NEST simulator Documentation

Release 1.0.0

steffengraber

Mar 18, 2019

Contents

CHAPTER 1

Welcome to the NEST simulator documentation!

Download	*Install*

NEST is a simulator for **spiking neural network models**, ideal for networks of any size, for example:

1. Models of information processing e.g. in the visual or auditory cortex of mammals,
2. Models of network activity dynamics, e.g. laminar cortical networks or balanced random networks,
3. Models of learning and plasticity.

New to NEST? Start here at our *Getting Started* page

Have an idea of the type of model you need? Click on one of the images to access our model directory:

Create complex networks using the Topology Module or the Microcircuit Model:

Need a different model? Check out how you can create you own model here.

Have a question or issue with NEST? See our *Getting Help* page.

1.1 How the documentation is organized

- *Tutorials* show you step by step instructions using NEST. If you haven't used NEST before, the PyNEST tutorial is a good place to start.
- *Example Networks* demonstrate the use of dozens of the neural network models implemented in NEST.
- *Topical Guides* provide deeper insight into several topics and concepts from *Parallel Computing* to handling *Gap Junction Simulations* and setting up a topological network.
- *Reference Material* provides a quick look up of definitions, functions and terms.

1.2 Contribute

- Have you used NEST in an article or presentation? *Let us know* and we will add it to our list of publications. Find out how to cite NEST in your work.
- If you have any comments or suggestions, please share them on our *Mailing List*.
- Want to contribute code? Check out our Developer Space to get started!
- For more info about our larger community and the history of NEST check out the NEST Initiative website

1.2.1 Links to other projects:

The NeuralEnsemble is a community-based initiative to promote and co-ordinate open-source software development in neuroscience. They host numerous software including PyNN, a simulator-independent language for building neuronal network models and Elephant (Electrophysiology Analysis Toolkit), a package for the analysis of neurophysiology data, using Neo data structures.

CHAPTER 2

Download NEST

NEST is available under the *GNU General Public License 2 or later*. This means that you can

- use NEST for your research,
- modify and improve NEST according to your needs,
- distribute NEST to others under the same license.

If you use NEST for your project, don't forget to cite NEST!

2.1 Download the current version of NEST here:

2.1.1 Current Release NEST 2.16.0

Release Notes

Latest developer version

2.2 Download the NEST Live Media for Virtual Machines

Live media is available in the OVA format, and is suitable, for example, for importing into VirtualBox. If you run **Windows**, this is the option for you OR if you just want to run NEST without installing it on your computer.

NEST Live Media 2.14.0 (OVA, 2.5G)

Checksum 2.14.0

See the *install instructions for Live Media*

2.3 Previous Releases

We continuously aim to improve NEST and implement features and fix bugs with every new version; thus, we strongly encourage our users to use the **most recent version of NEST**. However, if you do need an older version you can find all NEST releases here.

Older Versions of Live Media

- Ubuntu 16.04 Live Media with NEST 2.12.0
  - Download 2.12.0 (OVA, 3.2G)
  - Checksum 2.12.0 (sha512sum)
- Ubuntu 16.04 Live Media with NEST 2.10.0
  - Download 2.10.0 (OVA, ~3.7G)
  - Checksum 2.10.0 (sha512sum)
- Ubuntu 15.10 Live Media with NEST 2.8.0
  - Download 2.8.0 (OVA, ~2.5G)
  - Checksum 2.8.0 (sha512sum)

CHAPTER 3

Installation Instructions

3.1 Ubuntu / Debian Installation

3.1.1 Standard Installation

The following are the basic steps to compile and install NEST from source code:

- For most users, the following additional packages will likely be needed (see also the *Dependencies* section)

```
sudo apt-get install -y build-essential cmake libltdl7-dev libreadline6-dev \
libncurses5-dev libgsl0-dev python-all-dev python-numpy python-scipy \
python-matplotlib ipython openmpi-bin libopenmpi-dev python-nose
```

- Unpack the tarball

```
tar -xzvf nest-simulator-x.y.z.tar.gz
```

- Create a build directory:

```
mkdir nest-simulator-x.y.z-build
```

- Change to the build directory:

```
cd nest-simulator-x.y.z-build
```

- Configure NEST:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=</install/path> </path/to/NEST/src>
```

Note: `/install/path` should be an absolute path

You may need additional `cmake` options and you can find the configuration options *here*

- Compile and install NEST:

```
make
make install
make installcheck
```

NEST should now be successfully installed on your system. You should now be able to `import nest` from a python or ipython shell.

Note: If your operating system does not find the `nest` executable or if Python does not find the `nest` module, your path variables may not be set correctly. This may also be the case if Python cannot load the `nest` module due to missing or incompatible libraries. In this case, please run:

```
source </path/to/nest_install_dir>/bin/nest_vars.sh
```

to set the necessary environment variables. You may want to include this line in your `.bashrc` file, so that the environment variables are set automatically.

See the *Getting started* pages to find out how to get going with NEST or check out our example networks.

3.1.2 Dependencies

To build NEST, you need a recent version of CMake and libtool; the latter should be available for most systems and is probably already installed.

Note: NEST requires at least version v2.8.12 of cmake, but we recommend v3.4 or later. You can type `cmake --version` on the commandline to check your current version.

The GNU readline library is recommended if you use NEST interactively **without Python**. Although most Linux distributions have GNU readline installed, you still need to install its development package if want to use GNU readline with NEST. GNU readline itself depends on libncurses (or libtermcap on older systems). Again, the development packages are needed to compile NEST.

The GNU Scientific Library is needed by several neuron models, in particular those with conductance based synapses. If you want these models, please install the GNU Scientific Library along with its development packages.

If you want to use PyNEST, we recommend to install the following along with their development packages:

- Python
- NumPy
- SciPy
- matplotlib
- IPython

See the *Configuration Options* or the *High Performance Computing* instructions to further adjust settings for your system.

3.1.3 What gets installed where

By default, everything will be installed to the subdirectories `/install/path/{bin,lib,share}`, where `/install/path` is the install path given to `cmake`:

- Executables `/install/path/bin`
- Dynamic libraries `/install/path/lib/`
- SLI libraries `/install/path/share/nest/sli`
- Documentation `/install/path/share/doc/nest`
- Examples `/install/path/share/doc/nest/examples`
- PyNEST `/install/path/lib/pythonX.Y/site-packages/nest`
- PyNEST examples `/install/path/share/doc/nest/examples/pynest`
- Extras `/install/path/share/nest/extras/`

If you want to run the `nest` executable or use the `nest` Python module without providing explicit paths, you have to add the installation directory to your search paths. For example, if you are using bash:

```
export PATH=$PATH:/install/path/bin
export PYTHONPATH=/install/path/lib/pythonX.Y/site-packages:$PYTHONPATH
```

The script `/install/path/bin/nest_vars.sh` can be sourced in `.bashrc` and will set these paths for you. This also allows to switch between NEST installations in a convenient manner.

3.2 Installation on MAC OS X

On the Mac, you can install NEST either via Homebrew or manually. If you want to use PyNEST, you need to have a version of Python with some science packages installed, see the section Python on Mac for details.

3.2.1 Installation via Homebrew

The easiest way to install NEST on a Mac is to install it via the Homebrew package manager:

- To install homebrew, follow the instructions at brew.sh
- Then, in a terminal
  - Add the homebrew/science tap by running:

    ```
    brew tap brewsci/science
    ```

  - For information on what options NEST has and what will be installed, run:

    ```
    brew info nest
    ```

  - To install nest, execute

    ```
    brew install nest
    ```

Options have to be appended, so for example, to install NEST with PyNEST run:

```
brew install nest --with-python
```

This will install the most recent release version of NEST. To build NEST from the most recent sources on Github, use:

```
brew install nest --HEAD
```

3.2.2 Manual installation

The clang/clang++ compiler that ships with OS X/macOS does not support OpenMP threads and creates code that fails some tests. You therefore need to use **GCC** to compile NEST under OS X/macOS.

Installation instructions here have been tested under OS X 10.11 *El Capitan* and macOS 10.12 *Sierra* with Anaconda Python 2 and 3 and all other dependencies installed via Homebrew. See below for *Manual installation with dependencies from MacPorts*.

- Install Xcode from the AppStore.
- Install the Xcode command line tools by executing the following line in the terminal and following the instructions in the windows that will pop up:

```
xcode-select --install
```

- Install dependencies via Homebrew:

```
brew install gcc cmake gsl open-mpi libtool
```

- Create a directory for building and installing NEST (you should always build NEST outside the source code directory; installing NEST in a "place of its own" makes it easy to remove NEST later).
- Extract the NEST tarball as a subdirectory in that directory or clone NEST from GitHub into a subdirectory:

```
mkdir NEST       # directory for all NEST stuff
cd NEST
tar zxf nest-simulator-x.y.z.tar.gz
mkdir bld
cd bld
```

- Configure and build NEST inside the build directory:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=</install/path> \
      -DCMAKE_C_COMPILER=gcc-6 \
      -DCMAKE_CXX_COMPILER=g++-6 \
      </path/to/NEST/src>
```

```
make -j4         # -j4 builds in parallel using 4 processes
make install
make installcheck
```

To compile NEST with MPI support, add `-Dwith-mpi=ON` as `cmake` option.

Manual installation with dependencies from MacPorts

The following should work if you install dependencies using MacPorts (only steps that differ from the instructions above are shown):

- Install dependencies via MacPorts:

```
sudo port install gcc6 cmake gsl openmpi-default libtool \
python27 py27-cython py27-nose doxygen
```

- Configure and build NEST inside the build directory

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=</install/path> \
      -DPYTHON_LIBRARY=/opt/local/lib/libpython2.7.dylib \
      -DPYTHON_INCLUDE_DIR=/opt/local/Library/Frameworks/Python.framework/
↪Versions/2.7/include/python2.7 \
      -DCMAKE_C_COMPILER=/opt/local/bin/gcc-mp-6 \
      -DCMAKE_CXX_COMPILER=/opt/local/bin/g++-mp-6 \
      </path/to/NEST/src>
```

```
make -j4         # -j4 builds in parallel using 4 processes
make install
make installcheck
```

To compile NEST with MPI support, add `-Dwith-mpi=ON` as `cmake` option.

3.2.3 Python on Mac

The version of Python shipping with OS X/macOS is rather dated and does not include key packages such as NumPy. Therefore, you need to install Python via a channel that provides scientific packages.

One well-tested source is the Anaconda Python distribution for both Python 2 and 3. If you do not want to install the full Anaconda distribution, you can also install Miniconda and then install the packages needed by NEST by running:

```
conda install numpy scipy matplotlib ipython cython nose
```

Alternatively, you should be able to install the necessary Python packages via Homebrew, but this has not been tested.

3.3 High Performance Computer Systems Installation

3.3.1 Minimal configuration

NEST can be compiled without any external packages; such a configuration may be useful for initial porting to a new supercomputer. However, this implies several restrictions:

- Some neuron and synapse models will not be available, as they depend on ODE solvers from the GNU Scientific Library.
- The Python extension will not be available
- Multi-threading and parallel computing facilities will be disabled.

To configure NEST for compilation without external packages, use the following command:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=</install/path> \
      -Dwith-python=OFF \
      -Dwith-gsl=OFF \
      -Dwith-readline=OFF \
      -Dwith-ltdl=OFF \
      -Dwith-openmp=OFF \
      </path/to/nest/source>
```

See the *Configuration Options* to further adjust settings for your system.

3.3.2 Compiling for BlueGene/Q

NEST provides a cmake tool-chain file for cross compilation for BlueGene/Q. When configuring NEST use the following `cmake` line:

```
cmake -DCMAKE_TOOLCHAIN_FILE=Platform/BlueGeneQ_XLC \
      -DCMAKE_INSTALL_PREFIX:PATH=</install/path> \
      -Dwith-python=OFF \
      -Dstatic-libraries=ON \
      </path/to/NEST/src>
```

If you compile dynamically, be aware that the BlueGene/Q system might not provide an `ltdl` library. If you want to dynamically load an external user module, you have to compile and install an `ltdl` yourself and add `-Dwith-ltdl=<ltdl-install-dir>` to the `cmake` line. Otherwise add `-Dwith-ltdl=OFF`.

Additionally, the design of `cmake`'s MPI handling has a broken design, which is brittle in the case of BGQ and certain libraries (flags to use SIONlib, for example).

If you run into that, you must force `cmake` to use the wrappers rather than it's attempts to extract the proper flags for the underlying compiler as in:

```
-DCMAKE_C_COMPILER=/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlc_r
-DCMAKE_CXX_COMPILER=/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlcxx_r
```

BlueGene/Q and PyNEST

Building PyNEST on BlueGene/Q requires you to compile dynamically, i.e. `-Dstatic-libraries=OFF`. Furthermore, you have to cythonize the `pynest/pynestkernel.pyx/.pyx` on a machine with Cython installed:

```
cythonize pynestkernel.pyx
```

Copy the generated file `pynestkernel.cpp` into `</path/to/NEST/src>/pynest` on BlueGene/Q and point `-Dwith-python=<...>` to a valid python version for cross compilation, either Python 2:

```
-Dwith-python=/bgsys/tools/Python-2.7/bin/hostpython
```

or (much better) Python 3:

```
-Dwith-python=/bgsys/local/python3/3.4.2/bin/python3
```

CMake <3.4 is buggy when it comes to finding the matching libraries (for many years). Thus, you also have to specify `PYTHON_LIBRARY` and `PYTHON_INCLUDE_DIR` if they are not found OR the incorrect libraries are found, e.g.:

```
-DPYTHON_LIBRARY=/bgsys/tools/Python-2.7/lib64/libpython2.7.so.1.0
-DPYTHON_INCLUDE_DIR=/bgsys/tools/Python-2.7/include/python2.7
```

or (much better):

```
-DPYTHON_LIBRARY=/bgsys/local/python3/3.4.2/lib/libpython3.4m.a
-DPYTHON_INCLUDE_DIR=/bgsys/local/python3/3.4.2/include/python3.4m
```

A complete `cmake` line for PyNEST could look like this:

```
module load gsl

cmake -DCMAKE_TOOLCHAIN_FILE=Platform/BlueGeneQ_XLC \
```

(continues on next page)

(continued from previous page)

```
  -DCMAKE_INSTALL_PREFIX=</install/path> \
  -Dstatic-libraries=OFF \
  -Dcythonize-pynest=OFF \
      -DCMAKE_C_COMPILER=/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlc_r \
      -DCMAKE_CXX_COMPILER=/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlcxx_r \
      -Dwith-python=/bgsys/local/python3/3.4.2/bin/python3 \
      -DPYTHON_LIBRARY=/bgsys/local/python3/3.4.2/lib/libpython3.4m.a \
      -DPYTHON_INCLUDE_DIR=/bgsys/local/python3/3.4.2/include/python3.4m \
  -Dwith-ltdl=OFF \
  <nest-src>
```

Furthermore, for running PyNEST, make sure all python dependencies are installed and environment variables are set properly:

```
module load python3/3.4.2

# adds PyNEST to the PYTHONPATH
source <nest-install-dir>/bin/nest_vars.sh

# makes HOME and PYTHONPATH available for python
runjob \
  --exp-env HOME \
  --exp-env PATH \
  --exp-env LD_LIBRARY_PATH \
  --exp-env PYTHONUNBUFFERED \
  --exp-env PYTHONPATH \
  ... \
  : /bgsys/local/python3/3.4.2/bin/python3.4 script.py
```

BlueGene/Q and GCC

Compiling NEST with GCC (`-DCMAKE_TOOLCHAIN_FILE=Platform/BlueGeneQ_GCC`) might require you to use a GSL library compiled using GCC, otherwise undefined symbols break your build. After the GSL is built with GCC and installed in `gsl-install-dir`, add `-Dwith-gsl=<gsl-install-dir>` to the `cmake` line.

BlueGene/Q and Non-Standard Allocators

To use NEST with non-standard allocators on BlueGene/Q (e.g., tcmalloc), you should compile NEST and the allocator with the same compiler, usually GCC. Since static linking is recommended on BlueGene/Q, the allocator also needs to be linked statically. This requires specifying linker flags and the allocator library as shown in the following example:

```
cmake -DCMAKE_TOOLCHAIN_FILE=Platform/BlueGeneQ_GCC \
      -DCMAKE_INSTALL_PREFIX:PATH=$PWD/install \
      -Dstatic-libraries=ON -Dwith-warning=OFF \
      -DCMAKE_EXE_LINKER_FLAGS="-Wl,--allow-multiple-definition" \
      -Dwith-libraries=$HOME/tcmalloc/install/lib/libtcmalloc.a
```

3.3.3 Compiling for Fujitsu Sparc64

On the K Computer: The preinstalled `cmake` version is 2.6, which is too old for NEST. Please install a newer version, for example:

```
wget https://cmake.org/files/v3.4/cmake-3.4.2.tar.gz
tar -xzf cmake-3.4.2.tar.gz
mv cmake-3.4.2 cmake.src
mkdir cmake.build
cd cmake.build
../cmake.src/bootstrap --prefix=$PWD/install --parallel=4
gmake -j4
gmake install
```

Also you might need a cross compiled GNU Scientific Library (GSL). For GSL 2.1 this is a possible installation scenario:

```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-2.1.tar.gz
tar -xzf gsl-2.1.tar.gz
mkdir gsl-2.1.build gsl-2.1.install
cd gsl-2.1.build
../gsl-2.1/configure --prefix=$PWD/../gsl-2.1.install/ \
                     CC=mpifccpx \
                     CXX=mpiFCCpx \
                     CFLAGS="-Nnoline" \
                     CXXFLAGS="--alternative_tokens -O3 -Kfast,openmp, -Nnoline, -
↪Nquickdbg -NRtrap" \
                     --host=sparc64-unknown-linux-gnu \
                     --build=x86_64-unknown-linux-gnu
gmake -j4
gmake install
```

To install NEST, use the following `cmake` line:

```
cmake -DCMAKE_TOOLCHAIN_FILE=Platform/Fujitsu-Sparc64 \
      -DCMAKE_INSTALL_PREFIX:PATH=</install/path> \
      -Dwith-gsl=/path/to/gsl-2.1.install/ \
      -Dwith-optimize="-Kfast" \
      -Dwith-defines="-DUSE_PMA" \
      -Dwith-python=OFF \
      -Dwith-warning=OFF \
      </path/to/NEST/src>
make -j4
make install
```

The compilation can take quite some time compiling the file `models/modelsmodule.cpp` due to generation of many template classes. To speed up the process, you can comment out all synapse models you do not need. The option `-Kfast` on the K computer enables many different options:

```
-O3 -Kdalign,eval,fast_matmul,fp_contract,fp_relaxed,ilfunc,lib,mfunc,ns,omitfp,
↪prefetch_conditional,rdconv -x-
```

Be aware that, with the option `-Kfast` an internal compiler error - probably an out of memory situation - can occur. One solution is to disable synapse models that you don't use in `models/modelsmodule.cpp`. From current observations this might be related to the `-x-` option; you can give it a fixed value, e.g `-x1`, and the compilation succeeds (the impact on performance was not analyzed):

```
-Dwith-optimize="-Kfast -x1"
```

3.4 NEST LIVE MEDIA Installation

Download and install a virtual machine if you do not already have one installed.

Note: Although, the following instructions are for Virtual Box, you can use a different virtual machine, such as VMWare.

For Linux users, it is possible to install Virtual Box from the package repositories.

Debian:

```
sudo apt-get install virtualbox
```

Fedora:

```
sudo dnf install virtualbox
```

SuSe:

```
sudo zypper install virtualbox
```

3.4.1 NEST image setup

- Download the NEST live medium
- Start Virtual Box and import the virtual machine image "lubuntu-16.04_nest-2.14.0.ova" (**File > Import Appliance**)
- Once imported, you can run the NEST image
- The user password is **nest**.

Notes

- For better performance you can increase the memory for the machine **Settings > System > Base Memory**
- To allow fullscreen mode of the virtual machine you also need to increase the video memory above 16MB. (**Settings > Display > Video Memory**)
- If you need to share folders between the virtual machine and your regular desktop environment, click on **Settings**. Choose **Shared Folder** and add the folder you wish to share. Make sure to mark **automount**.
- To install Guest Additions, select **Devices > Insert Guest Additions CD image...** (top left of the VirtualBox Window). Then, open a terminal (Ctrl+Alt+t), go to "/media/nest/VBOXADDITIONS.../" and run "sudo bash VboxLinuxAdditions.run".
- To set the correct language layout for your keyboard (e.g., from "US" to "DE"), you can use the program "lxkeymap", which you start by typing "lxkeymap" in the terminal.

3.5 Configuration Options

NEST is installed with `cmake` (at least v2.8.12). In the simplest case, the commands:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=</install/path> </path/to/NEST/src>
make
make install
```

should build and install NEST to `/install/path`, which should be an absolute path.

3.5.1 Choice of CMake Version

We recommend to use `cmake` v3.4 or later, even though installing NEST with `cmake` v2.8.12 will in most cases work properly. For more detailed information please see below: `Python3 Binding (PyNEST)`

3.5.2 Choice of compiler

The default compiler for NEST is GNU gcc/g++, but NEST has also successfully been compiled with other compilers, including Intel icc/icpc, Pathscale, Portland and IBM compilers.

To select a specific compiler, please add the following flags to your `cmake` line:

```
-DCMAKE_C_COMPILER=<C-compiler> -DCMAKE_CXX_COMPILER=<C++-compiler>
```

3.5.3 Options for configuring NEST

NEST allows for several configuration options for custom builds:

Change NEST behavior:

```
-Dtics_per_ms=[number]     Specify elementary unit of time. [default 1000.0]
-Dtics_per_step=[number]   Specify resolution. [default 100]
-Dwith-ps-arrays=[OFF|ON]  Use PS array construction semantics. [default=ON]
```

Add user modules:

```
-Dexternal-modules=[OFF|<list;of;modules>]  External NEST modules to be linked
                                            in, separated by ';'. [default=OFF]
```

Connect NEST with external projects:

```
-Dwith-libneurosim=[OFF|ON|</path/to/libneurosim>]  Request the use of libneurosim.
                                                    Optionally give the directory,
                                                    where libneurosim is installed.
                                                    [default=OFF]
-Dwith-music=[OFF|ON|</path/to/music>] Request the use of MUSIC. Optionally
                                       give the directory, where MUSIC is installed.
                                       [default=OFF]
```

Change parallelization scheme:

```
-Dwith-mpi=[OFF|ON|</path/to/mpi>]  Request compilation with MPI. Optionally
                                    give directory with MPI installation.
                                    [default=OFF]
-Dwith-openmp=[OFF|ON|<OpenMP-Flag>]  Enable OpenMP multi-threading.
                                      Optional: set OMP flag. [default=ON]
```

Set default libraries:

```
-Dwith-gsl=[OFF|ON|</path/to/gsl>]            Find a gsl library. To set a specific
                                              library, set install path.[default=ON]
-Dwith-readline=[OFF|ON|</path/to/readline>] Find a GNU Readline library. To set
                                              a specific library, set install path.
                                              [default=ON]
-Dwith-ltdl=[OFF|ON|</path/to/ltdl>]         Find an ltdl library. To set a specific
                                              ltdl, set install path. NEST uses the
                                              ltdl for dynamic loading of external
                                              user modules. [default=ON]
-Dwith-python=[OFF|ON|2|3]                   Build PyNEST. To set a specific Python
                                              version, set 2 or 3. [default=ON]
-Dcythonize-pynest=[OFF|ON]                  Use Cython to cythonize pynestkernel.pyx.
                                              If OFF, PyNEST has to be build from
                                              a pre-cythonized pynestkernel.pyx.
                                              [default=ON]
```

Change compilation behavior:

```
-Dstatic-libraries=[OFF|ON]     Build static executable and libraries. [default=OFF]
-Dwith-optimize=[OFF|ON|<list;of;flags>]  Enable user defined optimizations. Separate
                                          multiple flags by ';'.
                                          [default OFF, when ON, defaults to '-O3']
-Dwith-warning=[OFF|ON|<list;of;flags>]   Enable user defined warnings. Separate
                                          multiple flags by ';'.
                                          [default ON, when ON, defaults to '-Wall']
-Dwith-debug=[OFF|ON|<list;of;flags>]     Enable user defined debug flags. Separate
                                          multiple flags by ';'.
                                          [default OFF, when ON, defaults to '-g']
-Dwith-libraries=<list;of;libraries>      Link additional libraries. Give full path.
                                          Separate multiple libraries by ';'.
                                          [default OFF]
-Dwith-includes=<list;of;includes>        Add additional include paths. Give full
                                          path without '-I'. Separate multiple include
                                          paths by ';'. [default OFF]
-Dwith-defines=<list;of;defines>          Additional defines, e.g. '-DXYZ=1'.
                                          Separate multiple defines by ';'. [default␣
↪OFF]
```

3.5.4 NO-DOC option

On systems where help extraction is slow, the call to `make install` can be replaced by `make install-nodoc` to skip the generation of help pages and indices. Using this option can help developers to speed up development cycles, but is not recommended for production use as it renders the built-in help system useless.

3.5.5 Configuring NEST for Distributed Simulation with MPI

1. Try `-Dwith-mpi=ON` as argument for `cmake`. If it works, fine.
2. If 1 does not work, or you want to use a non-standard MPI, try `-Dwith-mpi=/path/to/my/mpi`. Directory mpi should contain include, lib, bin subdirectories for MPI.
3. If that does not work, but you know the correct compiler wrapper for your machine, try configure `-DMPI_CXX_COMPILER=myC++_CompilerWrapper -DMPI_C_COMPILER=myC_CompilerWrapper -Dwith-mpi=ON`
4. Sorry, you need to fix your MPI installation.

3.5.6 Tell NEST about your MPI setup

If you compiled NEST with support for distributed computing via MPI, you have to tell it how your `mpirun`/`mpiexec` command works by defining the function mpirun in your `~/.nestrc` file. This file already contains an example implementation that should work with OpenMPI library.

3.5.7 Disabling the Python Bindings (PyNEST)

To disable Python bindings use:

```
-Dwith-python=OFF
```

as an argument to `cmake`.

Please see also the file ../../pynest/README.md in the documentation directory for details.

3.5.8 Python3 Binding (PyNEST)

To force a Python3-binding in a mixed Python2/3 environment pass:

```
-Dwith-python=3
```

as an argument to `cmake`.

`cmake` usually autodetects your Python installation. In some cases `cmake` might not be able to localize the Python interpreter and its corresponding libraries correctly. To circumvent such a problem following `cmake` built-in variables can be set manually and passed to `cmake`:

```
 PYTHON_EXECUTABLE ..... path to the Python interpreter
 PYTHON_LIBRARY ........ path to libpython
 PYTHON_INCLUDE_DIR .... two include ...
 PYTHON_INCLUDE_DIR2 ... directories

e.g.: Please note ``-Dwith-python=ON`` is the default::
 cmake -DCMAKE_INSTALL_PREFIX=</install/path> \
       -DPYTHON_EXECUTABLE=/usr/bin/python3 \
       -DPYTHON_LIBRARY=/usr/lib/x86_64-linux-gnu/libpython3.4m.so \
       -DPYTHON_INCLUDE_DIR=/usr/include/python3.4 \
       -DPYTHON_INCLUDE_DIR2=/usr/include/x86_64-linux-gnu/python3.4m \
       </path/to/NEST/src>
```

3.6 Compiling for Apple OSX/macOS

NEST can currently not be compiled with the clang/clang++ compilers shipping with macOS. Therefore, you first need to install GCC 6.3 or later. The easiest way to install all required software is using Homebrew (from http://brew.sh):

```
brew install gcc cmake gsl open-mpi libtool
```

will install all required prequisites. You can then configure NEST with

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=</install/path> \
      -DCMAKE_C_COMPILER=gcc-6\
      -DCMAKE_CXX_COMPILER=g++-6 \
      </path/to/NEST/src>
```

For detailed information on installing NEST under OSX/macOS, please see the "macOS" section of http://www.nest-simulator.org/installation.

