NeMo user manual

Andreas K. Fidjeland ⟨andreas.fidjeland@imperial.ac.uk⟩

October 1, 2010 Version 0.6.2

Abstract

NeMo is a library for discrete-time simulation of spiking neural networks. It is aimed at real-time simulation of tens of thousands of neurons on a single workstation. NeMo runs on parallel hardware; In particular it can run on CUDA-enabled GPUs. No parallel programming is required, however, on the part of the end user, as parallelisation is handled by the library. The library has interfaces in C++, C, Python, and Matlab.

1 A short tutorial introduction

The NeMo library can be used to simulate a network of neurons based on the Izhikevich neuron model [1]. The library exposes three basic types of objects: network, configuration, and simulation. Setting up and running such a simulation involves:

1. Creating a network object and adding neurons and synapses;
2. Creating a configuration object and setting its parameters as appropriate;
3. Creating a simulation object from the network and configuration objects and running the simulation.

The following section shows basic usage of the library using the C++ interface. The other language interfaces (Section 3) have very similar usage.

1.1 Constructing a network

Network construction is performed using a low-level interface where neurons and synapses are added individually. Higher-level construction interfaces, e.g. using various forms of projections, can be built on top of this, but is not part of NeMo.

First create an empty network and add a few neurons:

```
nemo::Network net;
net.addNeuron(0, 0.02, 0.20, -61.3, 6.5, -13.0, -65.0, 0.0);
net.addNeuron(1, 0.06, 0.23, -65.0, 2.0, -14.6, -65.0, 0.0);
```

To create a neuron the user specifies a unique index (which can be used when adding connections) as well as the neuron parameters (Section 2.1).

Synapses can be added by specifying the source and target neurons as well as the weight, conductance delay (in milliseconds), and a plasticity flag. For example, to connect the two above neurons in a recurrent fashion with one excitatory synapse and one inhibitory synapse one could do something like:

```
net.addSynapse(0, 1, 10, 1.0, true);
net.addSynapse(1, 0, 1, -0.5, false);
```

This simple network is sufficient for illustrating the usage of the library.

1.2 Creating a configuration

The configuration object specifies simulation-wide parameters, such as a global STDP function (disabled by default). It also specifies which of the available backends (Section **??**) will be used. In many cases a default configuration can be used.

```
nemo::Configuration conf;
```

A default-constructed configuration object will choose the best backend, but if a specific backend is desired the user can set this explicitly:

```
conf.setCudaBackend();
```

1.3 Creating and running a simulation

We can now create a simulation object from the network and configuration objects.

```
nemo::Simulation* sim = simulation(net, conf);
```

We can then run the simulation by stepping through it one millisecond at a time, getting back a vector of fired neuron indices for each call:

```
for(unsigned ms=0; ms < 1000; ++ms) {
  const std::vector<unsigned>& fired = sim->step();
}
```

We can also provide external stimulus to the network, by forcing specific neurons to fire. For example, to force neuron 0 and 1 to fire synchronized at a steady

10Hz for 10 seconds one could do the following (ignoring firing output for the time being):

```
std::vector<unsigned> stimulus;
stimulus.push_back(0);
stimulus.push_back(1);
for(unsigned ms=0; ms < 10000; ++ms) {
  if(ms % 100 == 0) {
    sim->step(stimulus);
  } else {
    sim->step();
  }
}
```

The above shows the basic usage of the simulator. The user can perform other actions on the simulation object as well including querying synapse data, and use STDP.

For full details of library usage refer to the language-specific notes (Section 3) and the full function reference (Section 4).

2 Simulation model

2.1 Neuron model

Neurons are based on the Izhikevich neuron model [1]. The model consists of a two-dimensional system of ordinary differential equations defined by

$$\dot{v} = 0.04v^2 + 5v + 140 - u + I \tag{1}$$

$$\dot{u} = a(bv - u) \tag{2}$$

with an after-spike resetting

$$\text{if } v \geq 30 \text{ mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{3}$$

where v represents the membrane potential and u the membrane recovery variable, accounting for the activation of K^+ and the inactivation of Na^+ providing post-potential negative feedback to v. The parameter a describes the time scale of the recovery variable, b describes its sensitivity to sub-threshold fluctuations, c gives the after-spike reset value of the membrane potential, and d describes the after-spike reset of the recovery variable. The variables a–d can be set so as to reproduce the behaviour of different types of neurons. The term I in Equation 1 represents the combined current from spike arrivals from all presynaptic neurons, which are summed every simulation cycle.

