

# **MOOSE Documentation**

*Release chamcham (3.1.1)*

**Upinder Bhalla, Niraj Dudani, Aditya Gilra, Aviral Goel, Subhasis P**

**May 25, 2018**



---

# Contents

---

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>What is MOOSE and what is it good for?</b> | <b>1</b>  |
| 1.1      | Installation . . . . .                        | 2         |
| 1.2      | Quick Start . . . . .                         | 5         |
| 1.3      | Cook Book . . . . .                           | 33        |
| 1.4      | Graphics . . . . .                            | 74        |
| 1.5      | References . . . . .                          | 75        |
| 1.6      | Doxygen . . . . .                             | 85        |
| 1.7      | Release Notes . . . . .                       | 85        |
| 1.8      | Changes . . . . .                             | 85        |
| 1.9      | Known issues . . . . .                        | 85        |
| <b>2</b> | <b>Indices and tables</b>                     | <b>87</b> |
|          | <b>Python Module Index</b>                    | <b>89</b> |



## What is MOOSE and what is it good for?

MOOSE is the **Multiscale Object-Oriented Simulation Environment**. It is designed to simulate neural systems ranging from subcellular components and biochemical reactions to complex models of single neurons, circuits, and large networks. MOOSE can operate at many levels of detail, from stochastic chemical computations, to multicompartment single-neuron models, to spiking neuron network models.

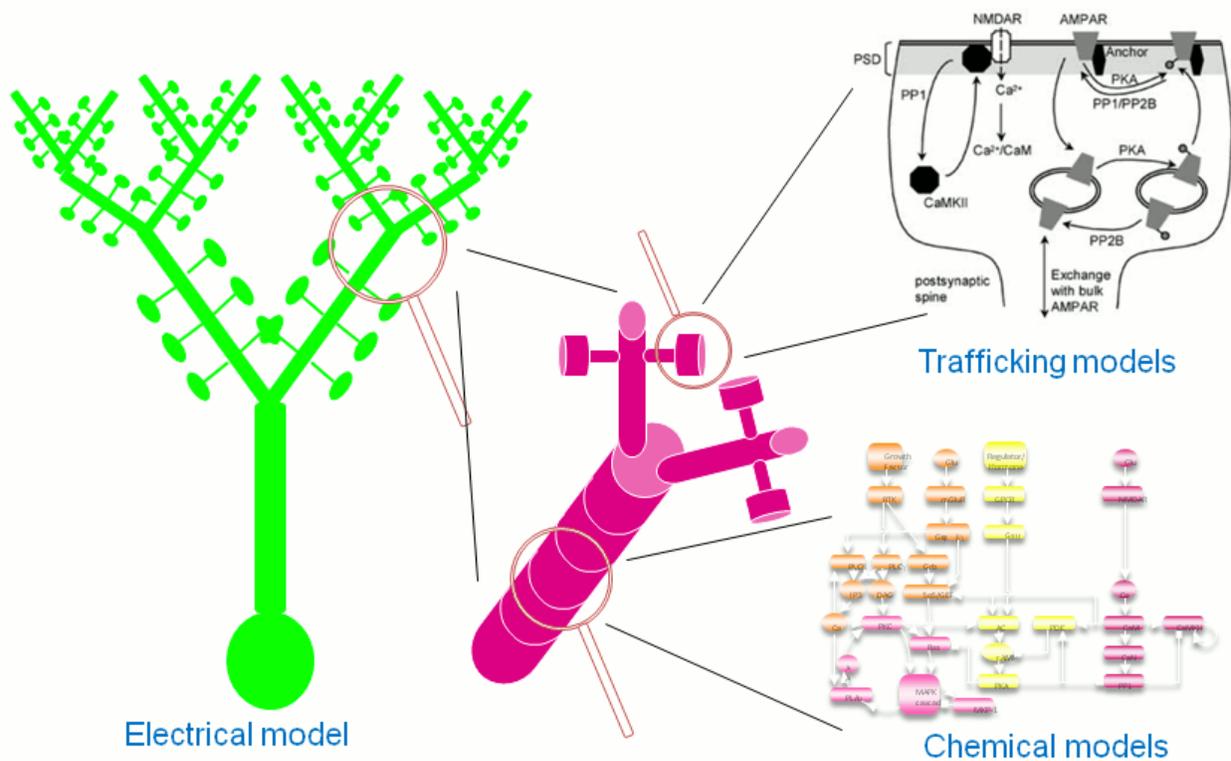


Fig. 1: Multiple scales can be modelled and simulated in MOOSE

MOOSE is multiscale: It can do all these calculations together. One of its major uses is to make biologically detailed models that combine electrical and chemical signaling.

MOOSE is object-oriented. Biological concepts are mapped into classes, and a model is built by creating instances of these classes and connecting them by messages. MOOSE also has numerical classes whose job is to take over difficult computations in a certain domain, and do them fast. There are such solver classes for stochastic and deterministic chemistry, for diffusion, and for multicompartment neuronal models.

MOOSE is a simulation environment, not just a numerical engine: It provides data representations and solvers (of course!), but also a scripting interface with Python, graphical displays with Matplotlib, PyQt, and OpenGL, and support for many model formats. These include SBML, NeuroML, GENESIS kkit and cell.p formats, HDF5 and NSDF for data writing.

## 1.1 Installation

### 1.1.1 Use pre-built packages

#### Linux

We recommend that you use our repositories hosted at [Open Build Service](#). We have build packages for most linux distributions. Visit [this page](#) to pick your distribution and follow instructions.

---

**Note:** `moogli` (tool to visualize network activity) is not available for CentOS-6.

---

#### MacOSX

MacOSX support is not complete yet. The GUI does not work due to compatibility issues with *PyQt* package available on homebrew. See [status of this ticket](#). Python interface is available (latest version 3.1.2. [Status](#)) and can be installed on MacOSX using homebrew

```
$ brew install homebrew/science/moose
```

### 1.1.2 Building MOOSE

In case your distribution is not listed on [our repository page](#), or if you want to build the latest development code, read on.

First, you need to get the source code. You can use `git` (clone the repository) or download snapshot of github repo by clicking on [this link](#).

```
$ git clone https://github.com/BhallaLab/moose --depth 100
```

This will create folder called “moose” with source code. Or,

```
$ wget https://github.com/BhallaLab/moose/archive/master.zip
$ unzip master.zip
```

You can download other released versions from [here](#).

## Install dependencies

Depending on your operating system, names of following packages may vary.

### Python interface for core MOOSE API (pymoose)

- **Required**

- Python For building the MOOSE Python bindings
- Python development headers and libraries, e.g. *python-dev* or *python-devel*
- NumPy (>= 1.6.x) For array interface, e.g. *python-numpy* or *numpy*

- **Optional**

- NetworkX (1.x). Layout is of chemical models is computed using networkx.
- Matplotlib (>=1.1.x) For plotting simulation results
- python-libsbnl For reading and writing chemical models in SBML format.

Most of the dependencies can be installed using package manager.

On Debian/Ubuntu

```
$ sudo apt-get install libhdf5-dev cmake libgsl0-dev libpython-dev python-numpy
```

**Note:** Ubuntu 12.04 does not have required version of *gsl* (required 1.16 or higher, available 1.15). On Ubuntu 16.04, package name is *libgsl-dev*.

On CentOS/Fedora/RHEL/Scientific Linux

```
$ sudo yum install hdf5-devel cmake libgsl-dev python-devel python-numpy
```

On SUSE/OpenSUSE

```
$ sudo zypper install hdf5-devel cmake libgsl-dev python-devel python-numpy
```

## build moose

```
$ cd /to/moose/source/code
$ mkdir _build
$ cd _build
$ cmake ..
$ make -j2      # using 2 core to build MOOSE
$ ctest --output-on-failure # optional
$ sudo make install
```

This will build MOOSE's python extension, *ctest* will run few tests to check if build process was successful.

