NEST Simulator Documentation

Release 1.0.0

NEST Developer Community

Jun 28, 2019
## CONTENTS

1 Welcome to the NEST simulator documentation! .................................................. 1
   1.1 How the documentation is organized ....................................................... 1
   1.2 Contribute ................................................................................................. 1

2 Download NEST ............................................................................................... 3
   2.1 Download the current version of NEST here: ............................................ 3
   2.2 Download the NEST Live Media for Virtual Machines ............................. 3
   2.3 Previous Releases ...................................................................................... 3

3 Installation Instructions ..................................................................................... 5
   3.1 Install NEST with Conda (Linux and macOS) - BETA ............................... 5
   3.2 Ubuntu/Debian Installation ....................................................................... 6
   3.3 Installation on macOS ............................................................................... 9
   3.4 High Performance Computer Systems Installation ................................. 11
   3.5 NEST LIVE MEDIA Installation .............................................................. 14
   3.6 Configuration Options ............................................................................. 15
   3.7 Compiling for Apple OS/macOS .............................................................. 18

4 Getting Started ................................................................................................ 21
   4.1 A quick overview of simulating neural networks .................................. 21
   4.2 How do I use NEST? ................................................................................ 21
   4.3 Physical units in NEST ............................................................................. 22
   4.4 Next Steps ................................................................................................. 23

5 Tutorials ............................................................................................................. 25
   5.1 Part 1: Neurons and simple neural networks ........................................... 25
   5.2 Part 2: Populations of neurons ................................................................. 35
   5.3 Part 3: Connecting networks with synapses ......................................... 40
   5.4 Part 4: Topologically structured networks ............................................ 45
   5.5 Introduction to the MUSIC Interface ...................................................... 51
   5.6 Connect two NEST simulations using MUSIC ....................................... 54
   5.7 MUSIC Connections in C++ and Python ............................................. 57
   5.8 The pymusic interface .......................................................................... 62
   5.9 Practical Tips ......................................................................................... 64
   5.10 Video Tutorial Series ............................................................................ 65

6 Model Directory ............................................................................................... 67

7 Example Neural Networks in NEST ................................................................. 69

8 Topology ......................................................................................................... 71
8.1 Topology User Manual ................................................................. 71
8.2 Topology Tutorial with Hill Tononi Model ................................ 106
8.3 Examples using Topology ........................................................... 120

9 Guides .................................................. 129
9.1 Connection Management ....................................................... 129
9.2 Running simulations ............................................................... 140
9.3 Guide to parallel computing ..................................................... 144
9.4 Random numbers ................................................................. 149
9.5 Analog recording with multimeter ............................................. 156
9.6 Simulations with gap junctions ............................................... 158
9.7 Simulations with precise spike times ....................................... 159
9.8 Using NEST with MUSIC ....................................................... 161

10 Getting Help ......................................................... 169
10.1 Have a specific question or problem with NEST? .................... 169
10.2 Getting help on the command line interface ............................ 169
10.3 Set up the integrated helpdesk ............................................... 170

11 Reference Material ..................................................... 171

12 NEST Community ...................................................... 173
12.1 Mailing List ................................................................. 173
12.2 Contributing to NEST ....................................................... 173
12.3 Reporting bugs .............................................................. 173
12.4 Become a NEST member ................................................... 173

13 License ......................................................... 175
13.1 GNU GENERAL PUBLIC LICENSE ...................................... 175
13.2 Preamble ................................................................. 175
13.3 GNU GENERAL PUBLIC LICENSE ...................................... 176

Index ................................................................. 179
WELCOME TO THE NEST SIMULATOR DOCUMENTATION!

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 your 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.
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
THREE

INSTALLATION INSTRUCTIONS

3.1 Install NEST with Conda (Linux and macOS) - BETA

The NEST simulator is now available as a package on conda-forge. If you don’t have Conda installed, we recommend you follow the instructions on Miniconda.

If you need to install NEST with additional configuration options, please see our install guide for Linux or install guide for macOS

Note: The conda-forge package of nest-simulator is still being tested, let us know if you come across any problems by submitting an issue on GitHub.

3.1.1 Install the latest conda-forge package for NEST

1. Create your conda environment and install NEST. We recommend that you create a dedicated environment for NEST, which should ensure there are no conflicts with previously installed packages.

IMPORTANT!

We strongly recommend that you install all programs you’ll need, (such as ipython or jupyter-lab) in the environment (ENVNAME) at the same time, by appending them to the following command. Installing packages later may override previously installed dependencies and potentially break packages! See managing environments in the Conda documentation for more information.

WITHOUT openmpi:

```bash
conda create --name ENVNAME -c conda-forge nest-simulator python
```

WITH openmpi:

```bash
conda create --name ENVNAME -c conda-forge nest-simulator=*mpi_openmpi* python
```

Where the syntax for this install follows the pattern: nest-simulator=<version>=<build_string>

2. Activate your environment:

```bash
conda activate ENVNAME
```
NEST Simulator Documentation, Release 1.0.0

3. Once installation is complete, you can open up Python or IPython in the terminal and import nest:

```
python
import nest
```

If installation was successful, you should see the NEST splash screen:

The installation is now complete!

### 3.1.2 Next Steps

Once you have completed installation, take a look at our PyNEST tutorials page find out how to create your first simulation or checkout some of our example networks!

Source Code: https://github.com/conda-forge/nest-simulator-feedstock/
Anaconda cloud package: https://anaconda.org/conda-forge/nest-simulator
The conda-forge package was tested on macOS 10.14 (Mojave) and Ubuntu 18.04 (Bionic Beaver)

### 3.2 Ubuntu/Debian Installation

#### 3.2.1 Standard Installation

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

- If not already installed on your system, the following packages are recommended (see also the Dependencies section)

```sh
sudo apt-get install -y 
build-essential 
cmake 
cython 
```

(continues on next page)
Unpack the tarball

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

Create a build directory:

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

Change to the build directory:

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

Configure NEST.

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

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

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

Compile and install NEST:

```bash
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.

**Important!**

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:

```bash
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.
### 3.2.2 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 libcurses (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.

### 3.2.3 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:

```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.
3.3 Installation on macOS

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.

3.3.1 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
```

3.3.2 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.

### 3.3.3 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.
3.4 High Performance Computer Systems Installation

3.4.1 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:

```bash
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.

3.4.2 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:

```bash
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:

```bash
-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 \ 
-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:

```bash
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
```

3.4.3 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:

```bash
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:

```bash
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:

```bash
cmake -DCMAKE_TOOLCHAIN_FILE=Platform/Fujitsu-Sparc64 \
    -DCMAKE_INSTALL_PREFIX:PATH=/install/path\n```

(continues on next page)
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"
```

### 3.5 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
```

### 3.5.1 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.

3.6 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.

3.6.1 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)

3.6.2 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>
```

3.6.3 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]
```

3.6. Configuration Options
Add user modules:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Dexternal-modules=[OFF</td>
<td>&lt;list;of;modules&gt;]</td>
</tr>
</tbody>
</table>

Connect NEST with external projects:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Dwith-libneurosim=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-music=[OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>

Change parallelization scheme:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Dwith-mpi=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-openmp=[OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>

Set default libraries:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Dwith-gsl=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-readline=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-ltdl=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-python=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dcythonize-pynest=[OFF</td>
<td>ON]</td>
</tr>
</tbody>
</table>

Change compilation behavior:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Dstatic-libraries=[OFF</td>
<td>ON]</td>
</tr>
<tr>
<td>-Dwith-optimize=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-warning=[OFF</td>
<td>ON</td>
</tr>
<tr>
<td>-Dwith-debug=[OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>
3.6.4 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.

3.6.5 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.

3.6.6 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.

3.6.7 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.
3.6.8 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>
```

3.7 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 prerequisites. 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 https://www.nest-simulator.org/installation.

3.7.1 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.

**When compiling with the Intel compiler:** To ensure that computations obey the IEEE754 standard for floating point numerics, the `-fp-model strict` flag is used by default, but can be overridden with `-Dwith-intel-compiler-flags="<intel-flags>"`

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

Please choose on which system you want to install NEST:

- **BETA** Conda installation for Linux and macOS
- Ubuntu/Debian installation
- Standard macOS installation
- High Performance Computer Systems
- NEST Live Media for Virtual Machines

For more information regarding installation see our Configuration Options.

**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.
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:

```python
# 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
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 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 an interface to Python\(^2\). Figure 5.1.1 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

---

\(^1\) Gewaltig MO. and Diesmann M. 2007. NEural Simulation Tool. 2(4):1430.
Figure 5.1.1: 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.
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.

```python
import nest
```

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

```python
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.

```python
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`.

```python
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.

The documentation of the NEST Simulator shows how to query and modify properties of neurons. The `nest.GetStatus(neuron)` function prints out the corresponding dictionary in the Python console. Many 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`:

```python
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.

```python
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:

```python
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`.

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

We now create a `spike_detector`, 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).

```python
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.

```python
nest.Connect(multimeter, neuron)
nest.Connect(neuron, spikedetector)
```
Figure 5.1.2: Membrane potential of integrate-and-fire neuron with constant input current.

Figure 5.1.3: Spikes of the neuron.
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.

```python
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.

```python
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.

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`.

```python
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.

```python
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 Figure 5.1.2 and Figure 5.1.3. 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:

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

now connect this newly created neuron to the multimeter:

```python
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.

```python
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.

```python
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:

```python
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.

```python
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 Figure 5.1.4 and Figure 5.1.5.

In the next part of the introduction (Part 2: Populations of neurons) we will look at more methods for connecting many neurons at once.
Figure 5.1.4: Membrane potential of integrate-and-fire neuron with Poisson noise as input.

Figure 5.1.5: Spikes of the neuron with noise.
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.

```python
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`.

```python
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:

```python
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 Figure 5.1.6.

![Figure 5.1.6: Postsynaptic potentials in neuron2 evoked by the spikes of neuron1](image.png)
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 parameters (such as "mu" and "sigma").

Simulation control

- \texttt{Simulate(t)} Simulate the network for \texttt{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: \texttt{pynest/examples/}.

5.2.2 Creating parameterised populations of nodes

In the previous handout, we introduced the function \texttt{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 \texttt{n}, which gives the number of nodes to be created (default: 1) and \texttt{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 \texttt{Create()}:

\begin{verbatim}
ndict = {"I_e": 200.0, "tau_m": 20.0}
neuronpop = nest.Create("iaf_psc_alpha", 100, params=ndict)
\end{verbatim}
The variable `neuronpop` is a tuple 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:

```python
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()`:

```python
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:

```python
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()`.

```python
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:

```python
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:
NEST Simulator Documentation, Release 1.0.0

\[ Vth = -55. \]
\[ Vrest = -70. \]

\[
\text{for neuron in epop1:} \\
\quad \text{nest.SetStatus([neuron], \{"V_m": Vrest+\(Vth-Vrest\)*numpy.random.rand()}))
\]

However, \texttt{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:

\[
dVms = \left[ \{"V_m": Vrest+(Vth-Vrest)*\text{numpy.random.rand()} \} \text{ for } x \text{ in epop1} \right]
\]

\[
\text{nest.SetStatus(epop1, dVms)}
\]

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:

\[
Vms = Vrest+(Vth-Vrest)*\text{numpy.random.rand(len(epop1))}
\]

\[
\text{nest.SetStatus(epop1, "V_m", Vms)}
\]

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.

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

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

\[
\text{nest.Connect(pop1, pop2, syn_spec=\{"weight":20.0\})}
\]

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

\[
\text{nest.Connect(pop1, pop2, \"one_to_one", syn_spec=\{"weight":20.0, \"delay":1.0\})}
\]

Finally, the multimeters are connected using the default rule

\[
\text{nest.Connect(multimeter, pop2)}
\]

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 \texttt{rule} and the key for parameters associated to the rule. Please see \textit{Connection management} for an illustrated guide to the usage of \texttt{Connect}.

5.2. Part 2: Populations of neurons 37
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:

```python
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:

```python
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:

```python
recdict = {"to_memory" : False, "to_file" : True, "label" : "epop_mp"}
mml = 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 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)`:}

```python
nest.SetDefaults("stdp_synapse", {"tau_plus": 15.0})
```
Any synapse generated from this model will then have all the standard parameters except for the \( \text{tau}_{\text{plus}} \), which will have the value given above.

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

\begin{verbatim}
nest.CopyModel("stdp_synapse","layer1_stdp_synapse",{"Wmax": 90.0})
\end{verbatim}

Now \texttt{layer1_stdp_synapse} will appear in the list returned by \texttt{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 \texttt{GetDefaults()} and \texttt{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:

\begin{verbatim}
nest.Create("iaf_psc_alpha", params={"tau_{minus}": 30.0})
\end{verbatim}

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.

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

If no synapse model is given, connections are made using the model \texttt{static_synapse}.

### 5.3.4 Distributing synapse parameters

The synapse parameters are specified in the synapse dictionary which is passed to the \texttt{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 \texttt{distribution} setting the target distribution of the parameters (for example \texttt{normal}). Optionally, parameters associated with the distribution can be set (for example \texttt{mu}). Here we show an example where the parameters \texttt{alpha} and \texttt{weight} of the stdp synapse are uniformly distributed.

\begin{verbatim}
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)
\end{verbatim}
Available distributions and associated parameters are described in Connection Management, the most common ones are:

<table>
<thead>
<tr>
<th>Distributions</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>mu, sigma</td>
</tr>
<tr>
<td>lognormal</td>
<td>mu, sigma</td>
</tr>
<tr>
<td>uniform</td>
<td>low, high</td>
</tr>
<tr>
<td>uniform_int</td>
<td>low, high</td>
</tr>
<tr>
<td>binomial</td>
<td>n, p</td>
</tr>
<tr>
<td>exponential</td>
<td>lambda</td>
</tr>
<tr>
<td>gamma</td>
<td>order, scale</td>
</tr>
<tr>
<td>poisson</td>
<td>lambda</td>
</tr>
</tbody>
</table>

5.3.5 Querying the synapses

The function `nest.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:

```python
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:

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

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

```python
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:

```python
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 `nest.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()`:

```python
combs = nest.GetConnections(epop1, synapse_model="stdp_synapse")
targets = nest.GetStatus(combs, "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 multimeters 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:

```python
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.
The best way to avoid this is to define a function:

```python
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:

```python
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 explicitly:

```python
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:

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

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

```python
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:

```python
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:

```python
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*
More advanced examples can be found at Example Networks, or have a look 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 visual-
   ization 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:

```python
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 be 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 is 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**:

```python
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
```

### 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**:

```python
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:
## 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:

```python
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.

```python
# Circular mask, gaussian kernel.
conn1 = {
    "connection_type":"divergent",
    "mask": {"circular":{"radius":0.75}},
    "kernel": {"gaussian":{"p_center":1.,"sigma":0.2}},
    "allow_autapses":False
}
```
# Rectangular mask, constant kernel, non-centered anchor

cconn2 = {  
    "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)

cconn3 = {  
    "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

cconn4 = {  
    "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": 0.5,
                "sigma": 0.25
            },
            "delays": {  
                "linear": {  
                    "c": 0.1,
                    "a": 0.2
                }
            }
        }
    },
    "allow_autapses": False
}
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection_type</td>
<td>Determines how nodes are selected when connections are made</td>
<td>convergent, divergent</td>
</tr>
<tr>
<td>mask</td>
<td>Spatially selected subset of neurons considered as (potential) targets</td>
<td>circular, rectangular, doughnut, grid</td>
</tr>
<tr>
<td>kernel</td>
<td>Function that determines the likelihood of a neuron being chosen as a target. Can be distance-dependent or -independent.</td>
<td>constant, uniform, linear, gaussian, exponential, gaussian2D</td>
</tr>
<tr>
<td>weights</td>
<td>Distribution of weight values of connections. Can be distance-dependent or -independent. <strong>NB:</strong> this value overrides any value currently used by synapse_model, and therefore unless defined will default to 1.</td>
<td>constant, uniform, linear, gaussian, exponential</td>
</tr>
<tr>
<td>delays</td>
<td>Distribution of delay values for connections. Can be distance-dependent or -independent. <strong>NB:</strong> like weights, this value overrides any value currently used by synapse_model!</td>
<td>constant, uniform, linear, gaussian, exponential</td>
</tr>
<tr>
<td>synapse_model</td>
<td>Define the type of synapse model to be included.</td>
<td>any synapse model included in nest. Models(), or currently user-defined</td>
</tr>
<tr>
<td>sources</td>
<td>Defines the sources (presynaptic) neurons for this connection.</td>
<td>any neuron label that is currently user-defined</td>
</tr>
<tr>
<td>targets</td>
<td>Defines the target (postsynaptic) neurons for this connection.</td>
<td>any neuron label that is currently user-defined</td>
</tr>
<tr>
<td>number_of_connections</td>
<td>Fixes the number of connections that this neuron is to send, ensuring we have a fixed out-degree distribution.</td>
<td>int</td>
</tr>
<tr>
<td>allow_multipapses</td>
<td>Whether we want to have multiple connections between the same source-target pair, or ensure unique connections.</td>
<td>boolean</td>
</tr>
<tr>
<td>allow_selfapses</td>
<td>Whether we want to allow a neuron to connect to itself</td>
<td>boolean</td>
</tr>
</tbody>
</table>

### 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():

```python
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:

```python
# 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:
```
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`* [https://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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nest.GetNodes(layer)</code></td>
<td>Returns GIDs of layer elements: either nodes or top-level subnets (for composite)</td>
</tr>
<tr>
<td><code>nest.GetLeaves(layer)</code></td>
<td>Returns GIDs of leaves of a structure, which is always going to be neurons rather subnets</td>
</tr>
<tr>
<td><code>topp.GetPosition(gids)</code></td>
<td>Returns position of elements specified in input</td>
</tr>
<tr>
<td><code>nest.GetStatus(layer, &quot;topology&quot;)</code></td>
<td>Returns the layer dictionary for a layer</td>
</tr>
</tbody>
</table>

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.

```
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
```
5.5.2 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.