In addition to the basic model parameters a–d and state variables u and v, the user can specify a random input current to each neuron. The input current is drawn from $\mathcal{N}(0, \sigma)$, where σ is set separately for each neuron. If σ is set to zero, no input current is generated.

2.2 Basic synapse model

Synapses are specified by a conductance delay and a weight. Conductance delays are specified in whole milliseconds, with a minimum delay of 1 ms and the maximum supported delay is to 64 ms.

Synapses can be either static or plastic, using spike-timing synaptic plasticity, the details of which can be found in the next section.

2.3 STDP model

`NeMo` supports spike-timing dependant plasticity, i.e. synapses can change during simulation depending on the temporal relationship between the firing of the pre- and post-synaptic neurons. To make use of STDP the user must enable STDP globally by specifying an STDP function and enable plasticity for each synapse as appropriate when constructing the network. A single STDP function is applied to the whole network.

Synapses can be either potentiated or depressed. With STDP enabled, the simulation accumulates a weight change which is the sum of potentiation and depression for each synapse. Potentiation always moves the synaptic weight away from zero, which for excitatory synapses is more positive, and for inhibitory synapses is more negative. Depression always moves the synapses weight towards zero. The accumulation of potentiation and depression statistics takes place every cycle, but the modification of the weight only takes place when explicitly requested.

Generally a synapse is potentiated if a spike arrives shortly before the postsynaptic neuron fires. Conversely, if a spike arrives shortly after the postsynaptic firing the synapse is depressed. Also, the effect of either potentiation or depression generally weakens as the time difference, dt, between spike arrival and firing increases. Beyond certain values of dt before or after the firing, STDP has no effect. These limits for dt specify the size of the STDP window.

The user can specify the following aspects of the STDP function:

- the size of the STDP window;
- what values of dt cause potentiation and which cause depression;
- the strength of either potentiation or depression for each value of dt, i.e. the shape of the discretized STDP function;
- maximum weight of plastic excitatory synapses; and
- minimum weight of plastic inhibitory synapses.

Since the simulation is discrete-time, the STDP function can be specified by providing values of the underlying function sampled at integer values of dt. For any value of dt a positive value of the function denotes potentiation, while a negative value denotes depression. The STDP function is described using two vectors: one for spike arrivals *before* the postsynaptic firing (pre-post pair), and one for spike arrivals *after* the postsynaptic firing (post-pre pair). The total length of these two vectors is the size of the STDP window. The typical scheme is to have positive values for pre-post pairs and negative values for post-pre pairs, but other schemes can be used.

When accumulating statistics a pairwise nearest-neighbour protocol is used. For each postsynaptic firing potentiation and depression statistics are updated based on the nearest pre-post spike pair (if any inside STDP window) and the nearest post-pre spike pair (if any inside the STDP window).

Excitatory synapses are never potentiated beyond the user-specified maximum weight, and are never depressed below zero. Likewise, inhibitory synapses are never potentiated beyond the user-specified minimum weight, and are never depressed above zero. Synapses can thus be deactivated, but never change from excitatory to inhibitory or vice versa.

2.4 Discrete-time simulation

The simulation is discrete-time with a fixed one millisecond step size. Within each step the following actions take place in a fixed order:

1. Compute accumulated current for incoming spikes;
2. Update the neuron state;
3. Determine if any neurons fired. The user can specify neurons which should be forced to fire at this point;
4. Update the state of the fired neurons
5. Accumulate STDP statistics, if STDP is enabled

2.5 Neuron and synapse indices

The user specifies the unique index of each neuron. These are just regular unsigned integers. The neuron indices does not *have* to start from zero and lie in a contiguous range, but in the current implementation such a simple indexing scheme may lead to better memory usage.

Synapses also have unique indices, but these are assigned by the library itself. Synapse indices are only required if querying the synapse state at run-time.

2.6 Numerical precision

The state update step uses the Euler method with a step size of 0.25ms.

The weights are stored internally in a fixed-point format (Q11.20) for two reasons. First, it is then possible to get repeatable results regardless of the order in which synapses are processed in a parallel setting (fixed-point addition is commutative, unlike floating point addition). Second, it results in better performance, at least on the CUDA backend, where atomic operations are available for integer/fixed-point but not for floating point. The fixed-point format should not overflow for synapses with remotely plausible weights, but the current accumulation uses saturating arithmetic nonetheless.

Neuron parameters are stored as single-precision floating point.

