install LCG. We assume that you have a working Linux installation: if that is not the case, refer to the documentation of one of the multiple distributions available. In our laboratory we use Debian and therefore the following instructions will be based on this distribution.

2.1 Configuration of the Linux box

LCG can be installed on a variety of Linux distributions. Here, we will refer to the stable version of Debian at the time of this writing (*Squeeze 6.0*).

2.1.1 Patching and installing the real-time kernel

If you don't want to use the real-time capabilities of LCG you can skip this paragraph.

This is potentially the most difficult part of the installation. In order to achieve nanosecond precision, LCG requires a kernel with real-time capabilities. Both RTAI or **PREEMPT_RT** can be used. The latter is advisable since RTAI does not include support for the latest kernels, which might work better with the most recent hardware.

You may need to install tools to build the kernel. In Debian you can type, as root:

```
apt-get update
apt-get install build-essential binutils-dev libelf-dev libncurses5
    libncurses5-dev git-core make gcc subversion libc6 libc6-dev
  automake libtool bison flex autoconf libgsl0-dev
```

1. **Check which kernels and patches are available.** It is advisable to install the real-time patch on a "vanilla" kernel. To do so, browse the kernel.org "rt" project page and find out which patches are available for the kernel that you wish to install. In the following, we will describe how to patch and compile the 3.8.4 version of the Linux kernel.

2. **Download the kernel and the real-time patch** The directory **/usr/src** is a common place to install the Linux kernel. You will need to have root privileges to perform these operations.

   ```
   cd /usr/src
   wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.8.4.
       tar.bz2
   wget http://www.kernel.org/pub/linux/kernel/projects/rt/3.8/
       patch-3.8.4-rt2.patch.bz2
   ```

3. **Decompress and patch the kernel.** Patching the kernel will add real-time support to the kernel you have downloaded.

   ```
   tar xjf linux-3.8.4.tar.bz2
   ln -s linux-3.8.4 linux
   cd linux
   bzcat ../patch-3.8.4-rt2.patch.bz2 | patch -p1
   ```

4. **Configure the kernel.** The easiest way to do this is to use the configuration file from a kernel that was previously installed in your system or from a similar installation/system.

   ```
   cp /boot/config-`uname -r` .config
   make oldconfig
   make menuconfig
   ```

   This will evoke a user interface to configure the kernel options. You need to set the `Preemption model` to `Fully Preemptible Kernel (RT)` in the `Processor type and features`. Disable `CPU Frequency scaling` under `Power Management and ACPI options` and `Check for stack overflow` and all options under `Tracers` (the latter to make the kernel smaller) in `Kernel hacking`. Additionally you should also disable the `Comedi drivers` under the `Staging drivers` of the `Device drivers` menu. Save the configuration and exit.

5. **Compile and install the kernel.** This step might take from several minutes to a few hours, depending on the size of the kernel and on the speed of your machine.

   ```
   make && make modules && make modules_install && make install
   ```

   When the installation is complete you will need to update the boot loader.

```
cd /boot
update-initramfs -c -k 3.8.4-rt2
update-grub
```

Check that grub has been updated. You should see an entry with the name of the newly installed kernel in `/etc/grub/menu.lst`. You can now reboot into your new kernel.

2.1.2 Installation of the dependencies

LCG makes use of several libraries:

1. **BOOST.** The BOOST C++ library is used for parsing XML files. BOOST can be installed by using the system's package manager: on Debian, enter the following command as root:

```
apt-get install libboost-dev
```

2. **HDF5.** The HDF Group designed a set of libraries for data storage and management with high performance, efficiency and high volume/complexity in mind. **lcg** uses this library to save binary data files. HDF5 can be installed by using the system's package manager: on Debian, enter the following command as root:

```
apt-get install hdf5-tools hdf5-serial-dev
```

3. **Comedi.** By default, LCG uses Comedi to interface to supported data acquisition cards. The installation of Comedi consists of three parts: a kernel-space library (`comedi` itself), a user-space interface (`comedilib`) and some additional programs for the calibration of the board (`comedi_calibrate`). They can be downloaded from the Git repository of the Comedi project. First of all, make a directory in your home folder, where all the source files will be downloaded and compiled, and move to that directory:

```
mkdir -p $HOME/local/src
cd $HOME/local/src
```

Then, download Comedi's source code from the repository:

```
git clone git://comedi.org/git/comedi/comedi.git
git clone git://comedi.org/git/comedi/comedilib.git
git clone git://comedi.org/git/comedi/comedi_calibrate.git
```

You now need to install each of these modules, starting with the kernel module:

```
cd comedi
./autogen.sh
./configure --with-linuxdir=/usr/src/linux
make
```

Then as root:

```
make install
depmod -a
```

Now for comedilib, the user space interface to the kernel module (run `make install` as root):

```
cd comedilib
./autogen.sh
./configure --prefix=/usr/local
make
make install
```

And similarly for comedi_calibrate, the tools to calibrate the DAQ cards (install the Boost headers before this step):

```
cd comedi_calibrate
./autogen.sh
./configure --prefix=/usr/local
make
make install
```

Now that the drivers are installed you need to create the rules to allow users to access the devices. To do that, create, as root, a file called `/etc/udev/rules.d/99-comedi.rules` and add the following line to it: `KERNEL=="comedi0", MODE="0666"`. In case you have multiple acquisition cards, add a line for each of them.

2.2 Installation of LCG

To install LCG, start by getting it from the GitHub repository. We recommend installing LCG in a local folder, so that it is easier to have multiple installations for different users.

```
cd $HOME/local/src
git clone https://github.com/danielelinaro/dynclamp.git lcg
cd lcg
autoreconf -i
./configure --prefix=$HOME/local
make
make install
```

If you are installing LCG on a real-time machine, you need to create a realtime group to allow non-root users to run LCG commands. To do so, as root add the following lines to `/etc/security/limits.conf`.

```
@realtime       -       rtprio  99
@realtime       -       memlock unlimited
```

Then, create a group `realtime` and add the users that you want to be able to use LCG.

```
groupadd realtime
usermod -a -G realtime USER
```

where USER is the actual login name of an existing user. You need to log out and log back in for the changes to take effect.

2.2.1 Installation of the Python bindings

LCG comes with a set of Python scripts that can be used both to perform standard electrophysiology experiments and as starting point for developing new protocols.

The full list of available protocols (called *commands*, in LCG terminology) is described in Chapter 4: here, we only describe how to install the Python bindings and their dependencies.

Start by installing `numpy`, `scipy`, `matplotlib` and `pytables` if they are not already present on your system. In Debian this can be accomplished with the following command:

```
apt-get install python-setuptools python-numpy python-scipy python-
    matplotlib python-tables python-lxml
```

Alternatively, to have the latest version of the modules, you can install them from source.

To install LCG Python modules, from the base directory (i.e., where you cloned the Git repository) type:

```
export PYTHONPATH=$PYTHONPATH:$HOME/local/lib/python2.7/site-
    packages
cd python
python setup.py build
python setup.py install --prefix=$HOME/local
```

In case you are using a different version of Python (2.6, for instance), change the directory accordingly (i.e., to $HOME/local/lib/python2.6/site-packages).

This will install the module in the directory `$HOME/local/lib/python2.7/site-packages`, which needs to be permanently added to your `PYTHONPATH` environment variable, by adding the previous `export` line to your `.bashrc` file. This can be accomplished with the following command:

```
echo 'export PYTHONPATH=$PYTHONPATH:$HOME/local/lib/python2.7/site-
    packages' >> $HOME/.bashrc
```

Again, change the directory accordingly if you are using a different Python version. You can test whether the installation was successful by issuing the following commands:

```
cd $HOME
python
import lcg
```

If the last command produces no error, the installation was successfull.

2.3 Final configuration

In order for LCG to function properly, a few configuration steps must be completed. First of all, add the directory where LCG binaries were installed to your path. To do this, add the following line to your `.bashrc` file:

```
export PATH=$PATH:$HOME/local/bin
```

After this, copy the files `lcg-env.sh` and `lcg-completion.bash` from LCG base directory to your home directory:

```
cp lcg-env.sh ~/.lcg-env.sh
cp lcg-completion.bash ~/.lcg-completion.bash
```

Source them any time you log in by adding the following lines to your `.bashrc` file:

```
source ~/.lcg-env.sh
source ~/.lcg-completion.bash
```

The script in `lcg-completion.bash` provides autocomplete capabilities to LCG but is not required for correct functioning. The environment variables exported in `lcg-env.sh`, on the other hand, provide necessary defaults to LCG and should be tailored to your system. In particular, the file exports the following variables:

- **COMEDI_DEVICE** The path to the device file from which data is read.
- **AI_CONVERSION_FACTOR_CC** The conversion factor to be used for the analog input, in current clamp mode.
- **AO_CONVERSION_FACTOR_CC** The conversion factor to be used for the analog output, in current clamp mode.
- **AI_CONVERSION_FACTOR_VC** The conversion factor to be used for the analog input, in voltage clamp mode.
- **AO_CONVERSION_FACTOR_VC** The conversion factor to be used for the analog output, in current clamp mode.
- **RANGE** The range of the output to the analog card.
- **AI_SUBDEVICE** The analog input subdevice on the acquisition card.
- **AI_CHANNEL** The default channel used for analog input.
- **AO_SUBDEVICE** The analog output subdevice on the acquisition card.
- **AO_CHANNEL** The default channel used for analog output.
- **AI_UNITS_CC** The units for the analog input, in current clamp mode.
- **AO_UNITS_CC** The units for the analog output, in current clamp mode.
- **AI_UNITS_VC** The units for the analog input, in voltage clamp mode.
- **AO_UNITS_VC** The units for the analog output, in voltage clamp mode.
- **SAMPLING_RATE** The default sampling rate of the acquisition.
- **GROUND_REFERENCE** The ground reference of the acquisition card. At present, Ground-Referenced Single Ended (GRSE) and Non-Referenced Single Ended (NRSE) are supported.