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 Figure 5.5.1 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 \( t + d \) (Figure 5.5.1 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 (Figure 5.5.1 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
Figure 5.5.1: 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 $t_0 + d$.

A 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.

Figure 5.5.2: 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 Figure 5.5.2 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 Figure 5.5.2 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.

Figure 5.5.3: 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 Figure 5.5.3. 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,

5.5. Introduction to the MUSIC Interface
there is a single MUSIC event handler device $E_v$ 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


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 Figure 5.5.3 from the introduction to this tutorial.

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

```python
#!/usr/bin/env python
import nest
nest.SetKernelStatus({"overwrite_files": True})
n = nest.Create('iaf_neuron', 2, [{'I_e': 400.0}, {'I_e': 405.0}])
music_out = nest.Create('music_event_out_proxy', 1,
                        params = {'port_name': 'p_out'})
for i, n in enumerate(neurons):
    nest.Connect([n], music_out, "one_to_one", {'music_channel': i})
sdetector = nest.Create("spike_detector")
nest.SetStatus(sdetector, {"withgid": True, "withtime": True, "to_file": True, "label": "send", "file_extension": "spikes"})
nest.Connect(neurons, sdetector)
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 integrate-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.
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.

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:

```plaintext
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:

<table>
<thead>
<tr>
<th>send.spikes</th>
<th>receive.spikes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 26.100</td>
<td>4 28.100</td>
</tr>
<tr>
<td>1 27.800</td>
<td>3 29.800</td>
</tr>
<tr>
<td>2 54.200</td>
<td>4 56.200</td>
</tr>
<tr>
<td>1 57.600</td>
<td>3 59.600</td>
</tr>
<tr>
<td>2 82.300</td>
<td>4 84.300</td>
</tr>
<tr>
<td>1 87.400</td>
<td>3 89.400</td>
</tr>
<tr>
<td>2 110.40</td>
<td>4 112.40</td>
</tr>
<tr>
<td>1 117.20</td>
<td>3 119.20</td>
</tr>
</tbody>
</table>

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:

```
#!/usr/bin/python

import nest

mcip = nest.Create('music_cont_in_proxy')
nest.SetStatus(mcip, {'port_name' : 'contdata'})

while time < 1000:
    nest.Simulate (10)
    data = nest.GetStatus (mcip, 'data')
    print data
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:

```cpp
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;
    } else
        firstId += rest;

    // (continues on next page)
```
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:

```cpp
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.

```cpp
[ ... continued from above ... ]

// Start runtime phase
MUSIC::Runtime runtime = MUSIC::Runtime(setup, TICK);
double tickt = runtime.time();

while (tickt < simt) {
    for (int idx = firstId; idx<(firstId+nLocal); idx++) {
        // send poisson spikes to every channel.
        send_poisson(outdata, RATE*(idx+1), tickt, idx);
    }
    runtime.tick(); // Give control to MUSIC
tickt = runtime.time();
}
runtime.finalize(); // clean up and end
}

double frand(double rate) { return -(1./rate) * log(random()/double(RAND_MAX));}
```
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 \texttt{tick()} function so it can communicate, until the simulation time exceeds the end time.

\texttt{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.

\texttt{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 \texttt{runtime.finalize()}. Music will shut down the communications and clean up after itself before exiting.

```c
void send_poisson(MUSIC::EventOutputPort* outport,
        double rate, double tickt, int index) {
    double t = frand(rate);
    while (t<TICK) {
        outport -> insertEvent(tickt+t, MUSIC::GlobalIndex(index));
        t = t + frand(rate);
    }
}
```

5.7. MUSIC Connections in C++ and Python

```c
MPI::Intracomm comm;
FILE *fout;

struct eventtype {
    double t;
    int id;
};
std::queue <eventtype> in_q;

class InHandler : public MUSIC::EventHandlerGlobalIndex {
    public:
    void operator () (double t, MUSIC::GlobalIndex id) {
        struct eventtype ev = {t, (int)id};
        in_q.push(ev);
    }
};

int main(int argc, char **argv)
{
    MUSIC::Setup* setup = new MUSIC::Setup (argc, argv);
    comm = setup->communicator();
    double simt;
    setup->config ("simtime", &simt);
    MUSIC::EventInputPort *indata =
```

(continues on next page)
setup->publishEventInput("p_in");

InHandler inhandler;

[ ... get processes, rank and channel width as in send.cpp ... ]

char *fname;

int dummy = asprintf(&fname, "output-%d.spk", rank);
fout = fopen(fname, "w");

[ ... calculate channel allocation as in send.cpp ... ]

MUSIC::LinearIndex inindex(firstId, nLocal);
indata->map(&inindex, &inhandler, IN_LATENCY);

The setup phase for the receiving 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.

```c
int main(int argc, char **argv)
{
    MUSIC::Runtime runtime = MUSIC::Runtime(setup, TICK);
    double tickt = runtime.time();

    while (tickt < simt) {
        runtime.tick(); // Give control to MUSIC
        tickt = runtime.time();
        while (!in_q.empty()) {
            struct eventtype ev = in_q.front();
            fprintf (fout, "%d\t%.4f\n", ev.id, ev.t);
            in_q.pop();
        }
      fclose(fout);
      runtime.finalize();
    }
}
```

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`.

```plaintext
AC_INIT(simple, 1.0)
AC_PREREQ([2.59])
AM_INIT_AUTOMAKE([1.11 -Wall subdir-objects no-define foreign])
AC_LANG([C++])
AC_CONFIG_HEADERS([config.h])

# set OpenMPI compiler wrapper
AC_PROG_CXX(mpicxx)

AC_CHECK_LIB([music], [_init])
AC_CHECK_HEADER([music.hh])
AC_CONFIG_FILES([Makefile])
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.

```plaintext
bin_PROGRAMS = send recv
send_SOURCES = send.cpp
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:

```plaintext
simtime=1.0
[from]
   binary=./send
   np=2
[to]
```

(continues on next page)
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:

```bash
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 acquired 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.

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

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

```python
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 acceptable 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
\rightarrow
-mpirun -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 `binary=<something>` line for each process. The parser expects it and will complain (or crash) otherwise. Also, if you try to process command 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:
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:

```bash
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
Here you can find the list of all the models implemented in NEST for neurons, synapses, devices and topological networks.

We also have a list of user-contributed modules on Github for you to check out.

- **Neuron model categories**
  - iaf
  - binary
  - conductance
  - current-based
  - hodgkin_huxley
  - hill_tononi_neuron
  - rate
  - point_process
  - clopath_neuron
  - parrot
  - neurons

- **Synapse model categories**
  - stdp
  - stp
  - gapjunction
  - static
  - hill_tononi_synapse
  - cont_delay
  - inst_rate
  - clopath_synapse
  - synapses

- **Device categories**
  - generator
  - detector
– music
– devices

• Network models
  – Topology

• create_model
EXAMPLE NEURAL NETWORKS IN NEST

You can find a list of networks using NEST at the following link: https://www.nest-simulator.org/more-example-networks/
The Topology Module is designed to build complex, layered networks in NEST.

- If you’ve never used topologically-structured networks before, we recommend you check out the PyNEST tutorial on Topologically-structured Networks
- We also recommend another tutorial using the Hill Tononi model
- For a comprehensive guide into topology, please see our Topology User Manual
- We have a large selection of examples using topology, check them out in our Examples using Topology section.

### 8.1 Topology User Manual

The Topology Module provides the NEST simulator\(^1\) with a convenient interface for creating layers of neurons placed in space and connecting neurons in such layers with probabilities and properties depending on the relative placement of neurons. This permits the creation of complex networks with spatial structure.

This user manual provides an introduction to the functionality provided by the Topology Module. It is based exclusively on the PyNEST, the Python interface to NEST. NEST users using the SLI interface should be able to map instructions to corresponding SLI code. This manual is not meant as a comprehensive reference manual. Please consult the online documentation in PyNEST for details; where appropriate, that documentation also points to relevant SLI documentation.

This manual describes the Topology Module included with NEST 2.16; the user interface and behavior of the module has not changed significantly since NEST 2.2.

In the next chapter of this manual, we introduce Topology layers, which place neurons in space. In Chapter 3 we then describe how to connect layers with each other, before discussing in Chapter 4 how you can inspect and visualize Topology networks. Chapter 5 deals with the more advanced topic of extending the Topology module with custom kernel functions and masks provided by C++ classes in an extension module.

You will find the Python scripts used in the examples in this manual in the NEST source code directory under doc/topology/user_manual_scripts.

### 8.1.1 Limitations and Disclaimer

**Undocumented features** The Topology Module provides a number of undocumented features, which you may discover by browsing the code. These features are highly experimental and should not be used for simulations, as they have not been validated.

\(^1\) NEST is available under an open source license at www.nest-simulator.org.
Layers

The Topology Module (just Topology for short in the remainder of this document) organizes neuronal networks in layers. We will first illustrate how Topology places elements in simple layers, where each element is a single model neuron. Layers with composite elements are discussed in the following section.

We will illustrate the definition and use of layers using examples.

Topology distinguishes between two classes of layers:

- **grid-based layers** in which each element is placed at a location in a regular grid;
- **free layers** in which elements can be placed arbitrarily in the plane.

Grid-based layers allow for more efficient connection-generation under certain circumstances.

### 8.1.2 Grid-based Layers

We create a first, grid-based simple layer with the following commands:

```python
import nest.topology as tp
l = tp.CreateLayer({'rows': 5,
                    'columns': 5,
                    'elements': 'iaf_psc_alpha'})
```

The layer is shown in Figure 8.1.1. Note the following properties:

- The layer has five rows and five columns.
- The 'elements' entry of the dictionary passed to CreateLayer determines the elements of the layer. In this case, the layer contains iaf_psc_alpha neurons.
- The center of the layer is at the origin of the coordinate system, (0, 0).
- The extent or size of the layer is $1 \times 1$. This is the default size for layers. The extent is marked by the thin square in Figure 8.1.1.
- The grid spacing of the layer is

\[
\begin{align*}
dx &= \frac{x\text{-extent}}{\text{number of columns}} \\
dy &= \frac{y\text{-extent}}{\text{number of rows}}
\end{align*}
\]

In the layer shown, we have $dx = dy = 0.2$, but the grid spacing may differ in x- and y-direction.

- Layer elements are spaced by the grid spacing and are arranged symmetrically about the center.
- The outermost layer elements are placed $dx/2$ and $dy/2$ from the borders of the extent.
- Element positions in the coordinate system are given by $(x, y)$ pairs. The coordinate system follows that standard mathematical convention that the $x$-axis runs from left to right and the $y$-axis from bottom to top.
- Each element of a grid-based layer has a row- and column-index in addition to its $(x, y)$-coordinates. Indices are shown in the top and right margin of Figure 8.1.1. Note that row-indices follow matrix convention, i.e., run from top to bottom. Following pythonic conventions, indices run from 0.

Layers have a default extent of $1 \times 1$. You can specify a different extent of a layer, i.e., its size in $x$- and $y$-direction by adding an 'extent' entry to the dictionary passed to CreateLayer:
Figure 8.1.1: Simple grid-based layer centered about the origin. Blue circles mark layer elements, the thin square the extent of the layer. Row and column indices are shown in the right and top margins, respectively.
l = tp.CreateLayer({'rows': 5, 'columns': 5, 'extent': [2.0, 0.5], 'elements': 'iaf_psc_alpha'})

Figure 8.1.2: Same layer as in Figure 8.1.1, but with different extent.

The resulting layer is shown in Figure 8.1.2. The extent is always a two-element tuple of floats. In this example, we have grid spacings $dx = 0.4$ and $dy = 0.1$. Changing the extent does not affect grid indices.

The size of 'extent' in $x$- and $y$-directions should be numbers that can be expressed exactly as binary fractions. This is automatically ensured for integer values. Otherwise, under rare circumstances, subtle rounding errors may occur and trigger an assertion, thus stopping NEST.

Layers are centered about the origin $(0,0)$ by default. This can be changed through the 'center' entry in the dictionary specifying the layer. The following code creates layers centered about $(0,0)$, $(-1,1)$, and $(1.5,0.5)$, respectively:

```
l1 = tp.CreateLayer({'rows': 5, 'columns': 5, 'elements': 'iaf_psc_alpha'})
l2 = tp.CreateLayer({'rows': 5, 'columns': 5, 'elements': 'iaf_psc_alpha', 'center': [-1., 1.]})
l3 = tp.CreateLayer({'rows': 5, 'columns': 5, 'elements': 'iaf_psc_alpha', 'center': [1.5, 0.5]})
```

The center is given as a two-element tuple of floats. Changing the center does not affect grid indices: For each of the three layers in Figure 8.1.3, grid indices run from 0 to 4 through columns and rows, respectively, even though elements in these three layers have different positions in the global coordinate system.

The 'center' coordinates should be numbers that can be expressed exactly as binary fractions. For more information, see Sec. 2.1.2.

To see how to construct a layer, consider the following example:

- a layer with $n_r$ rows and $n_c$ columns;
- spacing between nodes is $d$ in $x$- and $y$-directions;
- the left edge of the extent shall be at $x = 0$;
- the extent shall be centered about $y = 0$. 

Figure 8.1.3: Three layers centered about different positions in the global coordinate system.
From Eq. $dx\_dy\_extent$, we see that the extent of the layer must be $(n_c d, n_r d)$. We now need to find the coordinates $(c_x, c_y)$ of the center of the layer. To place the left edge of the extent at $x = 0$, we must place the center of the layer at $c_x = n_c d/2$ along the $x$-axis, i.e., half the extent width to the right of $x = 0$. Since the layer is to be centered about $y = 0$, we have $c_y = 0$. Thus, the center coordinates are $(n_c d/2, 0)$. The layer is created with the following code and shown in Figure 8.1.4:

```python
nc, nr = 5, 3
d = 0.1
l = tp.CreateLayer({'columns': nc,
                    'rows': nr,
                    'elements': 'iaf_psc_alpha',
                    'extent': [nc * d, nr * d],
                    'center': [nc * d / 2., 0.]})
```

### 8.1.3 Free layers

**Free layers** do not restrict node positions to a grid, but allow free placement within the extent. To this end, the user needs to specify the positions of all nodes explicitly. The following code creates a layer of 50 `iaf_psc_alpha` neurons uniformly distributed in a layer with extent $1 \times 1$, i.e., spanning the square $[-0.5, 0.5] \times [-0.5, 0.5]$:

```python
import numpy as np
pos = [[np.random.uniform(-0.5, 0.5), np.random.uniform(-0.5, 0.5)]
       for j in range(50)]
l = tp.CreateLayer({'positions': pos,
                    'elements': 'iaf_psc_alpha'})
```

Note the following points:
Figure 8.1.4: Layer with $n_c = 5$ rows and $n_r = 3$ columns, spacing $d = 0.1$ and the left edge of the extent at $x = 0$, centered about the $y$-axis. The cross marks the point on the extent placed at the origin $(0, 0)$, the circle the center of the layer.
Figure 8.1.5: A free layer with 50 elements uniformly distributed in an extent of size $1 \times 1$. 
• For free layers, element *positions* are specified by the 'positions' entry in the dictionary passed to `CreateLayer`. 'positions' is mutually exclusive with 'rows'/'columns' entries in the dictionary.

• The 'positions' entry must be a Python list (or tuple) of element coordinates, i.e., of two-element tuples of floats giving the \((x, y)\)-coordinates of the elements. One layer element is created per element in the 'positions' entry.

• All layer element positions must be within the layer’s extent. Elements may be placed on the perimeter of the extent as long as no periodic boundary conditions are used; see Sec. 2.4.

• Element positions in free layers are *not* shifted when specifying the 'center' of the layer. The user must make sure that the positions given lie within the extent when centered about the given center.

### 8.1.4 3D layers

Although the term “layer” suggests a 2-dimensional structure, the layers in NEST may in fact be 3-dimensional. The example from the previous section may be easily extended with another component in the coordinates for the positions:

```python
import numpy as np

pos = [[np.random.uniform(-0.5, 0.5), np.random.uniform(-0.5, 0.5),
       np.random.uniform(-0.5, 0.5)]
       for j in range(200)]
l = tp.CreateLayer({'positions': pos,
                   'elements': 'iaf_psc_alpha'})
```

Figure 8.1.6: A free 3D layer with 200 elements uniformly distributed in an extent of size \(1 \times 1 \times 1\).
8.1.5 Periodic boundary conditions

Simulations usually model systems much smaller than the biological networks we want to study. One problem this entails is that a significant proportion of neurons in a model network is close to the edges of the network with fewer neighbors than nodes properly inside the network. In the $5 \times 5$-layer in Figure 8.1.1, e.g., 16 out of 25 nodes form the border of the layer.

One common approach to reducing the effect of boundaries on simulations is to introduce periodic boundary conditions, so that the rightmost elements on a grid are considered nearest neighbors to the leftmost elements, and the topmost to the bottommost. The flat layer becomes the surface of a torus. Figure 8.1.7 illustrates this for a one-dimensional layer, which turns from a line to a ring upon introduction of periodic boundary conditions.

You specify periodic boundary conditions for a layer using the dictionary entry `edge_wrap`:

```python
lp = tp.CreateLayer({'rows': 1, 'columns': 5, 'extent': [5., 1.],
                    'elements': 'iaf_psc_alpha',
                    'edge_wrap': True})
```

Figure 8.1.7: Top left: Layer with single row and five columns without periodic boundary conditions. Numbers above elements show element coordinates. Colors shifting from blue to magenta mark increasing distance from the element at $(-2, 0)$. Bottom left: Same layer, but with periodic boundary conditions. Note that the element at $(2, 0)$ now is a nearest neighbor to the element at $(-2, 0)$. Right: Layer with periodic boundary condition arranged on a circle to illustrate neighborhood relationships.

Note that the longest possible distance between two elements in a layer without periodic boundary conditions is

$$\sqrt{x_{\text{ext}}^2 + y_{\text{ext}}^2}$$

but only

$$\frac{\sqrt{x_{\text{ext}}^2 + y_{\text{ext}}^2}}{2}$$

for a layer with periodic boundary conditions; $x_{\text{ext}}$ and $y_{\text{ext}}$ are the components of the extent size.

We will discuss the consequences of periodic boundary conditions more in Chapter 3.

From the perspective of NEST, a Topology layer is a special type of subnet. From the user perspective, the following points may be of interest:
Grid-based layers have the NEST model type `topology_layer_grid`, free layers the model type `topology_layer_free`.

The status dictionary of a layer has a 'topology' entry describing the layer properties (l is the layer created above):