3 Application programming interface

`NeMo` is implemented as a C++ class library and can thus be used directly in programs written in C++. There are also bindings in C (3.2), Python (**??**), and Matlab (3.3). The different language APIs follow largely the same programming model. The following sections specify the language-specific issues (linking, naming schemes, etc) while a full function reference, applying to all language interfaces can be found in Section 4.

3.1 C++ API

The C++ API is used by including the header file `nemo.hpp` and linking against the nemo dynamic library (`libnemo.so` or `nemo.dll`).

All classes and functions are found in the `nemo` namespace. Class names use initial upper-case. Function names use camelCase with initial lower-case letter.

The library is not thread safe.

Errors are reported via exceptions of type `nemo::exception`. These are subclasses of `std::exception`, so a descriptive error messages is availble using `const char* nemo::exception::what()`. Additionally, internally generated exceptions also carry an error number (`int nemo::exception::errorNumber()`) which are listed in `<nemo/types.hpp>`. If disambiguation between different `NeMo`-generated error types is not required, it is sufficient to simply catch `std::exception&`.

The following code snippet shows basic usage. The `NeMo` distribution contains an example directory with more advanced examples.

```
#include <nemo.hpp>

...

try {
  nemo::Network net;
  net.addNeuron(0,0.02,0.20,-61.3,6.5,-13.0,-65.0,0.0);
  net.addNeuron(1,0.06,0.23,-65.0,2.0,-14.6,-65.0,0.0);
  net.addSynapse(0, 1, 10, 1.0, true);
  net.addSynapse(1, 0, 1, -0.5, false);

  nemo::Configuration conf;

  boost::scoped_ptr<nemo::Simulation>
    sim(nemo::simulation(net, conf));

  for(unsigned ms=0; ms < 1000; ++ms) {
    const vector<unsigned>& fired = sim->step();
    for(vector<unsigned>::const_iterator n = fired.begin();
        n != fired.end(); ++n) {
      cout << ms << "␣" << *n << endl;
    }
  }

} catch(exception& e) {
  cerr << e.what() << endl;
}
```

3.2 C API

The C API follows the general object-model as outlined above.

To use the C API, include the header file `nemo.h` instead of `nemo.hpp`, and then link to libnemo.

All names use lower case and are separated by underscores. Both function and type names are prefixed '`nemo_`' and type names are also suffixed '`_t`'.

In the C API the network, configuration, and simulation objects are controlled via opaque pointers with typedefed names `nemo_network_t`, `nemo_configuration_t`, and `nemo_simulation_t`. These objects are generated with methods `nemo_new_x` (`x` = `network`, `configuration`, or `simulation`), and should be explicitly destroyed with the corresponding methods `nemo_delete_x`.

Methods on specific objects take the relevant opaque pointer as the first parameter.

Error handling is done via return codes. All API functions return a value of type `nemo_status_t`, which will be `NEMO_OK` if everything went fine and some other value (see `<nemo/types.h>`) otherwise.

The C API is not thread-safe.

The following C program program snippet shows basic usage of the `NeMo` library (without any error handling):

```
#include <nemo.h>

...

nemo_network_t net = nemo_new_network();
nemo_add_neuron(net,0,0.02,0.20,-61.3,6.5,-13.0,-65.0,0.0);
nemo_add_neuron(net,1,0.06,0.23,-65.0,2.0,-14.6,-65.0,0.0);
nemo_add_synapse(net, 0, 1, 10, 1.0, true);
nemo_add_synapse(net, 1, 0, 1, -0.5, false);

nemo_configuration_t conf = nemo_new_configuration();
nemo_simulation_t sim = nemo_new_simulation(net, conf);

for(unsigned ms=0; ms < 1000; ms++) {
  unsigned *fired, nfired;
  nemo_step(sim, NULL, 0, &fired, &nfired);
  for(unsigned i=0; i < nfired; i++) {
    printf("%u␣%u\n", ms, nfired[i]);
  }
}

nemo_delete_simulation(sim);
nemo_delete_configuration(conf);
nemo_delete_network(net);
```

3.3 Matlab API

The Matlab API follows the same general object-model as outlined above. It is implemented using MEX and the object-oriented features of Matlab. Note that object-orientation is not supported in older versions of Matlab (prior to circa 2007).

Both classes and member functions use camelCased identifiers. Since Matlab lacks namespaces, or indeed any module system, the class names are prefixed with '`nemo`': `nemoNetwork`, `nemoConfiguration`, and `nemoSimulation`.