3.6.1 Choice of compiler

Most NEST developers use the GNU gcc/g++ compilers. We also regularly compile NEST using the IBM xlc/xlC compilers. You can find the version of your compiler by, e.g.:

```
g++ -v
```

To select a specific compiler, please add the following flags to your `cmake` line:

```
-DCMAKE_C_COMPILER=<C-compiler> -DCMAKE_CXX_COMPILER=<C++-compiler>
```

Compiler-specific options

NEST has reasonable default compiler options for the most common compilers.

When compiling with the Portland compiler, use the `-Kieee` flag to ensure that computations obey the IEEE754 standard for floating point numerics.

If you have not done so, you can *download NEST here*.

Please choose on which system you want to install NEST:

- *Debian/ Ubuntu installation*
- *Mac OS X*
- *High Performance Computer Systems*
- *NEST Live Media for Virtual Machines*

Note: NEST is not supported natively on Microsoft Windows. However, it is possible to use NEST in Windows using a *virtual machine*

Note: These installation instructions are for **NEST 2.12 and later** as well as the most recent version obtained from GitHub. Installation instructions for NEST 2.10 and earlier are provided here, but we strongly encourage all our users to stay up-to-date with most recent version of NEST. We cannot support out-dated versions.

CHAPTER 4

Getting Started

4.1 A quick overview of simulating neural networks

A NEST simulation tries to follow the logic of an electrophysiological experiment - the difference being it takes place inside the computer rather than in the physical world.

In NEST, the neural system is a collection of **nodes** and their *Connections*. Nodes correspond to `neurons` and `devices` and connections by `synapses`. Different neuron and synapse models can coexist in the same network.

To measure or observe the network activity, you can define so-called *Devices* that represent the various instruments (for measuring and stimulating) found in an experiment. These devices write their data either to memory or to file.

The network and its configuration are defined at the level of the simulation language interpreter (SLI) as well as the PyNEST level.

Check out our *PyNEST tutorial*, which will explain how to build your first neural network simulation in NEST.

See Also

- List of Models in NEST
- Create your own model
- *Examples of Network Models*

4.2 How do I use NEST?

As the experimenter, you need a clear idea of *what* you want to learn from the experiment. In the context of a network *Simulation*, this means that you have to know *which input* you want to give to your network and *which output* you're interested in.

You can use NEST either with Python (PyNEST) or as a stand alone application (`nest`). PyNEST provides a *set of commands* to the Python interpreter which give you access to NEST's simulation kernel. With these commands, you describe and run your network simulation.

4.2.1 A basic network setup in PyNEST

You can use PyNEST interactively from the Python prompt or from within ipython. This is very helpful when you are exploring PyNEST, trying to learn a new functionality or debugging a routine. Once out of the exploratory mode, you will find it saves a lot of time to write your simulations in text files. These can in turn be run from the command line or from the Python or ipython prompt.

Fundamentally, you can build a basic network with the following functions:

```
# Create the models you want to simulate
neuron = nest.Create("model_name")

# Create the device to stimulate or measure the simulation
device = nest.Create("device_name")

# Modify properties of the neuron and device
nest.SetStatus(neuron, {"key" : value})
nest.SetStatus(device, {"key" : value})

# Tell NEST how they are connected to each other (synapse properties can be
# added here)
nest.Connect(device, neuron, syn_spec={"key": [value1, value2]})

# Simulate network providing a specific timeframe.
nest.Simulate(time_in_ms)
```

NEST is extensible and new models for neurons, synapses, and devices can be added. You can find out how to create your own model using NESTML and c++.

4.2.2 Connections

Connections between nodes (neurons, devices or synapses) define possible channels for interactions between them. A connection between two nodes is established, using the command `Connect`.

Each connection has two basic parameters, *weight* and *delay*. The weight determines the strength of the connection, the delay determines how long an event needs to travel from the sending to the receiving node. The delay must be a positive number greater or equal to the simulation stepsize and is given in ms.

4.2.3 Devices

Devices are network nodes which provide input to the network or record its output. They encapsulate the stimulation and measurement process. If you want to extract certain information from a simulation, you need a device which is able to deliver this information. Likewise, if you want to send specific input to the network, you need a device which delivers this input.

Devices have a built-in timer which controls the period they are active. Outside this interval, a device will remain silent. The timer can be configured using the command `SetStatus`.

4.2.4 Simulation

NEST simulations are time driven. The simulation time proceeds in discrete steps of size `dt`, set using the property `resolution` of the root node. In each time slice, all nodes in the system are updated and pending events are delivered.

The simulation is run by calling the command `Simulate(t)`, where `t` is the simulation time in milliseconds. See below for list of physical units in NEST.

4.3 Physical units in NEST

- time - ms
- voltage - mV
- capacitance - pF
- current - pA
- conductance - nS
- Spike rates (e.g. poisson_generator) - spikes/s
- modulation frequencies (e.g. ac_generator) - Hz

4.4 Next Steps

- *Download* and *Install NEST*
- Follow the *PyNEST tutorial* and simulate a neural network

CHAPTER 5

Tutorials

5.1 Part 1: Neurons and simple neural networks

5.1.1 Introduction

In this handout we cover the first steps in using PyNEST to simulate neuronal networks. When you have worked through this material, you will know how to:

- start PyNEST
- create neurons and stimulating/recording devices
- query and set their parameters
- connect them to each other or to devices
- simulate the network
- extract the data from recording devices

For more information on the usage of PyNEST, please see the other sections of this primer:

- *Part 2: Populations of neurons*
- *Part 3: Connecting networks with synapses*
- *Part 4: Topologically structured networks*

More advanced examples can be found at Example Networks, or have a look at at the source directory of your NEST installation in the subdirectory: `pynest/examples/`.

5.1.2 PyNEST - an interface to the NEST simulator

The NEural Simulation Tool (NEST: www.nest-initiative.org)[1] is designed for the simulation of large heterogeneous networks of point neurons. It is open source software released under the GPL licence. The simulator comes with

[1] Gewaltig MO. and Diesmann M. 2007. NEural Simulation Tool. 2(4):1430.

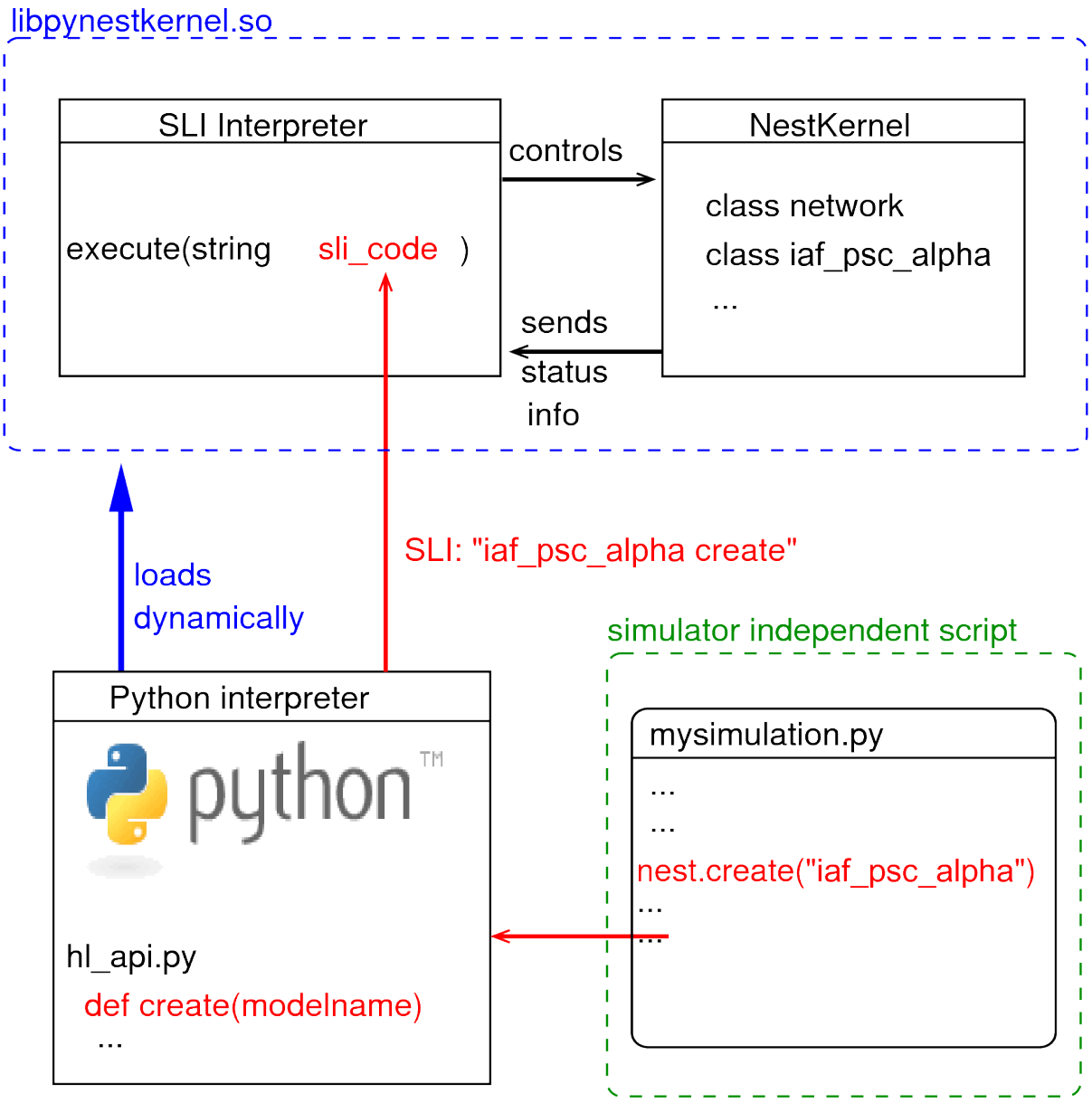


Fig. 5.1: Python Interface Figure. The Python interpreter imports NEST as a module and dynamically loads the NEST simulator kernel (`pynestkernel.so`). The core functionality is defined in `hl_api.py`. A simulation script of the user (`mysimulation.py`) uses functions defined in this high-level API. These functions generate code in SLI (Simulation Language Interpreter), the native language of the interpreter of NEST. This interpreter, in turn, controls the NEST simulation kernel.

an interface to Python[2]. Fig. 5.1 illustrates the interaction between the user's simulation script (`mysimulation.py`) and the NEST simulator. Eppler et al.[3] contains a technically detailed description of the implementation of this interface and parts of this text are based on this reference. The simulation kernel is written in C++ to obtain the highest possible performance for the simulation.

You can use PyNEST interactively from the Python prompt or from within ipython. This is very helpful when you are exploring PyNEST, trying to learn a new functionality or debugging a routine. Once out of the exploratory mode, you will find it saves a lot of time to write your simulations in text files. These can in turn be run from the command line or from the Python or ipython prompt.

Whether working interactively, semi-interactively, or purely executing scripts, the first thing that needs to happen is importing NEST's functionality into the Python interpreter.

```
import nest
```

It should be noted, however, that certain external packages must be imported *before* importing nest. These include scikit-learn and SciPy.

```
from sklearn.svm import LinearSVC
from scipy.special import erf

import nest
```

As with every other module for Python, the available functions can be prompted for.

```
dir(nest)
```

One such command is `nest.Models()` or in ipython `nest.Models?`, which will return a list of all the available models you can use. If you want to obtain more information about a particular command, you may use Python's standard help system.

This will return the help text (docstring) explaining the use of this particular function. There is a help system within NEST as well. You can open the help pages in a browser using `nest.helpdesk()` and you can get the help page for a particular object using `nest.help(object)`.

5.1.3 Creating Nodes

A neural network in NEST consists of two basic element types: nodes and connections. Nodes are either neurons, devices or sub-networks. Devices are used to stimulate neurons or to record from them. Nodes can be arranged in sub-networks to build hierarchical networks such as layers, columns, and areas - we will get to this later in the course. For now we will work in the default sub-network which is present when we start NEST, known as the `root node`.

To begin with, the root sub-network is empty. New nodes are created with the command `Create`, which takes as arguments the model name of the desired node type, and optionally the number of nodes to be created and the initialising parameters. The function returns a list of handles to the new nodes, which you can assign to a variable for later use. These handles are integer numbers, called *ids*. Many PyNEST functions expect or return a list of ids (see *command overview*). Thus, it is easy to apply functions to large sets of nodes with a single function call.

After having imported NEST and also the Pylab interface to Matplotlib[4], which we will use to display the results, we can start reating nodes. As a first example, we will create a neuron of type `iaf_psc_alpha`. This neuron is an integrate-and-fire neuron with alpha-shaped postsynaptic currents. The function returns a list of the ids of all the created neurons, in this case only one, which we store in a variable called `neuron`.

[2] Python Software Foundation. The Python programming language, 2008. http://www.python.org.
[3] Eppler JM et al. 2009 PyNEST: A convenient interface to the NEST simulator. 2:12. 10.3389/neuro.11.012.2008.
[4] Hunter JD. 2007 Matplotlib: A 2d graphics environment. 9(3):90–95.

```
import pylab
import nest
neuron = nest.Create("iaf_psc_alpha")
```

We can now use the id to access the properties of this neuron. Properties of nodes in NEST are generally accessed via Python dictionaries of key-value pairs of the form `{key: value}`. In order to see which properties a neuron has, you may ask it for its status.

```
nest.GetStatus(neuron)
```

This will print out the corresponding dictionary in the Python console. Many of these properties are not relevant for the dynamics of the neuron. To find out what the interesting properties are, look at the documentation of the model through the helpdesk. If you already know which properties you are interested in, you can specify a key, or a list of keys, as an optional argument to `GetStatus`:

```
nest.GetStatus(neuron, "I_e")
nest.GetStatus(neuron, ["V_reset", "V_th"])
```

In the first case we query the value of the constant background current `I_e`; the result is given as a tuple with one element. In the second case, we query the values of the reset potential and threshold of the neuron, and receive the result as a nested tuple. If `GetStatus` is called for a list of nodes, the dimension of the outer tuple is the length of the node list, and the dimension of the inner tuples is the number of keys specified.

To modify the properties in the dictionary, we use `SetStatus`. In the following example, the background current is set to 376.0pA, a value causing the neuron to spike periodically.

```
nest.SetStatus(neuron, {"I_e": 376.0})
```

Note that we can set several properties at the same time by giving multiple comma separated key:value pairs in the dictionary. Also be aware that NEST is type sensitive - if a particular property is of type `double`, then you do need to explicitly write the decimal point:

```
nest.SetStatus(neuron, {"I_e": 376})
```

will result in an error. This conveniently protects us from making integer division errors, which are hard to catch.

Next we create a `multimeter`, a *device* we can use to record the membrane voltage of a neuron over time. We set its property `withtime` such that it will also record the points in time at which it samples the membrane voltage. The property `record_from` expects a list of the names of the variables we would like to record. The variables exposed to the multimeter vary from model to model. For a specific model, you can check the names of the exposed variables by looking at the neuron's property `recordables`.

```
multimeter = nest.Create("multimeter")
nest.SetStatus(multimeter, {"withtime":True, "record_from":["V_m"]})
```

We now create a `spikedetector`, another device that records the spiking events produced by a neuron. We use the optional keyword argument `params` to set its properties. This is an alternative to using `SetStatus`. The property `withgid` indicates whether the spike detector is to record the source id from which it received the event (i.e. the id of our neuron).

```
spikedetector = nest.Create("spike_detector",
                params={"withgid": True, "withtime": True})
```

A short note on naming: here we have called the neuron `neuron`, the multimeter `multimeter` and so on. Of course, you can assign your created nodes to any variable names you like, but the script is easier to read if you choose names that reflect the concepts in your simulation.

5.1.4 Connecting nodes with default connections

Now we know how to create individual nodes, we can start connecting them to form a small network.

```
nest.Connect(multimeter, neuron)
nest.Connect(neuron, spikedetector)
```

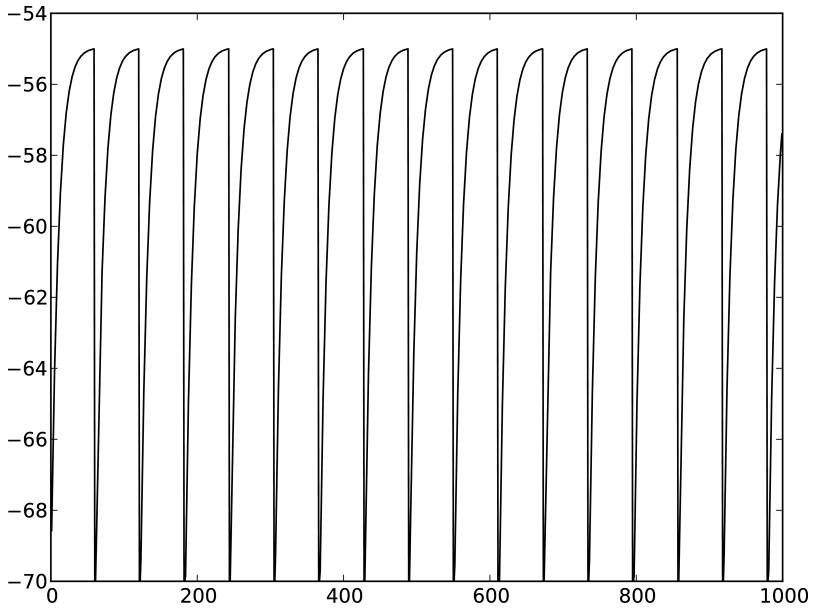


Fig. 5.2: Membrane potential of integrate-and-fire neuron with constant input current.

The order in which the arguments to `Connect` are specified reflects the flow of events: if the neuron spikes, it sends an event to the spike detector. Conversely, the multimeter periodically sends requests to the neuron to ask for its membrane potential at that point in time. This can be regarded as a perfect electrode stuck into the neuron.

Now we have connected the network, we can start the simulation. We have to inform the simulation kernel how long the simulation is to run. Here we choose 1000ms.

```
nest.Simulate(1000.0)
```

Congratulations, you have just simulated your first network in NEST!

5.1.5 Extracting and plotting data from devices

After the simulation has finished, we can obtain the data recorded by the multimeter.

```
dmm = nest.GetStatus(multimeter)[0]
Vms = dmm["events"]["V_m"]
ts = dmm["events"]["times"]
```

In the first line, we obtain the list of status dictionaries for all queried nodes. Here, the variable `multimeter` is the id of only one node, so the returned list just contains one dictionary. We extract the first element of this list by indexing it (hence the `[0]` at the end). This type of operation occurs quite frequently when using PyNEST, as most functions are designed to take in and return lists, rather than individual values. This is to make operations on groups of items (the usual case when setting up neuronal network simulations) more convenient.

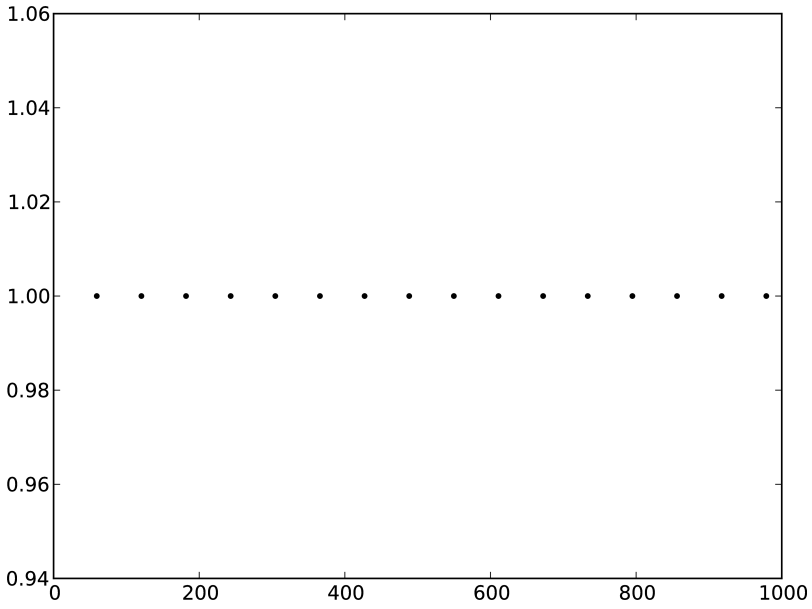


Fig. 5.3: Spikes of the neuron.

This dictionary contains an entry named `events` which holds the recorded data. It is itself a dictionary with the entries `V_m` and `times`, which we store separately in `Vms` and `ts`, in the second and third line, respectively. If you are having trouble imagining dictionaries of dictionaries and what you are extracting from where, try first just printing `dmm` to the screen to give you a better understanding of its structure, and then in the next step extract the dictionary `events`, and so on.

Now we are ready to display the data in a figure. To this end, we make use of `pylab`.

```
import pylab
pylab.figure(1)
pylab.plot(ts, Vms)
```

The second line opens a figure (with the number 1), and the third line actually produces the plot. You can't see it yet because we have not used `pylab.show()`. Before we do that, we proceed analogously to obtain and display the spikes from the spike detector.

```
dSD = nest.GetStatus(spikedetector,keys="events")[0]
evs = dSD["senders"]
ts = dSD["times"]
pylab.figure(2)
pylab.plot(ts, evs, ".")
pylab.show()
```

Here we extract the events more concisely by using the optional keyword argument `keys` to `GetStatus`. This extracts the dictionary element with the key `events` rather than the whole status dictionary. The output should look like Fig. 5.2 and Fig. 5.3. If you want to execute this as a script, just paste all lines into a text file named, say, `one-neuron.py` . You can then run it from the command line by prefixing the file name with `python`, or from the Python or ipython prompt, by prefixing it with `run`.

It is possible to collect information of multiple neurons on a single multimeter. This does complicate retrieving the information: the data for each of the n neurons will be stored and returned in an interleaved fashion. Luckily Python provides us with a handy array operation to split the data easily: array slicing with a step (sometimes called stride). To explain this you have to adapt the model created in the previous part. Save your code under a new name, in the next section you will also work on this code. Create an extra neuron with the background current given a different value:

```
neuron2 = nest.Create("iaf_psc_alpha")
nest.SetStatus(neuron2 , {"I_e": 370.0})
```

now connect this newly created neuron to the multimeter:

```
nest.Connect(multimeter, neuron2)
```

Run the simulation and plot the results, they will look incorrect. To fix this you must plot the two neuron traces separately. Replace the code that extracts the events from the `multimeter` with the following lines.

```
pylab.figure(2)
Vms1 = dmm["events"]["V_m"][::2] # start at index 0: till the end: each second entry
ts1 = dmm["events"]["times"][::2]
pylab.plot(ts1, Vms1)
Vms2 = dmm["events"]["V_m"][1::2] # start at index 1: till the end: each second entry
ts2 = dmm["events"]["times"][1::2]
pylab.plot(ts2, Vms2)
```

Additional information can be found at http://docs.scipy.org/doc/numpy-1.10.0/reference/arrays.indexing.html.

5.1.6 Connecting nodes with specific connections

A commonly used model of neural activity is the Poisson process. We now adapt the previous example so that the neuron receives 2 Poisson spike trains, one excitatory and the other inhibitory. Hence, we need a new device, the `poisson_generator`. After creating the neurons, we create these two generators and set their rates to 80000Hz and 15000Hz, respectively.

```
noise_ex = nest.Create("poisson_generator")
noise_in = nest.Create("poisson_generator")
nest.SetStatus(noise_ex, {"rate": 80000.0})
nest.SetStatus(noise_in, {"rate": 15000.0})
```

Additionally, the constant input current should be set to 0:

```
nest.SetStatus(neuron, {"I_e": 0.0})
```

Each event of the excitatory generator should produce a postsynaptic current of 1.2pA amplitude, an inhibitory event of -2.0pA. The synaptic weights can be defined in a dictionary, which is passed to the `Connect` function using the keyword `syn_spec` (synapse specifications). In general all parameters determining the synapse can be specified in the synapse dictionary, such as `"weight"`, `"delay"`, the synaptic model (`"model"`) and parameters specific to the synaptic model.

```
syn_dict_ex = {"weight": 1.2}
syn_dict_in = {"weight": -2.0}
nest.Connect([noise_ex], neuron, syn_spec=syn_dict_ex)
nest.Connect([noise_in], neuron, syn_spec=syn_dict_in)
```

The rest of the code remains as before. You should see a membrane potential as in Fig. 5.4 and Fig. 5.5.

In the next part of the introduction (*Part 2: Populations of neurons*) we will look at more methods for connecting many neurons at once.

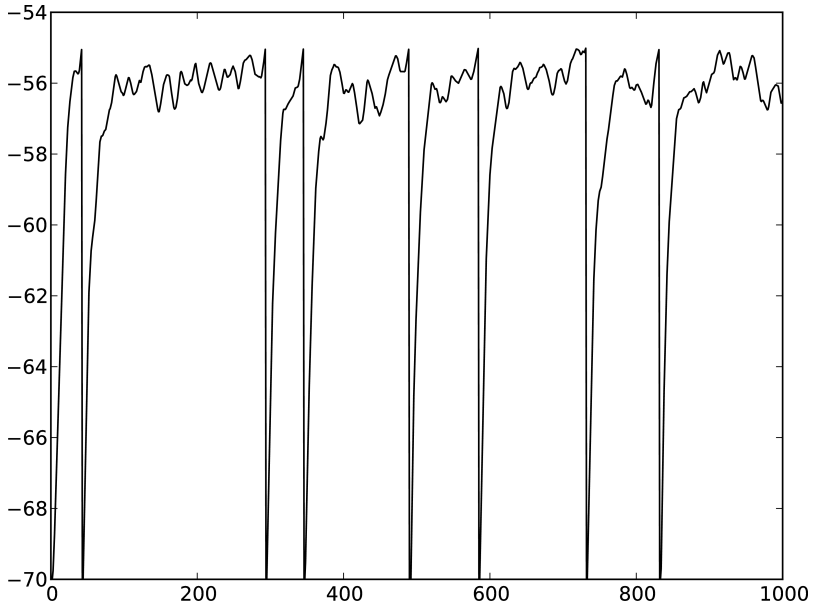


Fig. 5.4: Membrane potential of integrate-and-fire neuron with Poisson noise as input.