- **LCG_REALTIME** This variable tells whether the system should preferentially use the real-time kernel if it is available or not. The main advantage in using the real-time kernel (also for open loop experiments) is that it provides synchronous input and output, in contrast to the non-real-time mode, where input and output are asynchronously managed by the DAQ board.
- **LCG_RESET_OUTPUT** This variable tells whether the output of the DAQ board should automatically be reset to zero every time a trial ends. This is particularly useful when interrupting the program (with Ctrl-C for example) and is valid only for current clamp experiments.

Most of the previous values depend on how your amplifier is configured and on how it is wired to the acquisition card. It is also important to note that the conversion factors and the units provided in `.lcg-env.sh` are meaningful when only one input and output channel are present. In all other cases, input/output conversion factors will have to be specified either in the configuration file or when invoking a script. LCG provides a script that helps the user in finding the correct values for the conversion factors used on his/her system. To use it, turn on the amplifier and connect it to the board as you would during an experiment and run the following commands:

```
$ comedi_calibrate
$ lcg-find-conversion-factors --CC-channels 0,0 --VC-channels 1,1
```

where the `--CC-channels` and `--VC-channels` options specify the input and output channels to use in current or voltage clamp mode, respectively, and should reflect the values used in the system that is being configured. The script will ask the user a few questions and then update the values of the conversion factors in the `.lcg-env.sh` file: due to rounding errors, however, these values might have to be rounded by the user afterwards by editing manually the `.lcg-env.sh` file.

2.4 Installation without a realtime kernel

If you want to learn how to work with LCG or test configuration files, it is possible to install LCG on your personal computer. LCG can be compiled on any UNIX-like operating system, such as Linux and Mac OS X. To do so, you just need to install `BOOST` and the `HDF5` library (see Sec. 2.1.2). The procedure to install LCG is the same as described in Sec. 2.2: the `configure` script will detect the absence of Comedi and of the real-time kernel and not compile parts of LCG. If you are installing LCG on Mac OS X, note that the equivalent of the `.bashrc` file is called `.profile`.

2.5 Installing the live medium to a USB drive

2.5.1 From a Debian/Ubuntu distribution

This can be done from the command-line:

2.5.2 From Windows

In order to make a bootable LCG USB drive you can use UNetBootin.

Chapter 3

Getting started

This chapter will help you understanding some basic concepts related to LCG. It starts by giving a notion of the basic commands using an artificial neuron and continues to interfacing with live cells. While it does not intend to be a detailed tutorial on either Matlab or Oython, some indications on how to use these software packages will be provided.

3.1 Requirements

In order to get started you only need to boot your computer using the live DVD available at `http://www.tnb.ua.ac.be/LCGliveCD.iso`. Instructions about setting up a live USB medium are available in section 2.5. Although all experiments can be performed from the live medium, for production environments it is better to have a dedicated installation of LCG. Additional notes on the installation can be found in section 2.

If you are using a system that someone else installed you can check wether LCG is installed by typing `which lcg` in a terminal window: if the result is blank, the program has not been installed and you should either use the live DVD or install it yourself.

We assume that the reader is somehow familiar with the GNU/Linux operating system and in particular with the usage of a terminal prompt: however, please bear in mind that this is **not** a Linux tutorial. Check the webpage of your GNU/Linux distribution (e.g. Debian; Canonical Ubuntu; Fedora) or the linux.org tutorials for help on getting started with GNU/Linux. Additionally, the *Advanced Bash-Scripting guide* from The Linux Documentation Project is a great reference for how to use a UNIX terminal efficiently.

Additionally, some MATLAB and Python code will be used to load data generated with LCG. and therefore working knowledge of either of these languages is beneficial. Before getting started using LCG you may want to take a look at Appendix A for a short Matlab introduction and Appendix B for some notes on basic UNIX commands.

3.2 First steps in command-line electrophysiology

The most important concepts related to the everyday usage of LCG can be understood by using a neuron model: in this chapter we will use a leaky integrate-and-fire (LIF) model: for an excellent introduction to this and other neuron models, see Koch and Segev (1989). The initial part of this guide does not require a realtime kernel nor a data acquisiton board. We will illustrate some examples of usage of LCG as well as some basic concepts of data analysis of intracellular traces using Python and Octave/Matlab. This should allow a user that is not familiar with Python nor Matlab® to understand the basics and provide building blocks for more complete analyses. A good reference for getting started with Matlab® is Wallisch (2011) or the resources available on the Mathworks. For Python we suggest the Scipy lecture notes. We will also try to highlight some basic UNIX commands to handle file operations and suggest ways to structure and organize your experiments.

3.2.1 Creating an experiment folder

LCG does not require a particular folder structure to be used: when you run a command the recording will be executed in that very same directory. It is however beneficial to use a consistent folder structure. You can use `lcg-create-experiment-folder` for that:

```
lcg-create-experiment-folder -p YYYYMMDD_tutorial_[001] -s LIFsteps
   ,steps
```

`lcg-create-experiment-folder` can do more than creating a folder: among its features is the capability of adding information files with details about the experiment.

After running the previous command, the name of the folder that was just created will be printed to the terminal. Move into this folder using the UNIX command `cd <foldername>`. By using the command `ls` you can list the subfolders created by `lcg-create-experiment-folder`, namely LIFsteps and steps. `lcg-create-experiment-folder` also created a folder `01` inside each of these with the intention of storing different sets of trials related to the same protocol.

3.2.2 Recording from a simulated neuron

As a first example, we will compute the frequency-current (f-I) curve of a LIF model neuron. For this we will use the `lcg-steps` protocol with the `--model` option. Run the following commands in the terminal window (what follows # is a comment).

```
# Move into the stepsLIF/01 directory
cd LIFsteps/01
# Run the protocol
lcg-steps -a -200,800,50  -d 1 --model --no-shuffle -n 1
```

```
# Plot the results
lcg-plot -f all
```

Most LCG commands have multiple switches or options that usually have default parameters. Make sure that the default parameters fit your purpose, otherwise use switches to set them. All protocols have a `--help` switch that displays the supported options and default parameters.

In the above example you used LCG to inject DC current steps from −200 to 800 pA in 50 pA steps and 1 s duration into a simulated neuron. The `--no-shuffle` option tells `lcg-steps` not to shuffle the current amplitudes.

3.2.3 Loading files and analyzing data

Analyzing the recorded data is arguably on of the most important steps of an experiment. While this manual does not intend to go into details on the data analysis we will show examples of scripts to visualize and analyze data for this simple case. Before proceeding you should understand the basic structure of the data files recorded by LCG when using HDF5 files. For a detailed explanation read chapter 6. Each file contains at least 2 groups *Entities* and *Info*. The *Entities* group contains each element (entity; see 8) of the recording: in this case the recorded voltage trace and the injected current step. It should be noted that each entity can also contain **metadata** representing its parameters, such as the stimuli parameters for the *Waveform* entity. The *Info* group contains parameters of the experiment such as the *time step (dt)* used and the *duration (tend)* of the recording.

Python implementation

The scientific community often uses Python to analyze data and produce publication ready figures. We will show you how to produce a simple figure from the recorded traces using an `ipython notebook`: in essence this lets you write and annotate the data analysis script from a standard Internet browser. For details on how to use Python you should follow one of the many online tutorials. Start by launching the notebook and creating a new document.

```
ipython notebook --pylab
```

This should open a browser window and let you create a new notebook in the current working directory. You can now write Python code and execute it directly in the browser. Start by writing the following:

```
from glob import glob    # To list files
import numpy as np       # To handle numerical arrays and some math
import pylab as p        # To plot and visualize
import lcg               # To load experiment files and more
# List experiment files
files = glob('*.h5')
# Load and investigate last file
fname = files[-1]
ent,info = lcg.loadH5Trace(fname)
# Look at the names of the entities
```

```
print [e['name'] for e in ent]
# Thus the first entity is the Neuron and second entity the
    Waveform

# Create a time vector
#(This is done using size of the data and the sampling rate (1/dt))
time = np.arange(len(ent[0]['data']))*info['dt']
# Plot the data in 2 plots with shared x axis
fig, ax = p.subplots(len(entities),1, sharex=True)
for i,e in enumerate(ent):
    # Plot the data
    ax[i].plot(time, e['data'])
    # Add labels with the information from each entity
    ax[i].set_xlabel('Time (s)')
    ystr = '{0} entity ({1})'.format(e['name'],e['units'])
    ax[i].set_ylabel(ystr)
# Set the filename as title
ax[0].set_title('Data from file {0}'.format(fname))
```

Example 3.1: An example of a how to use Python to plot all entities in a HDF5 file recorded with LCG.