**Note:** To install MOOSE into non-standard directory, pass additional argument `-DCMAKE_INSTALL_PREFIX=path/to/install/dir` to *cmake*

```
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/.local ..
```

To use different version of python

```
$ cmake -DPYTHON_EXECUTABLE=/opt/python3/bin/python3 ..
```

After that installation is pretty easy

```
$ sudo make install
```

If everything went fine, you should be able to import moose in python shell.

```
>>> import moose
```

### 1.1.3 Graphical User Interface (GUI)

MOOSE-gui can be launched by running the following command

```
$ moosegui
```

Below are packages which you may need to install to be able to launch MOOSE Graphical User Interface.

- **Required:**

- PyQt4 (4.8.x)

On Ubuntu/Debian, these can be installed with

```
$ sudo apt-get install python-qt4
```

On CentOS/Fedora/RHEL

```
$ sudo yum install python-qt4
```

**Note:** If you have installed moose package, then GUI is launched by running following command:

```
$ moosegui
```

### 1.1.4 Building moogli

moogli is subproject of MOOSE for visualizing models. More details can be found [here](#).

Moogli is part of moose package. Building moogli can be tricky because of multiple dependencies it has.

- **Required**

- OSG (3.2.x) For 3D rendering and simulation of neuronal models
- Qt4 (4.8.x) For C++ GUI of Moogli

To get the latest source code of moogli, click on [this link](#).

Moogli depends on OpenSceneGraph (version 3.2.0 or higher) which may not be easily available for your operating system. For this reason, we distribute required OpenSceneGraph with moogli source code.

Depending on distribution of your operating system, you would need following packages to be installed.

On Ubuntu/Debian

```
$ sudo apt-get install python-qt4-dev python-qt4-gl python-sip-dev libqt4-dev
```

On Fedora/CentOS/RHEL

```
$ sudo yum install sip-devel PyQt4-devel qt4-devel libjpeg-devel PyQt4
```

On openSUSE

```
$ sudo zypper install python-sip python-qt4-devel libqt4-devel python-qt4
```

After this, building and installing `moogli` should be as simple as

```
$ cd /path/to/moogli
$ mkdir _build
$ cd _build
$ cmake ..
$ make
$ sudo make install
```

If you run into troubles, please report it on our [github repository](#).

## 1.2 Quick Start

### 1.2.1 MOOSE GUI: Graphical interface for MOOSE

Upinder Bhalla, Harsha Rani, Aviral Goel

MOOSE is the Multiscale Object-Oriented Simulation Environment. It can do all these calculations together. One of its major uses is to make biologically detailed models that combine electrical and chemical signaling.

This document describes the salient features of the GUI and Kinetickit of MOOSE.

---

#### Contents

- *Introduction* (page 6)
- *Interface* (page 6)
  - *Menu Bar* (page ??)
    - \* *File* (page ??)
      - *New* (page ??)
      - *Load Model* (page ??)
      - *Connect BioModels* (page ??)
      - *Quit* (page ??)
    - \* *View* (page ??)
      - *Editor View* (page ??)

- *Run View* (page ??)
- *Dock Widgets* (page ??)
- *SubWindows* (page ??)
- \* *Help* (page ??)
  - *About MOOSE* (page ??)
  - *Built-in Documentation* (page ??)
  - *Report a bug* (page ??)
- *Editor View* (page ??)
  - \* *Model Editor* (page ??)
  - \* *Property Editor* (page ??)
- *Run View* (page ??)
  - \* *Simulation Controls* (page 10)
  - \* *Plot Widget* (page 10)
    - *Toolbar* (page ??)
    - *Context Menu* (page ??)

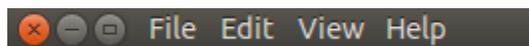
## Introduction

The Moose GUI currently allow you work on [chemical](#) models using a interface. This document describes the salient features of the GUI

## Interface

The interface layout consists of a *menu bar* (page ??) and two views, *editor view* (page ??) and *run view* (page ??).

## Menu Bar



The menu bar appears at the top of the main window. In Ubuntu 12.04, the menu bar appears only when the mouse is in the top menu strip of the screen. It consists of the following options -

## File

The File menu option provides the following sub options

- *New* (page ??) - Create a new chemical signalling model.
- *Load Model* (page ??) - Load a chemical signalling or compartmental neuronal model from a file.
- *Paper\_2015\_Demos Model* (page ??) - Loads and Runs chemical signalling or compartmental neuronal model from a file.
- *Recently Loaded Models* (page ??) - List of models loaded in MOOSE. (Atleast one model should be loaded)

- *Connect BioModels* (page ??) - Load chemical signaling models from the BioModels database.
- *Save* (page ??) - Saves chemical model to Genesis/SBML format.
- *Quit* (page ??) - Quit the interface.

## View

View menu option provides the following sub options -

- *Editor View* (page ??) - Switch to the editor view for editing models.
- *Run View* (page ??) - Switch to run view for running models.
- *Dock Widgets* (page ??) - Following dock widgets are provided
- *Python* (page ??) - Brings up a full fledged python interpreter integrated with MOOSE GUI. You can interact with loaded models and load new models through the PyMoose API. The entire power of python language is accessible, as well as MOOSE-specific functions and classes.
- *Edit* (page ??) - A property editor for viewing and editing the fields of a selected object such as a pool, enzyme, function or compartment. Editable field values can be changed by clicking on them and overwriting the new values. Please be sure to press enter once the editing is complete, in order to save your changes.
- *SubWindows* (page ??) - This allows you to tile or tabify the run and editor views.

## Help

- *About Moose* (page ??) - Version and general information about MOOSE.
- *Built-in documentation* (page ??) - Documentation of MOOSE GUI.
- *Report a bug* (page ??) - Directs to the github bug tracker for reporting bugs.

## Editor View

The editor view provides two windows -

- *Model Editor* (page ??) - The model editor is a workspace to edit and create models. Using click-and-drag from the icons in the menu bar, you can create model entities such as chemical pools, reactions, and so on. A click on any object brings its property editor on screen (see below). In objects that can be interconnected, a click also brings up a special arrow icon that is used to connect objects together with messages. You can move objects around within the edit window using click-and-drag. Finally, you can delete objects by selecting one or more, and then choosing the delete option from the pop-up menu. The links below is the screenshots point to the details for the chemical signalling model editor.
- *Property Editor* (page ??) - The property editor provides a way of viewing and editing the properties of objects selected in the model editor.

## Run View

The Run view, as the name suggests, puts the GUI into a mode where the model can be simulated. As a first step in this, you can click-and-drag an object to the graph window in order to create a time-series plot for that object. For example, in a chemical reaction, you could drag a pool into the graph window and subsequent simulations will display a graph of the concentration of the pool as a function of time. Within the Run View window, the time-evolution of the simulation is displayed as an animation. For chemical kinetic models, the size of the icons for reactant pools scale to