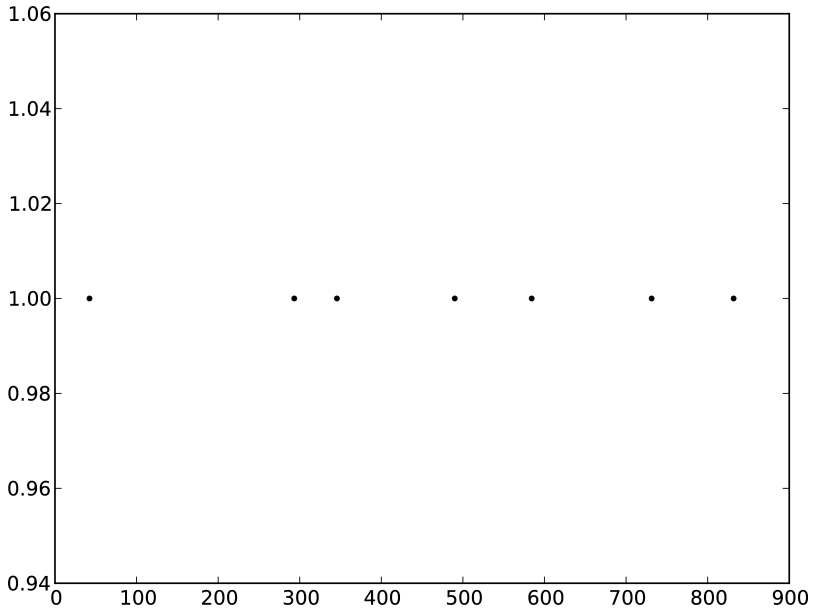


Fig. 5.5: Spikes of the neuron with noise.

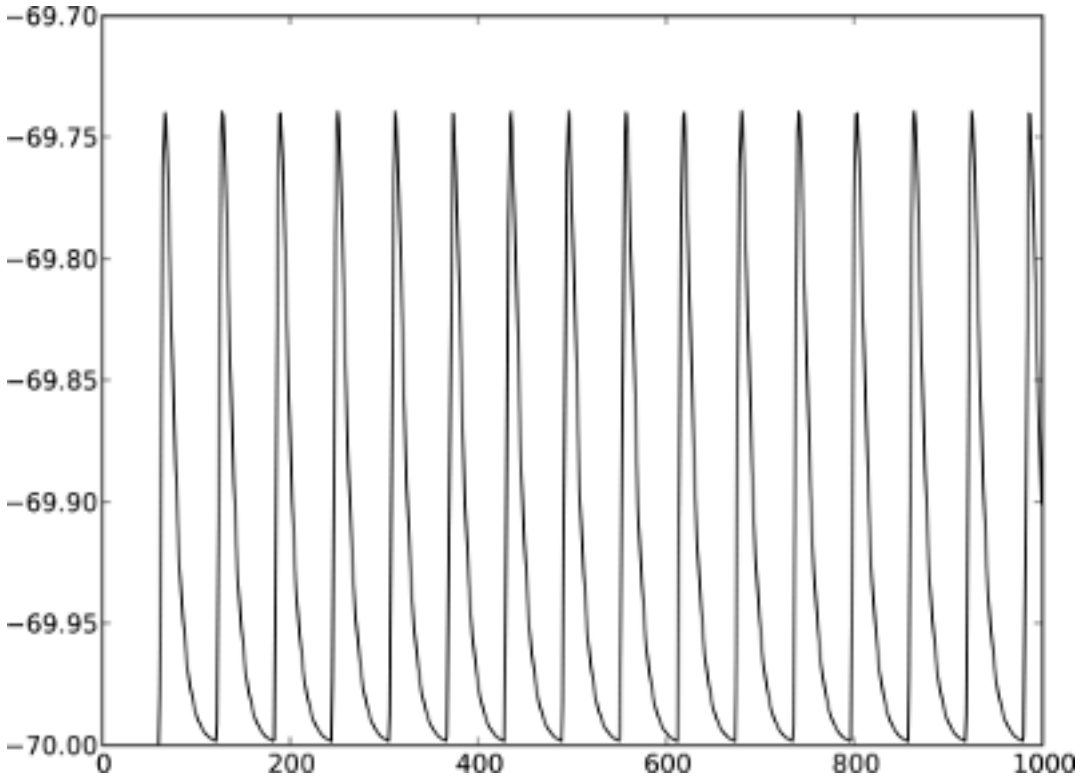


Fig. 5.6: Postsynaptic potentials in neuron2 evoked by the spikes of neuron1

5.1.7 Two connected neurons

There is no additional magic involved in connecting neurons. To demonstrate this, we start from our original example of one neuron with a constant input current, and add a second neuron.

```
import pylab
import nest
neuron1 = nest.Create("iaf_psc_alpha")
nest.SetStatus(neuron1, {"I_e": 376.0})
neuron2 = nest.Create("iaf_psc_alpha")
multimeter = nest.Create("multimeter")
nest.SetStatus(multimeter, {"withtime":True, "record_from":["V_m"]}
```

We now connect `neuron1` to `neuron2`, and record the membrane potential from `neuron2` so we can observe the postsynaptic potentials caused by the spikes of `neuron1`.

```
nest.Connect(neuron1, neuron2, syn_spec = {"weight":20.0})
nest.Connect(multimeter, neuron2)
```

Here the default delay of 1ms was used. If the delay is specified in addition to the weight, the following shortcut is available:

```
nest.Connect(neuron1, neuron2, syn_spec={"weight":20, "delay":1.0})
```

If you simulate the network and plot the membrane potential as before, you should then see the postsynaptic potentials of `neuron2` evoked by the spikes of `neuron1` as in Fig. 5.6.

5.1.8 Command overview

These are the functions we introduced for the examples in this handout; the following sections of this introduction will add more.

Getting information about NEST

See the *Getting Help Section*

Nodes

- **`Create(model, n=1, params=None)`** Create `n` instances of type `model` in the current sub-network. Parameters for the new nodes can be given as `params` (a single dictionary, or a list of dictionaries with size `n`). If omitted, the `model`'s defaults are used.
- **`GetStatus(nodes, keys=None)`** Return a list of parameter dictionaries for the given list of `nodes`. If `keys` is given, a list of values is returned instead. `keys` may also be a list, in which case the returned list contains lists of values.
- **`SetStatus(nodes, params, val=None)`** Set the parameters of the given `nodes` to `params`, which may be a single dictionary, or a list of dictionaries of the same size as `nodes`. If `val` is given, `params` has to be the name of a property, which is set to `val` on the `nodes`. `val` can be a single value, or a list of the same size as `nodes`.

Connections

This is an abbreviated version of the documentation for the `Connect` function, please see NEST's online help for the full version and *Connection Management* for an introduction and worked examples.

- *Connect(pre, post, conn_spec=None, syn_spec=None, model=None)*‘ Connect pre neurons to post neurons.Neurons in pre and post are connected using the specified connectivity (`"one_to_one"` by default) and synapse type (`"static_synapse"` by default). Details depend on the connectivity rule. Note: Connect does not iterate over subnets, it only connects explicitly specified nodes. `pre` - presynaptic neurons, given as list of GIDs `post` - presynaptic neurons, given as list of GIDs `conn_spec` - name or dictionary specifying connectivity rule, see below `syn_spec` - name or dictionary specifying synapses, see below

Connectivity

Connectivity is either specified as a string containing the name of a connectivity rule (default: `"one_to_one"`) or as a dictionary specifying the rule and rule-specific parameters (e.g. `"indegree"`), which must be given. In addition switches allowing self-connections (`"autapses"`, default: `True`) and multiple connections between a pair of neurons (`"multapses"`, default: `True`) can be contained in the dictionary.

Synapse

The synapse model and its properties can be inserted either as a string describing one synapse model (synapse models are listed in the synapsedict) or as a dictionary as described below. If no synapse model is specified the default model `"static_synapse"` will be used. Available keys in the synapse dictionary are `"model"`, `"weight"`, `"delay"`, `"receptor_type"` and parameters specific to the chosen synapse model. All parameters are optional and if not specified will use the default values determined by the current synapse model. `"model"` determines the synapse type, taken from pre-defined synapse types in NEST or manually specified synapses created via `CopyModel()`.

All other parameters can be scalars or distributions. In the case of scalar parameters, all keys take doubles except for `"receptor_type"` which has to be initialised with an integer. Distributed parameters are initialised with yet another dictionary specifying the distribution (`"distribution"`, such as `"normal"`) and distribution-specific paramters (such as `"mu"` and `"sigma"`).

Simulation control

- **`Simulate(t)`** Simulate the network for `t` milliseconds.

5.1.9 References

5.2 Part 2: Populations of neurons

5.2.1 Introduction

In this handout we look at creating and parameterising batches of `neurons`, and connecting them. When you have worked through this material, you will know how to:

- create populations of neurons with specific parameters
- set model parameters before creation
- define models with customised parameters
- randomise parameters after creation
- make random connections between populations
- set up devices to start, stop and save data to file
- reset simulations

For more information on the usage of PyNEST, please see the other sections of this primer:

- *Part 1: Neurons and simple neural networks*
- *Part 3: Connecting networks with synapses*
- *Part 4: Topologically structured networks*

More advanced examples can be found at Example Networks, or have a look at at the source directory of your NEST installation in the subdirectory: `pynest/examples/`.

5.2.2 Creating parameterised populations of nodes

In the previous handout, we introduced the function `Create(model, n=1, params=None)`. Its mandatory argument is the model name, which determines what type the nodes to be created should be. Its two optional arguments are `n`, which gives the number of nodes to be created (default: 1) and `params`, which is a dictionary giving the parameters with which the nodes should be initialised. So the most basic way of creating a batch of identically parameterised neurons is to exploit the optional arguments of `Create()`:

```
ndict = {"I_e": 200.0, "tau_m": 20.0}
neuronpop = nest.Create("iaf_psc_alpha", 100, params=ndict)
```

The variable `neuronpop` is a list of all the ids of the created neurons.

Parameterising the neurons at creation is more efficient than using `SetStatus()` after creation, so try to do this wherever possible.

We can also set the parameters of a neuron model *before* creation, which allows us to define a simulation more concisely in many cases. If many individual batches of neurons are to be produced, it is more convenient to set the defaults of the model, so that all neurons created from that model will automatically have the same parameters. The defaults of a model can be queried with `GetDefaults(model)`, and set with `SetDefaults(model, params)`, where `params` is a dictionary containing the desired parameter/value pairings. For example:

```
ndict = {"I_e": 200.0, "tau_m": 20.0}
nest.SetDefaults("iaf_psc_alpha", ndict)
neuronpop1 = nest.Create("iaf_psc_alpha", 100)
neuronpop2 = nest.Create("iaf_psc_alpha", 100)
neuronpop3 = nest.Create("iaf_psc_alpha", 100)
```

The three populations are now identically parameterised with the usual model default values for all parameters except `I_e` and `tau_m`, which have the values specified in the dictionary `ndict`.

If batches of neurons should be of the same model but using different parameters, it is handy to use `CopyModel(existing, new, params=None)` to make a customised version of a neuron model with its own default parameters. This function is an effective tool to help you write clearer simulation scripts, as you can use the name of the model to indicate what role it plays in the simulation. Set up your customised model in two steps using `SetDefaults()`:

```
edict = {"I_e": 200.0, "tau_m": 20.0}
nest.CopyModel("iaf_psc_alpha", "exc_iaf_psc_alpha")
nest.SetDefaults("exc_iaf_psc_alpha", edict)
```

or in one step:

```
idict = {"I_e": 300.0}
nest.CopyModel("iaf_psc_alpha", "inh_iaf_psc_alpha", params=idict)
```

Either way, the newly defined models can now be used to generate neuron populations and will also be returned by the function `Models()`.

```
epop1 = nest.Create("exc_iaf_psc_alpha", 100)
epop2 = nest.Create("exc_iaf_psc_alpha", 100)
ipop1 = nest.Create("inh_iaf_psc_alpha", 30)
ipop2 = nest.Create("inh_iaf_psc_alpha", 30)
```

It is also possible to create populations with an inhomogeneous set of parameters. You would typically create the complete set of parameters, depending on experimental constraints, and then create all the neurons in one go. To do this supply a list of dictionaries of the same length as the number of neurons (or synapses) created:

```
parameter_list = [{"I_e": 200.0, "tau_m": 20.0}, {"I_e": 150.0, "tau_m": 30.0}]
epop3 = nest.Create("exc_iaf_psc_alpha", 2, parameter_list)
```

5.2.3 Setting parameters for populations of neurons

It is not always possible to set all parameters for a neuron model at or before creation. A classic example of this is when some parameter should be drawn from a random distribution. Of course, it is always possible to make a loop over the population and set the status of each one:

```
Vth=-55.
Vrest=-70.
for neuron in epop1:
    nest.SetStatus([neuron], {"V_m": Vrest+(Vth-Vrest)*numpy.random.rand()})
```

However, `SetStatus()` expects a list of nodes and can set the parameters for each of them, which is more efficient, and thus to be preferred. One way to do it is to give a list of dictionaries which is the same length as the number of nodes to be parameterised, for example using a list comprehension:

```
dVms =  [{"V_m": Vrest+(Vth-Vrest)\*numpy.random.rand()} for x in epop1]
nest.SetStatus(epop1, dVms)
```

If we only need to randomise one parameter then there is a more concise way by passing in the name of the parameter and a list of its desired values. Once again, the list must be the same size as the number of nodes to be parameterised:

```
Vms = Vrest+(Vth-Vrest)\*numpy.random.rand(len(epop1))
nest.SetStatus(epop1, "V_m", Vms)
```

Note that we are being rather lax with random numbers here. Really we have to take more care with them, especially if we are using multiple threads or distributing over multiple machines. We will worry about this later.

5.2.4 Generating populations of neurons with deterministic connections

In the previous handout two neurons were connected using synapse specifications. In this section we extend this example to two populations of ten neurons each.

```
import pylab
import nest
pop1 = nest.Create("iaf_psc_alpha", 10)
nest.SetStatus(pop1, {"I_e": 376.0})
pop2 = nest.Create("iaf_psc_alpha", 10)
multimeter = nest.Create("multimeter", 10)
nest.SetStatus(multimeter, {"withtime":True, "record_from":["V_m"]})
```

If no connectivity pattern is specified, the populations are connected via the default rule, namely `all_to_all`. Each neuron of `pop1` is connected to every neuron in `pop2`, resulting in 10^2 connections.

```
nest.Connect(pop1, pop2, syn_spec={"weight":20.0})
```

Alternatively, the neurons can be connected with the `one_to_one`. This means that the first neuron in `pop1` is connected to the first neuron in `pop2`, the second to the second, etc., creating ten connections in total.

```
nest.Connect(pop1, pop2, "one_to_one", syn_spec={"weight":20.0, "delay":1.0})
```

Finally, the multimeters are connected using the default rule

```
nest.Connect(multimeter, pop2)
```

Here we have just used very simple connection schemes. Connectivity patterns requiring the specification of further parameters, such as in-degree or connection probabilities, must be defined in a dictionary containing the key `rule` and the key for parameters associated to the rule. Please see **:doc:'Connection management <../../guides/connection_management>'_** for an illustrated guide to the usage of `Connect`.

5.2.5 Connecting populations with random connections

In the previous handout we looked at the connectivity patterns `one_to_one` and `all_to_all`. However, we often want to look at networks with a sparser connectivity than all-to-all. Here we introduce four connectivity patterns which generate random connections between two populations of neurons.

The connection rule `fixed_indegree` allows us to create `n` random connections for each neuron in the target population `post` to a randomly selected neuron from the source population `pre`. The variables `weight` and `delay` can be left unspecified, in which case the default weight and delay are used. Alternatively we can set them in the `syn_spec` , so each created connection has the same weight and delay. Here is an example:

```
d = 1.0
Je = 2.0
Ke = 20
Ji = -4.0
Ki = 12
conn_dict_ex = {"rule": "fixed_indegree", "indegree": Ke}
conn_dict_in = {"rule": "fixed_indegree", "indegree": Ki}
syn_dict_ex = {"delay": d, "weight": Je}
syn_dict_in = {"delay": d, "weight": Ji}
nest.Connect(epop1, ipop1, conn_dict_ex, syn_dict_ex)
nest.Connect(ipop1, epop1, conn_dict_in, syn_dict_in)
```

Now each neuron in the target population `ipop1` has `Ke` incoming random connections chosen from the source population `epop1` with weight `Je` and delay `d`, and each neuron in the target population `epop1` has `Ki` incoming random connections chosen from the source population `ipop1` with weight `Ji` and delay `d`.

The connectivity rule `fixed_outdegree` works in analogous fashion, with `n` connections (keyword `outdegree`) being randomly selected from the target population `post` for each neuron in the source population `pre`. For reasons of efficiency, particularly when simulating in a distributed fashion, it is better to use `fixed_indegree` if possible.

Another connectivity pattern available is `fixed_total_number`. Here `n` connections (keyword `N`) are created by randomly drawing source neurons from the populations `pre` and target neurons from the population `post`.

When choosing the connectivity rule `pairwise_bernoulli` connections are generated by iterating through all possible source-target pairs and creating each connection with the probability `p` (keyword `p`).

In addition to the rule specific parameters `indegree`, `outdegree`, `N` and `p`, the `conn_spec` can contain the keywords `autapses` and `multapses` (set to `False` or `True`) allowing or forbidding self-connections and multiple connections between two neurons, respectively.

Note that for all connectivity rules, it is perfectly legitimate to have the same population simultaneously in the role of `pre` and `post`.

For more information on connecting neurons, please read the documentation of the `Connect` function and consult the guide at *Connection management*.

5.2.6 Specifying the behaviour of devices

All devices implement a basic timing capacity; the parameter `start` (default 0) determines the beginning of the device's activity and the parameter `stop` (default:) its end. These values are taken relative to the value of `origin` (default: 0). For example, the following example creates a `poisson_generator` which is only active between 100 and 150ms:

```
pg = nest.Create("poisson_generator")
nest.SetStatus(pg, {"start": 100.0, "stop": 150.0})
```

This functionality is useful for setting up experimental protocols with stimuli that start and stop at particular times.

So far we have accessed the data recorded by devices directly, by extracting the value of `events`. However, for larger or longer simulations, we may prefer to write the data to file for later analysis instead. All recording devices allow the specification of where data is stored over the parameters `to_memory` (default: `True`), `to_file` (default: `False`) and `to_screen` (default: `False`). The following code sets up a `multimeter` to record data to a named file:

```
recdict = {"to_memory" : False, "to_file" : True, "label" : "epop_mp"}
mm1 = nest.Create("multimeter", params=recdict)
```

If no name for the file is specified using the `label` parameter, NEST will generate its own using the name of the device, and its id. If the simulation is multithreaded or distributed, multiple files will be created, one for each process and/or thread. For more information on how to customise the behaviour and output format of recording devices, please read the documentation for `RecordingDevice`.

5.2.7 Resetting simulations

It often occurs that we need to reset a simulation. For example, if you are developing a script, then you may need to run it from the `ipython` console multiple times before you are happy with its behaviour. In this case, it is useful to use the function `ResetKernel()`. This gets rid of all nodes you have created, any customised models you created, and resets the internal clock to 0.

The other main use of resetting is when you need to run a simulation in a loop, for example to test different parameter settings. In this case there is typically no need to throw out the whole network and create and connect everything, it is enough to re-parameterise the network. A good strategy here is to create and connect your network outside the loop, and then carry out the parametrisation, simulation and data collection steps within the loop. Here it is often helpful to call the function `ResetNetwork()` within each loop iteration. It resets all nodes to their default configuration and wipes the data from recording devices.

5.2.8 Command overview

These are the new functions we introduced for the examples in this handout.

Getting and setting basic settings and parameters of NEST

- `GetKernelStatus(keys=none)`

  Obtain parameters of the simulation kernel. Returns:

  - Parameter dictionary if called without argument
  - Single parameter value if called with single parameter name
  - List of parameter values if called with list of parameter names
  - Set parameters for the simulation kernel.

Models

- `GetDefaults(model)`

  Return a dictionary with the default parameters of the given `model`, specified by a string.

- `SetDefaults(model, params)`

  Set the default parameters of the given `model` to the values specified in the `params` dictionary.

- `CopyModel(existing, new, params=None)`

  Create a `new` model by copying an `existing` one. Default parameters can be given as `params`, or else are taken from `existing`.

Simulation control

- `ResetKernel()`

  Reset the simulation kernel. This will destroy the network as well as all custom models created with `CopyModel()`. The parameters of built-in models are reset to their defaults. Calling this function is equivalent to restarting NEST.

- `ResetNetwork()`

  Reset all nodes and connections to the defaults of their respective model.

5.3 Part 3: Connecting networks with synapses

5.3.1 Introduction

In this handout we look at using synapse models to connect neurons. After you have worked through this material, you will know how to:

- set synapse model parameters before creation
- define synapse models with customised parameters
- use synapse models in connection routines
- query the synapse values after connection
- set synapse values during and after connection

For more information on the usage of PyNEST, please see the other sections of this primer:

- *Part 1: Neurons and simple neural networks*
- *Part 2: Populations of neurons*
- *Part 4: Topologically structured networks*

More advanced examples can be found at Example Networks, or have a look at at the source directory of your NEST installation in the subdirectory: `pynest/examples/`.

5.3.2 Parameterising synapse models

NEST provides a variety of different synapse models. You can see the available models by using the command `Models(synapses)`, which picks only the synapse models out of the list of all available models.

Synapse models can be parameterised analogously to neuron models. You can discover the default parameter settings using `GetDefaults(model)` and set them with `SetDefaults(model,params)`:

```
nest.SetDefaults("stdp_synapse",{"tau_plus": 15.0})
```

Any synapse generated from this model will then have all the standard parameters except for the `tau_plus`, which will have the value given above.

Moreover, we can also create customised variants of synapse models using `CopyModel()`, exactly as demonstrated for neuron models:

```
nest.CopyModel("stdp_synapse","layer1_stdp_synapse",{"Wmax": 90.0})
```

Now `layer1_stdp_synapse` will appear in the list returned by `Models()`, and can be used anywhere that a built-in model name can be used.

STDP synapses

For the majority of synapses, all of their parameters are accessible via `GetDefaults()` and `SetDefaults()`. Synapse models implementing spike-timing dependent plasticity are an exception to this, as their dynamics are driven by the post-synaptic spike train as well as the pre-synaptic one. As a consequence, the time constant of the depressing window of STDP is a parameter of the post-synaptic neuron. It can be set as follows:

```
nest.Create("iaf_psc_alpha", params={"tau_minus": 30.0})
```

or by using any of the other methods of parameterising neurons demonstrated in the first two parts of this introduction.

5.3.3 Connecting with synapse models

The synapse model as well as parameters associated with the synapse type can be set in the synapse specification dictionary accepted by the connection routine.

```
conn_dict = {"rule": "fixed_indegree", "indegree": K}
syn_dict = {"model": "stdp_synapse", "alpha": 1.0}
nest.Connect(epop1, epop2, conn_dict, syn_dict)
```

If no synapse model is given, connections are made using the model `static_synapse`.

5.3.4 Distributing synapse parameters

The synapse parameters are specified in the synapse dictionary which is passed to the `Connect`-function. If the parameter is set to a scalar all connections will be drawn using the same parameter. Parameters can be randomly distributed by assigning a dictionary to the parameter. The dictionary has to contain the key `distribution` setting the target distribution of the parameters (for example `normal`). Optionally, parameters associated with the distribution can be set (for example `mu`). Here we show an example where the parameters `alpha` and `weight` of the stdp synapse are uniformly distributed.

```
alpha_min = 0.1
alpha_max = 2.
w_min = 0.5
w_max = 5.

syn_dict = {"model": "stdp_synapse",
            "alpha": {"distribution": "uniform", "low": alpha_min, "high": alpha_max},
            "weight": {"distribution": "uniform", "low": w_min, "high": w_max},
            "delay": 1.0}
nest.Connect(epop1, neuron, "all_to_all", syn_dict)
```

Available distributions and associated parameters are described in Connection Management, the most common ones are:

Distributions	Keys
`normal`	`mu`, `sigma`
`lognormal`	`mu`, `sigma`
`uniform`	`low`, `high`
`uniform_int`	`low`, `high`
`binomial`	`n`, `p`
`exponential`	`lambda`
`gamma`	`order`, `scale`
`poisson`	`lambda`

5.3.5 Querying the synapses

The function `GetConnections(source=None, target=None, synapse_model=None)` returns a list of connection identifiers that match the given specifications. There are no mandatory arguments. If it is called without any arguments, it will return all the connections in the network. If `source` is specified, as a list of one or more nodes, the function will return all outgoing connections from that population:

```
nest.GetConnections(epop1)
```

Similarly, we can find the incoming connections of a particular target population by specifying `target` as a list of one or more nodes:

```
nest.GetConnections(target=epop2)
```

will return all connections beween all neurons in the network and neurons in `epop2`. Finally, the search can be restricted by specifying a given synapse model:

```
nest.GetConnections(synapse_model="stdp_synapse")
```

will return all the connections in the network which are of type `stdp_synapse`. The last two cases are slower than the first case, as a full search of all connections has to be performed.The arguments `source`, `target` and `synapse_model` can be used individually, as above, or in any conjunction:

```
nest.GetConnections(epop1, epop2, "stdp_synapse")
```

will return all the connections that the neurons in `epop1` have to neurons in `epop2` of type `stdp_synapse`. Note that all these querying commands will only return the local connections, i.e. those represented on that particular MPI process in a distributed simulation.

Once we have the array of connections, we can extract data from it using `GetStatus()`. In the simplest case, this returns a list of dictionaries, containing the parameters and variables for each connection found by `GetConnections`. However, usually we don't want all the information from a synapse, but some specific part of it. For example, if we want to check we have connected the network as intended, we might want to examine only the parameter `target` of each connection. We can extract just this information by using the optional `keys` argument of `GetStatus()`:

```
conns = nest.GetConnections(epop1, synapse_model="stdp_synapse")
targets = nest.GetStatus(conns, "target")
```

The variable `targets` is now list of all the `target` values of the connections found. If we are interested in more than one parameter, `keys` can be a list of keys as well:

```
conns = nest.GetConnections(epop1, synapse_model="stdp_synapse")
conn_vals = nest.GetStatus(conns, ["target","weight"])
```

The variable `conn_vals` is now a list of lists, containing the `target` and `weight` values for each connection found.