```python
print(nest.GetStatus(l)[0]['topology'])
```

```json
{'center': (0.0, 0.0), 'columns': 5, 'depth': 1, 'edge_wrap': False, 'extent': (1.0, -1.0), 'rows': 5}
```

The 'topology' entry is read-only.

- The NEST kernel sees the elements of the layer in the same way as the elements of any subnet. You will notice this when printing a network with a Topology layer:

```python
nest.PrintNetwork(depth=3)
```

```
+[-0] root dim=[1 25]
 | +[-1] topology_layer_grid dim=[25]
 | +[-1]...[25] iaf_psc_alpha
```

The 5 times 5 layer created above appears here as a `topology_layer_grid` subnet of 25 `iaf_psc_alpha` neurons. Only Topology connection and visualization functions heed the spatial structure of the layer.

### 8.1.6 Layers with composite elements

So far, we have considered layers in which each element was a single model neuron. Topology can also create layers with composite elements, i.e., layers in which each element is a collection of model neurons, or, in general NEST network nodes.

Construction of layers with composite elements proceeds exactly as for layers with simple elements, except that the 'elements' entry of the dictionary passed to `CreateLayer` is a Python list or tuple. The following code creates a 1 × 2 layer (to keep the output from `PrintNetwork()` compact) in which each element consists of one 'iaf_cond_alpha' and one 'poisson_generator' node

```python
l = tp.CreateLayer({'rows': 1, 'columns': 2, 'elements': ['iaf_cond_alpha', 'poisson_generator']})
```

```
+[-0] root dim=[1 4]
 | +[-1] topology_layer_grid dim=[4]
 | +[-1]...[2] iaf_cond_alpha
 | +[-3]...[4] poisson_generator
```

The network consist of one `topology_layer_grid` with four elements: two `iaf_cond_alpha` and two `poisson_generator` nodes. The identical nodes are grouped, so that the subnet contains first one full layer of `iaf_cond_alpha` nodes followed by one full layer of `poisson_generator` nodes.

You can create network elements with several nodes of each type by following a model name with the number of nodes to be created:
In this case, each layer element consists of 10 iaf_cond_alpha neurons, one poisson_generator, and two noise_generators.

Note the following points:

- Each element of a layer has identical components.
- All nodes within a composite element have identical positions, namely the position of the layer element.
- When inspecting a layer as a subnet, the different nodes will appear in groups of identical nodes.
- For grid-based layers, the function `GetElement` returns a list of nodes at a given grid position. See Chapter 4 for more on inspecting layers.
- In a previous version of the topology module it was possible to create layers with nested, composite elements, but such nested networks gobble up a lot of memory for subnet constructs and provide no practical advantages, so this is no longer supported. See the next section for design recommendations for more complex layers.

A paper on a neural network model might describe the network as follows:

The network consists of 20x20 microcolumns placed on a regular grid spanning 0.5° × 0.5° of visual space. Neurons within each microcolumn are organized into L2/3, L4, and L56 subpopulations. Each subpopulation consists of three pyramidal cells and one interneuron. All pyramidal cells are modeled as NEST iaf_psc_alpha neurons with default parameter values, while interneurons are iaf_psc_alpha neurons with threshold voltage $V_{th} = -52$ mV.

How should you implement such a network using the Topology module? The recommended approach is to create different models for the neurons in each layer and then define the microcolumn as one composite element:

```python
for lyr in ['L23', 'L4', 'L56']:
    nest.CopyModel('iaf_psc_alpha', lyr + 'pyr')
    nest.CopyModel('iaf_psc_alpha', lyr + 'in', {'V_th': -52.})
l = tp.CreateLayer({'rows': 20, 'columns': 20, 'extent': [0.5, 0.5], 'elements': ['L23pyr', 3, 'L23in', 'L4pyr', 3, 'L4in', 'L56pyr', 3, 'L56in']})
```

We will discuss in Chapter 3.1 how to connect selectively to different neuron models.

**Connections**

The most important feature of the Topology module is the ability to create connections between layers with quite some flexibility. In this chapter, we will illustrate how to specify and create connections. All connections are created using

---

2 See Nord (2009) for suggestions on how to describe network models.
the \texttt{ConnectLayers} function.

\section*{8.1.7 Basic principles}

We begin by introducing important terminology:

\textbf{Connection} In the context of connections between the elements of Topology layers, we often call the set of all connections between pairs of network nodes created by a single call to \texttt{ConnectLayers} a \textit{connection}.

\textbf{Connection dictionary} A dictionary specifying the properties of a connection between two layers in a call to \texttt{CreateLayers}.

\textbf{Source} The \textit{source} of a single connection is the node sending signals (usually spikes). In a projection, the source layer is the layer from which source nodes are chosen.

\textbf{Target} The \textit{target} of a single connection is the node receiving signals (usually spikes). In a projection, the target layer is the layer from which target nodes are chosen.

\textbf{Connection type} The \textit{connection type} determines how nodes are selected when \texttt{ConnectLayers} creates connections between layers. It is either 'convergent' or 'divergent'.

\textbf{Convergent connection} When creating a \textit{convergent connection} between layers, Topology visits each node in the target layer in turn and selects sources for it in the source layer. Masks and kernels are applied to the source layer, and periodic boundary conditions are applied in the source layer, provided that the source layer has periodic boundary conditions.

\textbf{Divergent connection} When creating a \textit{divergent connection}, Topology visits each node in the source layer and selects target nodes from the target layer. Masks, kernels, and boundary conditions are applied in the target layer.

\textbf{Driver} When connecting two layers, the \textit{driver} layer is the one in which each node is considered in turn.

\textbf{Pool} When connecting two layers, the \textit{pool} layer is the one from which nodes are chosen for each node in the driver layer. I.e., we have

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
Connection type & Driver & Pool \\
\hline
convergent & target layer & source layer \\
& source layer & target layer \\
\hline
\end{tabular}
\end{center}

\textbf{Displacement} The \textit{displacement} between a driver and a pool node is the shortest vector connecting the driver to the pool node, taking boundary conditions into account.

\textbf{Distance} The \textit{distance} between a driver and a pool node is the length of their displacement.

\textbf{Mask} The \textit{mask} defines which pool nodes are at all considered as potential targets for each driver node. See Sec. 3.3 for details.

\textbf{Kernel} The \textit{kernel} is a function returning a (possibly distance- or displacement-dependent) probability for creating a connection between a driver and a pool node. The default kernel is 1, i.e., connections are created with certainty. See Sec. 3.4 for details.

\textbf{Autapse} An \textit{autapse} is a synapse (connection) from a node onto itself. Autapses are permitted by default, but can be disabled by adding 'allow_autapses': False to the connection dictionary.

\textbf{Multapse} Node A is connected to node B by a \textit{multapse} if there are synapses (connections) from A to B. Multapses are permitted by default, but can be disabled by adding 'allow_multapses': False to the connection dictionary.
Connections between Topology layers are created by calling `ConnectLayers` with the following arguments:

1. The source layer.
2. The target layer (can be identical to source layer).
3. A connection dictionary that contains at least the following entry:
   ‘connection_type’ either 'convergent' or 'divergent'.

In many cases, the connection dictionary will also contain

‘mask’ a mask specification as described in Sec. 3.3.

Only neurons within the mask are considered as potential sources or targets. If no mask is given, all neurons in the respective layer are considered sources or targets.

Here is a simple example, cf. Figure 8.1.8

```python
                   'elements': 'iaf_psc_alpha'})
conndict = {'connection_type': 'divergent',
            'mask': {'rectangular': {'lower_left': [-2., -1.],
                                    'upper_right': [2., 1.]}},
            tp.ConnectLayers(l, l, conndict)
```

Figure 8.1.8: Left: Minimal connection example from a layer onto itself using a rectangular mask shown as red line for the node at (0, 0) (marked light red). The targets of this node are marked with red dots. The targets for the node at (4, 5) are marked with yellow dots. This node has fewer targets since it is at the corner and many potential targets are beyond the layer. Right: The effect of periodic boundary conditions is seen here. Source and target layer and connection dictionary were identical, except that periodic boundary conditions were used. The node at (4, 5) now has 15 targets, too, but they are spread across the corners of the layer. If we wrapped the layer to a torus, they would form a $5 \times 3$ rectangle centered on the node at (4, 5).

In this example, layer $l$ is both source and target layer. Connection type is divergent, i.e., for each node in the layer we choose targets according to the rectangular mask centered about each source node. Since no connection kernel is specified, we connect to all nodes within the mask. Note the effect of normal and periodic boundary conditions on the connections created for different nodes in the layer, as illustrated in Figure 8.1.8.

---

3 You can also use standard NEST connection functions to connect nodes in Topology layers.
8.1.8 Mapping source and target layers

The application of masks and other functions depending on the distance or even the displacement between nodes in the source and target layers requires a mapping of coordinate systems between source and target layers. Topology applies the following coordinate mapping rules:

1. All layers have two-dimensional Euclidean coordinate systems.
2. No scaling or coordinate transformation can be applied between layers.
3. The displacement \( d(D, P) \) from node \( D \) in the driver layer to node \( P \) in the pool layer is measured by first mapping the position of \( D \) in the driver layer to the identical position in the pool layer and then computing the displacement from that position to \( P \). If the pool layer has periodic boundary conditions, they are taken into account. It does not matter for displacement computations whether the driver layer has periodic boundary conditions.

8.1.9 Masks

A mask describes which area of the pool layer shall be searched for nodes to connect for any given node in the driver layer. We will first describe geometrical masks defined for all layer types and then consider grid-based masks for grid-based layers. If no mask is specified, all nodes in the pool layer will be searched.

Note that the mask size should not exceed the size of the layer when using periodic boundary conditions, since the mask would “wrap around” in that case and pool nodes would be considered multiple times as targets.

If none of the mask types provided in the topology library meet your need, you may add more mask types in a NEST extension module. This is covered in Chapter 5.

Topology currently provides four types of masks usable for 2-dimensional free and grid-based layers. They are illustrated in Figure 8.1.9. The masks are

**Rectangular** All nodes within a rectangular area are connected. The area is specified by its lower left and upper right corners, measured in the same unit as element coordinates. Example:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'rectangular': {'lower_left': [-2., -1.],
                                     'upper_right': [2., 1.]}},
```

**Circular** All nodes within a circle are connected. The area is specified by its radius.

```python
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 2.0}}},
```

**Doughnut** All nodes between an inner and outer circle are connected. Note that nodes on the inner circle are not connected. The area is specified by the radii of the inner and outer circles.

```python
conndict = {'connection_type': 'divergent',
            'mask': {'doughnut': {'inner_radius': 1.5,
                                  'outer_radius': 3.}}},
```

**Elliptical** All nodes within an ellipsis are connected. The area is specified by its major and minor axis. Note that this mask was added to NEST with NEST 2.14.

```python
conndict = {'connection_type': 'divergent',
            'mask': {'elliptical': {'major_axis': 7.,
                                    'minor_axis': 4.}}},
```
Figure 8.1.9: Masks for 2D layers. For all mask types, the driver node is marked by a wide light-red circle, the selected pool nodes by red dots and the masks by red lines. From left to right, top to bottom: rectangular, circular, doughnut and elliptical masks centered about the driver node.
By default, the masks are centered about the position of the driver node, mapped into the pool layer. You can change the location of the mask relative to the driver node by specifying an 'anchor' entry in the mask dictionary. The anchor is a 2D vector specifying the location of the mask center relative to the driver node, as in the following examples (cf. Figure 8.1.10).

```
conndict = {'connection_type': 'divergent',
            'mask': {'rectangular': {'lower_left': [-2., -1.],
                                    'upper_right': [2., 1.],
                                    'anchor': [-1.5, -1.5]}}
```

```
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 2.0},
                     'anchor': [-2.0, 0.0]}}
```

```
conndict = {'connection_type': 'divergent',
            'mask': {'doughnut': {'inner_radius': 1.5,
                                  'outer_radius': 3.},
                     'anchor': [1.5, 1.5]}}
```

```
conndict = {'connection_type': 'divergent',
            'mask': {'elliptical': {'major_axis': 7.,
                                   'minor_axis': 4.},
                     'anchor': [2.0, -1.0]}}
```

Figure 8.1.10: The same masks as in Figure 8.1.9, but centered about (−1.5, −1.5), (−2, 0), (1.5, 1.5) and (2, −1), respectively, using the 'anchor' parameter.

and elliptical masks, see Fig Figure 8.1.10. To do so, add an 'azimuth_angle' entry in the specific mask dictio-
The azimuth_angle is measured in degrees and is the rotational angle from the x-axis to the y-axis.

```
conndict = {
    'connection_type': 'divergent',
    'mask': {
        'rectangular': {
            'lower_left': [-2., -1.],
            'upper_right': [2., 1.],
            'azimuth_angle': 120.
        }
    }
}
```

```
conndict = {
    'connection_type': 'divergent',
    'mask': {
        'elliptical': {
            'major_axis': 7.,
            'minor_axis': 4.,
            'azimuth_angle': 45.
        }
    }
}
```

Figure 8.1.11: Rotated rectangle and elliptical mask from Figure 8.1.9 and Figure 8.1.10, where the rectangle mask is rotated 120° and the elliptical mask is rotated 45°.

Similarly, there are three mask types that can be used for 3D layers,

**Box** All nodes within a cuboid volume are connected. The area is specified by its lower left and upper right corners, measured in the same unit as element coordinates. Example:

```
conndict = {
    'connection_type': 'divergent',
    'mask': {
        'box': {
            'lower_left': [-2., -1., -1.],
            'upper_right': [2., 1., 1.]
        }
    }
}
```

**Spherical** All nodes within a sphere are connected. The area is specified by its radius.

```
conndict = {
    'connection_type': 'divergent',
    'mask': {
        'spherical': {
            'radius': 2.5
        }
    }
}
```

**Ellipsoidal** All nodes within an ellipsoid are connected. The area is specified by its major, minor, and polar axis. This mask has been part of NEST since NEST 2.14.

```
conndict = {
    'connection_type': 'divergent',
    'mask': {
        'ellipsoidal': {
            'major_axis': 7.,
            'minor_axis': 4.,
            'polar_axis': 4.5
        }
    }
}
```

As in the 2D case, you can change the location of the mask relative to the driver node by specifying a 3D vector in the 'anchor' entry in the mask dictionary. If you want to rotate the box or ellipsoidal masks, you can add
an 'azimuth_angle' entry in the specific mask dictionary for rotation from the x-axis towards the y-axis about the z-axis, or an 'polar_angle' entry, specifying the rotation angle in degrees from the z-axis about the (possibly rotated) x axis, from the (possibly rotated) y-axis. You can specify both at once of course. If both are specified, we first rotate about the z-axis and then about the new x-axis. NEST currently do not support rotation in all three directions, the rotation from the y-axis about the (possibly rotated) z-axis, from the (possibly rotated) x-axis is missing.

Figure 8.1.12: Masks for 3D layers. For all mask types, the driver node is marked by a wide light-red circle, the selected pool nodes by red dots and the masks by red lines. From left to right: box and spherical masks centered about the driver node.

Grid-based layers can be connected using rectangular grid masks. For these, you specify the size of the mask not by lower left and upper right corner coordinates, but give their size in rows and columns, as in this example:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'grid': {'rows': 3, 'columns': 5}}}
```

The resulting connections are shown in Figure 8.1.13. By default the top-left corner of a grid mask, i.e., the grid mask element with grid index [0, 0]⁴, is aligned with the driver node. You can change this alignment by specifying an anchor for the mask:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'grid': {'rows': 3, 'columns': 5},
                     'anchor': {'row': 1, 'column': 2}}}
```

You can even place the anchor outside the mask:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'grid': {'rows': 3, 'columns': 5},
                     'anchor': {'row': -1, 'column': 2}}}
```

The resulting connection patterns are shown in Figure 8.1.13.

Connections specified using grid masks are generated more efficiently than connections specified using other mask types.

Note the following:

- Grid-based masks are applied by considering grid indices. The position of nodes in physical coordinates is ignored.

⁴ See Sec. 2.1.1 for the distinction between layer coordinates and grid indices
In consequence, grid-based masks should only be used between layers with identical grid spacings.

The semantics of the 'anchor' property for grid-based masks differ significantly for general masks described in Sec. 3.3.1. For general masks, the anchor is the center of the mask relative to the driver node. For grid-based nodes, the anchor determines which mask element is aligned with the driver element.

### 8.1.10 Kernels

Many neuronal network models employ probabilistic connection rules. Topology supports probabilistic connections through kernels. A kernel is a function mapping the distance (or displacement) between a driver and a pool node to a connection probability. Topology then generates a connection according to this probability.

Probabilistic connections can be generated in two different ways using Topology:

- **Free probabilistic connections** are the default. In this case, `ConnectLayers` considers each driver node $D$ in turn. For each $D$, it evaluates the kernel for each pool node $P$ within the mask and creates a connection according to the resulting probability. This means in particular that each possible driver-pool pair is inspected exactly once and that there will be at most one connection between each driver-pool pair.

- **Prescribed number of connections** can be obtained by specifying the number of connections to create per driver node. See Sec. 3.7 for details.

Available kernel functions are shown in Table tbl_kernels. More kernel functions may be created in a NEST extension module. This is covered in Chapter 5.

$d$ is the distance and $(d_x, d_y)$ the displacement. All functions can be used to specify weights and delays, but only the constant and the distance-dependent functions, i.e., all functions above the double line, can be used as kernels.
Several examples follow. They are illustrated in Figure 8.1.14.

**Constant** The simplest kernel is a fixed connection probability:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 4.}},
            'kernel': 0.5}
```

**Gaussian** This kernel is distance dependent. In the example, connection probability is 1 for \( d = 0 \) and falls off with a “standard deviation” of \( \sigma = 1 \):

```python
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 4.}},
            'kernel': {'gaussian': {'p_center': 1.0, 'sigma': 1.}}}
```

**Eccentric Gaussian** In this example, both kernel and mask have been moved using anchors:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 4.}},
            'kernel': {'gaussian2D': {'p_center': 1.0, 'sigma_x': 1.0, 'sigma_y': 1.0, 'mean_x': 0.0, 'mean_y': 0.0, 'rho': 0.5, 'c': 0.5}}}
```

(continues on next page)
Figure 8.1.14: Illustration of various kernel functions. Top left: constant kernel, $p = 0.5$. Top center: Gaussian kernel, green dashed lines show $\sigma$, $2\sigma$, $3\sigma$. Top right: Same Gaussian kernel anchored at $(1.5, 1.5)$. Bottom left: Same Gaussian kernel, but all $p < 0.5$ treated as $p = 0$. Bottom center: 2D-Gaussian.
Note that the anchor for the kernel is specified inside the dictionary containing the parameters for the Gaussian.

**Cut-off Gaussian** In this example, all probabilities less than 0.5 are set to zero:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 4.}},
            'kernel': {'gaussian': {'p_center': 1.0,
                                     'sigma': 1.,
                                     'cutoff': 0.5}}}
```

**2D Gaussian** We conclude with an example using a two-dimensional Gaussian, i.e., a Gaussian with different widths in $x$- and $y$- directions. This kernel depends on displacement, not only on distance:

```python
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 4.}},
            'kernel': {'gaussian2D': {'p_center': 1.0,
                                      'sigma_x': 1.,
                                      'sigma_y': 3.}}}
```

Note that for pool layers with periodic boundary conditions, Topology always uses the shortest possible displacement vector from driver to pool neuron as argument to the kernel function.