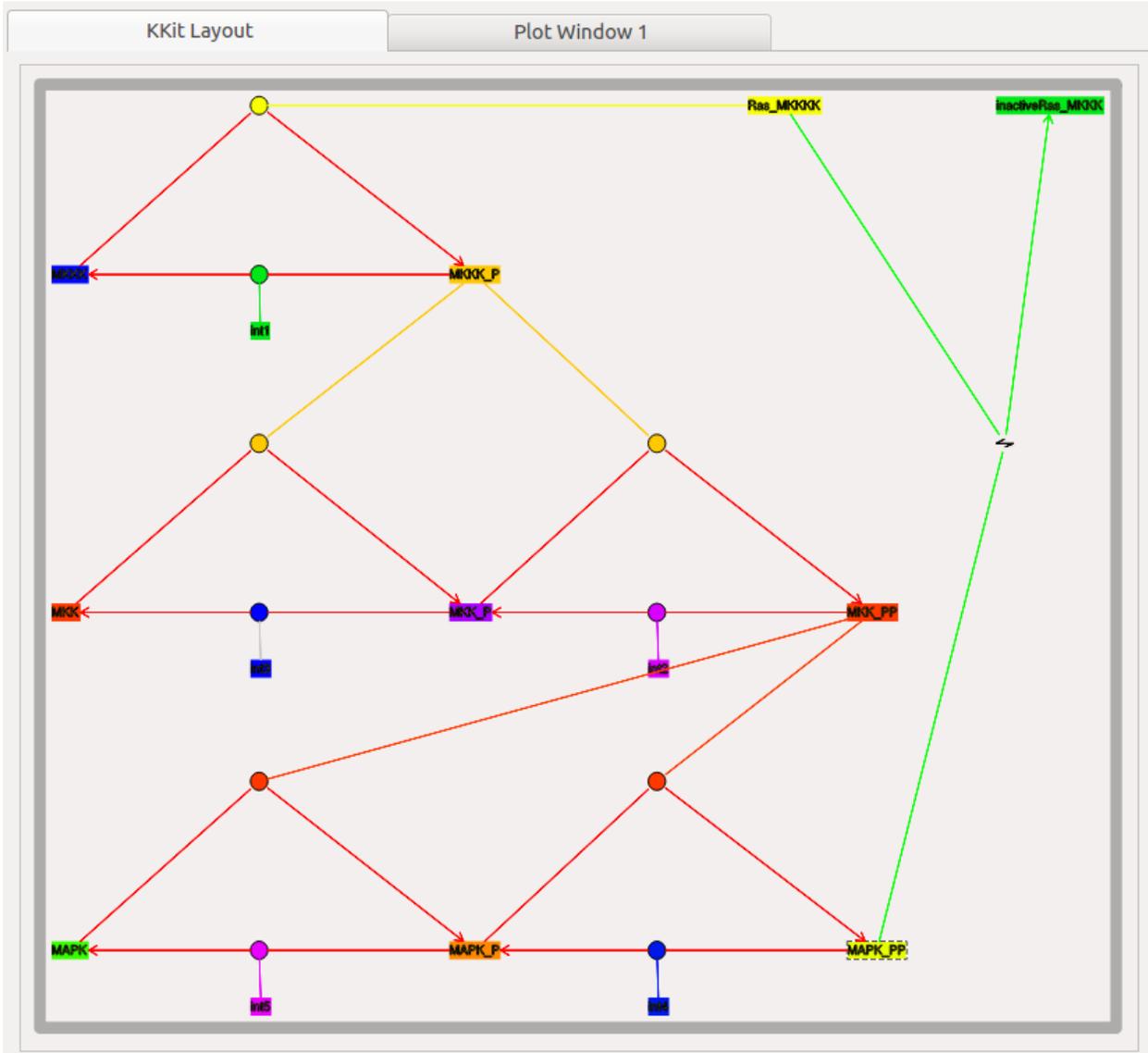
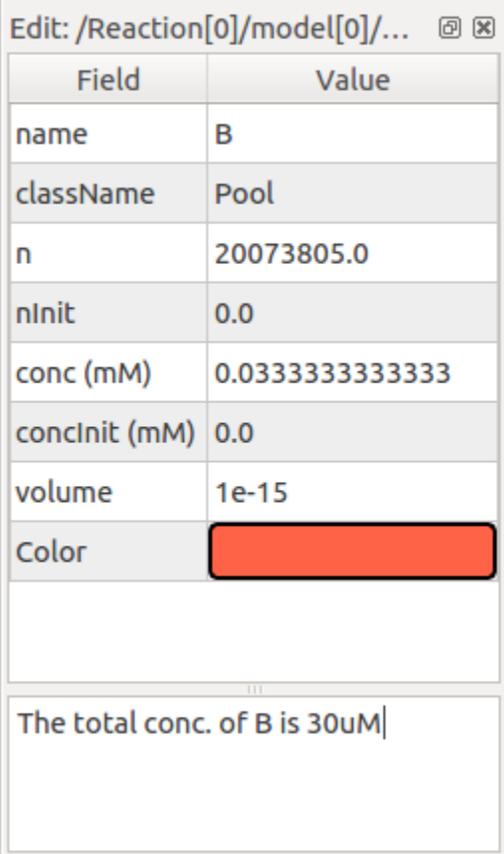


Fig. 2: Chemical Signalling Model Editor



The image shows a window titled "Edit: /Reaction[0]/model[0]/..." with a table of properties. The table has two columns: "Field" and "Value". The rows are as follows:

| Field         | Value                         |
|---------------|-------------------------------|
| name          | B                             |
| className     | Pool                          |
| n             | 20073805.0                    |
| nInit         | 0.0                           |
| conc (mM)     | 0.03333333333333              |
| conclnit (mM) | 0.0                           |
| volume        | 1e-15                         |
| Color         | <input type="text" value=""/> |

Below the table, there is a text area containing the text: "The total conc. of B is 30uM|".

Fig. 3: Property Editor

indicate concentration. Above the Run View window, there is a special tool bar with a set of simulation controls to run the simulation.

### Simulation Controls



Fig. 4: Simulation Control

This panel allows you to control the various aspects of the simulation.

- *Run Time* (page ??) - Determines duration for which simulation is to run. A simulation which has already run, runs further for the specified additional period.
- *Reset* (page ??) - Restores simulation to its initial state; re-initializes all variables to  $t = 0$ .
- *Stop* (page ??) - This button halts an ongoing simulation.
- *Current time* (page ??) - This reports the current simulation time.
- *Preferences* (page ??) - Allows you to set simulation and visualization related preferences.

### Plot Widget

#### Toolbar

On top of plot window there is a little row of icons:



These are the plot controls. If you hover the mouse over them for a few seconds, a tool-tip pops up. The icons represent the following functions:

-  - Add a new plot window
-  - Deletes current plot window
-  - Toggle X-Y axis grid
-  - Returns the plot display to its default position

-  - Undoes or re-does manipulations you have done to the display.
-  - The plots will pan around with the mouse when you hold the left button down. The plots will zoom with the mouse when you hold the right button down.
-  - With the “**left mouse button**“, this will zoom in to the specified rectangle so that the plots become bigger. With the “**right mouse button**“, the entire plot display will be shrunk to fit into the specified rectangle.
-  - You don't want to mess with these .
-  - Save the plot.

## Context Menu

The context menu is enabled by right clicking on the plot window. It has the following options -

- **Export to CSV** - Exports the plotted data to CSV format
- **Toggle Legend** - Toggles the plot legend
- **Remove** - Provides a list of plotted entities. The selected entity will not be plotted.

## 1.2.2 Getting started with python scripting for MOOSE

- *Introduction* (page 11)
- *Importing MOOSE and accessing built-in documentation* (page 12)
- *Setting the properties of elements: accessing fields* (page 14)
- *Putting them together: setting up connections* (page 16)
- *Scheduling* (page 18)
- *Running the simulation* (page 20)
- *Some more details* (page 20)
- *Moving on* (page 21)

## Introduction

This document describes how to use the `moose` module in Python scripts or in an interactive Python shell. It aims to give you enough overview to help you start scripting using MOOSE and extract farther information that may be required for advanced work. Knowledge of Python or programming in general will be helpful. If you just want to simulate existing models in one of the supported formats, you can fire the MOOSE GUI and locate the model file

using the `File` menu and load it. The GUI is described in separate document. If you are looking for recipes for specific tasks, take a look at *cookbook*. The example code in the boxes can be entered in a Python shell.