To get used to these methods of querying the synapses, it is recommended to try them out on a small network where all connections are known.

5.3.6 Coding style

As your simulations become more complex, it is very helpful to develop a clean coding style. This reduces the number of errors in the first place, but also assists you to debug your code and makes it easier for others to understand it (or even yourself after two weeks). Here are some pointers, some of which are common to programming in general and some of which are more NEST specific. Another source of useful advice is PEP-8, which, conveniently, can be automatically checked by many editors and IDEs.

Numbers and variables

Simulations typically have lots of numbers in them - we use them to set parameters for neuron models, to define the strengths of connections, the length of simulations and so on. Sometimes we want to use the same parameters in different scripts, or calculate some parameters based on the values of other parameters. It is not recommended to hardwire the numbers into your scripts, as this is error-prone: if you later decide to change the value of a given parameter, you have to go through all your code and check that you have changed every instance of it. This is particularly difficult to catch if the value is being used in different contexts, for example to set a weight in one place and to calculate the mean synaptic input in another.

A better approach is to set a variable to your parameter value, and then always use the variable name every time the value is needed. It is also hard to follow the code if the definitions of variables are spread throughout the script. If you have a parameters section in your script, and group the variable names according to function (e.g. neuronal parameters, synaptic parameters, stimulation parameters,...) then it is much easier to find and check them. Similarly, if you need to share parameters between simulation scripts, it is much less error-prone to define all the variable names in a separate parameters file, which the individual scripts can import. Thus a good rule of thumb is that numbers should only be visible in distinct parameter files or parameter sections, otherwise they should be represented by variables.

Repetitive code, copy-and-paste, functions

Often you need to repeat a section of code with minor modifications. For example, you have two `multimeter`s and you wish to extract the recorded variable from each of them and then calculate its maximum. The temptation is to write the code once, then copy-and-paste it to its new location and make any necessary modifications:

```
dma = nest.GetStatus(ma, keys="events")[0]
Vma = dma["Vm"]
amax = max(Vma)
dmb = nest.GetStatus(mb, keys="events")[0]
Vmb = dmb["Vm"]
bmax = max(Vmb)
print(amax-bmax)
```

There are two problems with this. First, it makes the main section of your code longer and harder to follow. Secondly, it is error-prone. A certain percentage of the time you will forget to make all the necessary modifications after the copy-and-paste, and this will introduce errors into your code that are hard to find, not only because they are semantically correct and so don't cause an obvious error, but also because your eye tends to drift over them:

```
dma = nest.GetStatus(multimeter1, keys="events")[0]
Vma = dma["Vm"]
amax = max(Vma)
dmb = nest.GetStatus(multimeter2, keys="events")[0]
Vmb = dmb["Vm"]
bmax = max(Vma)
print(amax-bmax)
```

The best way to avoid this is to define a function:

```
def getMaxMemPot(Vdevice):
    dm = nest.GetStatus(Vdevice, keys="events")[0]
    return max(dm["Vm"])
```

Such helper functions can usefully be stored in their own section, analogous to the parameters section. Now we can write down the functionality in a more concise and less error-prone fashion:

```
amax = getMaxMemPot(multimeter1)
bmax = getMaxMemPot(multimeter2)
print(amax-bmax)
```

If you find that this clutters your code, as an alternative you can write a `lambda` function as an argument for `map`, and enjoy the feeling of smugness that will pervade the rest of your day. A good policy is that if you find yourself about to copy-and-paste more than one line of code, consider taking the few extra seconds required to define a function. You will easily win this time back by spending less time looking for errors.

Subsequences and loops

When preparing a simulation or collecting or analysing data, it commonly happens that we need to perform the same operation on each node (or a subset of nodes) in a population. As neurons receive ids at the time of creation, it is possible to use your knowledge of these ids explictly:

```
Nrec = 50
neuronpop = nest.Create("iaf_psc_alpha", 200)
sd = nest.Create("spike_detector")
nest.Connect(range(1,N_rec+1),sd,"all_to_all")
```

However, this is *not at all recommended!*. This is because as you develop your simulation, you may well add additional nodes - this means that your initially correct range boundaries are now incorrect, and this is an error that is hard to catch. To get a subsequence of nodes, use a *slice* of the relevant population:

```
nest.Connect(neuronpop[:Nrec],spikedetector,"all_to_all")
```

An even worse thing is to use knowledge about neuron ids to set up loops:

```
for n in range(1,len(neuronpop)+1):
    nest.SetStatus([n], {"V_m": -67.0})
```

Not only is this error prone as in the previous example, the majority of PyNEST functions are expecting a list anyway. If you give them a list, you are reducing the complexity of your main script (good) and pushing the loop down to the faster C++ kernel, where it will run more quickly (also good). Therefore, instead you should write:

```
nest.SetStatus(neuronpop, {"V_m": -67.0})
```

See Part 2 for more examples on operations on multiple neurons, such as setting the status from a random distribution and connecting populations.

If you really really need to loop over neurons, just loop over the population itself (or a slice of it) rather than introducing ranges:

```
for n in neuronpop:
    my_weird_function(n)
```

Thus we can conclude: instead of range operations, use slices of and loops over the neuronal population itself. In the case of loops, check first whether you can avoid it entirely by passing the entire population into the function - you usually can.

5.3.7 Command overview

These are the new functions we introduced for the examples in this handout.

Querying Synapses

- `GetConnections(neuron, synapse_model="None"))`

  Return an array of connection identifiers.

  Parameters:

  - `source` - list of source GIDs
  - `target` - list of target GIDs
  - `synapse_model` - string with the synapse model

  If GetConnections is called without parameters, all connections in the network are returned. If a list of source neurons is given, only connections from these pre-synaptic neurons are returned. If a list of target neurons is given, only connections to these post-synaptic neurons are returned. If a synapse model is given, only connections with this synapse type are returned. Any combination of source, target and synapse_model parameters is permitted. Each connection id is a 5-tuple or, if available, a NumPy array with the following five entries: source-gid, target-gid, target-thread, synapse-id, port

  Note: Only connections with targets on the MPI process executing the command are returned.

5.4 Part 4: Topologically structured networks

5.4.1 Introduction

This handout covers the use of NEST's `topology` library to construct structured networks. When you have worked through this material you will be able to:

- Create populations of neurons with specific spatial locations
- Define connectivity profiles between populations
- Connect populations using profiles
- Visualise the connectivity

For more information on the usage of PyNEST, please see the other sections of this primer:

- *Part 1: Neurons and simple neural networks*
- *Part 2: Populations of neurons*

- *Part 3: Connecting networks with synapses*

More advanced examples can be found at Example Networks, or have a look at at the source directory of your NEST installation in the subdirectory: `pynest/examples/`.

5.4.2 Incorporating structure in networks of point neurons

If we use biologically detailed models of a neuron, then it's easy to understand and implement the concepts of topology, as we already have dendritic arbors, axons, etc. which are the physical prerequisites for connectivity within the nervous system. However, we can still get a level of specificity using networks of point neurons.

Structure, both in the topological and everyday sense, can be thought of as a set of rules governing the location of objects and the connections between them. Within networks of point neurons, we can distinguish between three types of specificity:

- Cell-type specificity – what sorts of cells are there?
- Location specificity – where are the cells?
- Projection specificity – which cells do they project to, and how?

In the previous handouts, we saw that we can create deterministic or randomly selected connections between networks using `Connect()`. If we want to create network models that incorporate the spatial location and spatial connectivity profiles, it is time to turn to the `topology` module. **NOTE:** Full documentation for usage of the topology module is present in NEST Topology Users Manual (NTUM)[1], which in the following pages is referenced as a full-source.

5.4.3 The nest.topology module

The `nest.topology` module allows us to create populations of nodes with a given spatial organisation, connection profiles which specify how neurons are to be connected, and provides a high-level connection routine. We can thus create structured networks by designing the connection profiles to give the desired specificity for cell-type, location and projection.

The generation of structured networks is carried out in three steps, each of which will be explained in the subsequent sections in more detail:

1. **Defining layers**, in which we assign the layout and types of the neurons within a layer of our network.
2. **Defining connection profiles**, where we generate the profiles that we wish our connections to have. Each connection dictionary specifies the properties for one class of connection, and contains parameters that allow us to tune the profile. These are related to the location-dependent likelihood of choosing a target (`mask` and `kernel`), and the cell-type specificity i.e. which types of cell in a layer can participate in the connection class (`sources` and `targets`).
3. **Connecting layers**, in which we apply the connection dictionaries between layers, equivalent to population-specificity. Note that multiple dictionaries can be applied between two layers, just as a layer can be connected to itself.
4. **Auxillary**, in which we visualise the results of the above steps either by `nest.PrintNetwork()` or visualization functions included in the topology module and query the connections for further analysis.

5.4.4 Defining layers

The code for defining a layer follows this template:

[1] Plesser HE and Enger H. NEST Topology User Manual, http://www.nest-simulator.org/wp-content/uploads/2015/04/Topology_UserManual.pdf

```
import nest.topology as topp
my_layer_dict = {...} # see below for options
my_layer = topp.CreateLayer(my_layer_dict)
```

where `my_layer_dict` will define the elements of the layer and their locations.

The choice of nodes to fill the `layer` is specified using the `elements` key. For the moment, we'll only concern ourselves with creating simple layers, where each element is from a homogeneous population. Then, the corresponding value for this dictionary entry should is the model type of the neuron, which can either be an existing model in the `NEST` collection, or one that we've previously defined using `CopyModel()`.

We next have to decide whether the nodes should be placed in a **grid-based** or **free** (off-grid) fashion, which is equivalent to asking ''can the elements of our network be regularly and evenly placed within a 2D network, or do we need to tell them where they should be located?".

1 - On-grid

we have to explicitly specify the size and spacing of the grid, by the number or rows *m* and columns *n* as well as the extent (layer size). The grid spacing i then determined from these, and *nxm* elements are arranged symmetrically. Note that we can also specify a center to the grid, else the default offset is the origin.

The following snippet produces `grid`:

```
layer_dict_ex = {"extent" : [2.,2.], # the size of the layer in mm
                 "rows" : 10, # the number of rows in this layer ...
                 "columns" : 10, # ... and the number of columns
                 "elements" : "iaf_psc_alpha"} # the element at each (x,y) coordinate␣
↪in the grid
```

2 - Off grid

we define only the elements, their positions and the extent. The number of elements created is equivalent to the length of the list of positions. This option allows much more flexibility in how we distribute neurons. Note that we should also specify the extent, if the positions fall outside of the default (extent size = [1,1] and origin as the center). See Section 2.2 in NUTM for more details.

The following snippet produces `free`:

```
import numpy as np
# grid with jitter
jit = 0.03
xs = np.arange(-0.5,.501,0.1)
poss = [[x,y] for y in xs for x in xs]
poss = [[p[0]+np.random.uniform(-jit,jit),p[1]+np.random.uniform(-jit,jit)] for p in␣
↪poss]
layer_dict_ex = {"positions": poss,
        "extent" : [1.1,1.1],
        "elements" : "iaf_psc_alpha"}
```

Note: The topology module does support 3D `layers`, but this is outside the scope of this handout.

An overview of all the parameters that can be used, as well as whether they are primarily used for grid-based or free layers, follows:

Parameter	Grid	Description	Possible values
elements	Both	Type of model to be included in the network	Any model listed in `nest.Models()` or user-defined model
extent	Both	Size of the layer in mm. Default is [1.,1.]	2D list
rows	On	Number of rows	int
columns	On	Number of columns	int
center	On	The center of the grid or free layer. Allows for grids to be structured independently of each other (see Fig. 2.3 in NTUM)	2D list
positions	Off	List of positions for each of the neurons to be created.	List of lists or tuples
edge_wrap	Both	Whether the layer should have a periodic boundary or not. Default: False	boolean

Advanced

Composite layers can also be created. This layer type extends the grid-based layer and allows us to define a number of neurons and other elements, such as `poisson_generators`, at each grid location. A full explanation is available in Section 2.5 of NTUM. The advantages in this approach is that, if we want to have a layer in which each element or subnetwork has the same composition of components, then it's very easy to define a layer which has these properties. For a simple example, let's consider a grid of elements, where each element comprises of 4 pyramidal cells, 1 interneuron, 1 poisson generator and 1 noise generator. The corresponding code is:

```
nest.CopyModel("iaf_psc_alpha","pyr")
nest.CopyModel("iaf_psc_alpha","inh", {"V_th": -52.})
comp_layer = topp.CreateLayer({"rows":5,"columns":5,
        "elements": ["pyr",4,"inh","poisson_generator","noise_generator"]})
```

5.4.5 Defining connection profiles

To define the types of connections that we want between populations of neurons, we specify a *connection dictionary*.

The only two mandatory parameters for any connection dictionary are `connection_type` and `mask`. All others allow us to tune our connectivity profiles by tuning the likelihood of a connection, the synapse type, the weight and/or delay associated with a connection, or the number of connections, as well as specifying restrictions on cell types that can participate in the connection class.

Chapter 3 in NTUM deals comprehensively with all the different possibilities, and it's suggested that you look there for learning about the different constraints, as well as reading through the different examples listed there. Here are some representative examples for setting up a connectivity profile, and the following table lists the parameters that can be used.

```
# Circular mask, gaussian kernel.
conn1 = {  "connection_type":"divergent",
           "mask": {"circular":{"radius":0.75}},
           "kernel": {"gaussian":{"p_center":1.,"sigma":0.2}},
           "allow_autapses":False
```

(continues on next page)

(continued from previous page)

```
            }

# Rectangular mask, constant kernel, non-centered anchor
conn2 = {   "connection_type":"divergent",
            "mask": {"rectangular":{"lower_left":[-0.5,-0.5],"upper_right":[0.5,0.5]},
                        "anchor": [0.5,0.5],
                    },
            "kernel": 0.75,
            "allow_autapses":False
            }

# Donut mask, linear kernel that decreases with distance
# Commented out line would allow connection to target the pyr neurons (useful for␣
↪composite layers)
conn3 = {   "connection_type": "divergent",
            "mask": {"doughnut":{"inner_radius":0.1,"outer_radius":0.95}},
            "kernel": {"linear": {"c":1.,"a":-0.8}},
            #"targets":"pyr"
        }

# Rectangular mask, fixed number of connections, gaussian weights, linear delays
conn4 = {   "connection_type":"divergent",
            "mask": {"rectangular":{"lower_left":[-0.5,-0.5],"upper_right":[0.5,0.5]}}
↪,
            "number_of_connections": 40,
            "weights": {"gaussian":{"p_center":J,"sigma":0.25}},
            "delays" : {"linear" :{"c":0.1,"a":0.2}},
            "allow_autapses":False
            }
```

Parameter	Description	Possible values
connection_type	Determines how nodes are selected when connections are made	convergent, divergent
mask	Spatially selected subset of neurons considered as (potential) targets	circular, rectangular, doughnut, grid
kernel	Function that determines the likelihood of a neuron being chosen as a target. Can be distance-dependent or -independent.	constant, uniform, linear, gaussian, exponential, gaussian2D
weights	Distribution of weight values of connections. Can be distance-dependent or -independent. **NB**: this value overrides any value currently used by synapse_model, and therefore unless defined will default to 1.!	constant, uniform, linear, gaussian, exponential
delays	Distribution of delay values for connections. Can be distance-dependent or -independent. **NB**: like weights, this value overrides any value currently used by synapse_model!	constant, uniform, linear, gaussian, exponential
synapse_model	Define the type of synapse model to be included.	any synapse model included in `nest.Models()`, or currently user-defined
sources	Defines the sources (presynaptic) neurons for this connection.	any neuron label that is currently user-defined
targets	Defines the target (postsynaptic) neurons for this connection.	any neuron label that is currently user-defined
number_of_connections	Fixes the number of connections that this neuron is to send, ensuring we have a fixed out-degree distribution.	int
allow_multapses	Whether we want to have multiple connections between the same source-target pair, or ensure unique connections.	boolean
allow_autapses	Whether we want to allow a neuron to connect to itself	boolean

5.4.6 Connecting layers

Connecting layers is the easiest step: having defined a source layer, a target layer and a connection dictionary, we simply use the function `topp.ConnectLayers()`:

```
ex_layer = topp.CreateLayer({"rows":5,"columns":5,"elements":"iaf_psc_alpha"})
in_layer = topp.CreateLayer({"rows":4,"columns":4,"elements":"iaf_psc_alpha"})
conn_dict_ex = {"connection_type":"divergent","mask":{"circular":{"radius":0.5}}}
# And now we connect E->I
topp.ConnectLayers(ex_layer,in_layer,conn_dict_ex)
```

Note that we can define several dictionaries, use the same dictionary multiple times and connect to the same layer:

```
# Extending the code from above ... we add a conndict for inhibitory neurons
conn_dict_in = {"connection_type":"divergent",
        "mask":{"circular":{"radius":0.75}},"weights":-4.}
# And finish connecting the rest of the layers:
```

(continues on next page)

(continued from previous page)

```
topp.ConnectLayers(ex_layer,ex_layer,conn_dict_ex) # Connect E->E
topp.ConnectLayers(in_layer,in_layer,conn_dict_in) # Connect I->I
topp.ConnectLayers(in_layer,ex_layer,conn_dict_in) # Connect I->E
```

5.4.7 Visualising and querying the network structure

There are two main methods that we can use for checking that our network was built correctly:

- `nest.PrintNetwork(depth=1)`

  which prints out all the neurons and subnetworks within the network in text form. This is a good manner in which to inspect the hierarchy of composite layers;

- *create plots using functions in ‘‘nest.topology‘* <http://www.nest-simulator.org/pynest-topology/>‘__

  There are three functions that can be combined:

  - `PlotLayer`
  - `PlotTargets`
  - `PlotKernel`

  which allow us to generate the plots used with NUTM and this handout. See Section 4.2 of NTUM for more details.

Other useful functions that may be of help, in addition to those already listed in NTUM Section 4.1, are:

Function	Description
nest.GetNodes(layer)	Returns GIDs of layer elements: either nodes or top-level subnets (for composite)
nest.GetLeaves(laye r)	Returns GIDs of leaves of a structure, which is always going to be neurons rather subnets
topp.GetPosition(gi ds)	Returns position of elements specified in input
nest.GetStatus(laye r,“topology”)	Returns the layer dictionary for a layer

5.4.8 References

5.5 Introduction to the MUSIC Interface

The MUSIC interface, a standard by the INCF, allows the transmission of data between applications at runtime. It can be used to couple NEST with other simulators, with applications for stimulus generation and data analysis and visualization and with custom applications that also use the MUSIC interface.

5.5.1 Setup of System

To use MUSIC with NEST, we first need to ensure MUSIC is installed on our system and NEST is configured properly.

Please install MUSIC using the instructions on the MUSIC website.

In the install of NEST, you need to add the following configuration option to your cmake.

```
cmake -Dwith-music=[ON </path/to/music>]
make
make install
```

5.5.2 A Quick Introduction to NEST and MUSIC

In this tutorial, we will show you how to use the MUSIC library together with NEST. We will cover how to use the library from PyNEST and from the SLI language interface. In addition, we'll introduce the use of MUSIC in a C++ application and how to connect such an application to a NEST simulation.

Our aim is to show practical examples of how to use MUSIC, and highlight common pitfalls that can trip the unwary. Also, we assume only a minimal knowledge of Python, C++ and (especially) SLI, so the examples will favour clarity and simplicity over elegance and idiomatic constructs.

Jump to

Part 1 - Connect 2 NEST simulations

While the focus here is on MUSIC, we need to know a few things about how NEST works in order to understand how MUSIC interacts with it.

Go straight to part 1 of tutorial - connect 2 simulations using PyNEST

The Basics of NEST

A NEST network consists of three types of elements: neurons, devices, and connections between them.

Neurons are the basic building blocks, and in NEST they are generally spiking point neuron models. Devices are supporting units that for instance generate inputs to neurons or record data from them. The Poisson spike generator, the spike detector recording device and the MUSIC input and output proxies are all devices. Neurons and devices are collectively called nodes, and are connected using connections.

Connections are unidirectional and carry events between nodes. Each neuron can get multiple input connections from any number of other neurons. Neuron connections typically carry spike events, but other kinds of events, such as voltages and currents, are also available for recording devices. Synapses are not independent nodes, but are part of the connection. Synapse models will typically modify the weight or timing of the spike sent on to the neuron. All connections have a synapse, by default the `static_synapse`.

Connections have a delay and a weight. All connections are implemented on the receiving side, and the interpretation of the parameters is ultimately up to the receiving node. In Fig. 5.7 A, neuron N_a has sent a spike to N_b at time t, over a connection with weight w_a and delay d. The spike is sent through the synapse, then buffered on the receiving side until $t + d$ (Fig. 5.7 B). At that time it's handed over to the neuron model receptor that converts the spike event to a current and applies it to the neuron model (Fig. 5.7 C).

5.5.3 Adding MUSIC connections

In NEST you use MUSIC with a pair of extra devices called *proxies* that create a MUSIC connection between them across simulations. The pair effectively works just like a regular connection within a single simulation. Each connection between MUSIC proxies is called a *port*, and connected by name in the MUSIC configuration file.

Each MUSIC port can carry multiple numbered *channels*. The channel is the smallest unit of transmission, in that you can distinguish data flowing in different channels, but not within a single channel. Depending on the application a

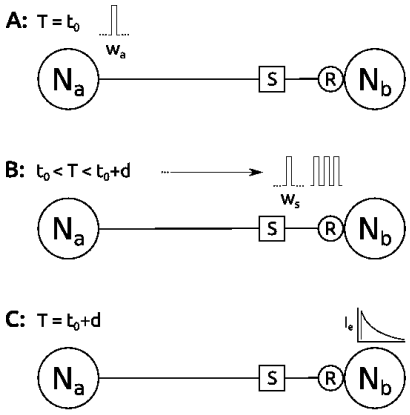


Fig. 5.7: A: Two connected neurons N_a and N_b, with a synapse S and a receptor R. A spike with weight W_a is generated at t_0. B: The spike traverses the synapse and is added to the queue in the receptor. C: The receptor processes the spike at time $t_0 + d$.

port may have one or many channels, and a single channel can carry the events from one single neuron model or the aggregate output of many neurons.

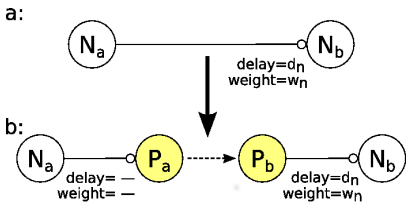


Fig. 5.8: A: Two connected neurons N_a and N_b, with delay d_n and weight w_n. B: We've added a MUSIC connection with an output proxy P_a on one end, and an input proxy P_b on the other.

In Fig. 5.8 A we see a regular NEST connection between two neurons N_a and N_b. The connection carries a weight w_n and a delay d_n. In Fig. 5.8 B we have inserted a pair of MUSIC proxies into the connection, with an output proxy P_a on one end, and input proxy P_b on the other.

As we mentioned above, MUSIC proxies are devices, not regular neuron models. Like most devices, proxies ignore weight and delay parameters on incoming connections. Any delay applied to the connection from N_a to the output proxy P_a is thus silently ignored. MUSIC makes the inter-simulation transmission delays invisible to the models themselves, so the connection from P_a to P_b is effectively zero. The total delay and weight of the connection from N_a to N_b is thus that set on the P_b to N_b connection.

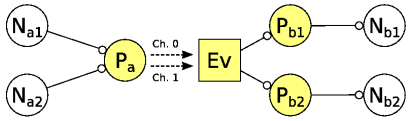


Fig. 5.9: A MUSIC connection with two outputs and two inputs. A single output proxy sends two channels of data to an input event handler that divides the channels to the two input proxies. They connect the recipient neuron models.

When we have multiple channels, the structure looks something like in Fig. 5.9. Now we have two neurons N_{a1} and N_{a2} that we want to connect to N_{b1} and N_{b2} respectively. As we mentioned above, NEST devices can accept connections from multiple separate devices, so we only need one output proxy P_a. We connect each input to a different channel.

Nodes can only output one connection stream, so on the receiving side we need one input proxy P_b per input. Internally,

there is a single MUSIC event handler device Ev that accepts all inputs from a port, then sends the appropriate channel inputs to each input proxy. These proxies each connect to the recipient neurons as above.

5.5.4 Publication

Djurfeldt M. et al. 2010. Run-time interoperability between neuronal network simulators based on the music framework. Neuroinformatics. 8(1):43–60. DOI: 10.1007/s12021-010-9064-z.

5.6 Connect two NEST simulations using MUSIC

Let's look at an example of two NEST simulations connected through MUSIC. We'll implement the simple network in Fig. 5.9 from *the introduction to this tutorial*.

We need a sending process, a receiving process and a MUSIC configuration file:

```
1  #!/usr/bin/env python
2
3  import nest
4  nest.SetKernelStatus({"overwrite_files": True})
5
6  neurons = nest.Create('iaf_neuron', 2, [{'I_e': 400.0}, {'I_e': 405.0}])
7
8  music_out = nest.Create('music_event_out_proxy', 1,
9      params = {'port_name':'p_out'})
10
11 for i, n in enumerate(neurons):
12     nest.Connect([n], music_out, "one_to_one",{'music_channel': i})
13
14 sdetector = nest.Create("spike_detector")
15 nest.SetStatus(sdetector, {"withgid": True, "withtime": True, "to_file": True,
16     "label": "send", "file_extension": "spikes"})
17
18 nest.Connect(neurons, sdetector)
19
20 nest.Simulate(1000.0)
```

The sending process is quite straightforward. We import the NEST library and set a useful kernel parameter. On line 6, we create two simple intergrate-and-fire neuron models, one with a current input of 400mA, and one with 405mA, just so they will respond differently. If you use ipython to work interactively, you can check their current status dictionary with `nest.GetStatus(neurons)`. The definitive documentation for NEST nodes is the header file, in this case `models/iaf_neuron.h` in the NEST source.

We create a single `music_event_out_proxy` for our output on line 8, and set the port name. We loop over all the neurons on lines 11-20 and connect them to the proxy one by one, each one with a different output channel. As we saw earlier, each MUSIC port can have any number of channels. Since the proxy is a device, it ignores any weight or delay settings here.