The code in listing 3.1 can be used to plot all entities in a file. We will now use the metadata to extract information about the stimulus. Of course this could also be extracted from the current waveforms but is much easier from the metadata (especially for more complicated protocols).

By analyzing the current trace plotted before, we know that our stimulus consists of 1 s of stimulation preceded and followed by 1 s in which no current is injected. This translates to a protocol description (metadata, see 5) with 3 lines. The protocol code (second column) is the same for all rows. The stimulation amplitude is therefore in the second row (third column).

```

# Get the metadata
metadata = ent[1]['metadata']
V = ent[0]['data']
# Cumulative sum of the first column is the duration
prot_time = np.cumsum(metadata[:,0])
# The third column is the amplitude of the stimuli (for the DC
    protocols); we are interested in the amplitude of the second
    line since that is the one of the step of current.
stim_amp = metadata[1,2]
# Find the peaks that cross threshold
threshold = 0.0
# The following line is sufficient to detect the spike indexes by
    in the simulated neuron in real data another approach must be
    taken.
indexes = np.where(V >= threshold)[0]
spikes = time[indexes]

rate = len(spikes[(spikes > prot_time[0]) & (spikes <= prot_time
    [1])])/(prot_time[1]-prot_time[0])
print('The spike rate is {0} spikes per second for a current of {1}
    pA.'.format( rate, stim_amp))
```

Example 3.2: Python example of the code to compute the spike rate during the stimulus for a LIF neuron.

The spike rate is then simply the number of spikes emitted during the application of the current step divided by its duration (1 s in this case). We can now iterate through all the files and compute the f-I curve, i.e., the spike rate versus the injected current.

```
def computeRateForLIF(time, V, metadata, threshold=0.0):
    # Function to compute the rate and return the stimulus
        amplitude
    # for a LIF neuron. See previous python code example.
    prot_time = np.cumsum(metadata[:,0])
    stim_amp = metadata[1,2]
    indexes = np.where(V >= threshold)[0]
    spikes = time[indexes]
    rate = len(spikes[(spikes > prot_time[0]) & (spikes <=
        prot_time[1])])/(prot_time[1]-prot_time[0])
    return rate, stim_amp

# List experiment files
files = glob('*.h5')
F = []
I = []
# Iterate through the files
for fname in files:
    ent,info = lcg.loadH5Trace(fname)
    V = ent[0]['data']
    metadata = ent[1]['metadata']
    time = np.arange(len(V))*info['dt']
    # Use our function
    tmp_rate,tmp_stim = computeRateForLIF(time,V,metadata)
    F.append(tmp_rate)
    I.append(tmp_stim)
# Make list an numpy.array to sort it so that plotting locks nicer
F = np.array(F)
F = F[np.argsort(I)]
I = np.sort(I)
# Plot and add axis labels
p.plot(I,F,'ko-', clip_on=False)
p.xlabel('Current␣(pA)')
p.ylabel('Rate␣(spikes␣per␣second)')
p.grid(True)
```

Example 3.3: Python example of the code to compute the FI curve of a LIF neuron.

Example 3.3 illustrates how to iterate through multiple files using Python. We would recommend that users not familiar with Python and willing to use it for data analysis engage in 2 simple exercises before continuing:

- compute the voltage-current (V-I) curve for those stimulations during which the LIF neuron did not emit spikes and compute the input resistance from the V-I curve.
- write a more general function for spike detection that uses the local maxima of the voltage trace to detect spikes.

Matlab implementation

We will now see how to use Matlab® to read this file and extract the timing of the spikes. First open Matlab® in a terminal window with the command:

```
matlab -nodesktop
```

You should see something like:

```
                < M A T L A B (R) >
Copyright 1984-2012 The MathWorks, Inc.
   R2012b (8.0.0.783) 64-bit (maci64)
                August 22, 2012

To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.

>>
```

We will use Matlab® without graphical user interface in these examples: if you prefer to use the GUI, you can launch Matlab® with the command `matlab &` .

Matlab® functions and the path Before continuining, you need to be familiar with the Matlab® concepts of `functions` and `path`. The terminal that you have just opened is interactive and the commands you type will be run as Matlab® commands. A Matlab® function can be seen as a generic box that receives inputs and returns outputs. The computations processed to transform the inputs into the outputs are the core of the function. In Matlab®, functions are defined by naming a file with the name of the function and placing the code `function [out] = functionName(in)` in the first line of that file. Having said this, Matlab® functions are just files which are named the same as the function: Matlab® does not search for functions in all directories of your disk and because of that you have to tell Matlab® where to search for functions. In order to do so you can use the `addpath` command. To add the functions that ship with LCG to the path for the current session, type at a terminal:

```
addpath([getenv('LCG_path'),'/matlab'])
```

Note that the above command will not work if you haven't defined the variable `LCG_path` in your environment (`$HOME/.bashrc` file) as the location of the source code of LCG. Although this command may seem complicated to the first time user of Matlab®, the only thing it does is retrieving the environment variable '`LCG_path`' and concatenating it with the `'/matlab'` string.

Loading and plotting the recorded traces Now that you have added the functions in the source path of LCG to the path of Matlab® with the previous `addpath` command, the function `loadH5Trace` will be available for Matlab®. Type `help loadH5Trace` to access the help of this function. You can load and plot the data with the commands:

```
files = dir('*.h5');
[entities,info] = loadH5Trace(files(end).name)
entities(1).name
Vm = entities(1).data;
time = [0:length(Vm)-1]*info.dt;
plot(time,Vm,'k')
```

The first line uses Matlab®'s command `dir` to list all directories. Then `loadH5Trace` loads the data into the variables `entities` and `info` - it loads one structure per entity connected to the H5Recorder.
The third line in the above code illustrates how you can find out the name of an entity in the `entities` array. This can be particularly useful since it lets you find a particular entity based on its type. Later we will see how to take advantage of this feature. The fourth and fifth line we associate the variable Vm to the recorded membrane potential of the LIFNeuron and create a time vector with the size of `Vm` and the time step saved in the `info` structure.
The last line plots the membrane voltage as a function of time using the color black.

Detecting the peak of the spikes There are several ways of detecting the peaks of the spikes. We will focus on a relatively robust yet simple method of doing so by taking advantage of the Matlab® function `findpeaks`.

```
% Define the refractory period of the peak detector (1ms); this can
    be useful when dealing with noisy signals.
refractory = 1.e-3/info.dt;
% The following uses the function find peaks to find the spikes
  with threshold crossing at 0mV and a refractory period.
[peaks, loc] = findpeaks(Vm,'THRESHOLD',0,'MINPEAKDISTANCE',
  refractory);
% The spikes are the locations of the peaks on the time vector
spks = time(loc);
% Plot time versus membrane potential
plot(time, Vm,'k')
hold on
% Plot the spike times and the peaks of the Action Potentials. The
  'hold on' command makes that the plots overlap.
plot(spks, peaks,'b.')
```

With the previous commands you can extract the peaks of the action potentials and the corresponding times and plot them on top of the membrane voltage trace. These commands will work also with very noisy signals, as long as the threshold and the refractory period are defined appropriately. We now want to compute the mean interspike interval: to do so, we first compute all the interspike intervals, which can be easily accomplished using the `diff` command. Then, we will use the `mean` command to compute the mean ISI.

```
% Compute te interspike intervals
isi = diff(spks);
% And the mean can be computed in a straight forward manner:
mean(isi)
% The reciprocal will give you the result in Hz
1./mean(isi)
```

3.2.4 Interfacing with a Patch Clamp amplifier

LCG is not bound to a particular amplifier or amplifier brand. While on one hand this adds versatility to the program, on the other it requires that the user be familiar with some basic concepts of data acquisition and inner loops of the amplifier. A recording system for patch clamp electrophysiology using LCG is composed of:

1. a patch-clamp Amplifier.
2. A Data Acquisition Board, referred to as DAQ throughout this manual.
3. A recording computer running LCG.

Assuming that the DAQ board is supported by the COMEDI drivers (or other drivers supported by LCG) then it is necessary that the recording computer understand what the values being read through the DAQ card mean in terms of the physical quantities being measured. This is accomplished in LCG by a set of environment variables.[1] In essence we need to know:

1. where to measure from and what that measure means.
2. Where to stimulate to and how to output something meaningful.

All this information is set up during the installation (see section 2.3) simply by editing a text file.

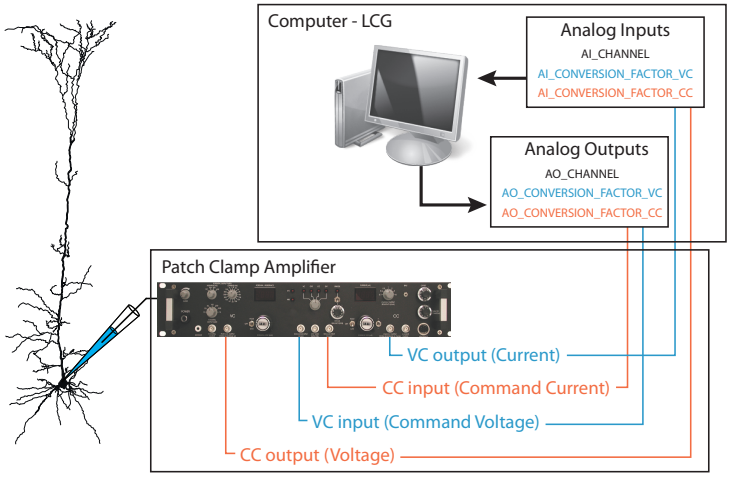

Figure 3.1: Correspondence between Digital Data Acquisition board and the patch clamp amplifier.