### Importing MOOSE and accessing built-in documentation

In a python script you import modules to access the functionalities they provide.

```
>>> import moose
```

This makes the `moose` module available for use in Python. You can use Python's built-in `help` function to read the top-level documentation for the `moose` module

```
>>> help(moose)
```

This will give you an overview of the module. Press `q` to exit the pager and get back to the interpreter. You can also access the documentation for individual classes and functions this way.

```
>>> help(moose.connect)
```

To list the available functions and classes you can use `dir` function<sup>1</sup>.

```
>>> dir(moose)
```

MOOSE has built-in documentation in the C++-source-code independent of Python. The `moose` module has a separate `doc` function to extract this documentation.

```
>>> moose.doc(moose.Compartment)
```

The class level documentation will show whatever the author/maintainer of the class wrote for documentation followed by a list of various kinds of fields and their data types. This can be very useful in an interactive session.

Each field can have its own detailed documentation, too.

```
>>> moose.doc('Compartment.Rm')
```

Note that you need to put the class-name followed by dot followed by field-name within quotes. Otherwise, `moose.doc` will receive the field value as parameter and get confused.

### Creating objects and traversing the object hierarchy

Different types of biological entities like neurons, enzymes, etc are represented by classes and individual instances of those types are objects of those classes. Objects are the building-blocks of models in MOOSE. We call MOOSE objects `element` and use `object` and `element` interchangeably in the context of MOOSE. Elements are conceptually laid out in a tree-like hierarchical structure. If you are familiar with file system hierarchies in common operating systems, this should be simple.

At the top of the object hierarchy sits the `Shell`, equivalent to the root directory in UNIX-based systems and represented by the path `/`. You can list the existing objects under `/` using the `le` function.

```
>>> moose.le()
Elements under /
/Msgs
/clock
```

(continues on next page)

---

<sup>1</sup> To list the classes only, use `moose.le('/classes')`

(continued from previous page)

```
/classes
/postmaster
```

`Msgs`, `clock` and `classes` are predefined objects in MOOSE. And each object can contain other objects inside them. You can see them by passing the path of the parent object to `le`

```
>>> moose.le('/Msgs')
Elements under /Msgs[0]
/Msgs[0]/singleMsg
/Msgs[0]/oneToOneMsg
/Msgs[0]/oneToAllMsg
/Msgs[0]/diagonalMsg
/Msgs[0]/sparseMsg
```

Now let us create some objects of our own. This can be done by invoking MOOSE class constructors (just like regular Python classes).

```
>>> model = moose.Neutral('/model')
```

The above creates a `Neutral` object named `model`. `Neutral` is the most basic class in MOOSE. A `Neutral` element can act as a container for other elements. We can create something under `model`

```
>>> soma = moose.Compartment('/model/soma')
```

Every element has a unique path. This is a concatenation of the names of all the objects one has to traverse starting with the root to reach that element.

```
>>> print soma.path
/model/soma
```

The name of the element can be printed, too.

```
>>> print soma.name
soma
```

The `Compartment` elements model small sections of a neuron. Some basic experiments can be carried out using a single compartment. Let us create another object to act on the `soma`. This will be a step current generator to inject a current pulse into the `soma`.

```
>>> pulse = moose.PulseGen('/model/pulse')
```

You can use `le` at any point to see what is there

```
>>> moose.le('/model')
Elements under /model
/model/soma
/model/pulse
```

And finally, we can create a `Table` to record the time series of the `soma`'s membrane potential. It is good practice to organize the data separately from the model. So we do it as below

```
>>> data = moose.Neutral('/data')
>>> vmtab = moose.Table('/data/soma_Vm')
```

Now that we have the essential elements for a small model, we can go on to set the properties of this model and the experimental protocol.

## Setting the properties of elements: accessing fields

Elements have several kinds of fields. The simplest ones are the `value` fields. These can be accessed like ordinary Python members. You can list the available value fields using `getFieldNames` function

```
>>> soma.getFieldNames('valueFinfo')
```

Here `valueFinfo` is the type name for value fields. `Finfo` is short form of *field information*. For each type of field there is a name ending with `-Finfo`. The above will display the following list

```
('this',
 'name',
 'me',
 'parent',
 'children',
 'path',
 'class',
 'linearSize',
 'objectDimensions',
 'lastDimension',
 'localNumField',
 'pathIndices',
 'msgOut',
 'msgIn',
 'Vm',
 'Cm',
 'Em',
 'Im',
 'inject',
 'initVm',
 'Rm',
 'Ra',
 'diameter',
 'length',
 'x0',
 'y0',
 'z0',
 'x',
 'y',
 'z')
```

Some of these fields are for internal or advanced use, some give access to the physical properties of the biological entity we are trying to model. Now we are interested in `Cm`, `Rm`, `Em` and `initVm`. In the most basic form, a neuronal compartment acts like a parallel RC circuit with a battery attached. Here `R` and `C` are resistor and capacitor connected in parallel, and the battery with voltage `Em` is in series with the resistor, as shown below:

The fields are populated with some defaults.

```
>>> print soma.Cm, soma.Rm, soma.Vm, soma.Em, soma.initVm
1.0 1.0 -0.06 -0.06 -0.06
```

You can set the `Cm` and `Rm` fields to something realistic using simple assignment (we follow SI unit)<sup>2</sup>.

```
>>> soma.Cm = 1e-9
>>> soma.Rm = 1e7
>>> soma.initVm = -0.07
```

<sup>2</sup> MOOSE is unit agnostic and things should work fine as long as you use values all converted to a consistent unit system.

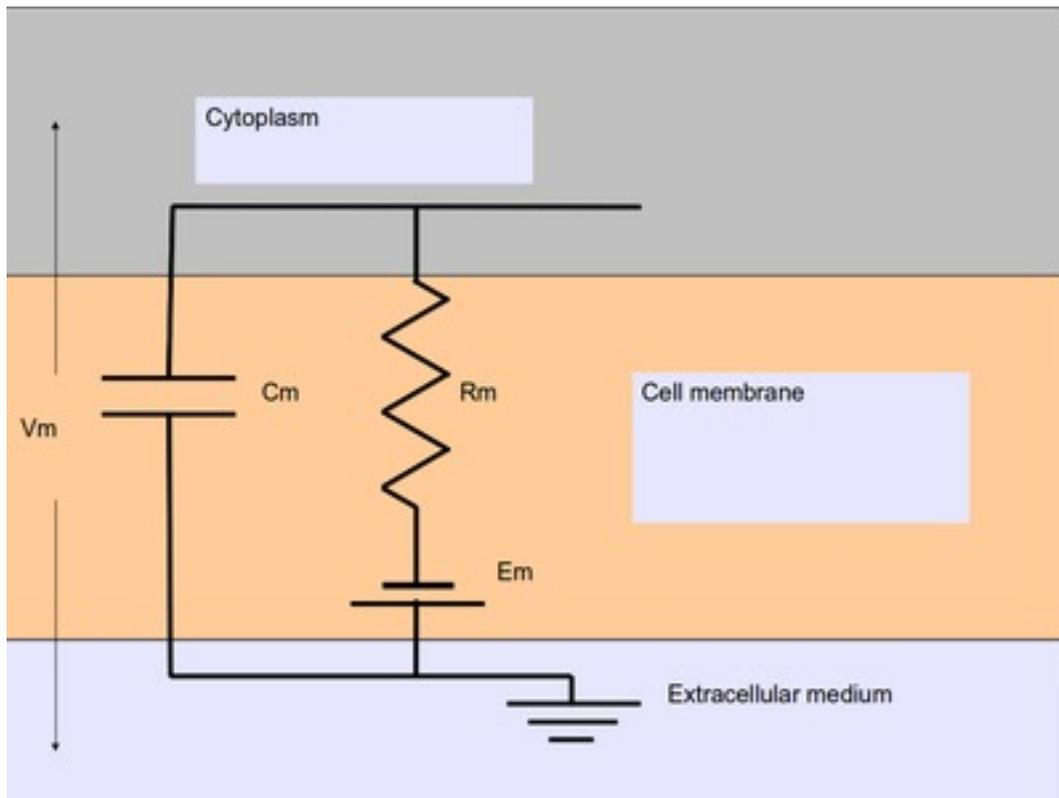


Fig. 5: Passive neuronal compartment

Instead of writing print statements for each field, you could use the utility function `showfield` to see that the changes took effect

```
>>> moose.showfield(soma)
[ /soma[0] ]
diameter      = 0.0
Ra            = 1.0
y0           = 0.0
Rm           = 10000000.0
numData      = 1
inject       = 0.0
initVm      = -0.07
Em          = -0.06
y           = 0.0
numField    = 1
path        = /soma[0]
dt          = 5e-05
tick       = 4
z0         = 0.0
name       = soma
Cm         = 1e-09
x0         = 0.0
Vm         = -0.06
className  = Compartment
length     = 0.0
Im         = 0.0
x          = 0.0
z          = 0.0
```