Lastly, we create a spike detector, set the parameters (which we could have done directly in the `Create` call) and connect the neurons to the spike detector so we can see what we're sending. Then we simulate for one second.

```
1  #!/usr/bin/env python
2
3  import nest
4  nest.SetKernelStatus({"overwrite_files": True})
5
```

(continues on next page)

(continued from previous page)

```
6  music_in = nest.Create("music_event_in_proxy", 2,
7      params = {'port_name': 'p_in'})
8
9  for i, n in enumerate(music_in):
10     nest.SetStatus([n], {'music_channel': i})
11
12 nest.SetAcceptableLatency('p_in', 2.0)
13
14 parrots = nest.Create("parrot_neuron", 2)
15
16 sdetector = nest.Create("spike_detector")
17 nest.SetStatus(sdetector, {"withgid": True, "withtime": True, "to_file": True,
18     "label": "receive", "file_extension": "spikes"})
19
20 nest.Connect(music_in, parrots, 'one_to_one', {"weight":1.0, "delay": 2.0})
21 nest.Connect(parrots, sdetector)
22
23 nest.Simulate(1000.0)
```

The receiving process follows the same logic, but is just a little more involved. We create two `music_event_in_proxy` — one per channel — on lines 6-7 and set the input port name. As we discussed above, a NEST node can accept many inputs but only emit one stream of data, so we need one input proxy per channel to be able to distinguish the channels from each other. On lines 9-10 we set the input channel for each input proxy.

The SetAcceptableLatency command on line 12 sets the maximum time, in milliseconds, that MUSIC is allowed to delay delivery of spikes transmitted through the named port. This should never be more than the *minimum* of the delays from the input proxies to their targets; that's the 2.0 ms we set on line 20 in our case.

On line 14 we create a set of parrot neurons. They simply repeat the input they're given. On lines 16-18 we create and configure a spike detector to save our inputs. We connect the input proxies one-to-one with the parrot neurons on line 20, then the parrot neurons to the spike detector on line 21. We will discuss the reasons for this in a moment. Finally we simulate for one second.

```
binary=./send.py
np=2

[to]
binary=./receive.py
np=2

from.p_out -> to.p_in [2]
```

The MUSIC configuration file structure is straightforward. We define one process `from` and one `to`. For each process we set the name of the binary we wish to run and the number of MPI processes it should use. On line 9 we finally define a connection from output port `p_out` in process `from` to input port `p_in` in process `to`, with two channels.

If our programs had taken command line options we could have added them with the `args` command:

```
binary=./send.py
args= --option -o somefile
```

Run the simulation on the command line like this:

```
mpirun -np 4 music python.music
```

You should get a screenful of information scrolling past, and then be left with four new data files, named something like `send-N-0.spikes`, `send-N-1.spikes`, `receive-M-0.spikes` and `receive-M-1.spikes`. The

names and suffixes are of course the same that we set in `send.py` and `receive.py` above. The first numeral is the node ID of the spike detector that recorded and saved the data, and the final numeral is the rank order of each process that generated the file.

Collate the data files:

```
cat send-*spikes | sort -k 2 -n  >send.spikes
cat receive-*spikes | sort -k 2 -n  >receive.spikes
```

We run the files together, and sort the output numerically ($-n$) by the second column ($-k$). Let's look at the beginning of the two files side by side:

```
send.spikes                receive.spikes

2   26.100                 4   28.100
1   27.800                 3   29.800
2   54.200                 4   56.200
1   57.600                 3   59.600
2   82.300                 4   84.300
1   87.400                 3   89.400
2   110.40                 4   112.40
1   117.20                 3   119.20
```

As expected, the received spikes are two milliseconds later than the sent spikes. The delay parameter for the connection from the input proxies to the parrot neurons in `receive.py` on line 20 accounts for the delay.

Also — and it may be obvious in a simple model like this — the neuron IDs on the sending side and the IDs on the receiving side have no fixed relationship. The sending neurons have ID 1 and 2, while the recipients have 3 and 4. If you need to map events in one simulation to events in another, you have to record this information by other means.

5.6.1 Continuous Inputs

MUSIC can send not just spike events, but also continuous inputs and messages. In NEST there are devices to receive, but not send, such inputs. The NEST documentation has a few examples such as this one below:

```
1  #!/usr/bin/python
2
3  import nest
4
5  mcip = nest.Create('music_cont_in_proxy')
6  nest.SetStatus(mcip, {'port_name' : 'contdata'})
7
8  time = 0
9  while time < 1000:
10     nest.Simulate (10)
11     data = nest.GetStatus (mcip, 'data')
12     print data
13     time += 10
```

The start mirrors our earlier receiving example: you create a continuous input proxy (a single input in this case) and set the port name.

NEST has no general facility to actually apply continuous-valued inputs directly into models. Its neurons deal only with spike events. To use the input you need to create a loop on lines 9-13 where you simulate for a short period, explicitly read the value on line 11, apply it to the simulation model, then simulate for a period again.

People sometimes try to use this pattern to control the rate of a Poisson generator from outside the simulation. You get the rate from outside as a continuous value, then apply it to the Poisson generator that in turn stimulates input neurons

in your network.

The problem is that you need to suspend the simulation every cycle, drop out to the Python interpreter, run a bit of code, then call back in to the simulator core and restart the simulation again. This is acceptable if you do it every few hundred or thousand milliseconds or so, but with an input that may change every few milliseconds this becomes very, very slow.

A much better approach is to forgo the use of the NEST Poisson generator. Generate a Poisson sequence of spike events in the *outside* process, and send the spike events directly into the simulation like we did in our earlier python example. This is far more effective, and the outside process is not limited to the generators implemented in NEST but can create any kind of spiking input. In the next section we will take a look at how to do this.

5.7 MUSIC Connections in C++ and Python

5.7.1 The C++ interface

The C++ interface is the lowest-level interface and what you would use to implement a MUSIC interface in simulators. But it is not a complicated API, so you can easily use it for your own applications that connect to a MUSIC-enabled simulation.

Let's take a look at a pair of programs that send and receive spikes. These can be used as inputs or outputs to the NEST models we created above with no change to the code. C++ code tends to be somewhat longwinded so we only show the relevant parts here. The C++ interface is divided into a setup phase and a runtime phase. Let's look at the setup phase first:

```
MPI::Intracomm comm;

int main(int argc, char **argv)
{
    MUSIC::Setup* setup = new MUSIC::Setup (argc, argv);
    comm = setup->communicator();

    double simt;                // read simulation time from the
    setup->config ("simtime", &simt);       // MUSIC configuration file

    MUSIC::EventOutputPort *outdata =       // set output port
    setup->publishEventOutput("p_out");

    int nProcs = comm.Get_size();       // Number of mpi processes
    int rank = comm.Get_rank();         // I am this process

    int width = 0;              // Get number of channels
    if (outdata->hasWidth()) {          // from the MUSIC configuration
    width = outdata->width();
    }
    // divide output channels evenly among MPI processes
    int nLocal = width / nProcs;    // Number of channels per process
    int rest = width % nProcs;
    int firstId = nLocal * rank;    // index of lowest ID
    if (rank < rest) {
    firstId += rank;
    nLocal += 1;
    } else
    firstId += rest;

```

(continues on next page)

(continued from previous page)

```
31      MUSIC::LinearIndex outindex(firstId, nLocal);    // Create local index
32      outdata->map(&outindex, MUSIC::Index::GLOBAL);   // apply index to port
33
34      [ ... continued below ... ]
35  }
```

At lines 5-6 we initialize MUSIC and MPI. The communicator is common to all processes running under MUSIC, and you'd use it instead of `COMM_WORLD` for your MPI processing.

Lines 7 and 8 illustrate something we haven't discussed so far. We can set and read free parameters in the MUSIC configuration file. We can for instance use that to set the simulation time like we do here; although this is of limited use with a NEST simulation as you can't read these configuration parameters from within NEST.

We set up an event output port and name it on line 11 and 12, then get the number of MPI processes and our process rank for later use. In lines 17-19 we read the number of channels specified for this port in the configuration file. We don't need to set the channels explicitly beforehand like we do in the NEST interface.

We need to tell MUSIC which channels should be processed by what MPI processes. Lines 22-29 are the standard way to create a linear index map from channels to MPI processes. It divides the set of channels into equal-sized chunks, one per MPI process. If channels don't divide evenly into processes, the lower-numbered ranks each get an extra channel. `firstId` is the index of the lowest-numbered channel for the current MPI process, and `nLocal` is the number of channels allocated to it.

On lines 31 and 32 we create the index map and then apply it to the output port we created in line 11. The `Index::GLOBAL` parameter says that each rank will refer to its channels by its global ID number. We could have used `Index::LOCAL` and each rank would refer to their own channels starting with 0. The linear index is the simplest way to map channels, but there is a permutation index type that lets you do arbitrary mappings if you want to.

The `map` method actually has one more optional argument: the `maxBuffered` argument. Normally MUSIC decides on its own how much event data to buffer on the receiving side before actually transmitting it. It depends on the connection structure, the amount of data that is generated and other things. But if you want, you can set this explicitly:

```
1  outdata->map(&outindex, MUSIC::Index::GLOBAL, maxBuffered)
```

With a `maxBuffered` value of 1, for instance, MUSIC will send emitted spike events every cycle. With a value of 2 it would send data every other cycle. This parameter can be necessary if the receiving side is time-sensitive (perhaps the input controls some kind of physical hardware), and the data needs to arrive as soon as possible.

```
1  [ ... continued from above ... ]
2
3  // Start runtime phase
4  MUSIC::Runtime runtime = MUSIC::Runtime(setup, TICK);
5  double tickt =  runtime.time();
6
7  while (tickt < simt) {
8  for (int idx = firstId; idx<(firstId+nLocal); idx++) {
9      // send poisson spikes to every channel.
10     send_poisson(outdata, RATE*(idx+1), tickt, idx);
11 }
12 runtime.tick();          // Give control to MUSIC
13 tickt = runtime.time();
14 }
15 runtime.finalize();          // clean up and end
16
17 }
18
19 double frand(double rate) {return -(1./rate)*log(random()/double(RAND_MAX));}
```

(continues on next page)

(continued from previous page)

```
20
21 void send_poisson(MUSIC::EventOutputPort* outport,
22           double rate, double tickt, int index) {
23     double t = frand(rate);
24     while (t<TICK) {
25     outport -> insertEvent(tickt+t, MUSIC::GlobalIndex(index));
26     t = t + frand(rate);
27     }
28 }
```

The runtime phase is short. On line 4 we create the MUSIC runtime object, and let it consume the setup. In the runtime loop on lines 7-14 we output data, then give control to MUSIC by its `tick()` function so it can communicate, until the simulation time exceeds the end time.

`runtime.time()` on lines 5 and 13 gives us the current time according to MUSIC. In lines 8-10 we loop through the channel indexes corresponding to our own rank (that we calculated during setup), and call a function defined from line 20 onwards that generates a poisson spike train with the rate we request.

The actual event insertion happens on line 24, and we give it the time and the global index of the channel we target. The loop on line 8 loops through only the indexes that belong to this rank, but that is only for performance. We could loop through all channels and send events to all of them if we wanted; MUSIC will silently ignore any events targeting a channel that does not belong to the current rank.

`runtime.tick()` gives control to MUSIC. Any inserted events will be sent to their destination, and any new incoming events will be received and available once the method returns. Be aware that this call is blocking and could take an arbitrary amount of time, if MUSIC has to wait for another simulation to catch up. If you have other time-critical communications you will need to put them in a different thread.

Once we reach the end of the simulation we call `runtime.finalize()`. Music will shut down the communications and clean up after itself before exiting.

```
1  MPI::Intracomm comm;
2  FILE *fout;
3
4  struct eventtype  {
5      double t;
6      int id;
7  };
8  std::queue <eventtype> in_q;
9
10 class InHandler : public MUSIC::EventHandlerGlobalIndex {
11     public:
12     void operator () (double t, MUSIC::GlobalIndex id) {
13         struct eventtype ev = {t, (int)id};
14         in_q.push(ev);
15     }
16 };
17
18 int main(int argc, char **argv)
19 {
20     MUSIC::Setup* setup = new MUSIC::Setup (argc, argv);
21     comm = setup->communicator();
22
23     double simt;
24     setup->config ("simtime", &simt);
25
26     MUSIC::EventInputPort *indata =
```

(continues on next page)

(continued from previous page)

```
27      setup->publishEventInput("p_in");
28
29      InHandler inhandler;
30
31      [ ... get processes, rank and channel width as in send.cpp ... ]
32
33      char *fname;
34      int dummy = asprintf(&fname, "output-%d.spk", rank);
35      fout = fopen(fname, "w");
36
37      [ ... calculate channel allocation as in send.cpp ... ]
38
39      MUSIC::LinearIndex inindex(firstId, nLocal);
40      indata->map(&inindex, &inhandler, IN_LATENCY);
41  }
```

The setup phase for the reveiving application is mostly the same as the sending one. The main difference is that we receive events through a callback function that we provide. During communication, MUSIC will call that function once for every incoming event, and that function stores those events until MUSIC is done and we can process them.

For storage we define a structure to hold time stamp and ID pairs on lines 4-7, and a queue of such structs on line 8. Lines 10-14 defines our callback function. The meat of it is lines 13-14, where we create a new event struct instance with the time stamp and ID we received, then push the structure onto our queue.

The actual setup code follows the same pattern as before: we create a setup object, get ourself a communicator, read any config file parameters and create a named input port. We also declare an instance of our callback event handler on line 29. We get our process and rank information and calculate our per-rank channel allocation in the exact same way as before.

The map for an input port that we create on line 40 needs two additional parameters that the output port map did not. We give it a reference to our callback function that we defined earlier. When events appear on the port, they get passed to the callback function. It also has an optional latency parameter. This is the same latency that we set with the separate `SetAcceptableLatency` function in the NEST example earlier, and it works the same way. Just remember that the MUSIC unit of time is seconds, not milliseconds.

```
1   int main(int argc, char **argv)
2   {
3       MUSIC::Runtime runtime = MUSIC::Runtime(setup, TICK);
4       double tickt = runtime.time();
5
6       while (tickt < simt) {
7       runtime.tick();      // Give control to MUSIC
8       tickt = runtime.time();
9       while (!in_q.empty()) {
10          struct eventtype ev = in_q.front();
11          fprintf (fout, "%d\t%.4f\n", ev.id, ev.t);
12          in_q.pop();
13      }
14      }
15      fclose(fout);
16      runtime.finalize();
17  }
```

The runtime is short. As before we create a runtime object that consumes the setup, then we loop until the MUSIC time exceeds our simulation time. We call `runtime.tick()` each time through the loop on line 8 and we process received events after the call to `tick()`. If you had a process with both sending and receiving ports you would submit the sending data before the `tick()` call, and process the receiving data after it in the same loop.

The `in_q` input queue we defined earlier holds any new input events. We take the first element on line 10, then process it — we write it out to a file — and finally pop it off the queue. When the queue is empty we're done and go back around the main loop again.

Lastly we call `runtime.finalize()` as before.

Building the Code

We have to build our `C++` code. The example code is already set up for the GNU Autotools, just to show how to do this for a MUSIC project. There's only two build-related files we need to care about (all the rest are autogenerated), `configure.ac` and `Makefile.am`.

```
1  AC_INIT(simple, 1.0)
2  AC_PREREQ([2.59])
3  AM_INIT_AUTOMAKE([1.11 -Wall subdir-objects no-define foreign])
4  AC_LANG([C++])
5  AC_CONFIG_HEADERS([config.h])
6  dnl # set OpenMPI compiler wrapper
7  AC_PROG_CXX(mpicxx)
8  AC_CHECK_LIB([music], [_init])
9  AC_CHECK_HEADER([music.hh])
10 AC_CONFIG_FILES([Makefile])
11 AC_OUTPUT
```

The first three lines set the project name and version, the minimum version of autotools we require and a list of options for Automake. Line 4 sets the current language, and line 5 that we want a config.h file.

Line 7 tells autoconf to use the `mpicxx` MPI wrapper as the C++ compiler. Lines 8-9 tells it to test for the existence of the `music` library, and look for the `music.hh` include file.

```
1 bin_PROGRAMS = send recv
2 send_SOURCES = send.cpp
3 recv_SOURCES = recv.cpp
```

`Makefile.am` has only three lines: `bin_PROGRAMS` lists the binaries we want to build. `send_SOURCES` and `recv_SOURCES` lists the source files each one needs.

Your project should already be set up, but if you start from nothing, you need to generate the rest of the build files. You'll need the Autotools installed for that. The easiest way to generate all build files is to use `autoreconf`:

```
autoreconf --install --force
```

Then you can build with the usual sequence of commands:

```
./configure
make
```

Try the Code

We can run these programs just like we did with the NEST example, using a Music configuration file:

```
1 simtime=1.0
2 [from]
3   binary=./send
4   np=2
5 [to]
```

(continues on next page)

(continued from previous page)

```
  binary=./recv
  np=2

  from.p_out -> to.p_in [2]
```

The structure is just the same as before. We have added a `simtime` parameter for the two applications to read, and the binaries are our two new programs. We run this the same way:

```
mpirun -np 4 music simple.music
```

You can change the simulation time by changing the `simtime` parameter at the top of the file. Also, these apps are made to deal with any number of channels, so you can change `[2]` to anything you like. If you have more channels than MPI processes for the `recv` app you will get more than one channel recorded per output file, just as the channel allocation code specified. If you have more MPI processes than input channels, some output files will be empty.

You can connect these with the NEST models that we wrote earlier. Copy them into the same directory. Then, in the `cpp.music` config file, change the `binary` parameter in `[from]` from `binary=./send` to `binary=./send.py`. You get two sets of output files. Concatenate them as before, and compare:

```
send.py             recv

2   26.100          1    0.0261
1   27.800          0    0.0278
2   54.200          1    0.0542
1   57.600          0    0.0576
2   82.300          1    0.0823
1   87.400          0    0.0874
2   110.40          1    0.1104
```

Indeed, we get the expected result. The IDs from the python process on the left are the originating neurons; the IDs on the right is the MUSIC channel on the receiving side. And of course NEST deals in milliseconds while MUSIC uses seconds.

This section has covered most things you need in order to use it for straightforward user-level input and output applications. But there is a lot more to the MUSIC API, especially if you intend to implement it as a simulator interface, so you should consult the documentation for more details.

5.8 The pymusic interface

MUSIC has recently aqcuired a plain Python interface to go along with the C++ API. If you just want to connect with a simulation rather than adding MUSIC capability to a simulator, this Python interface can be a lot more convenient than C++. You have Numpy, Scipy and other high-level libraries available, and you don't need to compile anything.

The interface is closely modelled on the C++ API; indeed, the steps to use it is almost exactly the same. You can mostly refer to the *C++ description* for explanation. Below we will only highlight the differences to the C++ API. The full example code is in the `pymusic` directory in the MUSIC repository.

```
#!/usr/bin/python
import music

[ ... ]

outdata.map(music.Index.GLOBAL,
    base=firstId,
```

(continues on next page)

(continued from previous page)

```
        size=nlocal)

[ ...]

runtime = setup.runtime(TICK)
tickt = runtime.time()
while tickt < simtime:

    for i in range(width):
    send_poisson(outdata, RATE, tickt, i)

    runtime.tick()
    tickt = runtime.time()
```

The sending code is almost completely identical to its C++ counterpart. Make sure python is used as interpreter for the code (and make sure this file is executable). Import music in the expected way.

Unlike the C++ API, the index is not an object, but simply a label indicating global or local indexing. The `map()` call thus need to get the first ID and the number of elements mapped to this rank directly. Also note that the `map()` functions have a somewhat unexpected parameter order, so it's best to use named parameters just in case.

The runtime looks the same as the C++ counterpart as well. We get the current simulation time, and repeatedly send new sets of events as long as the current time is smaller than the simulation time.

```
import Queue

in_q = Queue.Queue()

# Our input handler function
def inhandler(t, indextype, channel_id):
    in_q.put([t, channel_id])

    [ ... ]

indata.map(inhandler,
        music.Index.GLOBAL,
        base=firstId,
        size=nlocal,
        accLatency=IN_LATENCY)

tickt = runtime.time()
while tickt < simtime:

    runtime.tick()
    tickt = runtime.time()

    while not in_q.empty():
        ev = in_q.get()
        f.write("{0}\t{1:8.4f}\n".format(ev[1], ev[0]))
```

Here is the structure for the receiving process, modelled on the C++ code. We use a Python `Queue` class to implement our event queue.

The input handler function has signature `(float time, int indextype, int channel_id)`. The `time` and `channel_id` are the event times and IDs as before. The `indextype` is the type of the map index for this input and is `music.Index.LOCAL` or `music.Index.GLOBAL`.

The `map()` function keyword for accepatable latency is `accLatency`, and the `maxBuffered` keyword we men-

tioned in the previous section is, unsurprisingly, `maxBuffered`. The runtime is, again, the same as for C++.

As the `pymusic` bindings are still quite new the documentation is still lagging behind. This quick introduction should nevertheless bee enough for you to get going with the bindings. And should you need further help, the authors are only an email away.

5.9 Practical Tips

5.9.1 Start MUSIC using mpirun

There is an alternative way to start a MUSIC simulation without the `music` binary. The logic for parsing the configuration file is built into the library itself. So we can start each binary explicitly using mpirun. We give the config file name and the corresponding app label as command line options:

```
mpirun -np 2 <binary> --music_config <config file> --app-label <label> : ...
```

So to start a simulation with the sendsimple.py and recv programs, we can do:

```
mpirun -np 2 ./sendsimple.py --music-config simplepy.music --app-label from␣
↪:/
   -np 2 ./recv --music-config simplepy.music --app-label to
```

This looks long and cumbersome, of course, but it can be useful. Since it's parsed by the shell you are not limited to what the `music` launcher can parse, but the binary can be anything the shell can handle, including an explicit interpreter invocation or a shell script.

As a note, the config file no longer needs to contain the right binary names. But it *does* need to have a non-empty `binary=<something>` line for each process. The parser expects it and will complain (or crash) otherwise. Also, if you try to process comand line options in your Pynest script, it is very likely you will confuse MUSIC.

5.9.2 Disable Messages

NEST can be quite chatty as it connects things, especially with large networks. If we don't want all that output, we can tell it to display only error messages:

```
nest.sli_run("M_ERROR setverbosity")
```

There is unfortunately no straightforward way to suppress the initial welcome message. That is somewhat unfortunate, as they add up quickly in the output of a simulation when you use more than a few hundred cores.

5.9.3 Comma as decimal point

Sorting output spikes may fail if you, like the authors, come from a country that uses a comma as decimal separator and runs your computer in your native language. The problem is that sort respects the language settings and expects the decimal separator to be a comma. When it sees the decimal point in the input it assumes the numeral has ended and sorts only on the integer part.

The way to fix this is to set `LC\_ALL=C` before running the sort command. In a script or in the terminal you can do:

```
export LC_ALL=C
cat output-*|sort -k 2 -n >output.spikes
```

Or, if you want to do this temporarily for only one command:

```
cat output-*|LC_ALL=C sort -k 2 -n >output.spikes
```

5.9.4 Build Autotool-enable project

To build an Autotool-enabled C/C++ project, you don't actually need to be in the main directory. You can create a subdirectory and build everything from there. For instance, with the simple C++ MUSIC project in section *C++ build*, we can do this:

```
mkdir build
cd build
../configure
make
```

Why do that? Because all files you generate when building the project ends up under the `build` subdirectory, keeping the source directories completely clean and untouched. You can have multiple builds `debug`, `noMPI` and so on with different build options enabled, and you can completely clean out a build simply by deleting the directory.

This is surely completely obvious to many of you, but this author is almost ashamed to admit just how many years it took before I realized you could do this. I sometimes actually kept two copies of projects checked out just so I could build a separate debug version.

5.10 Video Tutorial Series

5.10.1 Introduction to NEST: Simulating a single neuron

CHAPTER 6

Example Neural Networks in NEST

You can find a list of networks using NEST at the following link: http://www.nest-simulator.org/more-example-networks/

CHAPTER 7

Guides

Here you can find detailed look into a variety of topics in NEST.

7.1 Analog recording with multimeter

As of r89xx, NEST replaces a range of analog recording devices, such as voltmeter, conductancemeter and aeif_w_meter with a universal *multimeter*, which can record all analog quantities a model neuron makes available for recording. Multimeter works essentially as the old-style voltmeter, but with a few changes:

- The `/recordables` list of a neuron model will tell you which quantities can be recorded:

```
In [3]: nest.GetDefaults('iaf_cond_alpha')['recordables']
Out[3]: ['V_m', 'g_ex', 'g_in', 't_ref_remaining']
```

- You have to configure multimeter to record from a set of quantities:

```
nest.Create('multimeter', params={'record_from': ['V_m', 'g_ex']})
```

- By default, the recording interval is 1ms, but you can change this

```
nest.Create('multimeter', params={'record_from': ['V_m', 'g_ex'], 'interval' :0.1}
↪)
```

- The set of variables to record and the recording interval must be set **before** the multimeter is connected to any node, and cannot be changed afterwards.
- After one has simulated a little, the `events` entry of the multimeter status dictionary will contain one numpy array of data for each recordable.
- Any node can only be recorded from by one multimeter.

7.1.1 Adapting scripts using voltmeter

Many NEST users have scripts that use voltmeter to record membrane potential. To ease the transition to the new-style analog recording, NEST still provides a device called `voltmeter`. It is simply a multimeter pre-configured to record the membrane potential `V_m`. It can be used exactly as the old voltmeter. The only change you need to make to your scripts is that you collect data from events/V_m instead of from events/potentials, e.g.

```
In [24]: nest.GetStatus(m, 'events')[0]['V_m']

Out[24]:
array([-70.        , -70.        , -70.        , -70.        ,
       -70.        , -70.        , -70.        , -70.        ,
```

7.1.2 An example

As an example, here is the multimeter.py example from the PyNEST examples set:

```
import nest
import numpy as np
import pylab as pl

# display recordables for illustration
print 'iaf_cond_alpha recordables: ', nest.GetDefaults('iaf_cond_alpha')['recordables
↪']

# create neuron and multimeter
n = nest.Create('iaf_cond_alpha',  params = {'tau_syn_ex': 1.0, 'V_reset': -70.0})

m = nest.Create('multimeter', params = {'withtime': True, 'interval': 0.1, 'record_
↪from': ['V_m', 'g_ex', 'g_in']})

# Create spike generators and connect
gex = nest.Create('spike_generator', params = {'spike_times': np.array([10.0, 20.0,␣
↪50.0])})
gin = nest.Create('spike_generator',  params = {'spike_times': np.array([15.0, 25.0,␣
↪55.0])})

nest.Connect(gex, n, params={'weight':  40.0}) # excitatory
nest.Connect(gin, n, params={'weight': -20.0}) # inhibitory
nest.Connect(m, n)

# simulate
nest.Simulate(100)

# obtain and display data
events = nest.GetStatus(m)[0]['events']
t = events['times'];

pl.subplot(211)
pl.plot(t, events['V_m'])
pl.axis([0, 100, -75, -53])
pl.ylabel('Membrane potential [mV]')

pl.subplot(212)
pl.plot(t, events['g_ex'], t, events['g_in'])
pl.axis([0, 100, 0, 45])
```

(continues on next page)

(continued from previous page)

```
pl.xlabel('Time [ms]')
pl.ylabel('Synaptic conductance [nS]')
pl.legend(('g_exc', 'g_inh'))
```

Here is the result:

Fig. 7.1: MultimeterExample

7.2 Connection Management

From NEST 2.4 onwards the old connection routines (i.e. `(Random)ConvergentConnect`, `(Random)DivergentConnect` and plain `Connect`) are replaced by one unified `Connect` function. In SLI ,the old syntax of the function still works, while in PyNEST, the `Connect()` function has been renamed to `OneToOneConnect()`. However, simple cases, which are just creating one-to-one connections between two lists of nodes are still working with the new command without the need to change the code. Note that the topology-module is not effected by theses changes. The translation between the old and the new connect routines is described in Old Connection Routines.