Figure 3.1 illustrates the most important connections and environment variables required to control the amplifier both in current clamp and in voltage clamp mode. Protocols run in current clamp mode use the environment variables with **CC** suffix while those run in voltage clamp mode use the **VC** suffix. This allows different gains, channels and units to be used with minimal interaction from the user side.

The majority of LCG commands make sense in current clamp mode and therefore use the corresponding conversion factors. However, those protocols that can be used both in voltage and current clamp modes (for instance, the steps protocol described before) use by default the current clamp conversion factors, but have an additional –**voltage-clamp** switch that instructs the command

[1] If you do not know what a bash environment variable is, please refer to a bash introductory course.

to use the corresponding conversion factors. A small command called **lcg-find-conversion-factors** can help quickly discovering the conversion factors associated with the patch clamp amplifier.

3.2.5 Your first recording

Once you have configured the environmental variables as described in section 2.3, LCG is ready to be used for patch clamp recordings. We can now attempt to characterize the V-I and f-I curves of a real cell as we did for the LIF neuron at the beginning of this chapter. This can be accomplished simply by running the **lcg-steps** protocol.
Set the amplifier to current clamp (CC) mode and enable the external command (if required by the patch clamp amplifier). You can now use the Active Electrode Compensation to compensate pipette resistance errors so do not apply bridge compensation in the amplifier.[2]

Go to the proper recording folder (**¡folder name¿/steps/01** in the previous example) and type the following command:

```
lcg-steps -a -200,500,50  -d 1
# Plot the results
lcg-plot -f all -k kernel.dat
```

This will first inject noise into the cell to compute the kernel that will be used for the offline Active Electrode Compensation and prompt the user for the number of points that are to be considered the length of the Electrode Kernel. Type a number so that the kernel encompasses the large peak and its decay. If you intend to use Active Electrode Compensation you should read Brette et al. (2008). After extracting the electrode kernel the protocol will deliver shuffled current pulses between -200 and $500\,\mathrm{pA}$ in $50\,\mathrm{pA}$ steps. All traces can be plotted using the **lcg-plot -f all** command with the **-k kernel.dat** option that indicates the kernel file to use. LCG also provides functions (for Matlab® and Python) to perform the Active Electrode Compensation offline.

[2] if you do not wish to use Active Electrode Compensation use the **–no-kernel** switch.

Chapter 4

Electrophysiology protocols

In this chapter, we describe the electrophysiology protocols that come with LCG.

All protocols allow the user to specify the sampling frequency (usually with a `-F` switch, but be careful of exceptions) and the input and output channels (usually with the `-I` and `-O` options, respectively, but be *extremely* careful of exceptions).

Of the following protocols, only those for the computation of f-I curves with a Proportional-Integral-Derivative (PID) controller and for clamping the frequency of a neuron require a real-time kernel. For all other protocols, a Comedi installation is sufficient.

Note that, whenever in the following voltage clamp or current clamp are mentioned, LCG does not communicate with the amplifier to switch modes: it is up to the user to do so.

An up-to-date list of the more commonly used protocols can be obtained by entering

```
1 lcg help
```

4.1 Action potential protocol

This protocol consists in injecting a brief (the default value is 1 ms) depolarizing pulse of current to elicit a single action potential to extract its salient electrophysiological properties. The duration of the pulse should be short enough in order not to interfere with the actual shape of the action potential. The user can specify the amplitude and duration of the pulse, as well as the number of repetitions and the interval between them.
For example, the following command

```
lcg ap -a 2000 -d 1 -n 10 -i 5
```

injects 10 pulses of amplitude 2000 pA and duration 1 ms with intervals of 5 s. The voltage is recorded for 0.5 s before and after the application of the pulse.

4.2 V-I curve protocol

This protocol consists in injecting a series of hyperpolarizing and/or depolarizing steps of current to compute a V-I curve. The default is to inject steps of current from −300 to 50 pA in steps of 50 pA, which is suited for cortical pyramidal neurons or other large types of cells. Alternatively, the user can specify the values of current by using the `-a` option.
For example, the following command

```
lcg vi -a -100,40,20 -d 1
```

instructs the protocol to inject steps of current from −100 to 40 pA in steps of 20 pA with a duration of 1 s.

4.3 Time constant protocol

This protocol is identical to the action potential protocol described previously, with the sole exception that the injected pulses of current are hyperpolarizing and can therefore be used to compute the time constant of a cell. The default values of amplitude and duration of the pulses are −300 pA and 10 ms, but they can be changed using the `-a` and `-d` options, respectively.
For example, the command

```
lcg tau -n 20
```

instructs the protocol to inject 20 pulses, instead of the default 30.

4.4 Current ramp protocol

This protocol injects an increasing ramp of current into a cell, to find its threshold for spiking. The user can specify the initial and final values of the ramp, as well as its duration.
For example, the command

```
lcg ramp -a 50 -A 300 -d 15
```

instructs the protocol to inject a 15 s ramp, starting from 50 pA and ending at 300 pA.

4.5 White noise protocol

This protocol injects white noise into a cell. It is primarily used to compute the electrode kernel for the Active Electrode Compensation technique described in Brette et al. (2008).
For example, the command

```
lcg kernel -s 100
```

instructs the protocol to inject a noisy trace lasting 10 s, with zero mean and standard deviation of 100 pA. The default value of standard deviation is 250 pA, which is suited for cortical pyramidal cells. The script also calls `lcg-extract-kernels` to extract the electrode kernel and saves the result in a file called `kernel.dat`. An additional option is provided (`-H`) to inject an additional *holding* current that is not taken into account in the actual computation of the electrode kernel. This option is useful for spontaneously active cells, like Purkinje cells, or for *in vivo* experiments.

4.6 Filtered noise protocol

This protocol injects a Ornstein-Uhlenbeck process into a cell. It can be used either to analyze the reliability and precision of cells, as was done in Mainen and Sejnowski (1995), or to extract the parameters of an exponential integrate and fire model, as described in Badel et al. (2008).
For example, the command

```
lcg ou -n 20 -i 5 -m 100 -s 100 -t 20 -d 3
```

instructs the protocol to inject 20 repetitions of a noisy current with mean and standard deviation equal to 100 pA and time constant of 20 ms. The duration of each injection is 3 s and the interval between repetitions is 5 s.

4.7 Input resistance

This protocol injects a train of hyperpolarizing pulses suited to compute the input resistance of a cell *in vivo*, as described in Crochet and Petersen (2006).
For example, the command

```
lcg rin -a -100 -d 300 -f 1 -D 90
```

instructs the protocol to inject 300 ms-long pulses of amplitude −100 pA at a frequency of 1 Hz. The total duration of the recording is specified by the `-D` option, in this case 90 s. Alternatively, the user can specify, with the `-N` option, the number of pulses to inject.

4.8 DC steps protocol

This protocol injects DC steps of current into a cell, and is a generalization of the previously described protocol for the computation of the V-I curve. It has additional parameters that allow to specify the duration of the recording after the application of the stimulus and an optional holding value.
For example, the command

```
lcg steps -a 100,400,50 -d 2 -n 4 -i 5
```

instructs the protocol to inject steps of current from 100 to 400 pA in steps of 50 pA with a duration of 2 s and interval of 5 s. Each amplitude is repeated 4 times. Additionally, this protocol has a `--vclamp` option, which instructs the program to consider the steps as voltage ones and use, as a consequence, the appropriate default conversion factors (contained in the `.lcg-env.sh` file, see Sec. 2.3). In this case, the command

```
lcg steps -a -100,0,20 --vclamp
```

instructs the program to inject *voltage* steps at values ranging from −100 to 0 mV, in steps of 20 mV. For all the other options, default values are used.
Note that in both cases, the amplitudes of the steps are shuffled, in order to minimize the effects due to adaptation to increasing (or decreasing) stimulation amplitudes. However, an option is provided (`--no-shuffle`) to inject the steps in a linear order.

4.9 f-I curve protocol

Input-output curves, also known as f-I curves, can be easily computed with LCG in two ways: the most straightforward and traditional one is to use `lcg-steps` to inject steps of current and subsequently count the number of spikes emitted for each amplitude.

An alternative way is to use a PID controller and have the neuron follow an increasing ramp of frequency. The protocol that implements this algorithm is `lcg-fi`, which requires a real-time kernel to operate correctly and can be called as follows:

```
lcg fi -a 150 -m 5 -M 30 -T 30 -n 2 -w 60
```

which instructs the protocol to initially inject a current of 150 pA and drive the cell from a minimum of 5 to a maximum of 30 Hz. The protocol is repeated twice, with each trial lasting 30 s and an interval of 60 s between trials. Note that here the inter-trial interval option is specified by the `-w` option, instead of the more common `-i`, which is used, together with `-p` and `-d`, to specify the integral, proportional and derivative gains of the controller, respectively. Note also that, for optimal results, the initial value of current (150 pA in the example) should be chosen such that it elicits spiking in the cell at a frequency as close as possible to the minimum required value (here equal to 5 Hz).