Now we can setup the current pulse to be delivered to the soma

```
>>> pulse.delay[0] = 50e-3
>>> pulse.width[0] = 100e-3
>>> pulse.level[0] = 1e-9
>>> pulse.delay[1] = 1e9
```

This tells the pulse generator to create a 100 ms long pulse 50 ms after the start of the simulation. The amplitude of the pulse is set to 1 nA. We set the delay for the next pulse to a very large value (larger than the total simulation time) so that the stimulation stops after the first pulse. Had we set `pulse.delay = 0`, it would have generated a pulse train at 50 ms intervals.

### Putting them together: setting up connections

In order for the elements to interact during simulation, we need to connect them via messages. Elements are connected to each other using special source and destination fields. These types are named `srcFinfo` and `destFinfo`. You can query the available source and destination fields on an element using `getFieldNames` as before. This time, let us do it another way: by the class name

```
>>> moose.getFieldNames('PulseGen', 'srcFinfo')
('childMsg', 'output')
```

This form has the advantage that you can get information about a class without creating elements of that class.

Here `childMsg` is a source field that is used by the MOOSE internals to connect child elements to parent elements. The second one is of our interest. Check out the built-in documentation [here](#)

```
>>> moose.doc('PulseGen.output')
PulseGen.output: double - source field
Current output level.
```

so this is the output of the pulse generator and this must be injected into the soma to stimulate it. But where in the soma can we send it? Again, MOOSE has some introspection built in.

```
>>> soma.getFieldNames('destFinfo')
('parentMsg',
 'setThis',
 'getThis',
 ...
 'setZ',
 'getZ',
 'injectMsg',
 'randInject',
 'cable',
 'process',
 'reinit',
 'initProc',
 'initReinit',
 'handleChannel',
 'handleRaxial',
 'handleAxial')
```

Now that is a long list. But much of it are fields for internal or special use. Anything that starts with `get` or `set` are internal `destFinfo` used for accessing value fields (we shall use one of those when setting up data recording). Among the rest `injectMsg` seems to be the most likely candidate. Use the `connect` function to connect the pulse generator output to the soma input

```
>>> m = moose.connect(pulse, 'output', soma, 'injectMsg')
```

`connect(source, source_field, dest, dest_field)` creates a message from `source` element's `source_field` field to `dest` elements `dest_field` field and returns that message. Messages are also elements. You can print them to see their identity

```
>>> print m
<moose.SingleMsg: id=5, dataId=733, path=/Msgs/singleMsg[733]>
```

You can print any element as above and the string representation will show you the class, two numbers (`id` and `dataId`) uniquely identifying it among all elements, and its path. You can get some more information about a message

```
>>> print m.e1.path, m.e2.path, m.srcFieldsOnE1, m.destFieldsOnE2
/model/pulse /model/soma ('output',) ('injectMsg',)
```

will confirm what you already know.

A message element has fields `e1` and `e2` referring to the elements it connects. For single one-directional messages these are source and destination elements, which are `pulse` and `soma` respectively. The next two items are lists of the field names which are connected by this message.

You could also check which elements are connected to a particular field

```
>>> print soma.neighbors['injectMsg']
[<moose.vec: class=PulseGen, id=729, path=/model/pulse>]
```

Notice that the list contains something called `vec`. We discuss this *later* (page 20). Also `neighbors` is a new kind of field: `lookupFinfo` which behaves like a dictionary. Next we connect the table to the soma to retrieve its membrane potential `Vm`. This is where all those `destFinfo` starting with `get` or `set` come in use. For each value field `X`, there is a `destFinfo get {X}` to retrieve the value at simulation time. This is used by the table to record the values `Vm` takes.

```
>>> moose.connect (vmtab, 'requestOut', soma, 'getVm')
<moose.SingleMsg: id=5, dataIndex=0, path=/Msgs[0]/singleMsg[0]>
```

This finishes our model and recording setup. You might be wondering about the source-destination relationship above. It is natural to think that `soma` is the source of `Vm` values which should be sent to `vmtab`. But here `requestOut` is a `srcFinfo` acting like a reply card. This mode of obtaining data is called *pull mode*.<sup>3</sup>

You can skip the next section on fine control of the timing of updates and read *Running the simulation* (page 20).

## Scheduling

With the model all set up, we have to schedule the simulation. Different components in a model may have different rates of update. For example, the dynamics of electrical components require the update intervals to be of the order 0.01 ms whereas chemical components can be as slow as 1 s. Also, the results may depend on the sequence of the updates of different components. These issues are addressed in MOOSE using a clock-based update scheme. Each model component is scheduled on a clock tick (think of multiple hands of a clock ticking at different intervals and the object being updated at each tick of the corresponding hand). The scheduling also guarantees the correct sequencing of operations. For example, your `Table` objects should always be scheduled *after* the computations that they are recording, otherwise they will miss the outcome of the latest calculation.

MOOSE has a central clock element (`/clock`) to manage time. `Clock` has a set of `Tick` elements under it that take care of advancing the state of each element with time as the simulation progresses. Every element to be included in a simulation must be assigned a tick. Each tick can have a different ticking interval (`dt`) that allows different elements to be updated at different rates.

By default, every object is assigned a clock tick with reasonable default timesteps as soon it is created:

| Class type  | tick  | dt               |
|---|-------|------------------|
| Electrical computations:<br>electrical compartments,<br>V <b>and</b> ligand-gated ion channels,<br>Calcium conc <b>and</b> Nernst,<br>stimulus generators <b>and</b> tables,<br>HSolve. | 0-7   | 50 microseconds  |
| Table (to plot elec. signals)   | 8     | 100 microseconds |
| Diffusion solver  | 10    | 0.01 seconds     |
| Chemical computations:<br>Pool, Reac, Enz, MMenz,<br>Func, Function,<br>Gsolve, Ksolve,<br>Stats (to do stats on outputs)   | 11-17 | 0.1 seconds      |
| Table2 (to plot chem. signals)  | 18    | 1 second         |
| HDF5DataWriter  | 30    | 1 second         |
| Postmaster ( <b>for</b> parallel<br>computations)   | 31    | 0.01 seconds     |

<sup>3</sup> This apparently convoluted implementation is for performance reason. Can you figure out why? *Hint: the table is driven by a slower clock than the compartment.*

There are 32 available clock ticks. Numbers 20 to 29 are unassigned so you can use them for whatever purpose you like.

If you want fine control over the scheduling, there are three things you can do.

- Alter the ‘tick’ field on the object
- Alter the dt associated with a given tick, using the `moose.setClock( tick, newdt)` command
- Go through a wildcard path of objects reassigning their clock ticks, using `moose.useClock( path, newtick, function)`.

Here we discuss these in more detail.

### Altering the ‘tick’ field

Every object knows which tick and dt it uses:

```
>>> a = moose.Pool( '/a' )
>>> print a.tick, a.dt
13 0.1
```

The `tick` field on every object can be changed, and the object will adopt whatever clock dt is used for that tick. The `dt` field is readonly, because changing it would have side-effects on every object associated with the current tick.