The connectivity pattern is defined inside the `Connect()` function under the key 'rule'. The patterns available are described in Connection Rules. In addition the synapse model can be specified within the connect function and all synaptic parameters can be randomly distributed.

The `Connect()` function can be called in either of the following manners:

```
Connect(pre, post)
Connect(pre, post, conn_spec)
Connect(pre, post, conn_spec, syn_spec)
```

`pre` and `post` are lists of Global Ids defining the nodes of origin and termination.

`conn_spec` can either be a string containing the name of the connectivity rule (default: 'all_to_all') or a dictionary specifying the rule and the rule-specific parameters (e.g. 'indegree'), which must be given.

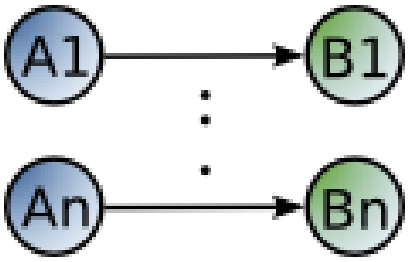
In addition switches allowing self-connections ('autapses', default: True) and multiple connections between pairs of neurons ('multapses', default: True) can be contained in the dictionary. The validity of the switches is confined by the Connect-call. Thus connecting the same set of neurons multiple times with the switch 'multapses' set to False, one particular connection might be established multiple times. The same applies to nodes being specified multiple times in the source or target vector. Here 'multapses' set to False will result in one potential connection between each occurring node pair.

`syn_spec` defines the synapse type and its properties. It can be given as a string defining the synapse model (default: 'static_synapse') or as a dictionary. By using the key-word variant (`Connect(pre, post, syn_spec=syn_spec_dict)`), the conn_spec can be omitted in the call to connect and 'all_to_all' is assumed as the default. The exact usage of the synapse dictionary is described in Synapse Specification.

7.2.1 Connection Rules

Connection rules are specified using the `conn_spec` parameter, which can be a string naming a connection rule or a dictionary containing a rule specification. Only connection rules requiring no parameters can be given as strings, for all other rules, a dictionary specifying the rule and its parameters, such as in- or out-degrees, is required.

7.2.2 one-to-one



The ith node in `pre` is connected to the ith node in `post`. The node lists pre and post have to be of the same length.

Example:

One-to-one connections

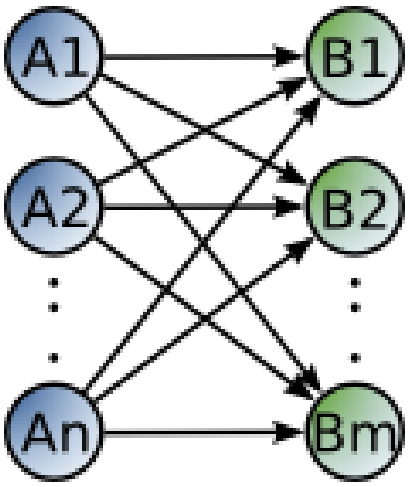
```
n = 10
A = Create("iaf_psc_alpha", n)
B = Create("spike_detector", n)
Connect(A, B, 'one_to_one')
```

This rule can also take two Global IDs A and B instead of integer lists. A shortcut is provided if only two nodes are connected with the parameters weight and delay such that weight and delay can be given as third and fourth argument to the Connect() function.

Example:

```
weight = 1.5
delay = 0.5
Connect(A[0], B[0], weight, delay)
```

7.2.3 all-to-all

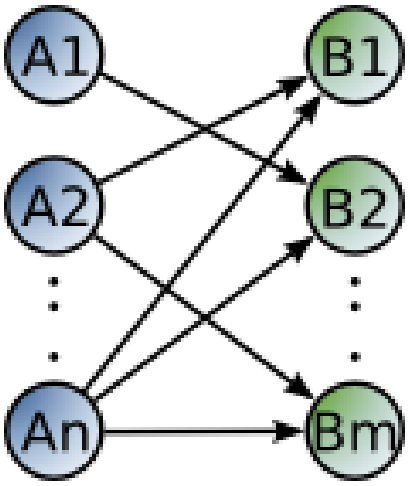


Each node in `pre` is connected to every node in `post`. Since 'all_to_all' is the default, 'rule' doesn't need to specified.

Example:

```
n, m = 10, 12
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", m)
Connect(A, B)
```

fixed-indegree

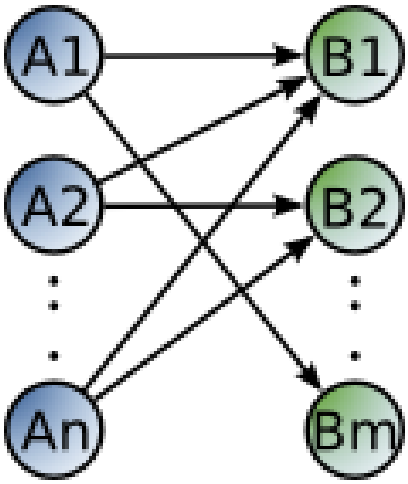


The nodes in `pre` are randomly connected with the nodes in `post` such that each node in `post` has a fixed `indegree`.

Example:

```
n, m, N = 10, 12, 2
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", m)
conn_dict = {'rule': 'fixed_indegree', 'indegree': N}
Connect(A, B, conn_dict)
```

fixed-outdegree



The nodes in `pre` are randomly connected with the nodes in `post` such that each node in `pre` has a fixed `outdegree`.

Example:

```
n, m, N = 10, 12, 2
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", m)
conn_dict = {'rule': 'fixed_outdegree', 'outdegree': N}
Connect(A, B, conn_dict)
```

fixed-total-number

The nodes in `pre` are randomly connected with the nodes in `post` such that the total number of connections equals `N`.

Example:

```
n, m, N = 10, 12, 30
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", m)
conn_dict = {'rule': 'fixed_total_number', 'N': N}
Connect(A, B, conn_dict)
```

pairwise-bernoulli

For each possible pair of nodes from `pre` and `post`, a connection is created with probability `p`.

Example:

```
n, m, p = 10, 12, 0.2
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", m)
conn_dict = {'rule': 'pairwise_bernoulli', 'p': p}
Connect(A, B, conn_dict)
```

Synapse Specification

The synapse properties can be given as a string or a dictionary. The string can be the name of a pre-defined synapse which can be found in the synapsedict (see Synapse Types) or a manually defined synapse via `CopyModel()`.

Example:

```
n = 10
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", n)
CopyModel("static_synapse","excitatory",{"weight":2.5, "delay":0.5})
Connect(A, B, syn_spec="excitatory")
```

Specifying the synapse properties in a dictionary allows for distributed synaptic parameter. In addition to the key 'model' the dictionary can contain specifications for 'weight', 'delay', 'receptor_type' and parameters specific to the chosen synapse model. The specification of all parameters is optional. Unspecified parameters will use the default values determined by the current synapse model. All parameters can be scalars, arrays or distributions (specified as dictionaries). One synapse dictionary can contain an arbitrary combination of parameter types, as long as they agree with the connection routine ('rule').

Scalar parameters must be given as floats except for the 'receptor_type' which has to be initialized as an integer. For more information on the receptor type see Receptor Types .

Example:

```
n = 10
neuron_dict = {'tau_syn': [0.3, 1.5]}
A = Create("iaf_psc_exp_multisynapse", n, neuron_dict)
B = Create("iaf_psc_exp_multisynapse", n, neuron_dict)
syn_dict ={"model": "static_synapse", "weight":2.5, "delay":0.5, 'receptor_type': 1}
Connect(A, B, syn_spec=syn_dict)
```

Array parameters can be used in conjunction with the rules 'one_to_one', 'all_to_all', 'fixed_indegree' and 'fixed_outdegree'. The arrays can be specified as numpy arrays or lists. As for the scalar parameters, all parameters but the receptor types must be specified as arrays of floats. For 'one_to_one' the array must have the same length as the population vector.

Example:

```
A = Create("iaf_psc_alpha", 2)
B = Create("spike_detector", 2)
conn_dict = {'rule': 'one_to_one'}
syn_dict = {'weight': [1.2, -3.5]}
Connect(A, B, conn_dict, syn_dict)
```

When connecting using 'all_to_all', the array must be of dimension len(post) x len(pre).

Example:

```
A = Create("iaf_psc_alpha", 3)
B = Create("iaf_psc_alpha", 2)
syn_dict = {'weight': [[1.2, -3.5, 2.5],[0.4, -0.2, 0.7]]}
Connect(A, B, syn_spec=syn_dict)
```

For 'fixed_indegree' the array has to be a two-dimensional NumPy array with shape (len(post), indegree), where indegree is the number of incoming connections per target neuron, therefore the rows describe the target and the columns the connections converging to the target neuron, regardless of the identity of the source neurons.

Example:

```
A = Create("iaf_psc_alpha", 5)
B = Create("iaf_psc_alpha", 3)
conn_dict = {'rule': 'fixed_indegree', 'indegree': 2}
syn_dict = {'weight': [[1.2, -3.5],[0.4, -0.2],[0.6, 2.2]]}
Connect(A, B, conn_spec=conn_dict, syn_spec=syn_dict)
```

For 'fixed_outdegree' the array has to be a two-dimensional NumPy array with shape (len(pre), outdegree), where outdegree is the number of outgoing connections per source neuron, therefore the rows describe the source and the columns the connections starting from the source neuron regardless of the identity of the target neuron.

Example:

```
A = Create("iaf_psc_alpha", 2)
B = Create("iaf_psc_alpha", 5)
conn_dict = {'rule': 'fixed_outdegree', 'outdegree': 3}
syn_dict = {'weight': [[1.2, -3.5, 0.4], [-0.2, 0.6, 2.2]]}
Connect(A, B, conn_spec=conn_dict, syn_spec=syn_dict)
```

Distributed parameters are initialized with yet another dictionary specifying the 'distribution' and the distribution-specific parameters, whose specification is optional.

Available distributions are given in the rdevdict, the most common ones are:

Distributions Keys 'normal' 'mu', 'sigma' 'normal_clipped' 'mu', 'sigma', 'low ', 'high' 'normal_clipped_to_boundary' 'mu', 'sigma', 'low ', 'high' 'lognormal' 'mu', 'sigma' 'lognormal_clipped' 'mu', 'sigma', 'low', 'high' 'lognormal_clipped_to_boundary' 'mu', 'sigma', 'low', 'high' 'uniform' 'low', 'high' 'uniform_int' 'low', 'high' 'binomial' 'n', 'p' 'binomial_clipped' 'n', 'p', 'low', 'high' 'binomial_clipped_to_boundary' 'n', 'p', 'low', 'high' 'gsl_binomial' 'n', 'p' 'exponential' 'lambda' 'exponential_clipped' 'lambda', 'low', 'high' 'exponential_clipped_to_boundary' 'lambda', 'low', 'high' 'gamma' 'order', 'scale' 'gamma_clipped' 'order', 'scale', 'low', 'high' 'gamma_clipped_to_boundary' 'order', 'scale', 'low', 'high' 'poisson' 'lambda' 'poisson_clipped' 'lambda', 'low', 'high' 'poisson_clipped_to_boundary' 'lambda', 'low', 'high' Example:

```
n = 10
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", n)
syn_dict = {'model': 'stdp_synapse',
            'weight': 2.5,
            'delay': {'distribution': 'uniform', 'low': 0.8, 'high': 2.5},
            'alpha': {'distribution': 'normal_clipped', 'low': 0.5, 'mu': 5.0, 'sigma
↪': 1.0}
           }
Connect(A, B, syn_spec=syn_dict)
```

In this example, the 'all_to_all' connection rule is applied by default, using the 'stdp_synapse' model. All synapses are created with weight 2.5, a delay uniformly distributed in [0.8, 2.5), while the alpha parameters is drawn from a normal distribution with mean 5.0 and std.dev 1.0; values below 0.5 are excluded by re-drawing any values below 0.5. Thus, the actual distribution is a slightly distorted Gaussian.

If the synapse is supposed to have a unique name and distributed parameters it needs to be defined in two steps:

```
n = 10
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", n)
CopyModel('stdp_synapse','excitatory',{'weight':2.5})
syn_dict = {'model': 'excitatory',
            'weight': 2.5,
            'delay': {'distribution': 'uniform', 'low': 0.8, 'high': 2.5},
```

(continues on next page)

(continued from previous page)

```
            'alpha': {'distribution': 'normal_clipped', 'low': 0.5, 'mu': 5.0, 'sigma
↪': 1.0}
          }
Connect(A, B, syn_spec=syn_dict)
```

For further information on the distributions see Random numbers in NEST.

7.2.4 Old Connection Routines

The old connection routines are still available in NEST 2.4, apart from the old `Connect()` which has been renamed to `OneToOneConnect()` and whose the support will end with the next release.

This section contains the documentation for the old connection routines and provides a manual on how to convert the old connection routines to the new `Connect()` function. The new connection routine doesn't yet support arrays or lists as input parameter other than `pre` and `post`. As a workaround we suggest to loop over the arrays.

One-to-one connections

`Connect(pre, post, params=None, delay=None, model='static_synapse')`: Make one-to-one connections of type *model* between the nodes in *pre* and the nodes in *post*. *pre* and *post* have to be lists of the same length. If *params* is given (as dictionary or list of dictionaries), they are used as parameters for the connections. If *params* is given as a single float or as list of floats, it is used as weight(s), in which case *delay* also has to be given as float or as list of floats.

Example old connection routine:

```
A = Create("iaf_psc_alpha", 2)
B = Create("spike_detector", 2)
weight = [1.2, -3.5]
delay = [0.3, 0.5]
Connect(A, B, weight, delay)
```

Note: Using `Connect()` with any of the variables `params`, `delay` and `model` will break the code. As a temporary fix the function `OnToOneConnect()` is provided which works in the same manner as the previous `Connect()`. However, `OneToOneConnect()` won't be supported in the next release.

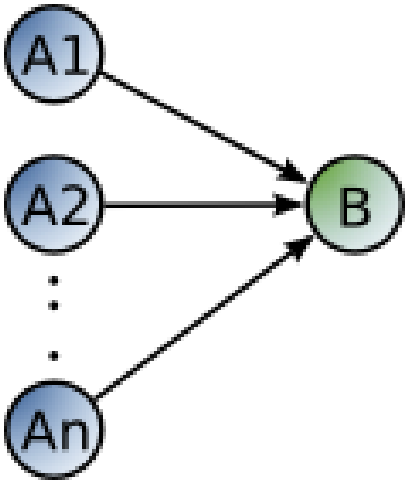
Example temporary fix for old connection routine:

```
A = Create("iaf_psc_alpha", 2)
B = Create("spike_detector", 2)
weight = [1.2, -3.5]
delay = [0.3, 0.5]
OneToOneConnect(A, B, weight, delay)
```

Example new connection routine:

```
A = Create("iaf_psc_alpha", 2)
B = Create("spike_detector", 2)
conn_dict = {'rule': 'one_to_one'}
syn_dict = {'weight': weight, 'delay', delay}
Connect(A, B, conn_dict, syn_dict)
```

Convergent connections



`ConvergentConnect(pre, post, weight=None, delay=None, model='static_synapse')`: Connect all neurons in *pre* to each neuron in *post*. *pre* and *post* have to be lists. If *weight* is given (as a single float or as list of floats), *delay* also has to be given as float or as list of floats.

Example old connection routine:

```
A = Create("iaf_psc_alpha", 2)
B = Create("spike_detector")
ConvergentConnect(A, B)
```

Example new connection routine:

```
A = Create("iaf_psc_alpha", 2)
B = Create("spike_detector")
Connect(A, B)
```

`RandomConvergentConnect(pre, post, n, weight=None, delay=None, model='static_synapse')`: Connect *n* randomly selected neurons from *pre* to each neuron in *post*. *pre* and *post* have to be lists. If *weight* is given (as a single float or as list of floats), *delay* also has to be given as float or as list of floats.

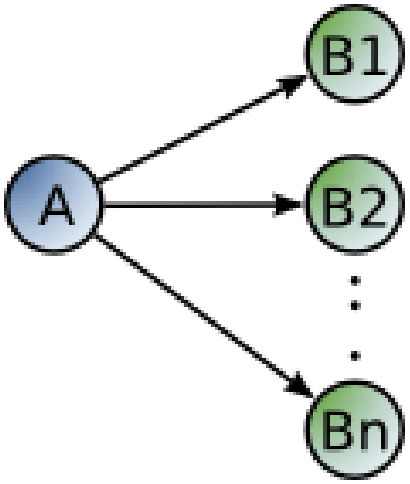
Example old connection routine:

```
option_dict = {'allow_autapses': True, 'allow_multapses': True}
model = 'my_synapse'
nest.RandomConvergentConnect(A, B, N, w0, d0, model, option_dict)
```

Example new connection routine:

```
conn_dict = {'rule': 'fixed_indegree', 'indegree': N, 'autapses': True, 'multapses':␣
↪True}
syn_dict = {'model': 'my_synapse', 'weight': w0, 'delay': d0}
nest.Connect(A, B, conn_dict, syn_dict)
```

Divergent connections



`DivergentConnect(pre, post, weight=None, delay=None, model='static_synapse')`: Connect each neuron in *pre* to all neurons in *post*. *pre* and *post* have to be lists. If *weight* is given (as a single float or as list of floats), *delay* also has to be given as float or as list of floats.

Example old connection routine:

```
A = Create("iaf_psc_alpha")
B = Create("spike_detector", 2)
DivergentConnect(A, B)
```

Example new connection routine:

```
A = Create("iaf_psc_alpha")
B = Create("spike_detector", 2)
Connect(A, B)
```

`RandomDivergentConnect(pre, post, n, weight=None, delay=None, model='static_synapse')`: Connect each neuron in *pre* to *n* randomly selected neurons from *post*. *pre* and *post* have to be lists. If *weight* is given (as a single float or as list of floats), *delay* also has to be given as float or as list of floats.

Example old connection routine:

```
option_dict = {'allow_autapses': True, 'allow_multapses': True}
model = 'my_synapse'
nest.RandomDivergentConnect(A, B, N, w0, d0, model, option_dict)
```

Example new connection routine:

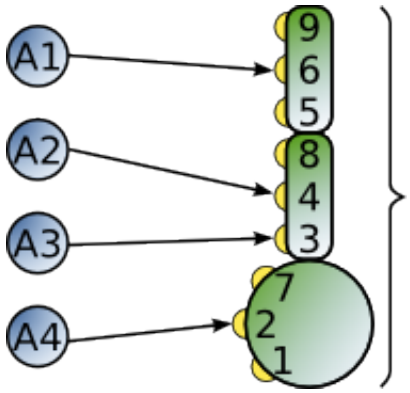
```
conn_dict = {'rule': 'fixed_outdegree', 'outdegree': N, 'autapses': True, 'multapses
↪': True}
syn_dict = {'model': 'my_synapse', 'weight': w0, 'delay': w0}
nest.Connect(A, B, conn_dict, syn_dict)
```

7.2.5 Topological Connections

If the connect functions above are not sufficient, the topology provides more sophisticated functions. For example, it is possible to create receptive field structures and much more! See Topological Connections for more information.

7.2.6 Receptor Types

Each connection in NEST targets a specific receptor type on the post-synaptic node. Receptor types are identified by integer numbers, the default receptor type is 0. The meaning of the receptor type depends on the model and is documented in the model documentation. To connect to a non-standard receptor type, the parameter *receptor_type* of the additional argument *params* is used in the call to the `Connect` command. To illustrate the concept of receptor types, we give an example using standard integrate-and-fire neurons as presynaptic nodes and a multi-compartment integrate-and-fire neuron (`iaf_cond_alpha_mc`) as post-synaptic node.




```
A1, A2, A3, A4 = Create("iaf_psc_alpha", 4)
B = Create("iaf_cond_alpha_mc")
receptors = GetDefaults("iaf_cond_alpha_mc")["receptor_types"]
print receptors

{'soma_exc': 1,
 'soma_inh': 2,
 'soma_curr': 7,
 'proximal_exc': 3
 'proximal_inh': 4,
 'proximal_curr': 8,
 'distal_exc': 5,
 'distal_inh': 6,
 'distal_curr': 9,}

Connect([A1], B, syn_spec={"receptor_type": receptors["distal_inh"]})
Connect([A2], B, syn_spec={"receptor_type": receptors["proximal_inh"]})
Connect([A3], B, syn_spec={"receptor_type": receptors["proximal_exc"]})
Connect([A4], B, syn_spec={"receptor_type": receptors["soma_inh"]})
```

The code block above connects a standard integrate-and-fire neuron to a somatic excitatory receptor of a multi-compartment integrate-and-fire neuron model. The result is illustrated in the figure.

7.2.7 Synapse Types

NEST supports multiple synapse types that are specified during connection setup. The default synapse type in NEST is `static_synapse`. Its weight does not change over time. To allow learning and plasticity, it is possible to use other synapse types that implement long-term or short-term plasticity. A list of available types is accessible via the command *Models("synapses")*. The output of this command (as of revision 11199) is shown below:

```
['cont_delay_synapse',
 'ht_synapse',
 'quantal_stp_synapse',
```

(continues on next page)

(continued from previous page)

```
 'static_synapse',
 'static_synapse_hom_wd',
 'stdp_dopamine_synapse',
 'stdp_facetshw_synapse_hom',
 'stdp_pl_synapse_hom',
 'stdp_synapse',
 'stdp_synapse_hom',
 'tsodyks2_synapse',
 'tsodyks_synapse']
```

All synapses store their parameters on a per-connection basis. An exception to this scheme are the homogeneous synapse types (identified by the suffix *_hom*), which only store weight and delay once for all synapses of a type. This means that these are the same for all connections. They can be used to save memory.

The default values of a synapse type can be inspected using the command `GetDefaults()`, which takes the name of the synapse as an argument, and modified with `SetDefaults()`, which takes the name of the synapse type and a parameter dictionary as arguments.

```
print GetDefaults("static_synapse")

{'delay': 1.0,
'max_delay': -inf,
'min_delay': inf,
'num_connections': 0,
'num_connectors': 0,
'receptor_type': 0,
'synapsemodel': 'static_synapse',
'weight': 1.0}

SetDefaults("static_synapse", {"weight": 2.5})
```

For the creation of custom synapse types from already existing synapse types, the command `CopyModel` is used. It has an optional argument `params` to directly customize it during the copy operation. Otherwise the defaults of the copied model are taken.

```
CopyModel("static_synapse", "inhibitory", {"weight": -2.5})
Connect(A, B, syn_spec="inhibitory")
```

Note: Not all nodes can be connected via all available synapse types. The events a synapse type is able to transmit is documented in the *Transmits* section of the model documentation.

7.2.8 Inspecting Connections

`GetConnections(source=None, target=None, synapse_model=None)`: Return an array of identifiers for connections that match the given parameters. source and target need to be lists of global ids, model is a string representing a synapse model. If GetConnections is called without parameters, all connections in the network are returned. If a list of source neurons is given, only connections from these pre-synaptic neurons are returned. If a list of target neurons is given, only connections to these post-synaptic neurons are returned. If a synapse model is given, only connections with this synapse type are returned. Any combination of source, target and model parameters is permitted. Each connection id is a 5-tuple or, if available, a NumPy array with the following five entries: source-gid, target-gid, target-thread, synapse-id, port.

The result of `GetConnections` can be given as an argument to the `GetStatus` function, which will then return a list with the parameters of the connections:

```
n1 = Create("iaf_psc_alpha")
n2 = Create("iaf_psc_alpha")
Connect(n1, n2)
conn = GetConnections(n1)
print GetStatus(conn)

[{'synapse_type': 'static_synapse',
  'target': 2,
  'weight': 1.0,
  'delay': 1.0,
  'source': 1,
  'receptor': 0}]
```

7.2.9 Modifying existing Connections

To modify the connections of an existing connection, one also has to obtain handles to the connections with `GetConnections()` first. These can then be given as arguments to the `SetStatus()` functions:

```
n1 = Create("iaf_psc_alpha")
n2 = Create("iaf_psc_alpha")
Connect(n1, n2)
conn = GetConnections(n1)
SetStatus(conn, {"weight": 2.0})
print GetStatus(conn)

[{'synapse_type': 'static_synapse',
  'target': 2,
  'weight': 2.0,
  'delay': 1.0,
  'source': 1,
  'receptor': 0}]
```

7.3 Parallel Computing

7.3.1 Introduction

Parallelization is a means to run simulations faster and use the capabilities of computer clusters and supercomputers to run large-scale simulations.

Since version 2.0, NEST is capable of running simulations on multi-core/-processor machines and computer clusters using two ways of parallelization: *thread-parallel simulation* and *distributed simulations*. The first is implemented using OpenMP (POSIX threads prior to NEST 2.2), while the second is implemented on top of the Message Passing Interface (MPI). Both ways of parallelism can be combined in a hybrid fashion.

Using threads allows to take advantage of multi-core and multi-processor computers without the need for additional software libraries. Using distributed computing allows to draw in more computers and thus enables larger simulations, than would fit into the memory of a single machine. The following paragraphs describe the facilities for parallel and distributed computing in detail.

See Plesser et al (2007) for more information on NEST parallelization and be sure to check the documentation on Random numbers in NEST.

7.3.2 Concepts and definitions

In order to ease the handling of neuron and synapse distribution with both thread and process based parallelization, we use the concept of local and remote threads, called *virtual processes*. A virtual process (VP) is a thread living in one of NEST's MPI processes. Virtual processes are distributed round-robin onto the MPI processes and counted continuously over all processes. The concept is visualized in the following figure:

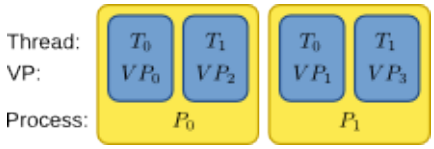


Fig. 7.2: Basic scheme for counting threads (T), virtual processes (VP) and MPI processes (P) in NEST

The status dictionary of each node (i.e. neuron or device) contains three entries that are related to parallel computing:

- *local* (boolean): indicating if the node exists on the local process or not
- *thread* (integer): id of the local thread the node is assigned to
- *vp* (integer): id of the virtual process the node is assigned to.