4.10 Frequency clamp protocol

In a sense, this protocol is the dual of an injection of suprathreshold current: in that case, a constant current is injected (clamped) and the firing frequency of the cell can be measured. Here, the protocol finds the current necessary to make the cell spike at a given frequency. To do so, it uses a PID controller in much the same way as the protocol for the computation of a f-I curve, with the difference that here the value of frequency is a constant instead of a ramp.
For example, the command

```
lcg fclamp -f 10 -a 100
```

instructs the protocol to clamp the firing frequency of the cell at a value of 10 Hz, starting with an injected current of 100 pA, which must be above threshold, in order for the frequency estimator to work properly.

4.11 Spontaneous activity protocol

This protocol can be used to record spontaneous activity from an arbitrary number of input channels and is suited both for *in vitro* and *in vivo* experiments. The user can specify the duration of the recording, the device, subdevice, channels and gains to be used in the recording. The only limitation is that all channels need be on the same device and subdevice.
For example, the command

```
lcg spontaneous -d 60 -I 0,1,2,3
```

instructs the protocol to record for 60 s from channels 0, 1, 2 and 3. If not specified, the default device file and input subdevice are taken from the `.lcg-env.sh` file, see Sec. 2.3.

4.12 Train of pulses protocol

This protocol injects a train of brief depolarizing protocols into a cell to elicit action potentials, while simultaneously recording from two cells, the stimulated one and another that might be connected to it. The purpose of this protocol is therefore to test the connectivity between pairs of cells and to measure their short-term synaptic properties.
For example, the command

```
lcg pulses -N 8 -O 0 -I 0,1 -f 30 -d 1 -A 3000
```

instructs the protocol to stimulate a presynaptic cell with a train of 8 pulses of duration 1 ms and amplitude 3000 pA at a frequency of 30 Hz. The output channel (i.e., the channel used to inject current into the presynaptic cell) is number 0, while the input channels are numbers 0 and 1, with 0 being the channel connected to the presynaptic cell and 1 that of the postsynaptic cell. The default is to record the presynaptic cell in current clamp and the postsynaptic

one in voltage clamp, but the option `--cclamp` allows to instruct the protocol to record both channels in current clamp mode.

4.13 Utility programs

LCG contains some additional programs that are not protocols, but that can be used in various occasions during an electrophysiology experiment. They are described in brief in the following, and we refer the user to their help (i.e., `lcg help utility_name`) for an in depth description of their options.

- `lcg zero` outputs zero on all channels of a DAQ board. It is particularly useful at the beginning of an experiment, when the output of the DAQ board is undefined.
- `lcg output` outputs a given value on some channels of the DAQ board. It can be useful when it is required to hold a cell with a certain current or to a certain voltage, for example in voltage clamp experiments.
- `lcg annotate` allows the user to add comments into an existing H5 data file. It accepts two options, `-f` for the filename and `-m` for the comment message. If no options are specified, the program prompts the user for the file name and for a message.

Chapter 5

Stimulus generator

Of course, if LCG were restricted only to the protocols described in Chapter 4, it would constitute a cheap replacement of commercial programs such as Pulse, which offer a vast array of additional functionalities. What makes LCG extremely powerful and *versatile* is the possibility to define arbitrary *stimulus* files to be used in voltage and current clamp experiments, described in this chapter, and the possibility to use *configuration* files to describe arbitrary experiments, (e.g., dynamic clamp or closed loop experiments) defined as the interactions of elementary objects. Configuration files and such objects, called *entities* in the context pf LCG, are described in detail in Chapters 7 and 8, respectively. This chapter describes how to use LCG to inject arbitrary waveforms, described in stimulus files.

In principle, all traditional electrophysiology protocols (i.e., those that do not require real-time control over the recorded quantities), can be implemented in LCG by using just two commands, `lcg-stimgen` and `lcg-stimulus`: the former produces stim-files[1] according to the specification of the user, and the latter applies the synthesized waveforms to the appropriate output channels while at the same time recording from an arbitrary number of input channels.

As a matter of fact, all the protocols described in Chapter 4, with the exception of f-I curve and frequency clamp protocols described in Sections 4.9 and 4.10, respectively, can be implemented using only `lcg-stimgen` and `lcg-stimulus`. The reason for their existence is simply the possibility to define appropriate default values, which make the application of the protocols even simpler.

We will first briefly summarize the rationale behind the meta-language used to generate stimulation files and then describe how to generate them using `lcg-stimgen` and how to use `lcg-stimulus` to stimulate and record using stimulation files.

[1] The syntax of stim-files is explained in great detail in the companion manual, which must be read to properly understand the following instructions.

5.1 Stimulation files

Stimulation files, or in short *stim-files*, are simply text files that contain a piecewise description of a particular waveform. More in detail, each stim-file is composed of a certain number of lines, with each line representing an elementary sub-waveform: the full waveform is therefore given by the concatenation of all sub-waveforms. Each elementary sub-waveform is described by 12 coefficients:

$$T \quad \text{code} \quad p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad \text{fix-seed} \quad \text{seed} \quad \text{subcode} \quad \text{op} \quad \text{exp}$$

The meaning of the coefficients is the following:

T	duration of the sub-waveform
code	code that identifies the sub-waveform
$p_1, \ldots, p_5$	parameters that specify the sub-waveform
fix-seed	whether or not the seed should be fixed (only for stochastic sub-waveforms)
seed	the seed to use (only for stochastic sub-waveforms)
subcode	the code of another sub-waveform, if algebraic operations are to be performed
op	the algebraic operation to perform
exp	the exponent to which the sub-waveform should be raised

Available codes range from 1 to 12 and identify 12 different types of elementary sub-waveforms, such as constants, sinusoids, square waves, Ornstein-Uhlenbeck stochastic process realizations, and so on. The full list of available waveforms is described in detailed in the companion manual, which can be found at `http://danielelinaro.github.io/dynclamp/stimgen_manual.pdf`.

If a noisy waveform is to be used, it is possible to set the seed used in the generation of the pseudo-random numbers. It is also possible to specify whether the seed should be used or not: in the latter case, a random seed is generated using the device `/dev/random`, available on most UNIX systems.

As mentioned previously, sub-waveforms can be combined to yield more complex stimulation waveforms: this is not limited only to the concatenation of elementary sub-waveforms, but can be implemented also by applying algebraic operations to sets of sub-waveforms. The way this is achieved is described in the companion manual.

Finally, each elementary sub-waveform can be raised to a particular exponent (for example 0.5 to obtain the square root of a waveform): this is achieved by setting the appropriate value for the last parameter of the sub-waveform.

This description of stimulus files is by no means exhaustive: we strongly encourage readers to familiarize themselves with the definition of stimulation waveforms by reading carefully through the companion manual.

5.2 Generating stimulation files

As mentioned previously, stimulation files can be generated in LCG using the command `lcg-stimgen`: this provides a convenient interface to the stim-file language, and saves the user the hassle of having to write stim-files manually. Also, it provides a way of interacting directly with `lcg-stimulus` by feeding the output of `lcg-stimgen` to `lcg-stimulus` by means of UNIX pipes, as will be described in the following section.

The syntax of `lcg-stimgen` is the following:

```
lcg-stimgen -o|--output filename [-a|--append]
          <waveform_1> [option <value>] p_1 p_2 ... p_5
          <waveform_2> [option <value>] p_1 p_2 ... p_5
          ...
          <waveform_N> [option <value>] p_1 p_2 ... p_5
```

Since `lcg-stimgen` can deal with an arbitrary number of waveforms, the options that specify the output file name (`-o` or `--output` switches) and whether the output should be appended to an existing file (`-a` or `--append` switches) should precede the definition of any sub-waveform. If no output file is specified, `lcg-stimgen` will output the resulting stim-file to standard output: this is useful both for testing purposes and for usage in conjuction with `lcg-stimulus` as will be described in the following.

Next comes the specification of each single sub-waveform with its parameters: the first parameter (i.e, `waveform_1`, `waveform_2` and so on) is a code that represent the type of waveform. At present, the accepted codes are `dc`, `ou`, `sine`, `square`, `saw`, `chirp`, `ramp`, `poisson-reg`, `poisson-exp`, `poisson-bi`, `gaussian`, `alpha`. For the meaning of each one, see the companion manual. The optional arguments for each sub-waveform are the following:

-d	duration of the sub-waveform in seconds
-s	seed of the sub-waveform, also sets `fix-seed` accordingly
-e	exponent of the sub-waveform
-p	sum this sub-waveform and the following
-m	subtract the following sub-waveform from this one
-M	multiply this sub-waveform and the following
-D	divide this sub-waveform by the following
-E	compute the composite waveform: this option is required for the last waveform in a series

Every short option has a corresponding long one, detailed in the online help of `lcg-stimgen`, accessible by issuing at a prompt the following command:

```
1 lcg-stimgen -h
```

Additionally, all other options —except for the duration— have default values, which are also detailed in the online help of the command.

