MIIND

Marc de Kamps, Hugh Osborne

May 12, 2021

MIIND

1 Installation **3**
- 1.1 Install MIIND package with pip . . . 3
- 1.2 Standalone package . . . 3
- 1.3 Standalone Docker . . . 3
- 1.4 Building Python MIIND From Source . . . 4
- 1.5 Building Standalone MIIND From Source . . . 5

2 Quick Start Guide **7**

3 Publications **11**

4 Indices and tables **13**

MIIND is a simulator that allows the creation, simulation and analysis of large-scale neural networks. It does not model individual neurons, but models populations directly, similarly to a neural mass models, except that we use population density techniques. Population density techniques are based on point model neurons, such as leaky-integrate-and-fire (LIF), quadratic-integrate-and-fire neurons (QIF), or more complex ones, such as adaptive-exponential-integrate-and-fire (AdExp), Izhikevich, Fitzhugh-Nagumo (FN). MIIND is able to model populations of 1D neural models (like LIF, QIF), or 2D models (AdExp, Izhikevich, FN, others). It does so by using statistical techniques to answer the question: "If I'd run a NEST or BRIAN simulation (to name some point model-based simulators), where in state space would my neurons be?" We calculate this distribution in terms of a density function, and from this density function we can infer many properties of the population, including its own firing rate. By modeling large-scale networks as homogeneous populations that exchange firing rate statistics, rather than spikes, remarkable efficiency can be achieved, whilst retaining a connection to spiking neurons that is not present in neural mass models.

CHAPTER

ONE

INSTALLATION

1.1 Install MIIND package with pip

To install with pip:

```
$ python -m pip install miind
```

MIIND is available on Windows, MacOS, and Linux for python versions >=3.6.

1.2 Standalone package

MIIND with CUDA Support

MIIND without CUDA Support

Additional python libraries which need to be installed using pip or conda:

- numpy
- matplotlib
- shapely
- descartes
- scipy

1.3 Standalone Docker

Pull MIIND from DockerHub:

```
$ docker pull hughosborne/miind:latest
```

CUDA is currently disabled for the MIIND Docker image.

1.4 Building Python MIIND From Source

Build and Install Python MIIND Locally:

```
$ python setup.py install
```

Python MIIND depends on:

- Boost
- GSL
- Freeglut
- OpenGL
- FFTW
- PugiXML
- Python3-Dev (Python.h)

Python MIIND optionally depends on:

- CUDA Toolkit
- OpenMP
- MPI
- ROOT

setup.py defines the cmake options for building MIIND in the variable cmake_args. When using setup.py to build MIIND, these options should be changed if a different configuration is required to the default (ENABLE_OPENMP, ENABLE_CUDA, ENABLE_TESTING). Note that platform specific versions of cmake_args are defined later in the script.

Listing 1: cmake-args

```
    cmake_args = (
            [
                    '-DCMAKE_BUILD_TYPE=Release',
                    '-DENABLE_OPENMP:BOOL=ON',
                    '-DENABLE_MPI:BOOL=OFF',
                    '-DENABLE_TESTING:BOOL=ON',
                    '-DENABLE_CUDA:BOOL=ON',
                    '-DENABLE_ROOT:BOOL=OFF',
                    '-DCMAKE_CUDA_FLAGS=--generate-code=arch=compute_30,code=[compute_
↪30,sm_30]'
            ]
    )
```

For example, to build MIIND with CUDA disabled and ROOT enabled.

Listing 2: cmake-args with CUDA disabled and ROOT enabled

```
    cmake_args = (
            [
                    '-DCMAKE_BUILD_TYPE=Release',
                    '-DENABLE_OPENMP:BOOL=ON',
```

(continues on next page)

(continued from previous page)

```
                    '-DENABLE_MPI:BOOL=OFF',
                    '-DENABLE_TESTING:BOOL=ON',
                    '-DENABLE_CUDA:BOOL=OFF',
                    '-DENABLE_ROOT:BOOL=ON'
            ]
    )
```

On Windows, vcpkg is used for building Python MIIND therefore only CUDA drivers and Ninja are required in addition to cmake and a compiler.

1.5 Building Standalone MIIND From Source

Standalone MIIND can also be built in the tranditional way (create a build directory and run cmake then install).

Create a build directory in the MIIND root directory:

```
$ mkdir build
```

Change directory:

```
$ cd build
```

Run ccmake to set the required cmake options and generate a cmake file:

```
$ cmake ..
```

Once generated, call make install (with admin permissions if required):

```
$ make install
```

Additional python libraries which need to be installed using pip or conda:

- numpy
- matplotlib
- shapely
- descartes
- scipy

Set the following environment variables:

- OMP_NUM_THREADS (See OpenMP documentation)
- Add <MIIND_Installation_Directory>/share/miind/python to PATH
- Add <MIIND_Installation_Directory>/share/miind/python to PYTHONPATH

CHAPTER

TWO

QUICK START GUIDE

This section demonstrates how to quickly set up a simulation for a simple E-I network of conductance based neurons using MIIND. A rudimentary level of python experience is needed to run the simulation. In most cases, MIIND can be installed via Python pip. The code for this example is given in full here but can also be found at https://github.com/dekamps/miind/tree/windows_pip_support/examples/quick_start. Once MIIND is installed, it can also be loaded into a working directory with the following command:

```
$ python -m miind.loadExamples
```

In the examples/quick_start directory, the generateCondFiles.py script can be run to generate the simulation files, cond.model and cond.tmat:

```
$ python generateCondFiles.py
```

The contents of generateCondFiles.py is given below. The two important parts of the script are the neuron model function, in this case named cond(), and the call to the MIIND function grid_generate.generate(). When run, the script generates two files, cond.model and cond.tmat.