### 8.1.11 Weights and delays

The functions presented in Table tbl_kernels can also be used to specify distance-dependent or randomized weights and delays for the connections created by ConnectLayers.

Figure Figure 8.1.15 illustrates weights and delays generated using these functions with the following code examples. All examples use a “layer” of 51 nodes placed on a line; the line is centered about $(25, 0)$, so that the leftmost node has coordinates $(0, 0)$. The distance between neighboring elements is 1. The mask is rectangular, spans the entire layer and is centered about the driver node.

**Linear example**

```python
ldict = {'rows': 1, 'columns': 51,
         'extent': [51., 1.], 'center': [25., 0.],
         'elements': 'iaf_psc_alpha'}

cdict = {'connection_type': 'divergent',
          'mask': {'rectangular': {'lower_left': [-25.5, -0.5],
                                    'upper_right': [25.5, 0.5]}},
          'weights': {'linear': {'c': 1.0,
                                 'a': -0.05,
                                 'cutoff': 0.0}},
          'delays': {'linear': {'c': 0.1, 'a': 0.02}}}
```

Results are shown in the top panel of Figure 8.1.15. Connection weights and delays are shown for the leftmost neuron as driver. Weights drop linearly from 1. From the node at $(20, 0)$ on, the cutoff sets weights to 0. There are no connections to nodes beyond $(25, 0)$, since the mask extends only 25 units to the right of the driver. Delays increase in a stepwise linear fashion, as NEST requires delays to be multiples of the simulation resolution.

**Linear example with periodic boundary conditions**
Results are shown in the middle panel of Figure 8.1.15. This example is identical to the previous, except that the (pool) layer has periodic boundary conditions. Therefore, the left half of the mask about the node at (0, 0) wraps back to the right half of the layer and that node connects to all nodes in the layer.

Various functions

<table>
<thead>
<tr>
<th>cdict</th>
</tr>
</thead>
</table>
| {'connection_type': 'divergent',
| 'mask': {'rectangular': {'lower_left': [-25.5, -0.5],
| 'upper_right': [25.5, 0.5]}},
| 'weights': {'linear': {'c': 1.0,
| 'a': -0.05,
| 'cutoff': 0.0}},
| 'delays': {'linear': {'c': 0.1, 'a': 0.02}} |

Results are shown in the bottom panel of Figure 8.1.15. It shows linear, exponential and Gaussian weight functions for the node at (25, 0).

Randomized weights and delays

<table>
<thead>
<tr>
<th>cdict</th>
</tr>
</thead>
</table>
| {'connection_type': 'divergent',
| 'mask': {'rectangular': {'lower_left': [-25.5, -0.5],
| 'upper_right': [25.5, 0.5]}},
| 'weights': {'uniform': {'min': 0.2, 'max': 0.8}} |

By using the 'uniform' function for weights or delays, one can obtain randomized values for weights and delays, as shown by the red circles in the bottom panel of Figure 8.1.15. Weights and delays can currently only be randomized with uniform distribution.

### 8.1.12 Periodic boundary conditions

Connections between layers with periodic boundary conditions are based on the following principles:

- Periodic boundary conditions are always applied in the pool layer. It is irrelevant whether the driver layer has periodic boundary conditions or not.

- By default, Topology does not accept masks that are wider than the pool layer when using periodic boundary conditions. Otherwise, one pool node could appear as multiple targets to the same driver node as the masks wraps several times around the layer. For layers with different extents in $x$- and $y$-directions this means that the maximum layer size is determined by the smaller extension.

- Kernel, weight and delay functions always consider the shortest distance (displacement) between driver and pool node.

In most physical systems simulated using periodic boundary conditions, interactions between entities are short-range. Periodic boundary conditions are well-defined in such cases. In neuronal network models with long-range interactions, periodic boundary conditions may not make sense. In general, we recommend to use periodic boundary conditions only when connection masks are significantly smaller than the layers they are applied to.
Figure 8.1.15: Distance-dependent and randomized weights and delays. See text for details.
8.1.13 Prescribed number of connections

We have so far described how to connect layers by either connecting to all nodes inside the mask or by considering each pool node in turn and connecting it according to a given probability function. In both cases, the number of connections generated depends on mask and kernel.

Many neuron models in the literature, in contrast, prescribe a certain \textit{fan in} (number of incoming connections) or \textit{fan out} (number of outgoing connections) for each node. You can achieve this in Topology by prescribing the number of connections for each driver node. For convergent connections, where the target layer is the driver layer, you thus achieve a constant fan in, for divergent connections a constant fan out.

Connection generation now proceeds in a different way than before:

1. For each driver node, \texttt{ConnectLayers} randomly selects a node from the mask region in the pool layer, and creates a connection with the probability prescribed by the kernel. This is repeated until the requested number of connections has been created.

2. Thus, if all nodes in the mask shall be connected with equal probability, you should not specify any kernel.

3. If you specify a non-uniform kernel (e.g., Gaussian, linear, exponential), the connections will be distributed within the mask with the spatial profile given by the kernel.

4. If you prohibit multapses (cf Sec. 3.1.1) and prescribe a number of connections greater than the number of pool nodes in the mask, \texttt{ConnectLayers} may get stuck in an infinite loop and NEST will hang. Keep in mind that the number of nodes within the mask may vary considerably for free layers with randomly placed nodes.

The following code generates a network of 1000 randomly placed nodes and connects them with a fixed fan out of 50 outgoing connections per node distributed with a profile linearly decaying from unit probability to zero probability at distance 0.5. Multiple connections (multapses) between pairs of nodes are allowed, self-connections (autapses) prohibited. The probability of finding a connection at a certain distance is then given by the product of the probabilities for finding nodes at a certain distance with the kernel value for this distance. For the kernel and parameter values below we have

\[ p_{\text{conn}}(d) = \frac{12}{\pi} \times 2\pi r \times (1 - 2r) = 24r(1 - 2r) \quad \text{for} \quad 0 \leq r < \frac{1}{2}. \]

The resulting distribution of distances between connected nodes is shown in Figure 8.1.16.

```python
pos = [[np.random.uniform(-1., 1.), np.random.uniform(-1., 1.)] for j in range(1000)]
ldict = {'positions': pos, 'extent': [2., 2.],
         'elements': 'iaf_psc_alpha', 'edge_wrap': True}
ndict = {'connection_type': 'divergent',
         'mask': {'circular': {'radius': 1.0}},
         'kernel': {'linear': {'c': 1., 'a': -2., 'cutoff': 0.0}},
         'number_of_connections': 50,
         'allow_multapses': True, 'allow_autapses': False}
```

Functions determining weight and delay as function of distance/displacement work in just the same way as before when the number of connections is prescribed.

8.1.14 Connecting composite layers

Connections between layers with composite elements are based on the following principles:

- All nodes within a composite element have the same coordinates, the coordinates of the element.
Figure 8.1.16: Distribution of distances between source and target for a network of 1000 randomly placed nodes, a fixed fan out of 50 connections and a connection probability decaying linearly from 1 to 0 at $d = 0.5$. The red line is the expected distribution from Eq. eq_ptheo.
• All nodes within a composite element are treated equally. If, e.g., an element of the pool layer contains three nodes and connection probability is 1, then connections with all three nodes will be created. For probabilistic connection schemes, each of the three nodes will be considered individually.

• If only nodes of a given model within each element shall be considered as sources or targets then this can be achieved by adding a 'sources' or 'targets' entry to the connection dictionary, which specifies the model to connect.

This is exemplified by the following code, which connects pyramidal cells (pyr) to interneurons (in) with a circular mask and uniform probability and interneurons to pyramidal cells with a rectangular mask unit probability.

```python
nest.ResetKernel()
nest.CopyModel('iaf_psc_alpha', 'pyr')
nest.CopyModel('iaf_psc_alpha', 'in')
ldict = {'rows': 10, 'columns': 10, 'elements': ['pyr', 'in']}
cdict_p2i = {'connection_type': 'divergent',
             'mask': {'circular': {'radius': 0.5}},
             'kernel': 0.8,
             'sources': {'model': 'pyr'},
             'targets': {'model': 'in'}}
cdict_i2p = {'connection_type': 'divergent',
             'mask': {'rectangular': {'lower_left': [-0.2, -0.2],
                                      'upper_right': [0.2, 0.2]}},
             'sources': {'model': 'in'},
             'targets': {'model': 'pyr'}}
l = tp.CreateLayer(ldict)
tp.ConnectLayers(l, l, cdict_p2i)
tp.ConnectLayers(l, l, cdict_i2p)
```

### 8.1.15 Synapse models and properties

By default, ConnectLayers creates connections using the default synapse model in NEST, `static_synapse`. You can specify a different model by adding a 'synapse_model' entry to the connection dictionary, as in this example:

```python
nest.ResetKernel()
nest.CopyModel('iaf_psc_alpha', 'pyr')
nest.CopyModel('iaf_psc_alpha', 'in')
nest.CopyModel('static_synapse', 'exc', {'weight': 2.0})
nest.CopyModel('static_synapse', 'inh', {'weight': -8.0})
ldict = {'rows': 10, 'columns': 10, 'elements': ['pyr', 'in']}
cdict_p2i = {'connection_type': 'divergent',
             'mask': {'circular': {'radius': 0.5}},
             'kernel': 0.8,
             'sources': {'model': 'pyr'},
             'targets': {'model': 'in'},
             'synapse_model': 'exc'}
cdict_i2p = {'connection_type': 'divergent',
             'mask': {'rectangular': {'lower_left': [-0.2, -0.2],
                                      'upper_right': [0.2, 0.2]}},
             'sources': {'model': 'in'},
             'targets': {'model': 'pyr'},
             'synapse_model': 'inh'}
l = tp.CreateLayer(ldict)
tp.ConnectLayers(l, l, cdict_p2i)
tp.ConnectLayers(l, l, cdict_i2p)
```
You have to use synapse models if you want to set, e.g., the receptor type of connections or parameters for plastic synapse models. These can not be set in distance-dependent ways at present.

### 8.1.16 Connecting devices to subregions of layers

It is possible to connect stimulation and recording devices only to specific subregions of layers. A simple way to achieve this is to create a layer which contains only the device placed typically in its center. For connecting the device layer to a neuron layer, an appropriate mask needs to be specified and optionally also an anchor for shifting the center of the mask. As demonstrated in the following example, stimulation devices require the divergent connection type

```python
nrn_layer = tp.CreateLayer({'rows': 20, 'columns': 20, 'elements': 'iaf_psc_alpha'})
stim = tp.CreateLayer({'rows': 1, 'columns': 1, 'elements': 'poisson_generator'})
cdict_stim = {'connection_type': 'divergent', 'mask': {'circular': {'radius': 0.1}, 'anchor': [0.2, 0.2]}}

tp.ConnectLayers(stim, nrn_layer, cdict_stim)
```

while recording devices require the convergent connection type (see also Sec. 3.11):

```python
rec = tp.CreateLayer({'rows': 1, 'columns': 1, 'elements': 'spike_detector'})
cdict_rec = {'connection_type': 'convergent', 'mask': {'circular': {'radius': 0.1}, 'anchor': [-0.2, 0.2]}}

tp.ConnectLayers(nrn_layer, rec, cdict_rec)
```

### 8.1.17 Layers and recording devices

Generally, one should not create a layer of recording devices, especially spike detectors, to record from a topology layer. Instead, create a single spike detector, and connect all neurons in the layer to that spike detector using a normal connect command:

```python
rec = nest.Create('spike_detector')
nrns = nest.GetLeaves(nrn_layer, local_only=True)[0]
nest.Connect(nrns, rec)
```

Connections to a layer of recording devices as described in Sec. 3.10, such as spike detectors, are only possible using the convergent connection type without a fixed number of connections. Note that voltmeter and multimeter are not suffering from this restriction, since they are connected as sources, not as targets.

### Inspecting Layers

We strongly recommend that you inspect the layers created by Topology to be sure that node placement and connectivity indeed turned out as expected. In this chapter, we describe some functions that NEST and Topology provide to
query and visualize networks, layers, and connectivity.

### 8.1.18 Query functions

The following table presents some query functions provided by NEST (nest.) and Topology (tp.). For detailed information about these functions, please see the online Python and SLI documentation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nest.PrintNetwork()</code></td>
<td>Print structure of network or subnet from NEST perspective.</td>
</tr>
<tr>
<td><code>nest.GetConnections()</code></td>
<td>Retrieve connections (all or for a given source or target); see also <a href="http://www.nest-simulator.org/connection_management">http://www.nest-simulator.org/connection_management</a>.</td>
</tr>
<tr>
<td><code>nest.GetNodes()</code></td>
<td>Applied to a layer, returns GIDs of the layer elements. For simple layers, these are the actual model neurons, for composite layers the top-level subnets.</td>
</tr>
<tr>
<td><code>nest.GetLeaves()</code></td>
<td>Applied to a layer, returns GIDs of all actual model neurons, ignoring subnets.</td>
</tr>
<tr>
<td><code>tp.GetPosition()</code></td>
<td>Return the spatial locations of nodes.</td>
</tr>
<tr>
<td><code>tp.GetLayer()</code></td>
<td>Return the layer to which nodes belong.</td>
</tr>
<tr>
<td><code>tp.GetElement()</code></td>
<td>Return the node(s) at the location(s) in the given grid-based layer(s).</td>
</tr>
<tr>
<td><code>tp.GetTargetNodes()</code></td>
<td>Obtain targets of a list of sources in a given target layer.</td>
</tr>
<tr>
<td><code>tp.GetTargetPositions()</code></td>
<td>Obtain positions of targets of a list of sources in a given target layer.</td>
</tr>
<tr>
<td><code>tp.FindNearestElement()</code></td>
<td>Return the node(s) closest to the location(s) in the given layer(s).</td>
</tr>
<tr>
<td><code>tp.FindCenterElement()</code></td>
<td>Return GID(s) of node closest to center of layer(s).</td>
</tr>
<tr>
<td><code>tp.Displacement()</code></td>
<td>Obtain vector of lateral displacement between nodes, taking periodic boundary conditions into account.</td>
</tr>
<tr>
<td><code>tp.Distance()</code></td>
<td>Obtain vector of lateral distances between nodes, taking periodic boundary conditions into account.</td>
</tr>
<tr>
<td><code>tp.DumpLayerNodes()</code></td>
<td>Write layer element positions to file.</td>
</tr>
<tr>
<td><code>tp.DumpLayerConnections()</code></td>
<td>Write connectivity information to file. This function may be very useful to check that Topology created the correct connection structure.</td>
</tr>
<tr>
<td><code>tp.SelectNodesByMask()</code></td>
<td>Obtain GIDs of nodes/elements inside a masked area of a layer. Part of NEST since NEST 2.14.</td>
</tr>
</tbody>
</table>

### 8.1.19 Visualization functions

Topology provides three functions to visualize networks:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PlotLayer()</code></td>
<td>Plot nodes in a layer.</td>
</tr>
<tr>
<td><code>PlotTargets()</code></td>
<td>Plot all targets of a node in a given layer.</td>
</tr>
<tr>
<td><code>PlotKernel()</code></td>
<td>Add indication of mask and kernel to plot of layer. It does not wrap masks and kernels with respect to periodic boundary conditions. This function is usually called by <code>PlotTargets</code>.</td>
</tr>
</tbody>
</table>

The following code shows a practical example: A $21 \times 21$ network which connects to itself with divergent Gaussian connections. The resulting graphics is shown in Figure 8.1.17. All elements and the targets of the center neuron are shown, as well as mask and kernel.
Figure 8.1.17: 21 × 21 grid with divergent Gaussian projections onto itself. Blue circles mark layer elements, red circles connection targets of the center neuron (marked by large light-red circle). The large red circle is the mask, the dashed green lines mark $\sigma$, $2\sigma$ and $3\sigma$ of the Gaussian kernel.
Adding topology kernels and masks