The positional arguments `p_1` to `p_5` come after the optional arguments. Each waveform requires a certain number of parameters: for instance, the constant waveform requires just one parameter, whereas a sinusoidal waveform requires

4. The exact number of parameters required by each sub-waveform is accessible by issuing the command

```
lcg-stimgen help <code>
```

where code is one of the sub-waveform codes described before: for example, the command

```
lcg-stimgen help sine
```

produces the following output:

```
Waveform 'sine' takes 4 parameters:
   1) amplitude
   2) frequency (Hz)
   3) phase
   4) offset
```

5.2.1 Examples

Chapter 6

Data files

In this chapter we describe the structure of the H5 files used by LCG to store the recorded data.

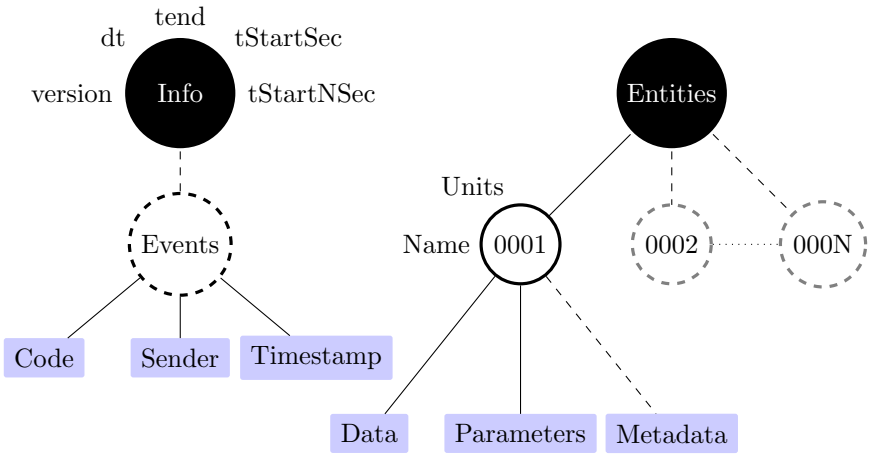

Figure 6.1: Diagram of the data file model implemented with the H5Recorder.

Chapter 7

Configuration files

This chapter describes how to write custom configuration files to be used by LCG.

7.1 Introduction

An LCG configuration file is a Extensible Markup Language *xml* file and can be opened in a standard text editor (`gedit <filename>` or `vi <filename>`). Examples can be found in the examples folder of the LCG source code or in the webpage of LCG.

The basic building blocks of LCG are called entities. An experiment consists of the interaction of several entities. We communicate with LCG by describing how the entities are connected to each other in a configuration file. Check section 8 for a description of the built in entities.

Lets examine one of these files closely:

```
<lcg>
  <simulation>
    <tend>5</tend>
    <rate>20000</rate>
  </simulation>
  <entities>
        <entity>
            <name>LIFNeuron</name>
            <id>1</id>
            <parameters>
            <C>0.08</C>
            <tau>0.0075</tau>
            <tarp>0.0014</tarp>
            <Er>-65.2</Er>
            <E0>-70</E0>
            <Vth>-50</Vth>
            <Iext>220</Iext>
            </parameters>
            <connections></connections>
```

```
20         </entity>
21   </entities>
22 </lcg>
```

Example 7.1: A simple example of a configuration file with a simulated integrate and fire neuron.

This file is composed layers. The most general is the `<lcg>` layer; this basically tells the program that this is a configuration file. Inside this layer there are two other:

- *<simulation>* - where the general parameters are defined e.g. the sampling rate (`rate`) and the duration (`tend`).
- *<entities>* - where the entities are listed and connected to each other. Generally each *<entity>* has a
  - *<name>* - the global descriptor of the entitiy.
  - *<id>* - the global identification number of an entity. This is used to connect entities and the ids have to be unique.
  - *<parameters>* - the parameters of the entity. These vary between each entity. You can use the manual section 8.
  - *<connections>* - the global `id`s to which this entity is connected.

Now lets create a file called `example.xml`. Open a terminal and copy the following lines one by one:

```
1 mkdir ~/examples  # this creates a directory called under your home
      folder (that's what the tilt is for)
2 cd ~/examples # goes inside this directory
3 gedit example1.xml # creates and opens a file called example.xml in
      the current folder using gnome text editor.
```

Note that if you do have a system without `gnome` like Mac OS the command `gedit` will not work; any text editor can be used. Now copy the example (7.1) above to this file. If you use copy and paste make sure that there are **no extra spaces**, this will depend on the your choice of editor. Save and close it. You can now run the example by doing: `LCG -c example.xml`

The `-c` option stands for **configuration file** can be used to tell LCG that we are going to pass a configuration file as an argument.

You have just ran a simulation of an integrate and fire neuron's membrane potential for 5 seconds. This was a very basic example, so the data was not recorded since there is no recording entity.

Chapter 8

Entities

Entities are one of the basic building blocks of real-time experiments with LCG. As mentioned in chapter 7, experiments are described in configuration files (that follow the XML standard) where entities are described and connected. Complex protocols can be designed by combining entities in different ways.

Going a little bit in the design of the entities, in its essence, these are C++ classes with particular defined methods that let them interface in the same way with the experiment engine. These methods are:

- initialise() - where the initial parameters of the entity are set as well as the initial input.
- firstStep() - executed in the first step of the experiment (where parameters that depend on precise timing can be initialised).
- step() - that is executed at every timestep of the simulation/experiment.
- handleEvent() - that is executed every time an event is received by the entity.
- terminate() - that is executed when the simulation/experiment ends.

Aside of the above described methods, entities have access to the *outputs* of all entities they are connected to and have an output of themselves. The output is nonetheless than a single double value that changes (or not) at every time-step depending on either the *step()* or the *handleEvent()* methods.

This chapter describes the entities contained in LCG, it provides short definitions of entity and the general properties. The descriptions can be used to create configuration files, however there are also Python helper classes to create configuration files. The helper classes are the ones used by most protocols and its usage is encouraged.

8.1 H5Recorder

The *H5Recorder* records the entities that are connected to it to HDF5 files. The entity saves the absolute time of the first step in the HDF5 file *Info* group in the *startTimeSec* and *startTimeNSec*. These times have nanosecond precision and are taken from the realtime clock.

Parameter	Type	Default	Description
`filename`	String	yyyymmddHHMMSS.h5	Name of the file to record
`compress`	Boolean	True	If true, file is saved with GZIP compression

Table 8.1: H5Recorder configuration file parameters

8.2 TriggeredH5Recorder

The *TriggeredH5Recorder* records the entities that are connected to it to HDF5 files when a TRIGGER event is received; each trigger adds a row in the dataset.

Parameter	Type	Default	Description
`before`	Double	Required	Time to record before trigger
`after`	Double	Required	Time to record after trigger
`filename`	String	yyyymmddHHMMSS.h5	Name of the file to record
`compress`	Boolean	True	If true, file is saved with GZIP compression

Table 8.2: TriggeredH5Recorder configuration file parameters

8.3 Waveform

The *Waveform* entity reads a stimulus file and generates the corresponding waveform. The stimulus description is saved as *metadata* when connected to a recorder.

Parameter	Type	Default	Description
`filename`	String	Required	Path to the stimulus file.
`units`	String	N/A	Sets the units of the entity.
`triggered`	Boolean	False	If true the entity waits to be triggered by TRIGGER events.

Table 8.3: Waveform configuration file parameters

8.4 Constant

The *Constant* entity holds a constant value with specified units.

Parameter	Type	Default	Description
`value`	Double	Required	Value to hold.
`untis`	String	N/A	Sets the units of the entity.

Table 8.4: Constant configuration file parameters

8.5 ConstantFromFile

The *ConstantFromFile* entity reads a value from a file and holds it constant in its output. Value is read from the file with the "%lf" formating.

Parameter	Type	Default	Description
`filename`	String	Required	Path of the filename to read the value from.
`units`	String	N/A	Sets the units of the entity.

Table 8.5: ConstantFromFile configuration file parameters

8.6 AnalogInput

The *AnalogInput* entity records data from an analog channel of the Data Acquisition board.

Parameter	Type	Default	Description
`deviceFile`	String	Required	Device File (example /dev/-comedi0 to record from the first comedi board).
`inputSubdevice`	Integer	Required	Subdevice (needs to be an analog input subdevice)
`readChannel`	Integer	Required	Input channel to read from.
`inputConversionFactor`	Double	Required	Conversion factor associated with the quantity being measured.
`range`	String	[-10,+10]	Input range to control the board internal gains.
`reference`	String	GRSE or NRSE	Refers to the grounding scheme being used, check the board manual.
`units`	String	mV	Units after conversion.

Table 8.6: AnalogInput configuration file parameters

8.7 AnalogOutput

The *AnalogOutput* entity outputs data to an analog channel of the Data Acquisition board.

Parameter	Type	Default	Description
`deviceFile`	String	Required	Device File (example /dev/-comedi0 to record from the first comedi board).
`ouputSubdevice`	Integer	Required	Subdevice (needs to be an analog output subdevice)
`writeChannel`	Integer	Required	Output channel to write to.
`outputConversionFactor`	Double	Required	Conversion factor associated with the quantity being output.
`range`	String	[-10,+10]	Output range to control the board internal gains.
`reference`	String	GRSE or NRSE	Refers to the grounding scheme being used, check the board manual.
`units`	String	mV	Units after conversion.