Listing 1: generateCondFiles.py

```
import miind.grid_generate as grid_generate

def cond(y,t):
        E_r = -65e-3
        tau_m = 20e-3
        tau_s = 5e-3

        v = y[0];
        h = y[1];

        v_prime = ( -(v - E_r) - (h * v) ) / tau_m
        h_prime = -h / tau_s

        return [v_prime, h_prime]

grid_generate.generate(
        func = cond,
        timestep = 1e-04,
        timescale = 1,
        tolerance = 1e-6,
        basename = 'cond',
        threshold_v = -55.0e-3,
```

(continues on next page)

(continued from previous page)

```
            reset_v = -65e-3,
            reset_shift_h = 0.0,
            grid_v_min = -72.0e-3,
            grid_v_max = -54.0e-3,
            grid_h_min = -1.0,
            grid_h_max = 2.0,
            grid_v_res = 200,
            grid_h_res = 200,
            efficacy_orientation = 'h')
```

The cond() function should be familiar to those who have used Python numerical integration frameworks such as scipy.integrate. It takes the two time dependent variables defined by y[0] and y[1] and a placeholder t parameter for performing a numerical integration. In the function, the user may define how the derivatives of each variable are to be calculated. The generate() function requires a suitable time step, values for a threshold and reset if needed, and a description of the extent of the state space to be simulated. With this structure, the user may define any two dimensional neuron model. The generated files are then referenced in a second file which describes a network of populations to be simulated. The second file, cond.xml, describes an E-I network which uses the files generated by generateCondFiles.py.

Listing 2: cond.xml

```
    <Simulation>
            <WeightType>CustomConnectionParameters</WeightType>
            <Algorithms>
                    <Algorithm type="GridAlgorithm" name="COND" modelfile="cond.model"␣
↪tau_refractive="0.0" transformfile="cond_0_0_0_0_.tmat" start_v="-0.065" start_w="0.0"␣
↪>
                            <TimeStep>1e-04</TimeStep>
                    </Algorithm>
                    <Algorithm type="RateFunctor" name="ExcitatoryInput">
                            <expression>800.</expression>
                    </Algorithm>
            </Algorithms>
            <Nodes>
                    <Node algorithm="ExcitatoryInput" name="INPUT_E" type="EXCITATORY_
↪DIRECT" />
                    <Node algorithm="ExcitatoryInput" name="INPUT_I" type="EXCITATORY_
↪DIRECT" />
                    <Node algorithm="COND" name="E" type="EXCITATORY_DIRECT" />
                    <Node algorithm="COND" name="I" type="INHIBITORY_DIRECT" />
            </Nodes>
            <Connections>
                    <Connection In="INPUT_E" Out="E" num_connections="1" efficacy="0.1"␣
↪delay="0.0"/>
                    <Connection In="INPUT_I" Out="I" num_connections="1" efficacy="0.1"␣
↪delay="0.0"/>
                    <Connection In="E" Out="I" num_connections="1" efficacy="0.1" delay=
↪"0.001"/>
                    <Connection In="E" Out="E" num_connections="1" efficacy="0.1" delay=
↪"0.001"/>
                    <Connection In="I" Out="E" num_connections="1" efficacy="-0.1"␣
↪delay="0.001"/>
                    <Connection In="I" Out="I" num_connections="1" efficacy="-0.1"␣
↪delay="0.001"/>
```

(continues on next page)

(continued from previous page)

```
        </Connections>
        <Reporting>
                <Display node="E" />
                <Display node="I" />
                <Rate node="E" t_interval="0.001" />
                <Rate node="I" t_interval="0.001" />
        </Reporting>
        <SimulationRunParameter>
                <SimulationName>EINetwork</SimulationName>
                <t_end>0.2</t_end>
                <t_step>1e-04</t_step>
                <name_log>einetwork.log</name_log>
        </SimulationRunParameter>
    </Simulation>
```

The Algorithms section is used to declare specific simulation methods for one or more populations in the network. In this case, a GridAlgorithm named COND is set up which references the cond.model and cond.tmat files. A RateFunctor algorithm produces a constant firing rate. In the Nodes section, two instances of COND are created: one for the excitatory and inhibitory populations respectively. Two ExcitatoryInput nodes are also defined. The Connections section allows us to connect the input nodes to the two conductance populations. The populations are connected to each other and to themselves with a 1ms transmission delay. The remaining sections are used to define how the output of the simulation is to be recorded, and to provide important simulation parameters such as the simulation time. By running the following python command, the simulation can be run:

```
$ python -m miind.run cond.xml
```

The probability density plots for both populations will be displayed in separate windows as the simulation progresses. The firing rate of the excitatory population can be plotted using the following commands:

```
$ python -m miind.miindio sim cond.xml
$ python -m miind.miindio rate E
```

Finally, the density function of each population can be plotted as a heat map for a given time in the simulation:

```
$ python -m miind.miindio plot-density E 0.12
```

CHAPTER

THREE

PUBLICATIONS

- De Kamps, M., Lepperød, M. and Lai, Y.M., 2019. Computational geometry for modeling neural populations: From visualization to simulation. PLoS computational biology, 15(3), p.e1006729.
- Lai, Y.M. and de Kamps, M., 2017. Population density equations for stochastic processes with memory kernels. Physical Review E, 95(6), p.062125.
- de Kamps, M., 2013. A generic approach to solving jump diffusion equations with applications to neural populations. arXiv preprint arXiv:1309.1654.
- Kamps, M.D., 2003. A simple and stable numerical solution for the population density equation. Neural computation, 15(9), pp.2129-2146.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search