This chapter will show examples of how to extend the topology module by adding custom kernel functions and masks. Some knowledge of the C++ programming language is needed for this. The functions will be added as a part of an extension module which is dynamically loaded into NEST. For more information on writing an extension module, see the section titled “Writing an Extension Module” in the NEST Developer Manual. The basic steps required to get started are:

1. From the NEST source directory, copy directory examples/MyModule to somewhere outside the NEST source, build or install directories.
2. Change to the new location of MyModule and prepare by issuing ./bootstrap.sh
3. Leave MyModule and create a build directory for it, e.g., mmb next to it
4. Configure. The configure process uses the script nest-config to find out where NEST is installed, where the source code resides, and which compiler options were used for compiling NEST. If nest-config is not in your path, you need to provided it explicitly like this
cmake -Dwith-nest=${NEST_INSTALL_DIR}/bin/nest-config ../MyModule
5. MyModule will then be installed to \${NEST_INSTALL_DIR}. This ensures that NEST will be able to find initializing SLI files for the module. You should not use the --prefix to select a different installation destination. If you do, you must make sure to use addpath in SLI before loading the module to ensure that NEST will find the SLI initialization file for your module.
6. Compile.
make
make install

The previous command installed MyModule to the NEST installation directory, including help files generated from the source code.
8.1.20 Adding kernel functions

As an example, we will add a kernel function called 'affine2d', which will be linear (actually affine) in the
displacement of the nodes, on the form

\[ p(d) = ad_x + bd_y + c. \]

The kernel functions are provided by C++ classes subclassed from \texttt{nest::Parameter}. To enable subclassing, add
the following lines at the top of the file \texttt{mymodule.h}:

```cpp
#include "topologymodule.h"
#include "parameter.h"
```

Then, add the class definition, e.g. near the bottom of the file before the brace closing the namespace \texttt{mynest}:

```cpp
class Affine2DParameter: public nest::Parameter
{
public:
    Affine2DParameter(const DictionaryDatum& d):
        Parameter(d),
        a_(1.0),
        b_(1.0),
        c_(0.0)
    {
        updateValue<
        double
    >(d, "a", a_);
        updateValue<
        double
    >(d, "b", b_);
        updateValue<
        double
    >(d, "c", c_);
    }

    double raw_value(const nest::Position<2>& disp,
                      librandom::RngPtr&) const
    {
        return a_\*disp[0] + b_\*disp[1] + c_;
    }

    nest::Parameter * clone() const
    {
        return new Affine2DParameter(*this); }

private:
    double a_, b_, c_;}
```

The class contains a constructor, which reads the value of the parameters \(a, b\) and \(c\) from the dictionary provided by
the user. The function \texttt{updateValue} will do nothing if the given key is not in the dictionary, and the default values
\(a = b = 1, c = 0\) will be used.

The overridden method \texttt{raw_value()} will return the actual value of the kernel function for the displacement given
as the first argument, which is of type \texttt{nest::Position<2>}. The template argument 2 refers to a 2-dimensional
position. You can also implement a method taking a \texttt{nest::Position<3>} as the first argument if you want to
support 3-dimensional layers. The second argument, a random number generator, is not used in this example.

The class also needs to have a \texttt{clone()} method, which will return a dynamically allocated copy of the object. We
use the (default) copy constructor to implement this.

To make the custom function available to the Topology module, you need to register the class you have provided. To
do this, add the line

```cpp
#include "topologymodule.h"
#include "parameter.h"
```
nest::TopologyModule::register_parameter<Affine2DParameter>("affine2d");

to the function MyModule::init() in the file mymodule.cpp. Now compile and install the module by issuing

make
make install

To use the function, the module must be loaded into NEST using nest::Install(). Then, the function is available to be used in connections, e.g.

```cpp
class EllipticMask : public nest::Mask<2>
{
public:
    EllipticMask(const DictionaryDatum& d):
        rx_(1.0), ry_(1.0)
    {
        updateValue<double>(d, "r_x", rx_);
        updateValue<double>(d, "r_y", ry_);
    }

    using Mask<2>::inside;

    // returns true if point is inside the ellipse
    bool inside(const nest::Position<2> &p) const
    {
        return p[0]*p[0]/rx_/rx_ + p[1]*p[1]/ry_/ry_ <= 1.0; }

    // returns true if the whole box is inside the ellipse
    bool inside(const nest::Box<2> &b) const
    {
        nest::Position<2> p = b.lower_left;

        // Test if all corners are inside mask
        if (not inside(p)) return false; // (0,0)
        p[0] = b.upper_right[0];
        if (not inside(p)) return false; // (0,1)
    }
};
```

8.1.21 Adding masks

The process of adding a mask is similar to that of adding a kernel function. A subclass of nest::Mask<D> must be defined, where D is the dimension (2 or 3). In this case we will define a 2-dimensional elliptic mask by creating a class called EllipticMask. Note that elliptical masks are already part of NEST see Sec. 3.3. That elliptical mask is defined in a different way than what we will do here though, so this can still be used as an introductory example. First, we must include another header file:

```cpp
#include "mask.h"
```

Compared to the Parameter class discussed in the previous section, the Mask class has a few more methods that must be overridden:
p[1] = b.upper_right[1];
if (not inside(p)) return false; // (l,1)
p[0] = b.lower_left[0];
if (not inside(p)) return false; // (l,0)

return true;
}

// returns bounding box of ellipse
nest::Box<2> get_bbox() const
{
    nest::Position<2> ll(-rx_,-ry_);
    nest::Position<2> ur(rx_,ry_);
    return nest::Box<2>(ll,ur);
}

nest::Mask<2> * clone() const
{
    return new EllipticMask(*this);
}

protected:
    double rx_, ry_;
8.1.23 Changes from Topology 2.12 to 2.14

This is a short summary of the most important changes in the Topology Module from NEST version 2.12 to 2.14.

- Elliptical and ellipsoidal masks have been added to NEST with NEST 2.14. To specify the mask, the \texttt{major_axis}, \texttt{minor_axis} and (for ellipsoidal masks) \texttt{polar_axis} must be specified.
- It is now possible to obtain the GIDs inside a masked area with the function \texttt{SelectNodesByMask}.

8.1.24 Changes from Topology 2.0 to 2.2

This is a short summary of the most important changes in the Topology Module from NEST version 2.0 to 2.2.

- Nested layers are no longer supported.
- Subnets are no longer used inside composite layers. A call to \texttt{GetElement} for a composite layer will now return a list of GIDs for the nodes at the position rather than a single subnet GID.
- Positions in layers may now be 3-dimensional.
- The functions \texttt{GetPosition}, \texttt{Displacement} and \texttt{Distance} now only works for nodes local to the current MPI process, if used in a MPI-parallel simulation.
- It is now possible to add kernel functions and masks to the Topology module through an extension module. Please see Chapter 5 for examples.

8.1.25 Changes from Topology 1.9 to 2.0

This is a short summary of the most important changes in the NEST Topology Module from the 1.9-xxxx to the 2.0 version.

- \texttt{ConnectLayer} is now called \texttt{ConnectLayers}
- Several other functions changed names, and there are many new functions. Please see Ch. 4 for an overview.
- All \texttt{nest.topology} functions now require lists of GIDs as input, not “naked” GIDs
- There are a number of new functions in \texttt{nest.topology}, I tried to write good doc strings for them
- For grid based layers (ie those with /rows and /columns), we have changed the definition of “extent”: Previously, nodes were placed on the edges of the extent, so if you had an extend of 2 (in x-direction) and 3 nodes, these had x-coordinates -1, 0, 1. The grid constant was extent/(num_nodes - 1).

Now, we define the grid constant as extent/num_nodes, center the nodes about 0 and thus add a space of half a grid constant between the outermost nodes and the boundary of the extent. If you want three nodes at -1,0,1 you thus have to set the extent to 3, i.e., stretching from -1.5 to 1.5.

The main reason for this change was that topology always added this padding silently when you used periodic boundary conditions (otherwise, neurons are the left and right edge would have been in identical locations, not what one wants).

- The semantics of the \texttt{anchor} entry for kernel functions has changed: the anchor now specifies the center of the probability distribution relative to the driver node. This is consistent with the semantics for free masks, see Sec. 3.3 and 3.4.
- Functions computing connection probabilities, weights and delays as functions of distance between source and target nodes now handle periodic boundary conditions correctly.
- Masks with a diameter larger than the diameter of the layer they are applied to are now prohibited by default. This avoids multiple connections when masks overlap.
References

8.2 Topology Tutorial with Hill Tononi Model

8.2.1 NEST Topology Module: A Case-Based Tutorial

NOTE: The network generated by this script does generate dynamics in which the activity of the entire system, especially Rp and Vp oscillates with approx 5 Hz. This is different from the full model. Deviations are due to the different model type and the elimination of a number of connections, with no changes to the weights.

Introduction

This tutorial shows you how to implement a simplified version of the Hill-Tononi model of the early visual pathway using the NEST Topology module. The model is described in the paper


We simplify the model somewhat both to keep this tutorial a bit shorter, and because some details of the Hill-Tononi model are not currently supported by NEST. Simplifications include:

# We use the \texttt{iaf\_cond\_alpha} neuron model, which is simpler than the Hill-Tononi model.

# As the \texttt{iaf\_cond\_alpha} neuron model only supports two synapses (labeled “ex” and “in”), we only include AMPA and GABA\_A synapses.

# We ignore the secondary pathway (Ts, Rs, Vs), since it adds just more of the same from a technical point of view.

# Synaptic delays follow a Gaussian distribution in the HT model. This implies actually a Gaussian distributions clipped at some small, non-zero delay, since delays must be positive. Currently, there is a bug in the Topology module when using clipped Gaussian distribution. We therefore draw delays from a uniform distribution.

1. Some further adaptations are given at the appropriate locations in the script.

This tutorial is divided in the following sections:

\textit{Philosophy} Discusses the philosophy applied to model implementation in this tutorial

\textit{Preparations} Necessary steps to use NEST and the Topology Module

\texttt{‘Configurable Parameters’} Define adjustable network parameters

\texttt{‘Neuron Models’} Define the neuron models needed by the network model

\texttt{‘Populations’} Create Populations

\texttt{‘Synapse models’} Define the synapse models used in the network model

\texttt{‘Connections’} Create Connections

\texttt{‘Example simulation’} Perform a small simulation for illustration. This section also discusses the setup for recording.

\textit{Philosophy}

\textbf{A network models has two essential components: populations and projections.} We first use NEST’s \texttt{CopyModel()} mechanism to
create specific models for all populations and subpopulations in the network, and then create the populations using the Topology module `CreateLayer()` function.

We use a two-stage process to create the connections, mainly because the same configurations are required for a number of projections: we first define dictionaries specifying the connections, then apply these dictionaries later.

The way in which we declare the network model here is an example. You should not consider it the last word: we expect to see a significant development in strategies and tools for network descriptions in the future. The following contributions to CNS*09 seem particularly interesting:

- Ralf Ansorg & Lars Schwabe. Declarative model description and code generation for hybrid individual- and population-based simulations of the early visual system (P57);
- Sharon Crook, R. Angus Silver, & Padraig Gleeson. Describing and exchanging models of neurons and neuronal networks with NeuroML (F1);

as well as the following paper which will apply in PLoS Computational Biology shortly:

- Eilen Nordlie, Marc-Oliver Gewaltig, & Hans Ekkehard Plesser.

Towards reproducible descriptions of neuronal network models.

**Preparations**

Please make sure that your `PYTHONPATH` is set correctly, so that Python can find the NEST Python module.

**Note:** By default, the script does not show any graphics. Set `SHOW_FIGURES` to `True` to activate graphics.

This example uses the function `GetLeaves`, which is deprecated. A deprecation warning is therefore issued. For details about deprecated functions, see documentation.

```python
import pylab
SHOW_FIGURES = False

if not SHOW_FIGURES:
    pylab_show = pylab.show
    def nop(s=None):
        pass
    pylab.show = nop
else:
    pylab.ion()

# ! Introduction
# !============
# ! This tutorial gives a brief introduction to the ConnPlotter toolbox. It is by no means complete.
# ! Load pynest
import nest

# ! Load NEST Topology module (NEST 2.2)
import nest.topology as topo

# ! Make sure we start with a clean slate, even if we re-run the script
# ! in the same Python session.
nest.ResetKernel()
```

(continues on next page)
import math

# ! Configurable Parameters
# ! ================
# !
# ! Here we define those parameters that we take to be
# ! configurable. The choice of configurable parameters is obviously
# ! arbitrary, and in practice one would have far more configurable
# ! parameters. We restrict ourselves to:
# !
# ! - Network size in neurons \(N\), each layer is \(N \times N\).
# ! - Network size in subtended visual angle \(\text{visSize}\), in degree.
# ! - Temporal frequency of drifting grating input \(f_{\text{dg}}\), in Hz.
# ! - Spatial wavelength and direction of drifting grating input,
# ! \(\lambda_{\text{dg}}\) and \(\phi_{\text{dg}}\), in degree/radian.
# ! - Background firing rate of retinal nodes and modulation amplitude,
# ! \(\text{retDC}\) and \(\text{retAC}\), in Hz.
# ! - Simulation duration \(\text{simtime}\); actual simulation is split into
# ! intervals of \(\text{sim_interval}\) length, so that the network state
# ! can be visualized in those intervals. Times are in ms.

Params = {'N': 40,
          'visSize': 8.0,
          'f_dg': 2.0,
          'lambda_dg': 2.0,
          'phi_dg': 0.0,
          'retDC': 30.0,
          'retAC': 30.0,
          'simtime': 100.0,
          'sim_interval': 5.0
        }

# ! Neuron Models
# ! ===========
# !
# ! We declare models in two steps:
# !
# ! 1. We define a dictionary specifying the NEST neuron model to use
# ! as well as the parameters for that model.
# ! 2. We create three copies of this dictionary with parameters
# ! adjusted to the three model variants specified in Table 2 of
# ! Hill & Tononi (2005) (cortical excitatory, cortical inhibitory,
# ! thalamic)
# !
# ! In addition, we declare the models for the stimulation and
# ! recording devices.
# !
# ! The general neuron model
# ! =========
# !
# ! We use the \'iaf_cond_alpha\' neuron, which is an
# ! integrate-and-fire neuron with two conductance-based synapses which
# ! have alpha-function time course. Any input with positive weights
# ! will automatically directed to the synapse labeled \'ex\', any
# ! with negative weights to the synapses labeled \'in\'. We define
# ! **all** parameters explicitly here, so that no information is
# ! hidden in the model definition in NEST. \'V_m\' is the membrane
# potential to which the model neurons will be initialized.

## Model equations and parameters for the Hill-Tononi neuron model

- Hill & Tononi specify their model in terms of the membrane time constant, while the `iaf_cond_alpha` model is based on the membrane capacitance. Interestingly, conductances are unitless in the H&T model. We thus can use the time constant directly as membrane capacitance.

- The model includes sodium and potassium leak conductances. We combine these into a single one as follows:

$$ \begin{equation} -g_{NaL}(V-E_{Na}) - g_{KL}(V-E_{K}) \\ = -(g_{NaL}+g_{KL}) \\ \left(V-\frac{g_{NaL}E_{NaL}+g_{KL}E_K}{g_{NaL}g_{KL}}\right) \end{equation} $$

- We write the resulting expressions for $g_L$ and $E_L$ explicitly below, to avoid errors in copying from our pocket calculator.

- The paper gives a range of 1.0-1.85 for $g_{KL}$, we choose 1.5 here.

- The Hill-Tononi model has no explicit reset or refractory time. We arbitrarily set $V_{reset}$ and $t_{ref}$.

- The paper uses double exponential time courses for the synaptic conductances, with separate time constants for the rising and falling flanks. Alpha functions have only a single time constant: we use twice the rising time constant given by Hill and Tononi.

- In the general model below, we use the values for the cortical excitatory cells as defaults. Values will then be adapted below.

```python
nest.CopyModel('iaf_cond_alpha', 'NeuronModel',
params={'C_m': 16.0,
        'E_L': (0.2 * 30.0 + 1.5 * -90.0) / (0.2 + 1.5),
        'g_L': 0.2 + 1.5,
        'E_ex': 0.0,
        'E_in': -70.0,
        'V_reset': -60.0,
        'V_th': -51.0,
        't_ref': 2.0,
        'tau_syn_ex': 1.0,
        'tau_syn_in': 2.0,
        'I_e': 0.0,
        'V_m': -70.0})
```

# Adaptation of models for different populations

## Cortical excitatory cells

```python
nest.CopyModel('iaf_cond_alpha', 'CtxExNeuron')
```

## Cortical inhibitory cells

```python
nest.CopyModel('iaf_cond_alpha', 'CxtInhNeuron')
```
nest.CopyModel('NeuronModel', 'CtxInNeuron',
    params={'C_m': 8.0,
            'V_th': -53.0,
            't_ref': 1.0})

# ! Thalamic cells
# ! ..............
nest.CopyModel('NeuronModel', 'ThalamicNeuron',
    params={'C_m': 8.0,
            'V_th': -53.0,
            't_ref': 1.0,
            'E_in': -80.0})

# ! Input generating nodes
# ! ----------------------
# ! Input is generated by sinusoidally modulate Poisson generators,
# ! organized in a square layer of retina nodes. These nodes require a
# ! slightly more complicated initialization than all other elements of
# ! the network:
# !
# ! - Average firing rate `rate`, firing rate modulation depth `amplitude`,
# ! and temporal modulation frequency `frequency` are the same for all
# ! retinal nodes and are set directly below.
# ! - The temporal phase `phase` of each node depends on its position in
# ! the grating and can only be assigned after the retinal layer has
# ! been created. We therefore specify a function for initializing the
# ! `phase`. This function will be called for each node.

def phaseInit(pos, lam, alpha):
    '''Initializer function for phase of drifting grating nodes.

    pos : position (x,y) of node, in degree
    lam : wavelength of grating, in degree
    alpha: angle of grating in radian, zero is horizontal

    Returns number to be used as phase of sinusoidal Poisson generator.
    '''
    return 360.0 / lam * (math.cos(alpha) * pos[0] + math.sin(alpha) * pos[1])

nest.CopyModel('sinusoidal_poisson_generator', 'RetinaNode',
    params={'amplitude': Params['retAC'],
            'rate': Params['retDC'],
            'frequency': Params['f_dg'],
            'phase': 0.0,
            'individual_spike_trains': False})

# ! Recording nodes
# ! ------------------

# ! We use the new `multimeter` device for recording from the model
# ! neurons. At present, `iaf_cond_alpha` is one of few models
# ! supporting `multimeter` recording. Support for more models will
# ! be added soon; until then, you need to use `voltmeter` to record
# ! from other models.