Help is available for each class and for each function using Matlab's regular help system, i.e. via calls such as `help nemoNetwork` and `help addNeuron` (or `help nemoNetwork/addNeuron`).

Internal `NeMo` errors result in regular Matlab errors, (i.e. as when `error` is called in a script).

The Matlab path must contain the directory with the m-files defining the three classes and the MEX library that interfaces with libnemo. On Windows this directory defaults to `C:\Program Files\nemo-<version>\Matlab`, and on Linux to `/usr/share/nemo/matlab`. Use `addpath` from within Matlab to set the path.

Additionally, the nemo libraries (plus any dependencies such as possibly the CUDA runtime library) needs to be on the system path. Note that this is different from the Matlab path. If the system path is not set correctly Matlab will issue a rather unhelpful message about the MEX-file being invalid.

The following shows a simple Matlab session using `NeMo` to set up a minimal network and run it for 1000ms printing pairs of time and fired neuron number:

```
net = nemoNetwork;
net.addNeuron(0, 0.02, 0.20, -61.3, 6.5, -13.0, -65.0, 0.0);
net.addNeuron(1, 0.06, 0.23, -65.0, 2.0, -14.6, -65.0, 0.0);
net.addSynapse(0, 1, 10, 1.0, true);
net.addSynapse(1, 0, 1, -0.5, false);
conf = nemoConfiguration;
sim = nemoSimulation(net, conf);
for ms=1:1000
  fired = sim.step;
  [fired; ones*ms]
```

4 Function reference

4.1 Network class

A Network is constructed by adding individual neurons synapses to the network. Neurons are given indices (from 0) which should be unique for each neuron. When adding synapses the source or target neurons need not necessarily exist yet, but should be defined before the network is finalised.

C++:

```
Network::Network()
```

C:

```
nemo_network_t
nemo_new_network()

nemo_delete_network(nemo_network_t net)
```

Matlab:

```
net = nemoNetwork()
```

Functions

- addNeuron
- addSynapse
- neuronCount

Network::addNeuron

add a single neuron to network

C++:

```
void
Network::addNeuron(unsigned idx, float a, float b, float c,
        float d, float u, float v, float sigma)
```

C:

```
nemo_status_t
nemo_add_neuron(nemo_network_t net, unsigned idx, float a,
        float b, float c, float d, float u, float v, float sigma)
```

Matlab:

```
net.addNeuron(idx, a, b, c, d, u, v, sigma)
```

Inputs

idx Neuron index (0-based)

a Time scale of the recovery variable

b Sensitivity to sub-threshold fluctuations in the membrane potential v

c After-spike value of the membrane potential v

d After-spike reset of the recovery variable u

u Initial value for the membrane recovery variable

v Initial value for the membrane potential

sigma Parameter for a random gaussian per-neuron process which generates random input current drawn from an N(0, sigma) distribution. If set to zero no random input current will be generated

The neuron uses the Izhikevich neuron model. See E. M. Izhikevich "Simple model of spiking neurons", IEEE Trans. Neural Networks, vol 14, pp 1569-1572, 2003 for a full description of the model and the parameters.

Network::addSynapse

add a single synapse to the network

C++:

```
uint64_t
Network::addSynapse(unsigned source, unsigned target,
        unsigned delay, float weight, unsigned char plastic)
```

C:

```
nemo_status_t
nemo_add_synapse(nemo_network_t net, unsigned source,
        unsigned target, unsigned delay, float weight,
        unsigned char plastic, uint64_t* id)
```

Matlab:

```
id = net.addSynapse(source, target, delay, weight, plastic)
```

Inputs

source Index of source neuron

target Index of target neuron

delay Synapse conductance delay in milliseconds

weight Synapse weights

plastic Boolean specifying whether or not this synapse is plastic

Outputs

id Unique synapse ID

Network::neuronCount

C++:

```
unsigned
Network::neuronCount()
```

C:

```
nemo_status_t
nemo_neuron_count(nemo_network_t net, unsigned* ncount)
```

Matlab:

```
ncount = net.neuronCount()
```

Outputs

ncount number of neurons in the network

4.2 Configuration class

C++:

```
Configuration::Configuration()
```

C:

```
nemo_configuration_t
nemo_new_configuration()

nemo_delete_configuration(nemo_configuration_t conf)
```

Matlab:

```
conf = nemoConfiguration()
```

Functions

- setCpuBackend
- setCudaBackend
- setStdpFunction
- backendDescription

Configuration::setCpuBackend

specify that the CPU backend should be used

C++:

```
void
Configuration::setCpuBackend(int tcount)
```

C:

```
nemo_status_t
nemo_set_cpu_backend(nemo_configuration_t conf, int tcount)
```

Matlab:

```
conf.setCpuBackend(tcount)
```

Inputs

tcount number of threads

Specify that the CPU backend should be used and optionally specify the number of threads to use. If the default thread count of -1 is used, the backend will choose a sensible value based on the available hardware concurrency.