Ticks **-1** and **-2** are special: They both tell the object that it is disabled (not scheduled for any operations). An object with a tick of **-1** will be left alone entirely. A tick of **-2** is used in solvers to indicate that should the solver be removed, the object will revert to its default tick.

### Altering the dt associated with a given tick

We initialize the ticks and set their dt values using the `setClock` function.

```
>>> moose.setClock(0, 0.025e-3)
>>> moose.setClock(1, 0.025e-3)
>>> moose.setClock(2, 0.25e-3)
```

This will initialize tick #0 and tick #1 with  $dt = 25 \hat{\mu}s$  and tick #2 with  $dt = 250 \hat{\mu}s$ . Thus all the elements scheduled on ticks #0 and 1 will be updated every  $25 \hat{\mu}s$  and those on tick #2 every  $250 \hat{\mu}s$ . We use the faster clocks for the model components where finer timescale is required for numerical accuracy and the slower clock to sample the values of  $V_m$ .

Note that if you alter the dt associated with a given tick, this will affect the update time for *all* the objects using that clock tick. If you’re unsure that you want to do this, use one of the vacant ticks.

### Assigning clock ticks to all objects in a wildcard path

To assign tick #2 to the table for recording  $V_m$ , we pass its whole path to the `useClock` function.

```
>>> moose.useClock(2, '/data/soma_Vm', 'process')
```

Read this as “use tick # 2 on the element at path `/data/soma_Vm` to call its `process` method at every step”. Every class that is supposed to update its state or take some action during simulation implements a `process` method. And in most cases that is the method we want the ticks to call at every time step. A less common method is `init`, which is implemented in some classes to interleave actions or updates that must be executed in a specific order<sup>4</sup>. The `Compartment` class is one such case where a neuronal compartment has to know the  $V_m$  of its neighboring compartments before it can calculate its  $V_m$  for the next step. This is done with:

<sup>4</sup> In principle any function available in a MOOSE class can be executed periodically this way as long as that class exposes the function for scheduling following the MOOSE API. So you have to consult the class’ documentation for any nonstandard methods that can be scheduled this way.

```
>>> moose.useClock(0, soma.path, 'init')
```

Here we used the `path` field instead of writing the path explicitly.

Next we assign tick #1 to process method of everything under `/model`.

```
>>> moose.useClock(1, '/model/##', 'process')
```

Here the second argument is an example of wild-card path. The `##` matches everything under the path preceding it at any depth. Thus if we had some other objects under `/model/soma`, `process` method of those would also have been scheduled on tick #1. This is very useful for complex models where it is tedious to schedule each element individually. In this case we could have used `/model/#` as well for the path. This is a single level wild-card which matches only the children of `/model` but does not go farther down in the hierarchy.

### Running the simulation

Once the model is all set up, we can put the model to its initial state using

```
>>> moose.reinit()
```

You may remember that we had changed `initVm` from `-0.06` to `-0.07`. The `reinit` call we initialize `Vm` to that value. You can verify that

```
>>> print soma.Vm
-0.07
```

Finally, we run the simulation for 300 ms

```
>>> moose.start(300e-3)
```

The data will be recorded by the `soma_vm` table, which is referenced by the variable `vmtab`. The `Table` class provides a numpy array interface to its content. The field is `vector`. So you can easily plot the membrane potential using the `matplotlib` library.

```
>>> import pylab
>>> t = pylab.linspace(0, 300e-3, len(vmtab.vector))
>>> pylab.plot(t, vmtab.vector)
>>> pylab.show()
```

The first line imports the `pylab` submodule from `matplotlib`. This useful for interactive plotting. The second line creates the time points to match our simulation time and length of the recorded data. The third line plots the `Vm` and the fourth line makes it visible. Does the plot match your expectation?

### Some more details

#### `vec`, `melement` and `element`

MOOSE elements are instances of the class `melement`. `Compartment`, `PulseGen` and other MOOSE classes are derived classes of `melement`. All `melement` instances are contained in array-like structures called `vec`. Each `vec` object has a numerical `id_` field uniquely identifying it. An `vec` can have one or more elements. You can create an array of elements

```
>>> comp_array = moose.vec('/model/comp', n=3, dtype='Compartment')
```

This tells MOOSE to create an `vec` of 3 `Compartment` elements with path `/model/comp`. For `vec` objects with multiple elements, the index in the `vec` is part of the element path.

```
>>> print comp_array.path, type(comp_array)
```

shows that `comp_array` is an instance of `vec` class. You can loop through the elements in an `vec` like a Python list

```
>>> for comp in comp_array:
...     print comp.path, type(comp)
... 
```

shows

```
/model/comp[0] <type 'moose.melement'>
/model/comp[1] <type 'moose.melement'>
/model/comp[2] <type 'moose.melement'>
```

Thus elements are instances of class `melement`. All elements in an `vec` share the `id_` of the `vec` which can be retrieved by `melement.getId()`.

A frequent use case is that after loading a model from a file one knows the paths of various model components but does not know the appropriate class name for them. For this scenario there is a function called `element` which converts (“casts” in programming jargon) a path or any moose object to its proper MOOSE class. You can create additional references to `soma` in the example this way

```
x = moose.element('/model/soma')
```

Any MOOSE class can be extended in Python. But any additional attributes added in Python are invisible to MOOSE. So those can be used for functionalities at the Python level only. You can see `moose-examples/squid/squid.py` for an example.

## Finfos

The following kinds of `Finfo` are accessible in Python

- **“valueFinfo”**: simple values. For each readable `valueFinfo XYZ` there is a `destFinfo getXYZ` that can be used for reading the value at run time. If `XYZ` is writable then there will also be `destFinfo setXYZ`. Example: `Compartment.Rm`
- **“lookupFinfo”**: lookup tables. These fields act like Python dictionaries but iteration is not supported. Example: `Neutral.neighbors`.
- **“srcFinfo”**: source of a message. Example: `PulseGen.output`.
- **“destFinfo”**: destination of a message. Example: `Compartment.injectMsg`. Apart from being used in setting up messages, these are accessible as functions from Python. `HHGate.setupAlpha` is an example.
- **“sharedFinfo”**: a composition of source and destination fields. Example: `Compartment.channel`.

## Moving on

Now you know the basics of `pymoose` and how to access the help system. You can figure out how to do specific things by looking at the ‘cookbook’. In addition, the `moose-examples/snippets` directory in your MOOSE installation has small executable python scripts that show usage of specific classes or functionalities. Beyond that you can browse the code in the `moose-examples` directory to see some more complex models.

MOOSE is backward compatible with GENESIS and most GENESIS classes have been reimplemented in MOOSE. There is slight change in naming (MOOSE uses CamelCase), and setting up messages are different. But [GENESIS documentation](#) is still a good source for documentation on classes that have been ported from GENESIS.

If the built-in MOOSE classes do not satisfy your needs entirely, you are welcome to add new classes to MOOSE. The API documentation will help you get started.

### 1.2.3 Demonstration of basic functionalities

#### Load and Run a Model

`helloMoose.main()`

This is the Hello MOOSE program. It shows how to get MOOSE to do its most basic operations: to load, run, and graph a model defined in an external model definition file.

The `loadModel` function is the core of this example. It can accept a range of file and model types, including `kkit`, `cspace` and GENESIS `.p` files. It autodetects the file type and loads in the simulation.

The `wildcardFind` function finds all objects matching the specified path, in this case Table objects holding the simulation results. They were all defined in the model file.

#### Start, Stop, and setting clocks

`startstop.main()`

This demo shows how to start, stop, and continue a simulation. This is commonly done when we want to run a model till settling, then change a parameter or deliver a stimulus, and then continue the simulation.