(continues on next page)
# ! We configure multimeter to record membrane potential to membrane
# ! potential at certain intervals to memory only. We record the GID of
# ! the recorded neurons, but not the time.

nest.CopyModel('multimeter', 'RecordingNode',
               params=[
                   'interval': Params['sim_interval'],
                   'record_from': ['V_m'],
                   'record_to': ['memory'],
                   'withgid': True,
                   'withtime': False
               ])

# ! Populations
# ! =====

# ! We now create the neuron populations in the model, again in the
# ! form of Python dictionaries. We define them in order from eye via
# ! thalamus to cortex.
# !
# ! We first define a dictionary defining common properties for all
# ! populations

layerProps = {
    'rows': Params['N'],
    'columns': Params['N'],
    'extent': [Params['visSize'], Params['visSize']],
    'edge_wrap': True
}

# ! This dictionary does not yet specify the elements to put into the
# ! layer, since they will differ from layer to layer. We will add them
# ! below by updating the `elements` dictionary entry for each
# ! population.

# ! Retina
# ! ---

layerProps.update({'elements': 'RetinaNode'})
retina = topo.CreateLayer(layerProps)

# ! Thalamus
# ! ---

# ! We first introduce specific neuron models for the thalamic relay
# ! cells and interneurons. These have identical properties, but by
# ! treating them as different models, we can address them specifically
# ! when building connections.

# ! We use a list comprehension to do the model copies.

[nest.CopyModel('ThalamicNeuron', SpecificModel)
 for SpecificModel in
  ('TpRelay', 'TpInter')]

# ! Now we can create the layer, with one relay cell and one
# ! interneuron per location:

for n in retina_leaves

# ! Topology Tutorial with Hill Tononi Model

8.2. Topology Tutorial with Hill Tononi Model

111
layerProps.update({"elements": ['TpRelay', 'TpInter']})
Tp = topo.CreateLayer(layerProps)

# ! Reticular nucleus
# ! -----------------
# ! We follow the same approach as above, even though we have only a
# ! single neuron in each location.
[nest.CopyModel('ThalamicNeuron', SpecificModel) for SpecificModel in
 ('RpNeuron',)]
layerProps.update({"elements": 'RpNeuron'})
Rp = topo.CreateLayer(layerProps)

# ! Primary visual cortex
# ! ---------------------
# ! We follow again the same approach. We differentiate neuron types
# ! between layers and between pyramidal cells and interneurons. At
# ! each location, there are two pyramidal cells and one interneuron in
# ! each of layers 2-3, 4, and 5-6. Finally, we need to differentiate
# ! between vertically and horizontally tuned populations. When creating
# ! the populations, we create the vertically and the horizontally
# ! tuned populations as separate populations.

# ! We use list comprehesions to create all neuron types:
[nest.CopyModel('CtxExNeuron', layer + 'pyr')
 for layer in ('L23', 'L4', 'L56')]
[nest.CopyModel('CtxInNeuron', layer + 'in')
 for layer in ('L23', 'L4', 'L56')]

# ! Now we can create the populations, suffixes h and v indicate tuning
layerProps.update({"elements": ['L23pyr', 2, 'L23in', 1,
 'L4pyr', 2, 'L4in', 1,
 'L56pyr', 2, 'L56in', 1]})
Vp_h = topo.CreateLayer(layerProps)
Vp_v = topo.CreateLayer(layerProps)

# ! Collect all populations
# ! ------------------------

# ! For reference purposes, e.g., printing, we collect all populations
# ! in a tuple:
populations = (retina, Tp, Rp, Vp_h, Vp_v)

# ! Inspection
# ! --------

# ! We can now look at the network using 'PrintNetwork':
nest.PrintNetwork()

# ! We can also try to plot a single layer in a network. For
# ! simplicity, we use Rp, which has only a single neuron per position.
topo.PlotLayer(Rp)
pylab.title('Layer Rp')
pylab.show()

# ! Synapse models
# ! =-------------

(continues on next page)
# Actual synapse dynamics, e.g., properties such as the synaptic time course, time constants, reversal potentials, are properties of neuron models in NEST and we set them in section `Neuron models` above. When we refer to *synapse models* in NEST, we actually mean connectors which store information about connection weights and delays, as well as port numbers at the target neuron (`'rport'`) and implement synaptic plasticity. The latter two aspects are not relevant here.

We just use NEST's `static_synapse` connector but copy it to synapse models `AMPA` and `GABA_A` for the sake of explicitness. Weights and delays are set as needed in section `Connections` below, as they are different from projection to projection. De facto, the sign of the synaptic weight decides whether input via a connection is handle by the `'_ex'` or the `'_in'` synapse.

```python
nest.CopyModel('static_synapse', 'AMPA')
nest.CopyModel('static_synapse', 'GABA_A')
```

## Connections

Building connections is the most complex part of network construction. Connections are specified in Table 1 in the Hill-Tononi paper. As pointed out above, we only consider AMPA and GABA_A synapses here. Adding other synapses is tedious work, but should pose no new principal challenges. We also use a uniform in stead of a Gaussian distribution for the weights.

The model has two identical primary visual cortex populations, `Vp_v` and `Vp_h`, tuned to vertical and horizontal gratings, respectively. The *only* difference in the connection patterns between the two populations is the thalamocortical input to layers L4 and L5-6 is from a population of 8x2 and 2x8 grid locations, respectively. Furthermore, inhibitory connection in cortex go to the opposing orientation population as to the own.

To save us a lot of code doubling, we thus defined properties dictionaries for all connections first and then use this to connect both populations. We follow the subdivision of connections as in the Hill & Tononi paper.

**Note:** Hill & Tononi state that their model spans 8 degrees of visual angle and stimuli are specified according to this. On the other hand, all connection patterns are defined in terms of cell grid positions. Since the NEST Topology Module defines connection patterns in terms of the extent given in degrees, we need to apply the following scaling factor to all lengths in connections:

```python
dpc = Params['visSize'] / (Params['N'] - 1)
```

We will collect all same-orientation cortico-cortical connections in `ccConnections = []`

```python
the same-orientation cortico-cortical connections in
```

and all cortico-thalamic connections in `ctConnections = []`

```python
and all cortico-thalamic connections in
```
Horizontal intralaminar
-----------------------

*Note:* "Horizontal" means "within the same cortical layer" in this case.

We first define a dictionary with the (most) common properties for horizontal intralaminar connection. We then create copies in which we adapt those values that need adapting, and

```python
horIntraBase = {
    "connection_type": "divergent",
    "synapse_model": "AMPA",
    "mask": {"circular": {"radius": 12.0 * dpc}},
    "kernel": {"gaussian": {"p_center": 0.05, "sigma": 7.5 * dpc}},
    "weights": 1.0,
    "delays": {"uniform": {"min": 1.75, "max": 2.25}}
}
```

We use a loop to do the for for us. The loop runs over a list of dictionaries with all values that need updating

```python
for conn in [{
    "sources": {"model": "L23pyr"},
    "targets": {"model": "L23pyr"},
},
    {"sources": {"model": "L23pyr"},
    "targets": {"model": "L23in"},
},
    {"sources": {"model": "L4pyr"},
    "targets": {"model": "L4pyr"},
    "mask": {"circular": {"radius": 7.0 * dpc}}},
    {"sources": {"model": "L4pyr"},
    "targets": {"model": "L4in"},
    "mask": {"circular": {"radius": 7.0 * dpc}}},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "L56pyr"},
    "weights": 1.0},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "L56in"}},
]:
    ndict = horIntraBase.copy()
    ndict.update(conn)
    ccConnections.append(ndict)
```

Vertical intralaminar
-----------------------

*Note:* "Vertical" means "between cortical layers" in this case.

We proceed as above.

```python
verIntraBase = {
    "connection_type": "divergent",
    "synapse_model": "AMPA",
    "mask": {"circular": {"radius": 2.0 * dpc}},
    "kernel": {"gaussian": {"p_center": 1.0, "sigma": 7.5 * dpc}},
    "weights": 2.0,
    "delays": {"uniform": {"min": 1.75, "max": 2.25}}
}
```

```python
for conn in [{
    "sources": {"model": "L23pyr"},
    "targets": {"model": "L56pyr"},
    "weights": 1.0},
    {"sources": {"model": "L23pyr"},
    "targets": {"model": "L23in"},
    "weights": 1.0},
    {"sources": {"model": "L4pyr"},
    "targets": {"model": "L23pyr"}},
    {"sources": {"model": "L4pyr"},
    "targets": {"model": "L23in"}},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "L23pyr"}},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "L23in"}},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "L4pyr"}},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "L4in"}},
]:
    ndict = verIntraBase.copy()
    ndict.update(conn)
    ccConnections.append(ndict)
```
# ! Intracortical inhibitory
# ! ------------------------
# !
# ! We proceed as above, with the following difference: each connection
# ! is added to the same-orientation and the cross-orientation list of
# ! connections.
# !
# ! **Note:** Weights increased from -1.0 to -2.0, to make up for missing GabaB
# !
# ! Note that we have to specify the **weight with negative sign** to make
# ! the connections inhibitory.

```python
intraInhBase = {
    "connection_type": "divergent",
    "synapse_model": "GABA_A",
    "mask": {"circular": {"radius": 7.0 * dpc}},
    "kernel": {"gaussian": {"p_center": 0.25, "sigma": 7.5 * dpc}},
    "weights": -2.0,
    "delays": {"uniform": {"min": 1.75, "max": 2.25}}
}
```

# ! We use a loop to do the for for us. The loop runs over a list of
# ! dictionaries with all values that need updating

```python
for conn in [{
    "sources": {"model": "L23in"},
    "targets": {"model": "L23pyr"},
},
    {"sources": {"model": "L23in"},
    "targets": {"model": "L23in"},
},
    {"sources": {"model": "L4in"},
    "targets": {"model": "L4pyr"},
},
    {"sources": {"model": "L4in"},
    "targets": {"model": "L4in"},
},
    {"sources": {"model": "L56in"},
    "targets": {"model": "L56pyr"},
},
    {"sources": {"model": "L56in"},
    "targets": {"model": "L56in"}}]:
    ndict = intraInhBase.copy()
    ndict.update(conn)
    ccConnections.append(ndict)
    ccxConnections.append(ndict)
```

# ! Corticothalamic
# ! ---------------

```python
corThalBase = {
    "connection_type": "divergent",
    "synapse_model": "AMPA",
    "mask": {"circular": {"radius": 5.0 * dpc}},
    "kernel": {"gaussian": {"p_center": 0.5, "sigma": 7.5 * dpc}},
    "weights": 1.0,
    "delays": {"uniform": {"min": 7.5, "max": 8.5}}
}
```

# ! We use a loop to do the for for us. The loop runs over a list of
# ! dictionaries with all values that need updating

```python
for conn in [{
    "sources": {"model": "L56pyr"},
    "targets": {"model": "TpRelay"},
},
    {"sources": {"model": "L56pyr"},
    "targets": {"model": "TpInter"}}]:
    ndict = intraInhBase.copy()
    ndict.update(conn)
    ctConnections.append(ndict)
```

# ! Corticoreticular
# ! ----------------

# ! In this case, there is only a single connection, so we write the
# ! dictionary itself; it is very similar to the corThalBase, and to
# ! show that, we copy first, then update. We need no `targets` entry,
# ! since Rp has only one neuron per location.

(continues on next page)
corRet = corThalBase.copy()
corRet.update({"sources": {"model": "L56pyr"}, "weights": 2.5})

# ! Build all connections beginning in cortex
# ! -----------------------------------------

# ! Cortico-cortical, same orientation
print("Connecting: cortico-cortical, same orientation")
[topo.ConnectLayers(Vp_h, Vp_h, conn) for conn in ccConnections]
[topo.ConnectLayers(Vp_v, Vp_v, conn) for conn in ccConnections]

# ! Cortico-cortical, cross-orientation
print("Connecting: cortico-cortical, other orientation")
[topo.ConnectLayers(Vp_h, Vp_v, conn) for conn in ccxConnections]
[topo.ConnectLayers(Vp_v, Vp_h, conn) for conn in ccxConnections]

# ! Cortico-thalamic connections
print("Connecting: cortico-thalamic")
[topo.ConnectLayers(Vp_h, Tp, conn) for conn in ctConnections]
[topo.ConnectLayers(Vp_v, Tp, conn) for conn in ctConnections]
topo.ConnectLayers(Vp_h, Rp, corRet)
topo.ConnectLayers(Vp_v, Rp, corRet)

# ! Thalamo-cortical connections
# ! ----------------------------

# ! **Note:** According to the text on p. 1674, bottom right, of
# ! the Hill & Tononi paper, thalamocortical connections are
# ! created by selecting from the thalamic population for each
# ! L4 pyramidal cell, ie, are "convergent" connections.
# !
# ! We first handle the rectangular thalamocortical connections.
thalCorRect = {"connection_type": "convergent",
               "sources": {"model": "TpRelay"},
               "synapse_model": "AMPA",
               "weights": 5.0,
               "delays": {"uniform": {"min": 2.75, "max": 3.25}}}  
print("Connecting: thalamo-cortical")

# ! Horizontally tuned
thalCorRect.update(
    {"mask": {"rectangular": {"lower_left": [-4.0 * dpc, -1.0 * dpc],
                             "upper_right": [4.0 * dpc, 1.0 * dpc]}}})
for conn in [{"targets": {"model": "L4pyr"}, "kernel": 0.5},
             {"targets": {"model": "L56pyr"}, "kernel": 0.3}]:
    thalCorRect.update(conn)
topo.ConnectLayers(Tp, Vp_h, thalCorRect)

# ! Vertically tuned
thalCorRect.update(
    {"mask": {"rectangular": {"lower_left": [-1.0 * dpc, -4.0 * dpc],
                             "upper_right": [1.0 * dpc, 4.0 * dpc]}}})
for conn in [{"targets": {"model": "L4pyr"}, "kernel": 0.5},
             {"targets": {"model": "L56pyr"}, "kernel": 0.3}]:
    thalCorRect.update(conn)
topo.ConnectLayers(Tp, Vp_v, thalCorRect)
# ! Diffuse connections
thalCorDiff = {"connection_type": "convergent",
    "sources": {"model": "TpRelay"},
    "synapse_model": "AMPA",
    "weights": 5.0,
    "mask": {"circular": {"radius": 5.0 * dpc}},
    "kernel": {"gaussian": {"p_center": 0.1, "sigma": 7.5 * dpc}},
    "delays": {"uniform": {"min": 2.75, "max": 3.25}}}

for conn in [{"targets": {"model": "L4pyr"}},
               {"targets": {"model": "L56pyr"}}]:
    thalCorDiff.update(conn)
topo.ConnectLayers(Tp, Vp_h, thalCorDiff)
topo.ConnectLayers(Tp, Vp_v, thalCorDiff)

# ! Thalamic connections
# ! -------------------
# ! Connections inside thalamus, including Rp
# !
# ! *Note:* In Hill & Tononi, the inhibition between Rp cells is mediated by
# ! GABA_B receptors. We use GABA_A receptors here to provide some
# ! self-dampening of Rp.
# !
# ! *Note:* The following code had a serious bug in v. 0.1: During the first
# ! iteration of the loop, "synapse_model" and "weights" were set to "AMPA" and
# ! "0.1", respectively and remained unchanged, so that all connections were
# ! created as excitatory connections, even though they should have been
# ! inhibitory. We now specify synapse_model and weight explicitly for each
# ! connection to avoid this.

thalBase = {"connection_type": "divergent",
    "delays": {"uniform": {"min": 1.75, "max": 2.25}}}

print("Connecting: intra-thalamic")

for src, tgt, conn in [(Tp, Rp, {"sources": {"model": "TpRelay"},
                                   "synapse_model": "AMPA",
                                   "mask": {"circular": {"radius": 2.0 * dpc}},
                                   "kernel": {"gaussian": {"p_center": 1.0, "sigma": 7.5 * dpc}},
                                   "weights": 2.0}),
                       (Tp, Tp, {"sources": {"model": "TpInter"},
                                 "targets": {"model": "TpRelay"},
                                 "synapse_model": "GABA_A",
                                 "weights": -1.0,
                                 "mask": {"circular": {"radius": 2.0 * dpc}},
                                 "kernel": {"gaussian":
                                                  {"p_center": 0.25, "sigma": 7.5 * dpc}}}),
                       (Tp, Tp, {"sources": {"model": "TpInter"},
                                 "targets": {"model": "TpInter"},
                                 "synapse_model": "GABA_A",
                                 "weights": -1.0,
                                 "mask": {"circular": {"radius": 2.0 * dpc}},
                                 "kernel": {"gaussian":
                                 {"p_center": 0.25, "sigma": 7.5 * dpc}}})
                       ]:
    topo.ConnectLayers(Tp, Vp_h, thalCorDiff)
topo.ConnectLayers(Tp, Vp_v, thalCorDiff)

(continues on next page)
(continued from previous page)


dict({"p_center": 0.25,
    "sigma": 7.5 * dpc})),
(Rp, Tp, {"targets": {"model": "TpRelay"},
    "synapse_model": "GABA_A",
    "weights": -1.0,
    "mask": {"circular": {"radius": 12.0 * dpc}},
    "kernel": {"gaussian":
        {"p_center": 0.15,
         "sigma": 7.5 * dpc})),
(Rp, Tp, {"targets": {"model": "TpInter"},
    "synapse_model": "GABA_A",
    "weights": -1.0,
    "mask": {"circular": {"radius": 12.0 * dpc}},
    "kernel": {"gaussian":
        {"p_center": 0.15,
         "sigma": 7.5 * dpc})),
(Rp, Rp, {"targets": {"model": "RpNeuron"},
    "synapse_model": "GABA_A",
    "weights": -1.0,
    "mask": {"circular": {"radius": 12.0 * dpc}},
    "kernel": {"gaussian":
        {"p_center": 0.5,
         "sigma": 7.5 * dpc}}},
thalBase.update(conn)
topo.ConnectLayers(src, tgt, thalBase)

# ! Thalamic input
# ! -------------
# ! Input to the thalamus from the retina.
# !
# ! **Note:** Hill & Tononi specify a delay of 0 ms for this connection.
# ! We use 1 ms here.
retThal = {"connection_type": "divergent",
    "synapse_model": "AMPA",
    "mask": {"circular": {"radius": 1.0 * dpc}},
    "kernel": {"gaussian": {"p_center": 0.75, "sigma": 2.5 * dpc}},
    "weights": 10.0,
    "delays": 1.0}
print("Connecting: retino-thalamic")
for conn in [{"targets": {"model": "TpRelay"}},
    {"targets": {"model": "TpInter"}}]:
    retThal.update(conn)
topo.ConnectLayers(retina, Tp, retThal)

# ! Checks on connections
# ! ---------------------
# ! As a very simple check on the connections created, we inspect
# ! the connections from the central node of various layers.