Node distribution

The distribution of nodes is based on the type of the node. Neurons are assigned to one of the virtual processes in a round-robin fashion. On all other virtual processes, no object is created (the *proxy* object in the figure below is just a conceptual way of keeping the id of the real node free on remote processes). The virtual process *idVP* on which a neuron with global id *idNode* is allocated is given by *idVP = idNode % NVP*, where *NVP* is the total number of virtual processes in the simulation.

Devices for the stimulation and observation of the network are replicated once on each thread in order to balance the load of the different threads and minimize their interaction. Devices thus do not have proxies on remote virtual processes.

The node distribution for a small network consisting of `spike_generator`, four `iaf_psc_alpha`s, and a `spike_detector` in a scenario with two processes with two threads each is shown in the following figure:

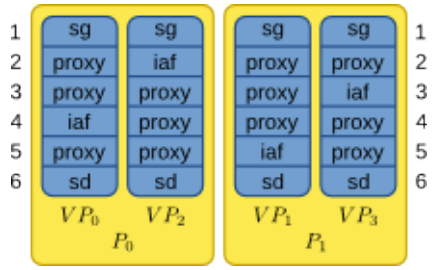


Fig. 7.3: sg=spike_generator, iaf=iaf_psc_alpha, sd=spike_detector. Numbers to the left and right indicate global ids.

For recording devices that are configured to record to a file (property *to_file* set to *true*), the distribution also results in multiple data files, each containing the data from one thread. The files names are composed according to the following scheme

```
[model|label]-gid-vp.[dat|gdf]
```

The first part is the name of the model (e.g. `voltmeter` or `spike_detector`) or, if set, the *label* of the recording device. The second part is the global id (GID) of the recording device. The third part is the id of the virtual process

the recorder is assigned to, counted from 0. The extension is `gdf` for spike files and `dat` for analog recordings from the `multimeter`. The `label` and `file_extension` of a recording device can be set like any other parameter of a node using `SetStatus`.

Spike exchange and synapse updates

Spike exchange in NEST takes different routes depending on the type of the sending and receiving node. There are two distinct cases.

Spikes between neurons are always exchanged through the global spike exchange mechanism. Neuron update and spike generation in the source neuron and spike delivery to the target neuron may be handled by different virtual process in this case. Spike delivery is always handled by the virtual process to which the *target* neuron is assigned (see property `vp` in the status dictionary).

Spike exchange to or from neurons over connections that either originate or terminate at a device (e.g., `spike_generator -> neuron` or `neuron -> spike_detector`) differs in that it bypasses the global spike exchange mechanism. Instead, spikes are delivered locally within the virtual process from or to a replica of the device. In this case, both the pre- and postsynaptic nodes are handled by the virtual process to which the neuron is assigned.

For synapse models supporting plasticity, synapse dynamics in the `Connection` object are always handled by the virtual process of the target node.

7.3.3 Using multiple threads

Thread-parallelism is compiled into NEST by default and should work on all MacOS and Linux machines without additional requirements. In order to keep results comparable and reproducible across different machines, however, the default mode is that only a single thread is used and multi-threading has to be turned on explicitly.

To use multiple threads for the simulation, the desired number of threads has to be set *before* any nodes or connections are created. The command for this is

```
nest.SetKernelStatus({"local_num_threads": T})
```

Usually, a good choice for T is the number of processor cores available on your machine. In some situations, over-subscribing can yield 20-30% improvement in simulation speed. Finding the optimal thread number for a specific situation might require a bit of experimenting.

7.3.4 Using distributed computing

Build requirements

To compile NEST for distributed computing, you need a library implementation of MPI on your system. If you are on a cluster, you most likely have this already. Note, that in the case of a pre-packaged MPI library you will need both, the library and the development packages. Please see the Installation instructions for general information on installing NEST. Please be advised that NEST should currently only be run in a homogeneous MPI environment. Running in a heterogenenous environment can lead to unexpected results or even crashes. Please contact the NEST community if you require support for exotic setups.

Compilation

If the MPI library and header files are installed to the standard directories of the system, it is likely that a simple

```
$NEST_SOURCE_DIR/configure --with-mpi
```

will find them (`$NEST_SOURCE_DIR` is the directory holding the NEST sources). If MPI is installed to a non-standard location `/path/to/mpi`, the command line looks like this:

```
$NEST_SOURCE_DIR/configure --with-mpi=/path/to/mpi
```

In some cases it might be necessary to specify MPI compiler wrappers explicitly:

```
$NEST_SOURCE_DIR/configure CC=mpicc CXX=mpicxx --with-mpi
```

Additional information concerning MPI on OSX can be found here.

Running distributed simulations

Distributed simulations cannot be run interactively, which means that the simulation has to be provided as a script. However, the script does not have to be changed compared to the script for serial simulation: inter-process communication and node distribution is managed transparently inside of NEST.

To distribute a simulation onto 128 processes of a computer cluster, the command line to execute looks like this:

```
mpirun -np 128 python simulation.py
```

Please refer to the MPI library documentation for details on the usage of `mpirun`.

MPI related commands

Although we generally advise strongly against writing process-aware code in simulation scripts (e.g. creating a neuron or device only on one process and such), in special cases it may be necessary to obtain information about the MPI application. One example would opening the right stimulus file for a specific rank. Therefore, some MPI specific commands are available:

`NumProcesses`
The number of MPI processes in the simulation

`ProcessorName`
The name of the machine. The result might differ on each process.

`Rank`
The rank of the MPI process. The result differs on each process.

`SyncProcesses`
Synchronize all MPI processes.

7.3.5 Reproducibility

To achieve the same simulation results even when using different parallelization strategies, the number of virtual processes has to be kept constant. A simulation with a specific number of virtual processes will always yield the same

results, no matter how they are distributed over threads and processes, given that the seeds for the random number generators of the different virtual processes are the same (see Random numbers in NEST).

In order to achieve a constant number of virtual processes, NEST provides the property *total_num_virtual_procs* to adapt the number of local threads (property *local_num_threads*, explained above) to the number of available processes.

The following listing contains a complete simulation script (*simulation.py*) with four neurons connected in a chain. The first neuron receives random input from a `poisson_generator` and the spikes of all four neurons are recorded to files.

```
from nest import *
SetKernelStatus({"total_num_virtual_procs": 4})
pg = Create("poisson_generator", params={"rate": 50000.0})
n = Create("iaf_psc_alpha", 4)
sd = Create("spike_detector", params={"to_file": True})
Connect(pg, [n[0]], syn_spec={'weight': 1000.0, 'delay': 1.0})
Connect([n[0]], [n[1]], syn_spec={'weight': 1000.0, 'delay': 1.0})
Connect([n[1]], [n[2]], syn_spec={'weight': 1000.0, 'delay': 1.0})
Connect([n[2]], [n[3]], syn_spec={'weight': 1000.0, 'delay': 1.0})
Connect(n, sd)
Simulate(100.0)
```

The script is run three times using different numbers of MPI processes, but 4 virtual processes in every run:

```
mkdir 4vp_1p; cd 4vp_1p
mpirun -np 1 python ../simulation.py
cd ..; mkdir 4vp_2p; cd 4vp_2p
mpirun -np 2 python ../simulation.py
cd ..; mkdir 4vp_4p; cd 4vp_4p
mpirun -np 4 python ../simulation.py
cd ..
diff 4vp_1p 4vp_2p
diff 4vp_1p 4vp_4p
```

Each variant of the experiment produces four data files, one for each virtual process (*spike_detector-6-0.gdf*, *spike_detector-6-1.gdf*, *spike_detector-6-2.gdf*, and *spike_detector-6-3.gdf*). Using diff on the three data directories shows that they all contain the same spikes, which means that the simulation results are indeed the same independently of the details of parallelization.

7.4 Random numbers

7.4.1 Introduction

Random numbers are used for a variety of purposes in neuronal network simulations, e.g.

- to create randomized connections
- to choose parameter values randomly
- to inject noise into network simulations, e.g., in the form of Poissonian spike trains.

This document discusses how NEST provides random numbers for these purposes, how you can choose which random number generator (RNG) to choose, and how to set the seed of RNGs in NEST. We use the term "random number" here for ease of writing, even though we are always talking about pseudorandom numbers generated by some algorithm.

NEST is designed to support parallel simulation and this puts some constraints on the use and generation of random numbers. We discuss these in the next section, before going into the details of how to control RNGs in NEST.

On this page, we mainly discuss the use of random numbers in parallel NEST simulations, but the comments pertain equally to serial simulations (N_vp=1).

Random Numbers vs Random Deviates

NEST distinguishes between random number generators, provided by `rngdict` and random deviate generators provided by `rdevdict`. Random *number* generators only provide double-valued numbers uniformly distributed on [0, 1] and uniformly distributed integers in {0, 1, . . . , N}. Random *deviate* generators, on the other hand, provide random numbers drawn from a range of distributions, such as the normal or binomial distributions. In most cases, you will be using random deviate generators. They are in particular used to initialize properties during network construction, as described in the sections Changes in NEST 2.4 and Examples below.

7.4.2 Changes in random number generation in NEST 2.4

Random deviate generation has become significantly more powerful in NEST 2.4, to fully support randomization of connections parameters offered by the revised `Connect` function, as described in Connection Management and illustrated by the examples below. We have also made minor changes to make to achieve greater similarity between NEST, PyNN, and NumPy. For most users, these changes only add new features. Only existing scripts using

- `uniformint`
- `normal_clipped`, `normal_clipped_left`, `normal_clipped_right`

generators from NEST 2.2 need to be adapted as detailed below.

The changes are as follows:

- Uniform integer generator
  - renamed from `uniformint` to `uniform_int`
  - parameters renamed to `low` and `high`
  - returns uniformly distributed integers from `{low, low+1, ..., high}`
- Uniform continuous generator
  - new generator `uniform`
  - parameters `low` and `high`
  - generates numbers uniformly distributed in `[low, high)`
- Full parameter sets for generators
  - In the past, many random deviate generators returned values for fixed parameters, e.g., the `normal` generator could only return zero-mean, unit-variance normal random numbers.
  - Now, all parameters for each generator can be set, in particular:
    - normal: mu, sigma
    - lognormal: mu, sigma
    - exponential: lambda
    - gamma: order, scale
  - Parameter values are checked more systematically for unsuitable values.
- Clipped normal generators
  - parameter names changed to `mu` and `sigma`

  - clipping limits now called `low` and `high`
  - `_left` and `_right` variants removed: for one-sided clipping, just set the boundary you want to clip at, the other is positive or negative infinity

- Clipped variants for most generators
  - For most random deviate generators, `_clipped` variants exist now.
  - For all clipped variants, one can set a lower limit (`low`, default: -infinity) and an upper limit (`high`: +infinty).
  - Clipped variants will then return numbers strictly in `(low, high)` for continuous distributions (e.g. normal, exponential) or `{low, low+1, ..., high}` for discrete distributions (e.g. poisson, binomial). This is achieved by redrawing numbers until an acceptable number is drawn.
  - Note that the resulting distribution differs from the original one and that drawing may become very slow if `(low, high)` contains only very small probability mass. Clipped generator variants should therefore mostly be used to clip tails with very small probability mass when randomizing time constants or delays.
- Clipped-to-boundary variants for most generators
  - To facilitate reproduction of certain publications, NEST also provides `_clipped_to_boundary` variants of most generators.
  - Clipped-to-boundary variants return the value `low` if a number smaller than `low` is drawn, and `high` if a number larger than `high` is drawn.
  - We believe that these variants should *not* be used for new studies.

7.4.3 Basics of parallel simulation in NEST

For details of parallelization in NEST, please see Parallel Computing and Plesser et al (2007). Here, we just summarize a few basics.

- NEST can parallelize simulations through *multi-threading*, *distribution* or a combination of the two.
- A distributed simulation is spread across several processes under the control of MPI (Message Passing Interface). Each network node is *local* to exactly one process and complete information about the node is only available to that process. Information about each connection is stored by the process in which the connection target is local and is only available and changeable on that process.
- Multi-threaded simulations run in a single process in a single computer. As a consequence, all nodes in a multi-threaded simulation are local.
- Distribution and multi-threading can be combined by running identical numbers of threads in each process.
- A serial simulation has a single process with a single seed.
- From the NEST user perspective, distributed processes and threads are visible as **virtual processes**. A simulation distributed across \(M\) MPI processes with \(T\) threads each, has \(N_{vp} = M times T\) virtual processes. It is a basic design principle of NEST that simulations shall generate *identical* results when run with a fixed \(N_{VP}\), no matter how the virutal processes are broken down into MPI processes and threads.
- Useful information can be obtained like this

  import nest nest.NumProcesses() # number of MPI processes nest.Rank() # rank of MPI process executing command nest.GetKernelStatus([‘num_processes’]) # same as nest.NumProcesses() nest.GetKernelStatus([‘local_num_threads’]) # number of threads in present process (same for all processes) nest.GetKernelStatus([‘total_num_virtual_procs’]) # N_vp = M x T

- When querying neurons, only very limited information is available for neurons on other MPI processes. Thus, before checking for specific information, you need to check if a node is local:

  n = nest.Create('iaf_psc_alpha') if nest.GetStatus(n, 'local')[0]: # GetStatus() returns list, pick element print nest.GetStatus(n, 'vp') # virtual process "owning" node print nest.GetStatus(n, 'thread') # thread in calling process "owning" node

7.4.4 Random numbers in parallel simulations

Ideally, all random numbers in a simulation should come from a single RNG. This would require shipping truckloads of random numbers from a central RNG process to all simulations processes and is thus impractical, if not outright prohibitively costly. Therefore, parallel simulation requires an RNG on each parallel process. Advances in RNG technology give us today a range of RNGs that can be used in parallel, with a quite high level of certainty that the resulting parallel streams of random numbers are non-overlapping and uncorrelated. While the former can be guaranteed, we are not aware of any generator for which the latter can be proven.

How many generators in a simulation

In a typical PyNEST simulation running on \(N_{vp}\) virtual processes, we will encounter \(2 N_{vp} + 1\) random number generators:

The global NEST RNG
This generator is mainly used when creating connections using `RandomDivergentConnect`.

One RNG per VP in NEST
These generators are used when creating connections using `RandomConvergentConnect` and to provide random numbers to nodes generating random output, e.g. the `poisson_generator`.

One RNG per VP in Python
These generators are used to randomized node properties (e.g., the initial membrane potential) and connection properties (e.g., weights).

The generators on the Python level are not strictly necessary, as one could in principle access the per-VP RNGs built into NEST. This would require very tedious SLI-coding, though. We therefore recommend at present that you use additional RNGs on the Python side.

Why a Global RNG in NEST

In some situations, randomized decisions on different virtual processes are not independent of each other. The most important case are randomized divergent connections. The problem here is as follows. For the sake of efficiency, NEST stores all connection information in the virtual process (VP) to which the target of a connection resides (target process). Thus, all connections are generated by this target process. Now consider the task of generating 100 randomized divergent connections emanating from a given source neuron while using 4 VPs. Then there should be 25 targets on each VP *on average*, but actual numbers will fluctuate. If independent processes on all VPs tried to choose target neurons, we could never be sure that exactly 100 targets would be chosen in total.

NEST thus creates divergent connections using a global RNG. This random number generator provides the exact same sequence of random numbers on each virtual process. Using this global RNG, each VP chooses 100 targets from the

entire network, but actually creates connections only for those targets that reside on the VP. In practice, the global RNG is implemented using one "clone" on each VP; NEST checks occasionally that all these clones are synchronized, i.e., indeed generate identical sequences.

Seeding the Random Generators

Each of the \(N_{vp}\) random generators needs to be seeded with a different seed to generate a different random number sequences. We recommend that you choose a *master seed* `msd` and seed the \(2N_{vp}+1\) generators with seeds `msd`, `msd+1`, ..., `msd+2*N_vp`. Master seeds for for independent experiments must differ by at least \(2N_{vp}+1\) . Otherwise, the same sequence(s) would enter in several experiments.

Seeding the Python RNGs

You can create a properly seeded list of \(N_{vp}\) RNGs on the Python side using

```
import numpy
msd = 123456
N_vp = nest.GetKernelStatus(['total_num_virtual_procs'])[0]
pyrngs = [numpy.random.RandomState(s) for s in range(msd, msd+N_vp)]
```

`msd` is the master seed, choose your own!

Seeding the global RNG

The global NEST rng is seeded with a single, positive integer number:

```
nest.SetKernelStatus({'grng_seed' : msd+N_vp})
```

Seeding the per-process RNGs

The per-process RNGs are seeded by a list of \(N_{vp}\) positive integers:

```
nest.SetKernelStatus({'rng_seeds' : range(msd+N_vp+1, msd+2*N_vp+1)})
```

Choosing the random generator type

Python and NumPy have the MersenneTwister MT19937ar random number generator built in. There is no simple way of choosing a different generator in NumPy, but as the MT19937ar appears to be a very robust generator, this should not cause significant problems.

NEST uses by default Knuth's lagged Fibonacci random number generator (The Art of Computer Programming, vol 2, 3rd ed, 9th printing or later, ch 3.6). If you want to use other generators, you can exchange them as described below. If you have built NEST without the GNU Science Library (GSL), you will only have the Mersenne Twister MT19937ar and Knuth's lagged Fibonacci generator available. Otherwise, you will also have some 60 generators from the GSL at your disposal (not all of them particularly good). You can see the full list of RNGs using

```
nest.sli_run('rngdict info')
```

Setting a different global RNG

To set a different global RNG in NEST, you have to pass a NEST random number generator object to the NEST kernel. This can currently only be done by writing some SLI code. The following code replaces the current global RNG with MT19937 seeded with 101:

```
nest.sli_run('0 << /grng rngdict/MT19937 :: 101 CreateRNG >> SetStatus')
```

The following happens here:

- `rngdict/MT19937 ::` fetches a "factory" for MT19937 from the `rngdict`
- `101 CreateRNG` uses the factory to create a single MT19937 generator with seed 101
- This is generator is then passed to the `/grng` status variable of the kernel. This is a "write only" variable that is invisible in `GetKernelStatus()`.

Setting different per-processes RNGs

One always needs to exchange all \(N_{vp}\) per-process RNGs at once. This is done by (assuming \(N_{vp}=2\)):

```
nest.sli_run('0 << /rngs [102 103] { rngdict/MT19937 :: exch CreateRNG } Map >>␣
↪SetStatus')
```

The following happens here:

- `[102 103] { rngdict/MT19937 :: exch CreateRNG } Map` creates an array of two RNG objects seeded with 102 and 103, respectively.
- This array is then passed to the `/rngs` status variable of the kernel. This variable is invisible as well.

7.4.5 Examples

NOTE: These examples are not yet updated for NEST 2.4

No random variables in script

If no explicit random variables appear in your script, i.e., if randomness only enters in your simulation through random stimulus generators such as `poisson_generator` or randomized connection routines such as `RandomConvergentConnect`, you do not need to worry about anything except choosing and setting your random seeds, possibly exchanging the random number generators.

Randomizing the membrane potential

If you want to randomize the membrane potential (or any other property of a neuron), you need to take care that each node is updated by the process on which it is local using the per-VP RNG for the VP to which the node belongs. This is achieved by the following code

```
pyrngs = [numpy.random.RandomState(s) for s in range(msd, msd+N_vp)]
nodes    = nest.Create('iaf_psc_delta', 10)
node_info    = nest.GetStatus(nodes)
local_nodes = [(ni['global_id'], ni['vp']) for ni in node_info if ni['local']]
for gid,vp in local_nodes:
   nest.SetStatus([gid], {'V_m': pyrngs[vp].uniform(-70.0, -50.0)})
```

The first line generates \([N_{vp}\) properly seeded NumPy RNGs as discussed above. The next line creates 10 nodes, while the third line extracts status information about each node. For local nodes, this will be full information, for non-local nodes we only get the following fields: `local`, `model` and `type`. On the fourth line, we create a list of tuples, containing global ID and virtual process number for all local neurons. The for loop then sets the membrane potential of each local neuron drawn from a uniform distribution on \([-70, -50]\) using the Python-side RNG for the VP to which the neuron belongs.

Randomizing convergent connections

We continue the above example by creating random convergent connections, \(C_E\) connections per target node. In the process, we randomize the connection weights:

```
C_E = 10
nest.CopyModel("static_synapse", "excitatory")
for tgt_gid, tgt_vp in local_nodes:
    weights = pyrngs[tgt_vp].uniform(0.5, 1.5, C_E)
    nest.RandomConvergentConnect(nodes, [tgt_gid], C_E,
                                 weight=list(weights), delay=2.0,
                                 model="excitatory")
```

Here we loop over all local nodes considered as target nodes. For each target, we create an array of \(C_E\) randomly chosen weights, uniform on \([0.5, 1.5. We then call `RandomConvergentConnect()` with this weight list as argument. Note a few details:

- We need to put `tgt_gid` into brackets as PyNEST functions always expect lists of GIDs.
- We need to convert the NumPy array `weights` to a plain Python list, as most PyNEST functions currently cannot handle array input.
- If we specify `weight`, we must also provide `delay`.

You can check the weights selected by

```
print nest.GetStatus(nest.GetConnections(), ['source', 'target', 'weight'])
```

which will print a list containing a triple of source GID, target GID and weight for each connection in the network. If you want to see only a subset of connections, pass source, target, or synapse model to `GetConnections()`.

Randomizing divergent connections

Randomizing the weights (or delays or any other properties) of divergent connections is more complicated than for convergent connections, because the target for each connection is not known upon the call to `RandomDivergentConnect`. We therefore need to first create all connections (which we can do with a single call, passing lists of nodes and targets), and then need to manipulate all connections. This is not only more complicated, but also significantly slower than the example above.

```
nest.CopyModel('static_synapse', 'inhibitory', {'weight': 0.0, 'delay': 3.0})
nest.RandomDivergentConnect(nodes, nodes, C_E, model='inhibitory')
gid_vp_map = dict(local_nodes)
for src in nodes:
    conns = nest.GetConnections(source=[src], synapse_model='inhibitory')
    tgts = [conn[1] for conn in conns]
    rweights = [{'weight': pyrngs[gid_vp_map[tgt]].uniform(-2.5, -0.5)}
                for tgt in tgts]
    nest.SetStatus(conns, rweights)
```

In this code, we first create all connections with weight 0. We then create `gid_vp_map`, mapping GIDs to VP number for all local nodes. For each node considered as source, we then find all outgoing excitatory connections from that node and then obtain a flat list of the targets of these connections. For each target we then choose a random weight as above, using the RNG pertaining to the VP of the target. Finally, we set these weights. Note that the code above is **slow**. Future versions of NEST will provide better solutions.

Testing scripts randomizing node or connection parameters

To ensure that you are consistently using the correct RNG for each node or connection, you should run your simulation several times the same \(N_{vp}\), but using different numbers of MPI processes. To this end, add towards the beginning of your script

```
nest.SetKernelStatus({"total_num_virtual_procs": 4})
```

and ensure that spikes are logged to file in the current working directory. Then run the simulation with different numbers of MPI processes in separate directories

```
mkdir 41 42 44
cd 41
mpirun -np 1 python test.py
cd ../42
mpirun -np 2 python test.py
cd ../44
mpirun -np 4 python test.py
cd ..
```

These directories should now have identical content, something you can check with `diff`:

```
diff 41 42
diff 41 44
```

These commands should not generate any output. Obviously, this test checks only a necessary, by no means a sufficient condition for a correct simulation (Oh yes, do make sure that these directories contain data! Nothing easier that to pass a diff-test on empty dirs.)

7.5 Scheduling and simulation flow

7.5.1 Introduction

To drive the simulation, neurons and devices (*nodes*) are updated in a time-driven fashion by calling a member function on each of them in a regular interval. The spacing of the grid is called the *simulation resolution* (default 0.1ms) and can be set using `SetKernelStatus`:

```
SetKernelStatus("resolution", 0.1)
```

Even though a neuron model can use smaller time steps internally, the membrane potential will only be visible to a `multimeter` on the outside at time points that are multiples of the simulation resolution.

In contrast to the update of nodes, an event-driven approach is used for the synapses, meaning that they are only updated when an event is transmitted through them (Morrison et al. 2005). To speed up the simulation and allow the efficient use of computer clusters, NEST uses a *hybrid parallelization strategy*. The following figure shows the basic loop that is run upon a call to `Simulate`:

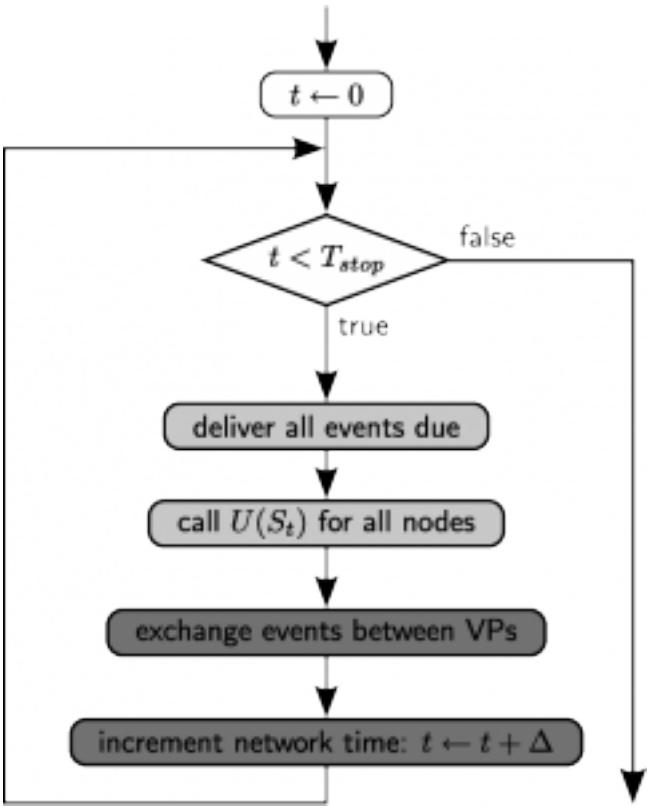


Fig. 7.4: Simulation Loop

The simulation loop. Light gray boxes denote thread parallel parts, dark gray boxes denote MPI parallel parts. U(St) is the update operator that propagates the internal state of a neuron or device.

7.5.2 Simulation resolution and update interval

Each connection in NEST has it's own specific *delay* that defines the time it takes until an event reaches the target node. We define the minimum delay *dmin* as the smallest transmission delay and *dmax* as the largest delay in the network. From this definition follows that no node can influence another node during at least a time of *dmin*, i.e. the elements are effectively decoupled for this interval.