Here, the model is just the output of a `PulseGen` object which generates periodic pulses. The demo shows how to start the simulation. using the `moose.reinit` command to reset the model to its initial state, and `moose.start` command to run the model for the specified duration. We issue multiple `moose.start` commands and do different things to the model between them. First, we change the delay of the pulseGen. Then we show a number of ways to assign the timestep (dt) to the table object in the simulation. Note that throughout this simulation the pulsegen is going at a uniform rate, it is just being sampled by the output table at different intervals.

`stimtable.main()`

Example of StimulusTable using Poisson random numbers.

Creates a StimulusTable and assigns it signal representing events in a Poisson process. The output of the StimTable is sent to a DiffAmp object for buffering and then recorded in a regular table.

#### The Hodgkin-Huxley demo

This is a self-contained graphical demo implemented by Subhasis Ray, closely based on the ‘Squid’ demo by Mark Nelson which ran in GENESIS.

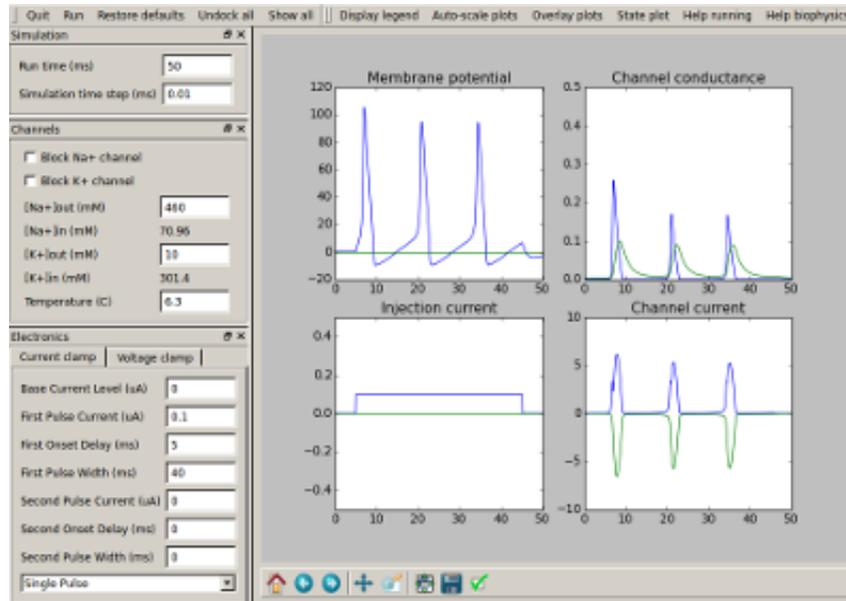
Simulation of Hodgkin-Huxley’s experiment on squid giant axon showing action potentials generated by a step current injection.

The demo has built-in documentation and may be run from the `moose-examples/squid` subdirectory of MOOSE.

#### Run Python from MOOSE

`pyrun.input_output()`

The PyRun class can take a double input through `trigger` field. Whenever another object sends an input to this field, the `runString` is executed.



The fun part of this is that you can use the input value in your python statements in `runString`. This is stored in a local variable called `input_`. You can rename this by setting `inputVar` field.

Things become even more interesting when you can send out a value computed using Python. PyRun objects allow you to define a local variable called `output` and whatever value you assign to this, will be sent out through the source field `output` on successful execution of the `runString`.

You can rename the output variable by setting `outputVar` field.

In this example, we send the output of a pulsegen object sending out the values 1, 2, 3 during each pulse and compute the square of these numbers in Python and set output to this square.

The calculated value is assigned to the `output` variable and in turn sent out to a Table object's input and gets recorded.

By default PyRun executes the `runString` whenever a `trigger` message is received and when its process method is called at each timestep. In both cases it sends out the `output` value. Since this may cause inaccuracies depending on what the Python statements in `runString` do, a `mode` can be specified to disable one of the above. We set `mode = 2` to disable the `process` method. Note that this could also have been done by setting its `tick = -1`. `mode = 1` will disable `trigger` message and `mode = 0`, the default, enables both.

```
pyrun.main()
```

You can use the PyRun class to run Python statements from MOOSE at runtime. This opens up many possibilities of interleaving computing in Python and MOOSE. You can also use this for debugging simulations.

```
pyrun.run_sequence()
```

In this example we demonstrate the use of PyRun objects to execute Python statements from MOOSE. Here is a couple of fun things to indicate the power of MOOSE-Python integration.

First we create a PyRun object called `Hello`. In its `initString` we put in Python statements that prints the element's string representation using `pymoose-API`. When `moose.reinit()` is called, this causes MOOSE to execute these Python statements which include Python calling a MOOSE function (Python->MOOSE->Python->MOOSE) - isn't that cool!

We also initialize a counter called `hello_count` to 0.

The statements in `initString` gets executed once, when we call `moose.reinit()`.

In the *runString* we put a couple of print statements to indicate the name of the object which is running and the current count. Then we increase the count directly.

When we call `moose.start()`, the *runString* gets executed at each time step.

The other PyRun object we create, is *World*. In its *initString* apart from ordinary print statements and initialization, we define a Python function called `incr_count`. This silly little function just increments the global `world_count` by 1.

The *runString* for *World* simply calls this function to increment the count and print it.

We may notice that we assign tick 0 to *Hello* and tick 1 to *World*. Looking at the output, you will realize that the sequences of the ticks strictly maintain the sequence of execution.

#### `pyrun1.main()`

You can use the PyRun class to run Python statements from MOOSE at runtime. This opens up many possibilities of interleaving computing in Python and MOOSE. You can also use this for debugging simulations.

The PyRun class can take a double input through *trigger* field. Whenever another object sends an input to this field, the *runString* is executed.

The fun part of this is that you can use the input value in your python statements in *runString*. This is stored in a local variable called *input\_*. You can rename this by setting *inputVar* field.

Things become even more interesting when you can send out a value computed using Python. PyRun objects allow you to define a local variable called *output* and whatever value you assign to this, will be sent out through the source field *output* on successful execution of the *runString*.

You can rename the output variable by setting *outputVar* field.

In this example, we send the output of a pulsegen object sending out the values 1, 2, 3 during each pulse and compute the square of these numbers in Python and set output to this square.

The calculated value is assigned to the *output* variable and in turn sent out to a Table object's input and gets recorded.

By default PyRun executes the *runString* whenever a *trigger* message is received and when its process method is called at each timestep. In both cases it sends out the *output* value. Since this may cause inaccuracies depending on what the Python statements in *runString* do, a *mode* can be specified to disable one of the above. We set `mode = 2` to disable the *process* method. Note that this could also have been done by setting its `tick = -1`. `mode = 1` will disable *trigger* message and `mode = 0`, the default, enables both.

#### `pyrun1.run_sequence()`

In this example we demonstrate the use of PyRun objects to execute Python statements from MOOSE. Here is a couple of fun things to indicate the power of MOOSE-Python integration.

First we create a PyRun object called *Hello*. In its *initString* we put in Python statements that prints the element's string representation using `pymoose-API`. When `moose.reinit()` is called, this causes MOOSE to execute these Python statements which include Python calling a MOOSE function (Python->MOOSE->Python->MOOSE) - isn't that cool!

We also initialize a counter called *hello\_count* to 0.

The statements in *initString* gets executed once, when we call `moose.reinit()`.

In the *runString* we put a couple of print statements to indicate the name of the object which is running and the current count. Then we increase the count directly.

When we call `moose.start()`, the *runString* gets executed at each time step.

The other PyRun object we create, is *World*. In its *initString* apart from ordinary print statements and initialization, we define a Python function called `incr_count`. This silly little function just increments the global `world_count` by 1.

The *runString* for *World* simply calls this function to increment the count and print it.

We may notice that we assign tick 0 to *Hello* and tick 1 to *World*. Looking at the output, you will realize that the sequences of the ticks strictly maintain the sequence of execution.