# ! Connections from Retina to TpRelay
topo.PlotTargets(topo.FindCenterElement(retina), Tp, 'TpRelay', 'AMPA')
pylab.title('Connections Retina -> TpRelay')
pylab.show()
Connections from TpRelay to L4pyr in Vp (horizontally tuned)

```
topo.PlotTargets(topo.FindCenterElement(Tp), Vp_h, 'L4pyr', 'AMPA')
pylab.title('Connections TpRelay -> Vp(h) L4pyr')
pylab.show()
```

Connections from TpRelay to L4pyr in Vp (vertically tuned)

```
topo.PlotTargets(topo.FindCenterElement(Tp), Vp_v, 'L4pyr', 'AMPA')
pylab.title('Connections TpRelay -> Vp(v) L4pyr')
pylab.show()
```

Recording devices

```
# This recording device setup is a bit makeshift. For each population
# we want to record from, we create one 'multimeter', then select
# all nodes of the right model from the target population and
# connect. `loc` is the subplot location for the layer.
print("Connecting: Recording devices")
recorders = {}
for name, loc, population, model in [('TpRelay', 1, Tp, 'TpRelay'),
   ('Rp', 2, Rp, 'RpNeuron'),
   ('Vp_v L4pyr', 3, Vp_v, 'L4pyr'),
   ('Vp_h L4pyr', 4, Vp_h, 'L4pyr')]:
    recorders[name] = (nest.Create('RecordingNode'), loc)
population_leaves = nest.GetLeaves(population)[0]
tgts = [nd for nd in population_leaves
    if nest.GetStatus([nd], 'model')[0] == model]
nest.Connect(recorders[name][0], tgts)  # one recorder to all targets
```

Example simulation

```
# This simulation is set up to create a step-wise visualization of
# the membrane potential. To do so, we simulate `sim_interval`
# milliseconds at a time, then read out data from the multimeters,
# clear data from the multimeters and plot the data as pseudocolor
# plots.

# show time during simulation
nest.SetStatus([0], {'print_time': True})

# lower and upper limits for color scale, for each of the four
# populations recorded.
vmn = [-80, -80, -80, -80]
vmx = [-50, -50, -50, -50]
nest.Simulate(Params['sim_interval'])
```

```
# loop over simulation intervals
for t in pylab.arange(Params['sim_interval'], Params['simtime'],
   Params['sim_interval']):

    # do the simulation
    nest.Simulate(Params['sim_interval'])
```

(continues on next page)
# clear figure and choose colormap
pylab.clf()
pylab.jet()

# now plot data from each recorder in turn, assume four recorders
for name, r in recorders.items():
    rec = r[0]
    sp = r[1]
    pylab.subplot(2, 2, sp)
    d = nest.GetStatus(rec)[0]['events']['V_m']

    if len(d) != Params['N'] ** 2:
        # cortical layer with two neurons in each location, take average
        d = 0.5 * (d[::2] + d[1::2])

    # clear data from multimeter
    nest.SetStatus(rec, {'n_events': 0})
    pylab.imshow(pylab.reshape(d, (Params['N'], Params['N'])),
                 aspect='equal', interpolation='nearest',
                 extent=(0, Params['N'] + 1, 0, Params['N'] + 1),
                 vmin=vmn[sp - 1], vmax=vmx[sp - 1])
    pylab.colorbar()
    pylab.title(name + ', t = %6.1f ms' % nest.GetKernelStatus()['time'])

pylab.draw()  # force drawing inside loop
pylab.show()   # required by 'pyreport'

# ! just for some information at the end
print(nest.GetKernelStatus())

Total running time of the script: ( 0 minutes 0.000 seconds)

8.3 Examples using Topology

8.3.1 Create two layers with one pyramidal cell and one interneuron

Create two 30x30 layers with nodes composed of one pyramidal cell and one interneuron. Connect with two projections, one pyr->pyr, one pyr->in, and visualize.

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import nest
import nest.topology as topo
import pylab
pylab.ion()

nest.ResetKernel()
nest.set_verbosity('M_WARNING')

# create two test layers
nest.CopyModel('iaf_psc_alpha', 'pyr')
nest.CopyModel('iaf_psc_alpha', 'in')
```

(continues on next page)
```
a = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': ['pyr', 'in']})
b = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': ['pyr', 'in']})
topo.ConnectLayers(a, b, {'connection_type': 'divergent',
                      'sources': {'model': 'pyr'},
                      'targets': {'model': 'pyr'},
                      'mask': {'circular': {'radius': 0.5}},
                      'kernel': 0.5,
                      'weights': 1.0,
                      'delays': 1.0})
topo.ConnectLayers(a, b, {'connection_type': 'divergent',
                      'sources': {'model': 'pyr'},
                      'targets': {'model': 'in'},
                      'mask': {'circular': {'radius': 1.0}},
                      'kernel': 0.2,
                      'weights': 1.0,
                      'delays': 1.0})
pylab.clf()

# plot targets of neurons in different grid locations
for ctr in [[15, 15]]:
    # obtain node id for center: pick first node of composite
    ctr_id = topo.GetElement(a, ctr)
    # get all projection targets of center neuron
    tgts = [ci[1] for ci in nest.GetConnections(ctr_id)]
    # get positions of targets
    tpyr = pylab.array(tuple(zip(*[topo.GetPosition([n])[0] for n in tgts
                                   if nest.GetStatus([n], 'model')[0] == 'pyr'])))
    tin = pylab.array(tuple(zip(*[topo.GetPosition([n])[0] for n in tgts
                                   if nest.GetStatus([n], 'model')[0] == 'in'])))
    # scatter-plot
    pylab.scatter(tpyr[0] - 0.02, tpyr[1] - 0.02, 20, 'b', zorder=10)
    pylab.scatter(tin[0] + 0.02, tin[1] + 0.02, 20, 'r', zorder=10)
    # mark locations with background grey circle
    pylab.plot(tpyr[0], tpyr[1], 'o', markerfacecolor=(0.7, 0.7, 0.7),
               markersize=10, markeredgewidth=0, zorder=1, label='_nolegend_')
    pylab.plot(tin[0], tin[1], 'o', markerfacecolor=(0.7, 0.7, 0.7),
               markersize=10, markeredgewidth=0, zorder=1, label='_nolegend_')
    # mark sender position with transparent red circle
    ctrpos = topo.GetPosition(ctr_id)[0]
    pylab.gca().add_patch(pylab.Circle(ctrpos, radius=0.15, zorder=99,
                                        fc='r', alpha=0.4, ec='none'))
    # mark mask positions with open red/blue circles
    pylab.gca().add_patch(pylab.Circle(ctrpos, radius=0.5, zorder=2,
                                        fc='none', ec='b', lw=3))
```
8.3.2 Create two layers of iaf_psc_alpha neurons from target perspective

Create two 30x30 layers of iaf_psc_alpha neurons and connect with convergent projection and rectangular mask, visualize connection from target perspective.

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import nest
import nest.topology as topo
import pylab

pylab.ion()

nest.ResetKernel()
nest.set_verbosity('M_WARNING')

# create two test layers
a = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': 'iaf_psc_alpha', 'edge_wrap': True})
b = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': 'iaf_psc_alpha', 'edge_wrap': True})
topo.ConnectLayers(a, b, {'connection_type': 'convergent',
                          'mask': {'rectangular': {'lower_left': [-0.2, -0.5],
                                             'upper_right': [0.2, 0.5]},
                                   'kernel': 0.5,
                                   'weights': {'uniform': {'min': 0.5, 'max': 2.0}},
                                   'delays': 1.0})

pylab.clf()

# plot sources of neurons in different grid locations
for tgt_pos in [[15, 15], [0, 0]]:
    # obtain node id for center
    tgt = topo.GetElement(b, tgt_pos)
    # obtain list of outgoing connections for ctr
    # int() required to cast numpy.int64
    spos = tuple(zip(*[topo.GetPosition([int(conn[0])])][0] for conn in
                       topo.GetOutgoingConnections(tgt)
                       if conn[0] == tgt_pos))
    print(spos)
```

Total running time of the script: (0 minutes 0.000 seconds)
8.3.3 Create two layers of iaf_psc_alpha neurons from source perspective

Create two 30x30 layers of iaf_psc_alpha neurons and connect with convergent projection and rectangular mask, visualize connections from source perspective.

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import pylab
import nest
import nest.topology as topo
pylab.ion()

nest.ResetKernel()

# create two test layers
a = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                      'elements': 'iaf_psc_alpha', 'edge_wrap': True})
b = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                      'elements': 'iaf_psc_alpha', 'edge_wrap': True})

conndict = {'connection_type': 'convergent',
            'mask': {'rectangular': {'lower_left': [-0.2, -0.5],
                                    'upper_right': [0.2, 0.5]}},
            'kernel': 0.5,
            'weights': {'uniform': {'min': 0.5, 'max': 2.0}},
            'delays': 1.0}

nest.GetConnections(target=tgt)

# scatter-plot
pylab.scatter(spos[0], spos[1], 20, zorder=10)

# mark sender position with transparent red circle
ctrpos = pylab.array(topo.GetPosition(tgt)[0])
pylab.gca().add_patch(pylab.Circle(ctrpos, radius=0.1, zorder=99,
                                    fc='r', alpha=0.4, ec='none'))

# mark mask position with open red rectangle
pylab.gca().add_patch(pylab.Rectangle(ctrpos - (0.2, 0.5), 0.4, 1.0, zorder=1,
                                       fc='none', ec='r', lw=3))

# mark layer edge
pylab.gca().add_patch(pylab.Rectangle((-1.5, -1.5), 3.0, 3.0, zorder=1,
                                       fc='none', ec='k', lw=3))

# beautify
pylab.axes().set_xticks(pylab.arange(-1.5, 1.55, 0.5))
pylab.axes().set_yticks(pylab.arange(-1.5, 1.55, 0.5))
pylab.grid(True)
pylab.axis([-2.0, 2.0, -2.0, 2.0])
pylab.axes().set_aspect('equal', 'box')
pylab.title('Connection sources')
```

Total running time of the script: ( 0 minutes 0.000 seconds)
topo.ConnectLayers(a, b, conndict)

# first, clear existing figure, get current figure
pylab.clf()
fig = pylab.gcf()

# plot targets of two source neurons into same figure, with mask
for src_pos in [[15, 15], [0, 0]]:
    # obtain node id for center
    src = topo.GetElement(a, src_pos)
    topo.PlotTargets(src, b, mask=conndict['mask'], fig=fig)

# beautify
pylab.axes().set_xticks(pylab.arange(-1.5, 1.55, 0.5))
pylab.axes().set_yticks(pylab.arange(-1.5, 1.55, 0.5))
pylab.grid(True)
pylab.axis([-2.0, 2.0, -2.0, 2.0])
pylab.axes().set_aspect('equal', 'box')
pylab.title('Connection targets')

# pylab.savefig('conncon_targets.pdf')

**Total running time of the script:** (0 minutes 0.000 seconds)

### 8.3.4 Create two 30x30 layers of iaf_psc_alpha neurons (circular mask)

Connect with circular mask, flat probability, visualize.

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import nest
import nest.topology as topo
import pylab
pylab.ion()

nest.ResetKernel()

# create two test layers
a = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': 'iaf_psc_alpha'})
b = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': 'iaf_psc_alpha'})
conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 0.5}},
            'kernel': 0.5,
            'weights': {'uniform': {'min': 0.5, 'max': 2.0}},
            'delays': 1.0}
topo.ConnectLayers(a, b, conndict)

# plot targets of neurons in different grid locations
# first, clear existing figure, get current figure
pylab.clf()
```

(continues on next page)
Total running time of the script: ( 0 minutes 0.000 seconds)

8.3.5 Connect layers using Gaussian probalistic kernel

Create two layers of 30x30 elements and connect them using a Gaussian probabilistic kernel, visualize.

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

import pylab
import nest
import nest.topology as topo

pylab.ion()
nest.ResetKernel()

# create two test layers
a = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': 'iaf_psc_alpha'})
b = topo.CreateLayer({'columns': 30, 'rows': 30, 'extent': [3.0, 3.0],
                     'elements': 'iaf_psc_alpha'})

conndict = {'connection_type': 'divergent',
            'mask': {'circular': {'radius': 3.0}},
            'kernel': {'gaussian': {'p_center': 1.0, 'sigma': 0.5}},
            'weights': 1.0,
            'delays': 1.0}
topo.ConnectLayers(a, b, conndict)

# plot targets of neurons in different grid locations

# first, clear existing figure, get current figure
pylab.clf()
fig = pylab.gcf()

# plot targets of two source neurons into same figure, with mask
# use different colors
for src_pos, color in [[[15, 15], 'blue'], [[0, 0], 'green']]:
    # obtain node id for center
    src = topo.GetElement(a, src_pos)
    topo.PlotTargets(src, b, mask=conndict['mask'], fig=fig)

    # beautify
    pylab.axes().set_xticks(pylab.arange(-1.5, 1.55, 0.5))
    pylab.axes().set_yticks(pylab.arange(-1.5, 1.55, 0.5))
    pylab.grid(True)
    pylab.axis([-2.0, 2.0, -2.0, 2.0])
    pylab.axes().set_aspect('equal', 'box')
    pylab.title('Connection targets')

# pylab.savefig('connex.pdf')
# obtain node id for center
src = topo.GetElement(a, src_pos)
topo.PlotTargets(src, b, mask=conndict['mask'], kernel=conndict['kernel'],
src_color=color, tgt_color=color, mask_color=color,
kernel_color=color, src_size=100,
fig=fig)

# beautify
pylab.axes().set_xticks(pylab.arange(-1.5, 1.55, 0.5))
pylab.axes().set_yticks(pylab.arange(-1.5, 1.55, 0.5))
pylab.grid(True)
pylab.axes().set_aspect('equal', 'box')
pylab.title('Connection targets, Gaussian kernel')

# pylab.savefig('gaussex.pdf')

Total running time of the script:  (0 minutes 0.000 seconds)

8.3.6 Create layer of 4x3 iaf_psc_alpha neurons

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import nest
import pylab
import nest.topology as topo

pylab.ion()
nest.ResetKernel()

l1 = topo.CreateLayer({'columns': 4, 'rows': 3,
        'extent': [2.0, 1.5],
        'elements': 'iaf_psc_alpha'})
nest.PrintNetwork()
nest.PrintNetwork(2)
nest.PrintNetwork(2, l1)
topo.PlotLayer(l1, nodesize=50)

# beautify
pylab.axis([-1.0, 1.0, -0.75, 0.75])
pylab.axes().set_aspect('equal', 'box')
pylab.yscale('equal')
pylab.axes().set_xticks([-0.75, -0.25, 0.25, 0.75])
pylab.axes().set_yticks([-0.5, 0, 0.5])
pylab.grid(True)
pylab.xlabel('4 Columns, Extent: 1.5')
pylab.ylabel('2 Rows, Extent: 1.0')

# pylab.savefig('grid_iaf.png')

Total running time of the script:  (0 minutes 0.000 seconds)
```
8.3.7 Create layer of 12 freely placed iaf_psc_alpha neuron

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import nest
import pylab
import random
import nest.topology as topo

pylab.ion()

nest.ResetKernel()

# generate list of 12 (x,y) pairs
pos = [[random.uniform(-0.75, 0.75), random.uniform(-0.5, 0.5)]
      for j in range(12)]

l1 = topo.CreateLayer({'extent': [2., 1.5],
                      'positions': pos,
                      'elements': 'iaf_psc_alpha'})

nest.PrintNetwork()
nest.PrintNetwork(2)
nest.PrintNetwork(2, l1)
topo.PlotLayer(l1, nodesize=50)

# beautify
pylab.axis([-1.0, 1.0, -0.75, 0.75])
pylab.axis().set_aspect('equal', 'box')
pylab.axis().set_xticks([-0.75, -0.25, 0.25, 0.75])
pylab.axis().set_yticks([-0.5, 0, 0.5])
pylab.grid(True)
pylab.xlabel('4 Columns, Extent: 1.5')
pylab.ylabel('2 Rows, Extent: 1.0')

# pylab.savefig('grid_iaf_irr.png')
```

Total running time of the script: ( 0 minutes 0.000 seconds)

8.3.8 Create three layers of 4x3 iaf_psc_alpha neurons, each with different center

BCCN Tutorial @ CNS*09 Hans Ekkehard Plesser, UMB

```python
import pylab
import time
import nest
import nest.topology as topo

pylab.ion()

for ctr in [(0.0, 0.0), (-2.0, 2.0), (0.5, 1.0)]:
    nest.ResetKernel()
    pylab.clf()
    l1 = topo.CreateLayer({'columns': 4, 'rows': 3,
                         'extent': [2.0, 1.5],
                         'positions': pos,
                         'elements': 'iaf_psc_alpha'})

    nest.PrintNetwork()
    nest.PrintNetwork(2)
    nest.PrintNetwork(2, l1)
    topo.PlotLayer(l1, nodesize=50)

    # beautify
    pylab.axis([-1.0, 1.0, -0.75, 0.75])
    pylab.axis().set_aspect('equal', 'box')
    pylab.axis().set_xticks([-0.75, -0.25, 0.25, 0.75])
    pylab.axis().set_yticks([-0.5, 0, 0.5])
    pylab.grid(True)
    pylab.xlabel('4 Columns, Extent: 1.5')
    pylab.ylabel('2 Rows, Extent: 1.0')

    # pylab.savefig('grid_iaf_irr.png')
```

(continues on next page)
topo.PlotLayer(l1, nodesize=50, fig=pylab.gcf())

# beautify
pylab.axis([-3, 3, -3, 3])
pylab.axes().set_aspect('equal', 'box')
pylab.axes().set_xticks(pylab.arange(-3.0, 3.1, 1.0))
pylab.axes().set_yticks(pylab.arange(-3.0, 3.1, 1.0))
pylab.grid(True)
pylab.xlabel('4 Columns, Extent: 1.5, Center: %.1f' % ctr[0])
pylab.ylabel('2 Rows, Extent: 1.0, Center: %.1f' % ctr[1])

pylab.draw()
Here you can find detailed look into a variety of topics in NEST.