Table 8.7: AnalogOutput configuration file parameters

8.8 AnalogIO

The *AnalogIO* entity that combines entities 8.6 and 8.7. Reads from an input channel and outputs what is connected to it to an analog channel of the Data Acquisition board.

Parameter	Type	Default	Description
`deviceFile`	String	Required	Device File (example /dev/-comedi0 to record from the first comedi board).
`inputSubdevice`	Integer	Required	Subdevice (needs to be an analog input subdevice)
`readChannel`	Integer	Required	Input channel to read from.
`inputConversionFactor`	Double	Required	Conversion factor associated with the quantity being measured.
`inputRange`	String	[-10,+10]	Output range to control the board internal gains.
`reference`	String	GRSE or NRSE	Refers to the grounding scheme being used, check the board manual.
`units`	String	mV	Input units after conversion.
`ouputSubdevice`	Integer	Required	Subdevice (needs to be an analog output subdevice)
`writeChannel`	Integer	Required	Output channel to write to.
`outputConversionFactor`	Double	Required	Conversion factor associated with the quantity being output.

Table 8.8: AnalogIO configuration file parameters

8.9 RealNeuron

The *RealNeuron* entity that combines entities 8.6 and 8.7 and is able to perform Active Electrode Compensation Online Brette et al. (2008). Emits spikes upon threshold crossing.

Parameter	Type	Default	Description
deviceFile	String	Required	Device File (example /dev/-comedi0 to record from the first comedi board).
inputSubdevice	Integer	Required	Subdevice (needs to be an analog input subdevice)
readChannel	Integer	Required	Input channel to read from.
inputConversionFactor	Double	Required	Conversion factor associated with the quantity being measured.
inputRange	String	[-10,+10]	Output range to control the board internal gains.
kernelFile	String	Empty	Location of the kernel file; sets AEC online.
spikeThreshold	Double	Required	Threshold for spike detection.
V0	Double	Required	Initial voltage guess (used to initialize the state of the entity).
reference	String	GRSE or NRSE	Refers to the grounding scheme being used, check the board manual.
units	String	mV	Input units after conversion.
ouputSubdevice	Integer	Required	Subdevice (needs to be an analog output subdevice)
writeChannel	Integer	Required	Output channel to write to.
outputConversionFactor	Double	Required	Conversion factor associated with the quantity being output.
holdLastValue	Boolean	False	Whether to hold the last value (offset).
holdLastValueFilename	String	filename	Path to the file where last value is to be written.

Table 8.9: RealNeuron configuration file parameters

The **holdLastValue** parameter allows the constant injection of an offset value in the output of the entity. This can be useful to hold the cell at a certain value, it is used for instance in the PRC protocol in conjuntion with a ConstantFromFile entity .

8.10 LIFNeuron

The *LIFNeuron* entity integrates the equation for a Linear Integrate and Fire neuron with refractory period and can be used to run independent simulations or

to interface simulated neurons with real cells. It is often used also for debuging of complex protocols.

Parameter	Type	Default	Description
`C`	Double	Required	Cell membrane capacitance.
`tau`	Double	Required	Cell membrane time constant.
`tarp`	Double	Required	Refractory period.
`Er`	Double	Required	Reset potential.
`E0`	Double	Required	Equilibrium potential.
`Vth`	Double	Required	Threshold for spike emission.
`Iext`	Double	Required	External current offset.
`holdLastValue`	Boolean	False	Whether to hold the last value (offset).
`holdLastValueFilename`	String	filename	File where last value is written.

Table 8.10: LIFNeuron configuration file parameters

8.11 IzhikevichNeuron

The *IzhickevichNeuron* entity integrates the equation for a Izhickevich neuron with refractory period and can be used to run independent simulations or to interface simulated neurons with real cells; similarly to entity 8.10. It is often used also for debuging of complex protocols.

Parameter	Type	Default	Description
`a`	Double	0.02	a parameter.
`b`	Double	0.2	b parameter.
`c`	Double	-65	c parameter.
`d`	Double	2	d parameter.
`Vspk`	Double	30	Spike voltage value.
`Iext`	Double	0	External current offset.

Table 8.11: IzhickevichNeuron configuration file parameters

8.12 FrequencyEstimator

The *FrequencyEstimator* entity estimates the instantaneous frequency of an event by iterating the following formula:

$$\tilde{F}_k = {t_k}^{-1} \cdot \left(1 - e^{-t_k/\tau)}\right) + \tilde{F}_{k-1} \cdot e^{-t_k/\tau}, \tag{8.1}$$

where t_k is the time of the k^{th} event and τ (in s) the time scale over which the instantaneous event rate is estimated, weighing each new event time and the previous event rate history $\tilde{F}_{k-1}$ Wallach et al. (2011); Linaro et al. (2014).

Parameter	Type	Default	Description
`tau`	Double	Required	Estimator time constant.
`initialFrequency`	Double	0.0	Initial frequency estimation ($\tilde{F}_{k-1}$).

Table 8.12: FrequencyEstimator configuration file parameters

8.13 PID

The *PID* entity is evolved by **TRIGGER** or **SPIKE** events, here referred by triggers. The output of the entity is kept constant between triggers. Furthermore, it can be set ON or OFF by a **TOGGLE** event; when OFF, the triggers have no effect on the output of the entity.

The entity requires **two inputs**; the inputs are compared to yield the error signal $e_k = \text{input}_2 - \text{input}_1$. The output of the PID is then updated at every trigger by the equation:

$$I_{k\,holding} = g_P \cdot e_k + g_I \cdot \sum_{i=0}^{k} e_i + g_D \cdot (e_k - e_{k-1}), \tag{8.2}$$

where g_P, g_I, g_D are the proportional, integral, and derivative gains, respectively.

Parameter	Type	Default	Description
`gp`	Double	Required	Proportional gain.
`gi`	Double	Required	Integral gain.
`gd`	Double	0.0	Derivative gain.

Table 8.13: PID configuration file parameters

8.14 EventCounter

The *EventCounter* entity counts events received and emits events when a certain count is reached. It can be used to trigger or toggle other entities or to stop the experiment/simulation.

LCG currently implements the following events:

1. **SPIKE** - Signal a spike in a Neuron.
2. **TRIGGER** - Trigger other entities.

3. **RESET** - Reset other entities.
4. **TOGGLE** - Toggle other entities.
5. **STOPRUN** - Terminate the simulation/experiment.
6. **DIGITAL_RISE** - Signal the rise of a digital line.

Parameter	Type	Default	Description
`maxCount`	Integer	Required	Number of events to count.
`autoReset`	Boolean	True	Start from zero when maxCount events is reach.
`eventToCount`	String	SPIKE	Type of the events to be counted.
`eventToSend`	String	TRIGGER	Type of the events to send when maxCount is reach.

Table 8.14: EventCounter configuration file parameters

8.15 Poisson

The *Poisson* entity emits spike events following a Poisson distribution at a fixed rate. This entity has no output (is always zero) and the output is not recorded.

Parameter	Type	Default	Description
`rate`	Double	Required	Rate of the poisson point Process. If the rate is negative the point process becomes deterministic (fixed rate).
`seed`	Integer	Optional	Seed of the random algorithm.

Table 8.15: Poisson configuration file parameters

8.16 Connection

The *Connection* is an entity that introduces a delay to an event. This entity has no output.

Parameter	Type	Default	Description
`delay`	Double	Required	Value of the delay in seconds.

Table 8.16: Connection configuration file parameters

8.17 VariableDelayConnection

The *VariableDelayConnection* is an entity that introduces a delay to an event. This entity has no output and no parameters, the behavior is dictated by the operators connected to it: see entities PhasicDelay, SobolDelay and RandomDelay.

8.18 SynapticConnection

The *SynapticConnection* is an entity that introduces a delay to a SPIKE event with synaptic weight. This entity has no output.

Parameter	Type	Default	Description
`delay`	Double	Required	Value of the delay in seconds.
`weight`	Double	Required	Weight of the synapse.

Table 8.17: SynapticConnection configuration file parameters

8.19 PhasicDelay

The *PhasicDelay* is a special type of entity (functor) that returns the input multiplied by a constant value. It can be used for instance to give the phase in an slightly noisy oscillatory cycle. This entity has no output.

Parameter	Type	Default	Description
`delay`	Double	0.0	Value of the delay (usually between zero and one)

Table 8.18: PhasicDelay configuration file parameters

8.20 SobolDelay

The *SobolDelay* is a special type of entity (functor) that returns numbers drawn from a Sobol Pseudo-Random distribution multiplied by the input to the entity. Its used for instance in the PRC protocol, use the `--dry-run` switch for an example. This entity has no output.

Parameter	Type	Default	Description
`min`	Double	0.0	Minimum of the Sobol distribution
`max`	Double	1.0	Maximum of the Sobol distribution
`startSample`	Integer	0	Sample to start drawing numbers from (can be used to have consistency across simulations/experiments).

Table 8.19: SobolDelay configuration file parameters

8.21 RandomDelay

The *RandomDelay* is a special type of entity (functor) that returns the input multiplied by a constant value. It can be defined using an interval or the mean and standard deviation of the gaussian distribution (see below). This entity has no output.

Parameter	Type	Default	Description
`interval`	Double,Double	Required	Interval between the numbers are drawn from

Table 8.20: RandomDelay configuration file parameters using intervals.