Fig. 7.5: Definitions of the minimimum delay and the simulation resolution.

Definitions of minimum delay (dmin) and simulation resolution (h).

Two major optimizations in NEST are built on this decoupling:

1. Every neuron is updated in steps of the simulation resolution, but always for *dmin* time in one go, as to keep neurons in cache as long as possible.
2. MPI processes only communicate in intervals of *dmin* as to minimize communication costs.

These optimizations mean that the sizes of spike buffers in nodes and the buffers for inter-process communication depend on *dmin*+*dmax* as histories that long back have to be kept. NEST will figure out the correct value of *dmin* and *dmax* based on the actual delays used during connection setup. Their actual values can be retrieved using `GetKernelStatus`:

```
GetKernelStatus("min_delay")    # (A corresponding entry exists for max_delay)
```

Setting *dmin* and *dmax* manually

In linear simulation scripts that build a network, simulate it, carry out some post-processing and exit, the user does not have to worry about the delay extrema *dmin* and *dmax* as they are set automatically to the correct values. However, NEST also allows subsequent calls to`Simulate`, which only work correctly if the content of the spike buffers is preserved over the simulations.

As mentioned above, the size of that buffer depends on *dmin*+*dmax* and the easiest way to assert its integrity is to not change its size after initialization. Thus, we freeze the delay extrema after the first call to `Simulate`. To still allow adding new connections inbetween calls to `Simulate`, the required boundaries of delays can be set manually using `SetKernelStatus` (Please note that the delay extrema are set as properties of the synapse model):

```
SetDefaults("static_synapse", {"min_delay": 0.5, "max_delay": 2.5})
```

These settings should be used with care, though: setting the delay extrema too wide without need leads to decreased performance due to more update calls and communication cycles (small *dmin*), or increased memory consumption of NEST (large *dmax*).

7.5.3 Spike generation and precision

A neuron fires a spike when the membrane potential is above threshold at the end of an update interval (i.e., a multiple of the simulation resolution). For most models, the membrane potential is then reset to some fixed value and clamped to that value during the refractory time. This means that the last membrane potential value at the last time step before the spike can vary, while the potential right after the step will usually be the reset potential (some models may deviate from this). This also means that the membrane potential recording will never show values above the threshold. The time of the spike is always the time at *the end of the interval* during which the threshold was crossed.

NEST also has a some models that determine the precise time of the threshold crossing during the interval. Please see the documentation on precise spike time neurons for details about neuron update in continuous time and the documentation on connection management for how to set the delay when creating synapses.

7.6 Simulations with gap junctions

Note: This documentation describes the usage of gap junctions in NEST 2.12. A documentation for NEST 2.10 can be found in Hahne et al. 2016. It is however recommended to use NEST 2.12 (or later), due to several improvements in terms of usability.

7.6.1 Introduction

Simulations with gap junctions are supported by the Hodgkin-Huxley neuron model `hh_psc_alpha_gap`. The synapse model to create a gap-junction connection is named `gap_junction`. Unlike chemical synapses gap junctions are bidirectional connections. In order to create **one** accurate gap-junction connection **two** NEST connections are required: For each created connection a second connection with the exact same parameters in the opposite direction is required. NEST provides the possibility to create both connections with a single call to `nest.Connect` via the `make_symmetric` flag (default value: `False`) of the connection dictionary:

```
import nest

a = nest.Create('hh_psc_alpha_gap')
b = nest.Create('hh_psc_alpha_gap')
# Create gap junction between neurons a and b
nest.Connect(a, b, {'rule': 'one_to_one', 'make_symmetric': True},
                   {'model': 'gap_junction', 'weight': 0.5})
```

In this case the reverse connection is created internally. In order to prevent the creation of incomplete or non-symmetrical gap junctions the creation of gap junctions is restricted to

- `one_to_one` connections with `'make_symmetric': True`
- `all_to_all` connections with equal source and target populations and default or scalar parameters

7.6.2 Create random connections

NEST random connection rules like `fixed_total_number`, `fixed_indegree` etc. cannot be employed for the creation of gap junctions. Therefore random connections have to be created on the Python level with e.g. the `random` module of the Python Standard Library:

```
import nest
import random
import numpy as np
```

(continues on next page)

(continued from previous page)

```
# total number of neurons
n_neuron = 100

# total number of gap junctions
n_gap_junction = 3000

n = nest.Create('hh_psc_alpha_gap', n_neuron)

random.seed(0)

# draw n_gap_junction pairs of random samples from the list of all
# neurons and reshaped data into two corresponding lists of neurons
m = np.transpose(
    [random.sample(n, 2) for _ in range(n_gap_junction)])

# connect obtained lists of neurons both ways
nest.Connect(m[0], m[1],
             {'rule': 'one_to_one', 'make_symmetric': True},
             {'model': 'gap_junction', 'weight': 0.5})
```

As each gap junction contributes to the total number of gap-junction connections of two neurons, it is hardly possible to create networks with a fixed number of gap junctions per neuron. With the above script it is however possible to control the approximate number of gap junctions per neuron. E.g. if one desires `gap_per_neuron = 60` the total number of gap junctions should be chosen as `n_gap_junction = n_neuron * gap_per_neuron / 2`.

Note: The (necessary) drawback of creating the random connections on the Python level is the serialization of the connection procedure in terms of computation time and memory in distributed simulations. Each compute node participating in the simulation needs to draw the identical full set of random numbers and temporarily represent the total connectivity in variable `m`. Therefore it is advisable to use the internal random connection rules of NEST for the creation of connections whenever possible. For more details see Hahne et al. 2016.

7.6.3 Adjust settings of iterative solution scheme

For simulations with gap junctions NEST uses an iterative solution scheme based on a numerical method called Jacobi waveform relaxation. The default settings of the iterative method are based on numerical results, benchmarks and previous experience with gap-junction simulations (see Hahne et al. 2015) and should only be changed with proper knowledge of the method. In general the following parameters can be set via kernel parameters:

```
nest.SetKernelStatus({'use_wfr': True,
                      'wfr_comm_interval': 1.0,
                      'wfr_tol': 0.0001,
                      'wfr_max_iterations': 15,
                      'wfr_interpolation_order': 3})
```

For a detailed description of the parameters and their function see (Hahne et al. 2016, Table 2).

7.7 Simulations with precise spike times

The simulation resolution *h* and the minimum synaptic transmission delay *dmin* define the two major time intervals of the scheduling and simulation flow of NEST: neurons update their state variables in steps of *h*, whereas spikes are communicated and delivered to their targets in steps of *dmin*, where *dmin* is a multiple of *h*.

Traditionally, spikes are constrained to the simulation grid such that neurons can propagate their state variables for an entire *h*-step without interruption by incoming spikes. This enables faster simulations of neurons with linear subthreshold dynamics as a precomputed propagator matrix for a time step of fixed size *h* can be employed (Rotter & Diesmann, 1999).

Neurons buffer the incoming spikes until they become due, where spikes can be lumped together provided that the corresponding synapses have the same post-synaptic dynamics. Within a *dmin*-interval, each neuron independently proceeds in steps of *h*: it retrieves the inputs that are due in the current time step from its spike buffers and updates its state variables such as the membrane potential.

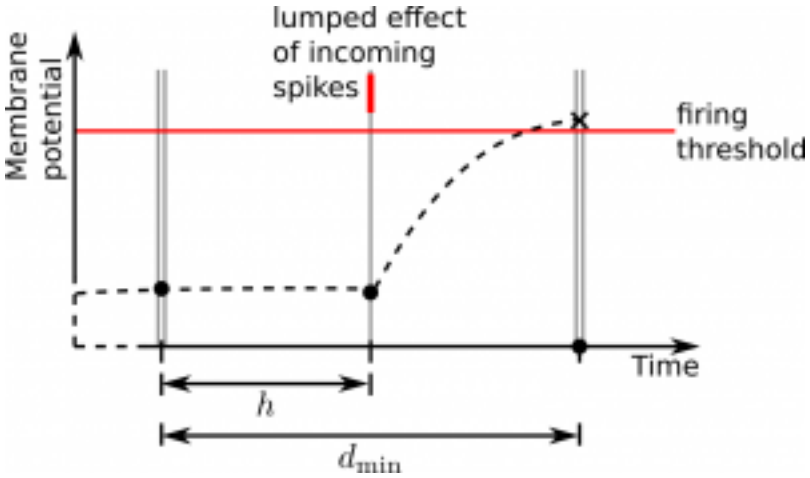


Fig. 7.6: Propagation of membrane potential in case of grid-constrained spiking. Filled dots indicate update of membrane potential; black cross indicates detection of threshold crossing. As visual guidance, dashed black curves indicate time course of membrane potential. For simplicity, d_min=2h.

If after an update the membrane potential is above the firing threshold, the neuron emits a spike and resets its membrane potential. Due to time discretization both spike and reset happen at the right border of the *h*-step in which the threshold crossing occurred; the spike is time stamped accordingly.

NEST enables also simulations with precise spike times, which are represented by an integer time stamp and a double precision offset. As the incoming spikes divide the *h*-steps into substeps, a neuron needs to update its state variables for each substep.

If after an update the membrane potential is above the firing threshold, the neuron determines the precise offset of the outgoing spike with respect to the next point on the time grid. This grid point marks the spike's time stamp. The neuron then emits the spike and resets its membrane potential.

7.7.1 Models with precise spike times in NEST

`poisson_generator_ps` creates Poissonian spike trains, where spike times have an integer time stamp and a double precision offset. It is hence dedicated to simulations with precise spike times. The device can also be connected to grid-constrained neuron models, which only use the time stamps of the spikes and ignore their offsets. However, spike generation with `poisson_generator_ps` is less efficient than with its grid-constrained counterpart `poisson_generator`.

`parrot_neuron_ps` repeats the incoming spikes just as its grid-constrained counterpart `parrot_neuron` but it is able to represent precise spike times.

`iaf_psc_delta_canon` is an integrate-and-fire neuron model with delta-shaped post-synaptic currents that employs precise spike times; its grid-constrained counterpart is `iaf_psc_delta`. In this model the precise location of an outgoing spike is determined analytically.

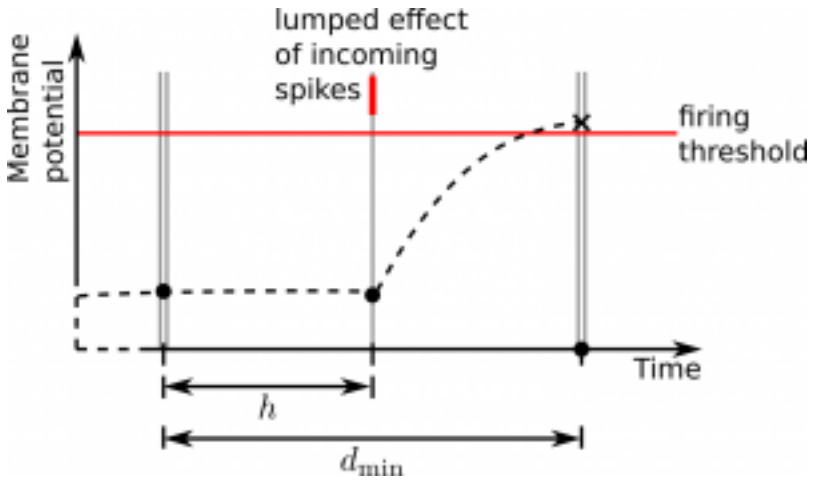


Fig. 7.7: Propagation of membrane potential in case of off-grid spiking. Dashed red line indicates precise time of threshold crossing.

`iaf_psc_alpha_canon` and `iaf_psc_alpha_presc` are integrate-and-fire neuron models with alpha-shaped post-synaptic currents that employ precise spike times; their grid-constrained counterpart is `iaf_psc_alpha`. The neuron models have been developed in the context of Morrison et al. (2007). As both models employ interpolation in order to determine the precise location of an outgoing spike, the achieved precision depends on the simulation resolution *h*. The models differ in the way they process incoming spikes, which also affects the attained precision (see Morrison et al. (2007) for details).

`iaf_psc_exp_ps` is an integrate-and-fire neuron model with exponentially shaped post-synaptic currents that employs precise spike times; its grid-constrained counterpart is `iaf_psc_exp`. It has been developed in the context of Hanuschkin et al. (2010), which is a continuation of the work presented in Morrison et al. (2007). As the neuron model employs an iterative search in order to determine the precise location of an outgoing spike, the achieved precision does not depend on the simulation resolution h. The model can also be used through the PyNN interface.

The source code of these models is in the *precise* module of NEST.

7.7.2 Questions and answers about precise neurons

During the review process of the above mentioned papers, we came up with a list of questions and answers pertaining to the implementation and usage of precise spiking neurons. This list can be found here.

7.8 Using NEST with MUSIC

7.8.1 Introduction

NEST supports the MUSIC interface, a standard by the INCF, which allows the transmission of data between applications at runtime[1]. It can be used to couple NEST with other simulators, with applications for stimulus generation and data analysis and visualization and with custom applications that also use the MUSIC interface.

Basically, all communication with MUSIC is mediated via *proxies* that receive/send data from/to remote applications using MUSIC. Different proxies are used for the different types of data. At the moment, NEST supports sending and receiving spike events and receiving continuous data and string messages.

[1] Djurfeldt M, et al. 2010. Run-time interoperability between neuronal simulators based on the MUSIC framework. Neuroinformatics, 8. doi:10.1007/s12021-010-9064-z*.

You can find the installation instructions for MUSIC on their Github Page: INCF/MUSIC

Reference

7.8.2 Sending and receiving spike events

A minimal example for the exchange of spikes between two independent instances of NEST is given in the example `examples/nest/music/minimalmusicsetup.music`.

It sends spikes using the `music_event_out_proxy` script and receives the spikes using a `music_event_in_proxy`.

```
stoptime=0.01

[from]
  binary=./minimalmusicsetup_sendnest.py
  np=1

[to]
  binary=./minimalmusicsetup_receivenest.py
  np=1

from.spikes_out -> to.spikes_in [1]
```

This configuration file sets up two applications, `from` and `to`, which are both instances of NEST. The first runs a script to send spike events on the MUSIC port `spikes_out` to the second, which receives the events on the port `spikes_in`. The width of the port is 1.

The content of `minimalmusicsetup_sendnest.py` is contained in the following listing.

First, we import nest and set up a check to ensure MUSIC is installed before continuing.

```
import nest

nest.sli_run("statusdict/have_music ::")
if not nest.spp():
    import sys

    print("NEST was not compiled with support for MUSIC, not running.")
    sys.exit()

nest.set_verbosity("M_ERROR")
```

Next we create a `spike_generator` and set the spike times. We then create our neuron model (`iaf_psc_alpha`) and connect the neuron with the spike generator.

```
sg = nest.Create('spike_generator')
nest.SetStatus(sg, {'spike_times': [1.0, 1.5, 2.0]})

n = nest.Create('iaf_psc_alpha')

nest.Connect(sg, n, 'one_to_one', {'weight': 750.0, 'delay': 1.0})
```

We then create a voltmeter, which will measure the membrane potenial, and connect it with the neuron.

```
vm = nest.Create('voltmeter')
nest.SetStatus(vm, {'to_memory': False, 'to_screen': True})
```

(continues on next page)

(continued from previous page)

```
nest.Connect(vm, n)
```

Finally, we create a `music_event_out_proxy`, which forwards the spikes it receives directly to the MUSIC event output port `spikes_out`. The spike generator is connected to the `music_event_out_proxy` on channel 0 and the network is simulated for 10 milliseconds.

```
meop = nest.Create('music_event_out_proxy')
nest.SetStatus(meop, {'port_name': 'spikes_out'})

nest.Connect(sg, meop, 'one_to_one', {'music_channel': 0})

nest.Simulate(10)
```

The next listing contains the content of `minimalmusicsetup_receivenest.py`, which is set up similarly to the above script, but without the spike generator.

```
import nest

nest.sli_run("statusdict/have_music ::")
if not nest.spp():
    import sys

    print("NEST was not compiled with support for MUSIC, not running.")
    sys.exit()

nest.set_verbosity("M_ERROR")

meip = nest.Create('music_event_in_proxy')
nest.SetStatus(meip, {'port_name': 'spikes_in', 'music_channel': 0})

n = nest.Create('iaf_psc_alpha')

nest.Connect(meip, n, 'one_to_one', {'weight': 750.0})

vm = nest.Create('voltmeter')
nest.SetStatus(vm, {'to_memory': False, 'to_screen': True})

nest.Connect(vm, n)

nest.Simulate(10)
```

Running the example using `mpirun -np 2 music minimalmusicsetup.music` yields the following output, which shows that the neurons in both processes receive the same input from the `spike_generator` in the first NEST process and show the same membrane potential trace.

```
NEST v1.9.svn (C) 1995-2008 The NEST Initiative
-70
-70
-68.1559
-61.9174
-70
-70
-70
-65.2054
-62.1583
```

(continues on next page)

(continued from previous page)

```
NEST v1.9.svn (C) 1995-2008 The NEST Initiative
-70
-70
-68.1559
-61.9174
-70
-70
-70
-65.2054
-62.1583
```

7.8.3 Receiving string messages

Currently, NEST is only able to receive messages, and unable to send string messages. We thus use MUSIC's `messagesource` program for the generation of messages in the following example. The configuration file (`msgtest.music`) is shown below

```
stoptime=1.0
np=1
[from]
  binary=messagesource
  args=messages
[to]
  binary=./msgtest.py

from.out -> to.msgdata [0]
```

This configuration file connects MUSIC's `messagesource` program to the port `msgdata` of a NEST instance. The `messagesource` program needs a data file, which contains the messages and the corresponding time stamps. For this example, we use the data file, `messages0.dat`:

```
0.3     Hello
0.7     !
```

Note: In MUSIC, the default unit for time is seconds for the specification of times, while NEST uses miliseconds.

The script that sets up the receiving side (`msgtest.py`) of the example is shown in the following script.

We first import NEST and create an instance of the `music_message_in_proxy`. We then set the name of the port it listens on to `msgdata`. The network is simulated in steps of 10 ms.

```
#!/usr/bin/python

import nest

mmip = nest.Create ('music_message_in_proxy')
nest.SetStatus (mmip, {'port_name' : 'msgdata'})

# Simulate and get message data with a granularity of 10 ms:
time = 0
while time < 1000:
    nest.Simulate (10)
```

(continues on next page)

(continued from previous page)

```
    data = nest.GetStatus(mmip, 'data')
    print data
    time += 10
```

We then run the example using

```
mpirun -np 2 music msgtest.music
```

which yields the following output:

```
           -- N E S T 2 beta --
          Neural Simulation Tool
  Copyright 1995-2009 The NEST Initiative
   Version 1.9-svn Sep 22 2010 16:50:01

This program is provided AS IS and comes with
NO WARRANTY. See the file LICENSE for details.

Problems or suggestions?
  Website     : <a class="external free" href="http://www.nest-initiative.org" rel=
↪"nofollow">http://www.nest-initiative.org</a>
  Mailing list: nest_user@nest-initiative.org

Type 'nest.help()' to find out more about NEST.

Sep 23 16:09:12 Simulate [Info]:
    Simulating 10 ms.

Sep 23 16:09:12 Scheduler::prepare_nodes [Info]:
    Please wait. Preparing elements.

Sep 23 16:09:12 music_message_in_proxy::calibrate() [Info]:
    Mapping MUSIC input port 'msgdata' with width=0 and acceptable latency=0
    ms.

Sep 23 16:09:12 Scheduler::prepare_nodes [Info]:
    Simulating 1 nodes.

Sep 23 16:09:12 Scheduler::resume [Info]:
    Entering MUSIC runtime with tick = 0.1 ms

Sep 23 16:09:12 Scheduler::resume [Info]:
    Simulation finished.
[{'messages': [], 'message_times': array([], dtype=float64)}]

:

Sep 23 16:13:36 Simulate [Info]:
    Simulating 10 ms.

Sep 23 16:13:36 Scheduler::prepare_nodes [Info]:
    Please wait. Preparing elements.

Sep 23 16:13:36 Scheduler::prepare_nodes [Info]:
    Simulating 1 nodes.

Sep 23 16:13:36 Scheduler::resume [Info]:
```

(continues on next page)

(continued from previous page)

```
    Simulation finished.
[{'messages': ['Hello', '!'], 'message_times': array([ 300.,  700.])}]
```

7.8.4 Receiving continuous data

As in the case of string message, NEST currently only supports receiving continuous data, but not sending. This means that we have to use another of MUSIC's test programs to generate the data for us. This time, we use `constsource`, which generates a sequence of numbers form 0 to w, where w is the width of the port. The MUSIC configuration file (`conttest.music`) is shown in the following listing:

```
stoptime=0.01

[from]
  binary=./minimalmusicsetup_sendnest.py
  np=1

[to]
  binary=./minimalmusicsetup_receivenest.py
  np=1

from.spikes_out -> to.spikes_in [1]
```

```
stoptime=1.0
[from]
np=1
binary=./cont_out.py
[to]
np=1
binary=./cont_in.py

from.cont_out -> to.cont_in [10]
```

The receiving side is again implemented using a PyNEST script (`conttest.py`). We first import the NEST and create an instance of the `music_cont_in_proxy`. we set the name of the port it listens on to `msgdata`. We then simulate the network in steps of 10 ms.

```
#!/usr/bin/python

import nest

mcip = nest.Create('music_cont_in_proxy')
nest.SetStatus(mcip, {'port_name' : 'cont_in'})

# Simulate and get vector data with a granularity of 10 ms:
time = 0
while time < 1000:
   nest.Simulate (10)
   data = nest.GetStatus (mcip, 'data')
   print data
   time += 10
```

The example is run using

```
mpirun -np 2 music conttest.music
```

which yields the following output:

```
           -- N E S T 2 beta --
          Neural Simulation Tool
  Copyright 1995-2009 The NEST Initiative
   Version 1.9-svn Sep 22 2010 16:50:01

This program is provided AS IS and comes with
NO WARRANTY. See the file LICENSE for details.

Problems or suggestions?
  Website     : <a class="external free" href="http://www.nest-initiative.org" rel=
↪"nofollow">http://www.nest-initiative.org</a>
  Mailing list: nest_user@nest-initiative.org

Type 'nest.help()' to find out more about NEST.

Sep 23 16:49:09 Simulate [Info]:
    Simulating 10 ms.

Sep 23 16:49:09 Scheduler::prepare_nodes [Info]:
    Please wait. Preparing elements.

Sep 23 16:49:09 music_cont_in_proxy::calibrate() [Info]:
    Mapping MUSIC input port 'contdata' with width=10.

Sep 23 16:49:09 Scheduler::prepare_nodes [Info]:
    Simulating 1 nodes.

Sep 23 16:49:09 Scheduler::resume [Info]:
    Entering MUSIC runtime with tick = 0.1 ms

Sep 23 16:49:09 Scheduler::resume [Info]:
    Simulation finished.
[array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])]

:

Sep 23 16:47:24 Simulate [Info]:
    Simulating 10 ms.

Sep 23 16:47:24 Scheduler::prepare_nodes [Info]:
    Please wait. Preparing elements.

Sep 23 16:47:24 Scheduler::prepare_nodes [Info]:
    Simulating 1 nodes.

Sep 23 16:47:24 Scheduler::resume [Info]:
    Simulation finished.
[array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])]
```

CHAPTER 8

Getting Help

8.1 Have a specific question or problem with NEST?

- Check out the FAQs for common issues.

If your question is not on there, ask our *Mailing List*.

8.2 Getting help on the command line interface

- The `helpdesk()` command will launch the documentation pages on your browser. See *Set up the integrated helpdesk* to specify the browser of your choice.
- To access the High-level Python API reference material you can use the commands:

```
# list all functions and attributes
dir(nest)

# Get docstring for function in python
help('nest.FunctionName')

# or in ipython
nest.FunctionName?
```

- To access a specific C++ or SLI reference page for an object, command or parameter you can use the command:

```
nest.help('name')
```

8.2.1 Model Information

- To get a complete list of the models available in NEST type:

```
nest.Models()
```

  - To get a list of only neuron models use:

```
nest.Models(mtype='nodes', sel=None)
```

  - To get a list of only synapse models use:

```
nest.Models(mtype='synapses', sel=None)
```

- To get details on model parameters and usage use:

```
nest.help('model_name')
```

8.3 Set up the integrated helpdesk

The command `helpdesk` needs to know which browser to launch in order to display the help pages. The browser is set as an option of `helpdesk`. Please see the file `~/.nestrc` for an example setting `firefox` as browser. Please note that the command `helpdesk` does not work if you have compiled NEST with MPI support, but you have to enter the address of the helpdesk (`file://$PREFIX/share/doc/nest`) manually into the browser. Please replace `$PREFIX` with the prefix you chose during the configuration of NEST. If you did not explicitly specify one, it is most likely set to `/usr` or `/usr/local` depending on what system you use.

CHAPTER 9

Reference Material

- Here you can find the PyNEST APIs
- The Command Index contains a list of all SLI and C++ related reference material.

CHAPTER 10

NEST Community

10.1 Mailing List

The NEST User Mailing list is intended to be a forum for questions on the usage of NEST, the exchange of code and general discussions about NEST. The philosophy is that all users profit by sharing their experience. All NEST core developers are subscribed to this list and will participate in the discussions as far as time allows.

Subscription http://mail.nest-initiative.org/cgi-bin/mailman/listinfo/nest_user

Archive (only subscribed users) http://mail.nest-initiative.org/cgi-bin/mailman/listinfo/nest_user

10.2 Contributing to NEST

NEST draws its strength from the many people that use and improve it. We are happy to consider your contributions (e.g. own models, bug or documentation fixes) for addition to the official version of NEST.

Please see the NEST developer space for information about the development workflow of NEST and for how to create a fork of our Git repository and make pull requests against it.

10.3 Reporting bugs

The primary place to go to if you find an error is the GitHub issue tracker for NEST. Please take the time to check if your issue has already been reported there before creating a new one.

To make it easier for the developers to understand and solve the problem, please include the following information in your bug report, if applicable:

1. Release version or Git revision of NEST.
2. Platform and operating system version.
3. The file `config.log` from the build directory and the output of the configure script.

4. The contents of the `reports` directory in the build directory.
5. A detailed transcript of how you got the error.
6. A minimal script to reproduce the error.

10.4 Become a NEST member

If you would like to be actively involved in the NEST Initiative and support its goals, please see our member page.

CHAPTER 11

License

11.1 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

11.2 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

11.3 GNU GENERAL PUBLIC LICENSE

11.3.1 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the

terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for alls it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

11.3.2 NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

11.3.3 END OF TERMS AND CONDITIONS