### 9.1 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:

```python
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](#).
9.1.1 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.

9.1.2 one-to-one

The $i$th node in $\text{pre}$ is connected to the $i$th node in $\text{post}$. The node lists $\text{pre}$ and $\text{post}$ have to be of the same length.

Example:

One-to-one connections

```python
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:

```python
weight = 1.5
delay = 0.5
Connect(A[0], B[0], weight, delay)
```
### 9.1.3 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:


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

9.1.4 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 OneToOneConnect() 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:

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

Example new connection routine:

```python
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:

```python
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:

```python
conn_dict = {'rule': 'fixed_indegree', 'indegree': N, 'autapses': True, 'multapses': False}
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:

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

Example new connection routine:

```python
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:

```python
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:

```python
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)
```

9.1.5 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.
9.1.6 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.

```python
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.

9.1.7 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:

```python
["cont_delay_synapse",
"bt_synapse",
"quantal_stp_synapse",
(continues on next page)
```
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.

```python
print GetDefaults("static_synapse")

{'delay': 1.0,
 'max_delay': -inf,
 'min_delay': inf,
 'num_connections': 0,
 'num_connectors': 0,
 'receptor_type': 0,
 'synapsemmodel': '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.

```python
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.

### 9.1.8 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:
9.1.9 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:

```python
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}]
```

9.2 Running simulations

9.2.1 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`:

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.
Figure 9.2.1: Simulation Loop
9.2.2 Simulation resolution and update interval

Each connection in NEST has its own specific delay that defines the time it takes until an event reaches the target node. We define the minimum delay $d_{\text{min}}$ as the smallest transmission delay and $d_{\text{max}}$ as the largest delay in the network. From this definition follows that no node can influence another node during at least a time of $d_{\text{min}}$, i.e. the elements are effectively decoupled for this interval.

![Diagram showing definitions of minimum delay and simulation resolution](image)

Figure 9.2.2: Definitions of the minimum delay and the simulation resolution.

Definitions of minimum delay ($d_{\text{min}}$) 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 $d_{\text{min}}$ time in one go, as to keep neurons in cache as long as possible.

2. MPI processes only communicate in intervals of $d_{\text{min}}$ 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 $d_{\text{min}}+d_{\text{max}}$ as histories that long back have to be kept. NEST will figure out the correct value of $d_{\text{min}}$ and $d_{\text{max}}$ based on the actual delays used during connection setup. Their actual values can be retrieved using `GetKernelStatus`:

```python
GetKernelStatus("min_delay")  # (A corresponding entry exists for max_delay)
```

**Setting $d_{\text{min}}$ and $d_{\text{max}}$ 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 $d_{\text{min}}$ and $d_{\text{max}}$ 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 $d_{\text{min}}+d_{\text{max}}$ 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 in between 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):

```python
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 $d_{\text{min}}$), or increased memory consumption of NEST (large $d_{\text{max}}$).

9.2.3 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.

### 9.2.4 Splitting a simulation into multiple intervals

In some cases, it may be useful to run a simulation in shorter intervals to extract information while the simulation is
running. The simplest way of doing this is to simply loop over `Simulate()` calls:

```python
for _ in range(20):
    nest.Simulate(10)
    # extract and analyse data
```

would run a simulation in 20 rounds of 10 ms. With this solution, NEST takes a number of preparatory and cleanup
steps for each `Simulate()` call. This makes the solution robust and entirely reliable, but comes with a performance
cost.

A more efficient solution doing exactly the same thing is

```python
nest.Prepare()
for _ in range(20):
    nest.Run(10)
    # extract and analyse data
nest.Cleanup()
```

For convenience, the `RunManager()` context manager can handle preparation and cleanup for you:

```python
with nest.RunManager():
    for _ in range(20):
        nest.Run(10)
        # extract and analyse data
```

**Note:**

- If you do not use `RunManager()`, you must call `Prepare()`, `Run()` and `Cleanup()` in that order.
- You can call `Run()` any number of times inside a `RunManager()` context or between `Prepare()` and `Cleanup()` calls.
- Calling `SetStatus()` inside a `RunManager()` context or between `Prepare()` and `Cleanup()` will
  lead to unpredictable results.
- After calling `Cleanup()`, you need to call `Prepare()` again before calling `Run()`.

### 9.2.5 Repeated simulations

The only reliable way to perform two simulations of a network from exactly the same starting point is to restart NEST
or to call `ResetKernel()` and then to build the network anew. If your simulations are rather large and you are working
on a computer with a job queueing system, it may be most efficient to submit individual jobs or a job array to simulate
network instances in parallel; don’t forget to use different random seeds!

The following example performs simulations of a single neuron driven by a Poisson spike train using different seeds
and output files for each run:
for n in range(10):
    nest.ResetKernel()
    nest.SetKernelStatus({'grng_seed': 100*n + 1,
                          'rng_seeds': [100*n + 2]})
    pg = nest.Create('poisson_generator', params={'rate': 100000.0})
    nrn = nest.Create('iaf_psc_alpha')
    sd = nest.Create('spike_detector',
                     params={'label': 'spikes-run{:02d}'.format(n),
                             'to_file': True})
    nest.Connect(pg, nrn)
    nest.Connect(nrn, sd)
    nest.Simulate(100)

The `ResetNetwork()` function available in NEST 2 is incomplete in that it only resets the state of neurons and devices to default values and deletes spikes that are in the delivery pipeline. It does not reset plastic synapses or delete spikes from the spike buffers of neurons. We will therefore remove the function in NEST 3 and already now advise against using `ResetNetwork()`.

### 9.3 Guide to parallel computing

- **What is parallelization?**
- **Virtual processes**
- **Node distributions**
  - Neuron distribution
  - Device Distribution
- **Spike exchange and synapse update**
  - Spikes between neurons
  - Spikes between neurons and devices
  - Synaptic plasticity models
- **Using multiple threads**
- **Using distributed computing**
  - Build requirements
  - Configure
  - Run distributed simulations
  - MPI related commands
- **Reproducibility**
9.3.1 What is parallelization?

Parallelization can improve the efficiency of running large-scale simulations by taking advantage of multicore/multiprocessor machines, computer clusters or supercomputers. Here we explain how parallelization is set up in NEST and how you can take advantage of it for your simulations.

NEST employs two methods for parallelization:

- **Thread-parallel simulation**
  - uses OpenMP
  - takes advantage of multicore and multiprocessor computers without the need for additional libraries

- **Distributed simulation (or distributed computing)**
  - uses the Message Passing Interface (MPI)
  - supports simulations over multiple computers

Both methods can be combined within a simulation.

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

9.3.2 Virtual processes

We use the concept of local and remote threads, called virtual processes. A virtual process (VP) is a thread residing in one of NEST’s MPI processes. For both thread and distributed parallelization, VPs simplify handling of neuron and synapses distributions. Virtual processes are distributed round-robin (i.e. each VP is allocated equal time slices, without any given a priority) onto the MPI processes and counted continuously over all processes.

![Thread: T₀, T₁, T₂, T₃
VP: V₁, V₂, V₁, V₃
Process: P₀, P₁](image)

Figure 9.3.1: Basic scheme showing how threads (T) and virtual processes (VP) reside in MPI processes (P) in NEST

9.3.3 Node distributions

The distribution of nodes depends on the type of node.

In the figure below, a node distribution for a small network consisting of spike_generator, four iaf_psc_alpha neurons, and a spike_detector in a scenario with two processes with two threads each.

**Note:** 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
Neuron distribution

Neurons are assigned to one of the virtual processes in a round-robin fashion. On all other virtual processes, no object is created. Proxies ensure the id of the real node on a given VP is kept free.

The virtual process $id_{vp}$ on which a neuron with global id $gid_{node}$ is allocated is given by $id_{vp} = gid_{node} \times \text{number of virtual processes in the simulation}$.

Device Distribution

Devices are replicated once on each thread in order to balance the load and minimize their interaction. Devices thus do not have proxies on remote virtual processes.

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

\[ [\text{model}|\text{label}]-gid-vp.[\text{dat}|\text{gdf}] \]

The first part is the name of the model (e.g. `voltmeter` or `spike_detector`) or, if set, the label of the recording device. Next is the global id (GID) of the recording device, followed by the id of the VP assigned to the recorder. Spike files have the file extension `gdf` and analog recordings from the `multimeter` have `dat` as file extension.

The label and file_extension of a recording device can be set like any other parameter of a node using SetStatus.

9.3.4 Spike exchange and synapse update

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

- 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.
- But the virtual process assigned to the target neuron, always handles the corresponding spike delivery (see property `vp` in the status dictionary).
Spikes between neurons and devices

- 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) bypasses the global spike exchange mechanism.
- 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.

Synaptic plasticity models

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

9.3.5 Using multiple threads

Thread-parallel simulation 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, the default mode is set to a single thread 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

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

Usually, a good choice for $T$ is the number of processor cores available on your machine.

**Note:** In some situations, oversubscribing (i.e., to specify a local_num_threads that is higher than available cores on your machine) can yield 20-30% improvement in simulation speed. Finding the optimal thread number for a specific situation might require a bit of experimenting.

9.3.6 Using distributed computing

Build requirements

To compile NEST for distributed computing, you will need

- a library implementation of MPI on your system. If you are on a cluster, you most likely have this already.
- NEST development packages in the case of pre-packaged MPI library.

**Note:** Please be advised that NEST should currently only be run in a homogeneous MPI environment. Running in a heterogeneous environment can lead to unexpected results or even crashes. Please contact the NEST community if you require support for exotic setups.
Configure

If using the standard installation instructions when calling `cmake`, add the option `Dwith-mpi=ON`. The build summary should report that MPI is linked.

Please see the Installation instructions for more information on installing NEST.

Run distributed simulations

Distributed simulations cannot be run interactively, which means that the simulation has to be provided as a script. However, the script can be the same as a script for any simulation. No changes are necessary for distributed simulation scripts: 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 should look 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.

9.3.7 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})
```
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.

### 9.4 Random numbers

#### 9.4.1 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-nest>` and `Examples` below.
9.4.2 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: +infinity)`.
  - 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 \((\text{low}, \text{high})\) 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 \texttt{clipped_to_boundary} variants of most generators.
  - Clipped-to-boundary variants return the value \texttt{low} if a number smaller than \texttt{low} is drawn, and \texttt{high} if a number larger than \texttt{high} is drawn.
  - We believe that these variants should \textit{not} be used for new studies.

### 9.4.3 Basics of parallel simulation in NEST

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

- NEST can parallelize simulations through \textit{multi-threading}, \textit{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 \textit{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 \textit{virtual processes}. A simulation distributed across \(M\) MPI processes with \(T\) threads each, has \(N_{\text{vp}} = M \times T\) virtual processes. It is a basic design principle of NEST that simulations shall generate \textit{identical results} when run with a fixed \(N_{\text{vp}}\), no matter how the virtual processes are broken down into MPI processes and threads.
- Useful information can be obtained like this

```python
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_vps = 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:

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

### 9.4.4 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_{\text{vp}}\$ virtual processes, we will encounter \$2 N_{\text{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_{\text{vp}}\$ random generators needs to be seeded with a different seed to generate a different random num-
ber sequences. We recommend that you choose a master seed `msd` and seed the \$(2 N_{\text{vp}} + 1)\$ generators with seeds
`msd, msd+1, ..., msd+2*N_{\text{vp}}`. Master seeds for for independent experiments must differ by at least \$(2 N_{\text{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

```python
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:

```python
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:

```python
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

```python
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:

```python
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()`.

9.4. Random numbers
Setting different per-processes RNGs

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

\[
\text{nest.sli.run('0 << /rngs [102 103] { rngdict/MT19937 :: exch CreateRNG } Map >> SetStatus')}
\]

The following happens here:

- \([102 103] \{ \text{rngdict/MT19937 :: exch CreateRNG } \} \text{ Map} \) 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.

9.4.5 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 \text{poisson\_generator} or randomized connection routines such as \text{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:

\[
\text{pyrngs = [numpy.random.RandomState(s) for s in range(msd, msd+N_{vp})]}
\text{nodes = nest.Create('iaf\_psc\_delta', 10)}
\text{node\_info = nest.GetStatus(nodes)}
\text{local\_nodes = [(ni['global\_id'], ni['\text{vp}\']) for ni in node\_info if ni['\text{local}\']]}$
\text{for gid, vp in local\_nodes:}$
\text{nest.SetStatus([gid], {'V_m': pyrngs[vp].uniform(-70.0, -50.0)})}
\]

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: \text{local, model} and \text{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:
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

```python
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.

```python
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
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

```bash
mkdir 41 42 44
cd 41
mpirun -np 1 python test.py
```

```bash
cd ../42
mpirun -np 2 python test.py
```

```bash
cd ../44
mpirun -np 4 python test.py
cd ..
```

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

```bash
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.)

## 9.5 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:

  ```python
  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:

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

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

  ```python
  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.

### 9.5.1 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.,
       9.5])
```

### 9.5.2 An example

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

```python
code
def main():
    import nest
    import numpy as np
    import matplotlib.pyplot as plt

    # 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']

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

    plt.subplot(212)
    plt.plot(t, events['g_ex'], t, events['g_in'])
    plt.axis([0, 100, 0, 45.0])
    plt.xlabel('Time [ms]')
    plt.ylabel('Synaptic conductance [nS]')
    plt.legend(('g_exc', 'g_inh'))
```
9.6 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.

9.6.1 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:

```python
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

9.6.2 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:

```python
import nest
import random
import numpy as np

# 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.

9.6.3 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:

```python
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).

9.7 Simulations with precise spike times

The simulation resolution $h$ and the minimum synaptic transmission delay $d_{\text{min}}$ 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 $d_{\text{min}}$, where $d_{\text{min}}$ 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 sub-
threshold 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 $d_{\text{min}}$-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.

![Diagram](image1)

Figure 9.7.1: 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, $d_{\text{min}}=2h$.

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.

![Diagram](image2)

Figure 9.7.2: Propagation of membrane potential in case of off-grid spiking. Dashed red line indicates precise time of threshold crossing.

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.
9.7.1 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_ps 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.

iaf_psc_alpha_ps 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.

9.7.2 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.

9.8 Using NEST with MUSIC

9.8.1 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.

You can find the installation instructions for MUSIC on their Github Page: INCF/MUSIC

Reference

9.8.2 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 potential, and connect it with the neuron.

```
vm = nest.Create('voltmeter')
nest.SetStatus(vm, {'to_memory': False, 'to_screen': True})
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.

```python
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.

```python
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.
9.8.3 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 milliseconds.

The script that sets up the receiving side (msgtest.py) of the example is shown in the following script:

```
#!/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:

while time < 1000:
    nest.Simulate (10)
    data = nest.GetStatus(mmip, 'data')
    print data
    time += 10
```

We then run the example using

```
-61.9174
-70
-70
-70
-65.2054
-62.1583
```

Note: In MUSIC, the default unit for time is seconds for the specification of times, while NEST uses milliseconds.
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="https://www.nest-initiative.org" rel="nofollow">https://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]:
  Simulation finished.

[{'messages': ['Hello', '!'], 'message_times': array([ 300., 700.])}]
```

9.8. Using NEST with MUSIC
9.8.4 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:

```
constsource

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': 'msgdata'})

# 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
```
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="https://www.nest-initiative.org" rel="nofollow">https://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.])]

9.8. Using NEST with MUSIC
10.1 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.

10.2 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')
  ```

10.2.1 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')
```

## 10.3 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.
REFERENCE MATERIAL

- Here you can find the PyNEST APIs
- The Command Index contains a list of all SLI and C++ related reference material.
12.1 Mailing List

The NEST users 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.

For more information, subscription instructions and access to the archives, please visit https://www.nest-initiative.org/mailinglist/.

12.2 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.

12.3 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.

12.4 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.
13.1 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.

13.2 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.
13.3 GNU GENERAL PUBLIC LICENSE

13.3.1 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.

13.3.2 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.

13.3.3 END OF TERMS AND CONDITIONS
INDEX

N
NumProcesses, 148

P
ProcessorName, 148

R
Rank, 148

S
SyncProcesses, 148