Parameter	Type	Default	Description
`mean`	Double	Required	Mean of the gaussian distribution.
`stdev`	Double	Required	Standard deviation of the gaussian distribution.

Table 8.21: RandomDelay configuration file parameters using distribution parameters.

8.22 PeriodicPulse

The *PeriodicPulse* entity emits pulses at fixed periods with a given duration and amplitude. Emits a TRIGGER event when a pulse starts.

Parameter	Type	Default	Description
`frequency`	Double	Required	Frequency of the pulses.
`duration`	Double	Required	Sets the duration of the pulses (in seconds).
`amplitude`	Double	Required	Sets the amplitude of the pulses.
`units`	String	pA	Units of the output.

Table 8.22: PeriodicPulse configuration file parameters

8.23 PeriodicTrigger

The *PeriodicTrigger* entity emits trigger events with a fixed period.

Parameter	Type	Default	Description
`frequency`	Double	Required	Frequency of the triggers.
`delay`	Double	0	Sets the Delay of the first trigger.
`tend`	Double	INFINITY	Sets the maximum time of the last trigger.

Table 8.23: PeriodicTrigger configuration file parameters

8.24 ExponentialSynapse

The *ExponentialSynapse* entity models a synapse as a single exponential. Needs to be connected to a Neuron and injects a conductance. Spike events increase the weight of the synapse.

Parameter	Type	Default	Description
`E`	Double	Required	Reversal Potential.
`tau`	Double	Required	Time constant.

Table 8.24: ExponentialSynapse configuration file parameters

8.25 Exp2Synapse

The *Exp2Synapse* entity models a synapse as a alpha function. Needs to be connected to a Neuron and injects a conductance. Spike events increase the weight of the synapse.

Parameter	Type	Default	Description
`E`	Double	Required	Reversal Potential.
`tauRise`	Double	Required	Rise time constant.
`tauDecay`	Double	Required	Decay time constant.

Table 8.25: Exp2Synapse configuration file parameters

8.26 TMGSynapse

The *TMGSynapse* entity models a Tsodyks-Markram synapse. Needs to be connected to a Neuron and injects a conductance. Spike events increase the weight of the synapse.

Parameter	Type	Default	Description
`E`	Double	Required	Reversal Potential.
`U`	Double	Required	Amplitude.
`tau1`	Double	Required	Time constant.
`tauRec`	Double	Required	Recovery time constant.
`tauFacil`	Double	Required	Facilitation time constant.

Table 8.26: TMGSynapse configuration file parameters

8.27 ConductanceStimulus

The *ConductanceStimulus* entity is used to inject a conductance into a neuron, it uses the first input as conductance waveform and must be connected to a RealNeuron to be used. The following formula is used to compute the conductance:

$$I_k = g \cdot (E - V), \tag{8.3}$$

with g the conductance waveform given by the first input, E the reversal potential of the conductance and V the voltage taken from the RealNeuron.

Parameter	Type	Default	Description
`E`	Double	Required	Reversal Potential.

Table 8.27: ConductanceStimulus configuration file parameters

8.28 NMDAConductanceStimulus

The *NMDAConductanceStimulus* entity is used to inject an NMDA conductance into a neuron, it uses the first input as conductance waveform and must be connected to a RealNeuron to be used (like entity 8.27). The following formula is used to compute the conductance:

$$I_k = \frac{g \cdot (E - V)}{1 + K_1 \cdot exp^{-K_2 \cdot V}}, \tag{8.4}$$

with g the conductance waveform given by the first input, E the reversal potential of the conductance and V the voltage taken from the RealNeuron.

Parameter	Type	Default	Description
`E`	Double	Required	Reversal Potential.
`K1`	Double	Required	NMDA amplitude.
`K2`	Double	Required	NMDA time constant.

Table 8.28: ConductanceStimulus configuration file parameters

8.29 OU

The *OU* entity outputs an Ornstein-Uhlenbeck fluctuating waveform with static mean, standard deviation and time constant.

Parameter	Type	Default	Description
`mean`	Mean	Required	Sets the mean of the OU process.
`stdev`	Double	Required	Sets the standard deviation of the OU process.
`tau`	Double	Required	Sets the time constant of the OU process.
`units`	String	Required	Sets the units of the entity.
`initialCondition`	Double	Required	Sets the initial Condition.
`interval`	Double,Double	Required	Sets the interval.

Table 8.29: OU configuration file parameters

8.30 OUNonStationary

The *OUNonStationary* entity outputs an Ornstein-Uhlenbeck fluctuating waveform with mean and standard deviation taken from the first two inputs.

Parameter	Type	Default	Description
`mean`	Mean	Required	Sets the mean of the OU process.
`stdev`	Double	Required	Sets the standard deviation of the OU process.
`tau`	Double	Required	Sets the time constant of the OU process.
`units`	String	Required	Sets the units of the entity.
`initialCondition`	Double	Required	Sets the initial Condition.
`interval`	Double,Double	Required	Sets the interval.

Table 8.30: OUNonStationary configuration file parameters

8.31 HHSodium

The *HHSodium* entity models a Hodgkin-Huxley sodium current.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.12	Maximum conductance.
`E`	Double	50.0	Reversal Potential.

Table 8.31: HHSodium configuration file parameters

8.32 HHPotassium

The *HHPotassium* entity models a Hodgkin-Huxley potassium current.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.036	Maximum conductance.
`E`	Double	-77.0	Reversal Potential.

Table 8.32: HHPotassium configuration file parameters

8.33 HHSodiumCN

The *HHSodiumCN* entity models a Hodgkin-Huxley sodium current with Channel Noise.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.12	Maximum conductance.
`E`	Double	50.0	Reversal Potential.
`gamma`	Double	10.0	Single channel conductance.

Table 8.33: HHSodiumCN configuration file parameters

8.34 HHPotassiumCN

The *HHPotassiumCN* entity models a Hodgkin-Huxley potassium current with channel noise.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.036	Maximum conductance.
`E`	Double	-77.0	Reversal Potential.
`gamma`	Double	10.0	Single channel conductance.

Table 8.34: HHPotassiumCN configuration file parameters

8.35 HH2Sodium

The *HH2Sodium* entity models a Hodgkin-Huxley sodium current with temperature correction.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.12	Maximum conductance.
`E`	Double	50.0	Reversal Potential.
`vtraub`	Double	-63	variable to adjust threshold.
`temperature`	Double	36	Reversal Potential.

Table 8.35: HH2Sodium configuration file parameters

8.36 HH2Potassium

The *HH2Potassium* entity models a Hodgkin-Huxley potassium current with temperature correction.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.036	Maximum conductance.
`E`	Double	-77.0	Reversal Potential.
`vtraub`	Double	-63	variable to adjust threshold.
`temperature`	Double	36	Reversal Potential.

Table 8.36: HH2Potassium configuration file parameters

8.37 MCurrent

The *MCurrent* entity models a persistent sodium current with temperature correction.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.12	Maximum conductance.
`E`	Double	50.0	Reversal Potential.
`taumax`	Double	1000	time constant.
`temperature`	Double	36	Reversal Potential.

Table 8.37: MCurrent configuration file parameters

8.38 WBSodium

The *WBSodium* entity models a Wang-Buzsaki sodium current.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.035	Maximum conductance.
`E`	Double	55.0	Reversal Potential.

Table 8.38: WBSodium configuration file parameters

8.39 WBPotassium

The *WBPotassium* entity models a Wang-Buzsaki potassium current.

Parameter	Type	Default	Description
`area`	Double	Required	Area to be used in the conductance scaling.
`gbar`	Double	0.009	Maximum conductance.
`E`	Double	-90.0	Reversal Potential.

Table 8.39: WBPotassium configuration file parameters

Chapter 9

Additional features

This chapter will describe some additional features present in LCG that most likely are not of daily use. However, they can prove rather useful in some particular scenarios.

9.1 Computation of the SHA checksum

9.2 Adding comments to data files

References

Bibliography

R. Brette, Z. Piwkowska, C. Monier, M. Rudolph-Lilith, J. Fournier, M. Levy, Y. Frégnac, T. Bal, and A. Destexhe, Neuron **59**, 379 (2008).

C. Koch and I. Segev, eds., *Methods in Neuronal Modeling: From Synapses to Networks (Computational Neuroscience)* (The MIT Press, 1989).

P. Wallisch, *Matlab for Neuroscientists*, Neuroscience 2011 (Academic Press, 2011), ISBN 9780123838360, URL `http://books.google.be/books?id=K3zFygAACAAJ`.

Z. F. Mainen and T. J. Sejnowski, Science **268**, 1503 (1995).

L. Badel, S. Lefort, R. Brette, C. C. H. Petersen, W. Gerstner, and M. J. E. Richardson, Journal of neurophysiology **99**, 656 (2008).

S. Crochet and C. C. H. Petersen, Nature neuroscience **9**, 608 (2006).

A. Wallach, D. Eytan, A. Gal, C. Zrenner, and S. Marom, Frontiers in neuroengineering **4**, 3 (2011).

D. Linaro, J. Couto, and M. Giugliano, Journal of Neuroscience Methods **230**, 5 (2014).

Appendix A

MATLAB reference

TODO

Appendix B

UNIX reference

TODO