Configuration::setCudaBackend

specify that the CUDA backend should be used

C++:

```
void
Configuration::setCudaBackend(int deviceNumber)
```

C:

```
nemo_status_t
nemo_set_cuda_backend(nemo_configuration_t conf, int deviceNumber)
```

Matlab:

```
conf.setCudaBackend(deviceNumber)
```

Inputs

deviceNumber

Specify that the CUDA backend should be used and optionally specify a desired device. If the (default) device value of -1 is used the backend will choose the best available device. If the cuda backend (and the chosen device) cannot be used for whatever reason, an exception is raised. The device numbering is the numbering used internally by nemo (see cudaDeviceCount and cudaDeviceDescription). This device numbering may differ from the one provided by the CUDA driver directly, since nemo ignores any devices it cannot use.

Configuration::setStdpFunction

enable STDP and set the global STDP function

C++:

```
void
Configuration::setStdpFunction(const vector<float>& prefire,
        const vector<float>& postfire, float minWeight, float maxWeight)
```

C:

```
nemo_status_t
nemo_set_stdp_function(nemo_configuration_t conf,
        float prefire[], size_t prefire_len,
        float postfire[], size_t postfire_len, float minWeight,
        float maxWeight)
```

Matlab:

```
conf.setStdpFunction(prefire, postfire, minWeight, maxWeight)
```

Inputs

prefire STDP function values for spikes arrival times before the postsynaptic firing, starting closest to the postsynaptic firing

postfire STDP function values for spikes arrival times after the postsynaptic firing, starting closest to the postsynaptic firing

minWeight Lowest (negative) weight beyond which inhibitory synapses are not potentiated

maxWeight Highest (positive) weight beyond which excitatory synapses are not potentiated

The STDP function is specified by providing the values sampled at integer cycles within the STDP window.

Configuration::backendDescription

Description of the currently selected simulation backend

C++:

```
std::string
Configuration::backendDescription()
```

C:

```
nemo_status_t
nemo_backend_description(nemo_configuration_t conf, const char** description)
```

Matlab:

```
description = conf.backendDescription()
```

Outputs

description Textual description of the currently selected backend

The backend can be changed using setCudaBackend or setCpuBackend

4.3 Simulation class

A simulation is created from a network and a configuration object. The simulation is run by stepping through it, providing stimulus as appropriate. It is possible to read back synapse data at run time. The simulation also maintains a timer for both simulated time and wallclock time.

C++:

```
Simulation*
simulation(const Network&, const Configuration&)
```

C:

```
nemo_simulation_t
nemo_new_simulation(nemo_network_t net, nemo_configuration_t conf)

nemo_delete_simulation(nemo_simulation_t sim)
```

Matlab:

```
sim = nemoSimulation(net, conf)
```

Functions

- step
- applyStdp
- getTargets
- getDelays
- getWeights
- getPlastic
- elapsedWallclock
- elapsedSimulation
- resetTimer

Simulation::step

run simulation for a single cycle (1ms)

C++:

```
const vector<unsigned>&
Simulation::step(const vector<unsigned>& fstim)
```

C:

```
nemo_status_t
nemo_step(nemo_simulation_t sim,
        unsigned fstim[], size_t fstim_len,
        unsigned* fired[], size_t* fired_len)
```

Matlab:

```
fired = sim.step(fstim)
```

Inputs

fstim An optional list of neurons, which will be forced to fire this cycle

Outputs

fired Neurons which fired this cycle

Simulation::applyStdp

update synapse weights using the accumulated STDP statistics

C++:

```
void
Simulation::applyStdp(float reward)
```

C:

```
nemo_status_t
nemo_apply_stdp(nemo_simulation_t sim, float reward)
```

Matlab:

```
sim.applyStdp(reward)
```

Inputs

reward Multiplier for the accumulated weight change

Simulation::getTargets

return the targets for the specified synapses

C++:

```
const vector<unsigned>&
Simulation::getTargets(const vector<uint64_t>& synapses)
```

C:

```
nemo_status_t
nemo_get_targets(nemo_simulation_t sim,
        uint64_t synapses[], size_t synapses_len, unsigned* targets[])
```

Matlab:

```
targets = sim.getTargets(synapses)
```

Inputs

synapses synapse ids (as returned by addSynapse)

Outputs

targets indices of target neurons

Simulation::getDelays

return the conductance delays for the specified synapses

C++:

```
const vector<unsigned>&
Simulation::getDelays(const vector<uint64_t>& synapses)
```

C:

```
nemo_status_t
nemo_get_delays(nemo_simulation_t sim,
        uint64_t synapses[], size_t synapses_len, unsigned* delays[])
```

Matlab:

```
delays = sim.getDelays(synapses)
```

Inputs

synapses synapse ids (as returned by addSynapse)

Outputs

delays conductance delays of the specified synpases

Simulation::getWeights

return the weights for the specified synapses

C++:

```
const vector<float>&
Simulation::getWeights(const vector<uint64_t>& synapses)
```

C:

```
nemo_status_t
nemo_get_weights(nemo_simulation_t sim,
        uint64_t synapses[], size_t synapses_len, float* weights[])
```

Matlab:

```
weights = sim.getWeights(synapses)
```

Inputs

synapses synapse ids (as returned by addSynapse)

Outputs

weights weights of the specified synapses

Simulation::getPlastic

return the boolean plasticity status for the specified synapses

C++:

```
const vector<unsigned char>&
Simulation::getPlastic(const vector<uint64_t>& synapses)
```

C:

```
nemo_status_t
nemo_get_plastic(nemo_simulation_t sim,
        uint64_t synapses[], size_t synapses_len, unsigned char* plastic[])
```

Matlab:

```
plastic = sim.getPlastic(synapses)
```

Inputs

synapses synapse ids (as returned by addSynapse)

Outputs

plastic plasticity status of the specified synpases

Simulation::elapsedWallclock

C++:

```
unsigned long
Simulation::elapsedWallclock()
```

C:

```
nemo_status_t
nemo_elapsed_wallclock(nemo_simulation_t sim, unsigned long* elapsed)
```

Matlab:

```
elapsed = sim.elapsedWallclock()
```

Outputs

elapsed number of milliseconds of wall-clock time elapsed since first simulation step (or last timer reset)

Simulation::elapsedSimulation

C++:

```
unsigned long
Simulation::elapsedSimulation()
```

C:

```
nemo_status_t
nemo_elapsed_simulation(nemo_simulation_t sim, unsigned long* elapsed)
```

Matlab:

```
elapsed = sim.elapsedSimulation()
```

Outputs

elapsed number of milliseconds of simulation time elapsed since first simulation step (or last timer reset)

Simulation::resetTimer

reset both wall-clock and simulation timer

C++:

```
void
Simulation::resetTimer()
```

C:

```
nemo_status_t
nemo_reset_timer(nemo_simulation_t sim)
```

Matlab:

```
sim.resetTimer()
```

5 Installation

5.1 Windows

The easiest way to install is to use the precompiled library (NSIS installer). Both header (C and C++) and library files are stored in `c:\Program Files\nemo-<version>`. Python bindings, Matlab bindings, and examples are stored in subdirectories of this.

Alternatively, the library can be built from source using `cmake` to generate a Visual Studio project file, and then building from within Visual Studio. Builds in Cygwin or MSys/MinGW have not been tested.

5.2 Linux

There are no precompiled binaries for linux, so the library should be built from source using `cmake`. By default, headers are installed to `/usr/local/include`, the library files to `/usr/local/lib`, Python bindings, Matlab bindings, examples and documentation to subdirectories of `/usr/loca/share/nemo`.

5.3 Building from source

`NeMo` relies on the boost headers. Additionally the following dependencies may be needed depending on what `cmake` configuration options are loaded:

Feature	`cmake` option	Dependency
CUDA backend	`NEMO_CUDA_ENABLED`	Cuda toolkit
dynamic loading	`NEMO_CUDA_ENABLED`	`libltdl` (on Linux only)
Timing function	`NEMO_TIMING_ENABLED`	boost date_time
Python bindings	`NEMO_PYTHON_ENABLED`	boost python
Matlab bindings	`NEMO_MATLAB_ENABLED`	Matlab (including `mex` compiler)
Example programs	`NEMO_EXAMPLES_ENABLED`	boost program_options

The basic `cmake` build procedure is

```
cd <nemo-directory>
mkdir build
cd build
cmake ..
make
sudo make install
```

References

[1] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. Neural Networks*, 14:1569–1572, 2003.