## 1.2.4 MOOSE Classes

### Messages

#### One-to-one message

```
onetoonemsg.main()
```

Demonstrates one-to-one connection between objects through ‘moose.connect’.

#### Show the message

```
showmsg.main()
```

This is to show a `_raw_` way of traversing messages.

#### Single Message Cross

```
singlemsgcross.main()
```

This example shows that you can have two ematrix objects and connect individual elements using *Single* message

```
singlemsgcross.test_crossing_single()
```

This function creates an ematrix of two PulseGen elements and another ematrix of two Table elements.

The two pulsegen elements have same amplitude but opposite phase.

Table[0] is connected to PulseGen[1] and Table[1] to Pulsegen[0].

In the plot you should see two square pulses of opposite phase.

### Time

#### Clocks

```
showclocks.main()
```

This snippet shows various ways of displaying scheduling information of moose model components.

The */clock/tick* ematrix has 10 elements, any of which can be setup by using the *moose.setClock(tickNo, dt)* function. This sets the interval between the ticking events for it to *dt* time.

Individual model components can be assigned ticks by *moose.useClock(tickNo, targetPath, targetFinfo)*. Commonly used target finfo is *process*, which causes the function of the same name in the ematrix at target path to be called at each ticking event of tick *tickNo*. Thus displaying the neighbors of *process* finfo of an element will show the tick assigned to it.

On the other hand, the tick ematrix has 10 finfos, *proc0 ... proc9* which connect to all the targets of the corresponding *tickNo*. You can display the neighbors of these finfos also to see what is scheduled on each tick.

### Generating Time Data Table

`timetable.generate_poisson_times (rate=20, simtime=100, seed=1)`  
Generate Poisson spike times using *rate*. Use *seed* for seeding the numpy rng

`timetable.main ()`  
Demonstrates the use of TimeTable elements in MOOSE.

This scripts creates two time tables, #1 is filled with entries in a numpy array and #2 is filled from a text file containing the event times.

The *state* field of #1, which becomes 1 when an event occurs and 0 otherwise, is recorded.

On the other hand, #2 is connected to a synapse (in a SynChan element) to demonstrate artificial spike event generation.

`timetable.timetable_file (filename='timetable.txt')`  
Create a TimeTable and populate it from file specified by *filename*. If *filename* does not exist, a file of the same name is created and a random series of spike times is saved in it

`timetable.timetable_nparray ()`  
Create a time table and populate it with numpy array. The *vec* field in a Table can be directly assigned a sequence to fill the table entries.

### Vectors

`vectors.main ()`  
Demonstrates how to use vectors of moose elements

### Data Entries

`wildcard.main ()`  
Explore the wildcard search in moose models

### Interpolation

#### 1-dimensional Interpolation

`interpol.main ()`  
Example of Interpol object.

#### 2-dimensional interpolation

Example of Interpol object in 2 dimensions.

### Function

`func.main ()`  
Demonstrate the use of Func class

`func.test_func ()`  
This function creates a Func object evaluating a function of a single variable. It both shows direct evaluation without running a simulation and a case where the x variable comes from another source.

```
func.test_func_nosim()
```

Create a Func object for computing function values without running a simulations.

## SymCompartment

```
symcompartment.main()
```

This example demonstrates the use of SymCompartment class of MOOSE.

## Tables

```
tabledemo.main()
```

Table can be used for recording and saving data in ascii text formats. In this example we create a square-pulse generator object and record the output using a table.

The steps are:

1. Create a PulseGen element *pulse*.
2. Set *delay[0]=1.0*, *width[0]=0.2*, *level[0]=0.5*, so it generates 0.2 s wide square pulses with 0.5 amplitude every 1 s.
3. Create a Table element *tab*.
4. Connect the *outputValue* field of *pulse* to *tab*.
5. We set tick-interval of ticks 0 and 1 to 0.01 and schedule *pulse* on tick 0 and *tab* on tick 1.
5. Run the simulation for 5 s and save data to the ascii file *output\_tabledemo.csv*.

## Data Types

### HDF Data Type

HDF5 is a self-describing file format for storing large datasets. MOOSE has an utility `HDF5DataWriter` for saving simulations data in HDF5 files.

```
hdfdemo.main()
```

In this example

1. We create a passive neuronal compartment *comp*.
2. We create an `HDF5DataWriter` object *hdfwriter* for writing to a file `output_hdfdemo.h5`. The *mode* of the `HDF5DataWriter` is set to 2 which means if a file of the same name exists, it will be overwritten.
3. The membrane voltage *Vm* and membrane current *Im* of *comp* are connected to *hdfwriter* for recording.
4. We set some attributes of the datasets and the file.
5. We run the simulation, *hdfwriter* records and stores the *Vm* and *Im* values over time.
6. We close the file by calling `close()` method of *hdfwriter*.

Running this snippet creates the file `output_hdfdemo.h5` which reflects the structure of the model:

```
model
|
|
c[0]
```

(continues on next page)

```

|
| ____im
|
| ____vm

```

*im* and *vm* are datasets containing *Im* and *Vm* field values recorded from *comp*.

## NSDF DataType

NSDF : Neuroscience Simulation Data Format

NSDF is an HDF5 based format for storing data from neuroscience simulation.

This script is for demonstrating the use of NSDFWriter class to dump data in NSDF format.

The present implementation of NSDFWriter puts all value fields connected to its requestData into /data/uniform/{className}/{fieldName} 2D dataset - each row corresponding to one object.

Event data are stored in /data/event/{className}/{fieldName}/{Id}\_{dataIndex}\_{fieldIndex} where the last component is the string representation of the ObjId of the source.

The model tree (starting below root element) is saved as a tree of groups under /model/modeltree (one could easily add the fields as attributes with a little bit of more code).

The mapping between data source and uniformly sampled data is stored as a dimension scale in /map/uniform/{className}/{fieldName}. That for event data is stored as a compound dataset in /map/event/{className}/{fieldName} with a [source, data] columns.

The start and end timestamps of the simulation are saved as file attributes: C/C++ time functions have this limitation that they give resolution up to a second, this means for simulation lasting < 1 s the two timestamps may be identical.

Much of the environment specification is set as HDF5 attributes (which is a generic feature from HDF5WriterBase).

MOOSE is unit agnostic at present so unit specification is not implemented in NSDFWriter. But units can be easily added as dataset attribute if desired as shown in this example.

References:

Ray, Chintaluri, Bhalla and Wojcik. NSDF: Neuroscience Simulation Data Format, Neuroinformatics, 2015.

<http://nsdf.readthedocs.org/en/latest/>

`nsdf.setup_model ()`

Setup a dummy model with a PulseGen and a SpikeGen. The SpikeGen detects the leading edges of the pulses created by the PulseGen and sends out the event times. We record the PulseGen outputValue as Uniform data and leading edge time as Event data in the NSDF file.

`nsdf_vec.main ()`

Example code to dump data from multiple elements in a vector.

In this demo we create a PulseGen vector where each element has a different set of pulse parameters. After saving the output vector directly using MOOSE NSDFWriter we open the NSDF file using h5py and plot the saved data.

You need h5py module installed to run this simulation.

References:

Ray, Chintaluri, Bhalla and Wojcik. NSDF: Neuroscience Simulation Data Format, Neuroinformatics, 2015.

<http://nsdf.readthedocs.org/en/latest/>

`nsdf_vec.read_nsdf (fname)`

Read the specific file we created in this example.

Note that the preferable way of associating source with data is to use the DimensionScale. But since there is one-to-one correspondence between the data rows and the map rows (source path), we are exploiting that here.

`nsdf_vec.write_nsdf ()`

Setup a dummy model with a PulseGen vec and dump the outputValue in NSDF file

## Threading

**class** `threading_demo.StatusThread (tab)`

This thread checks the status of the moose worker thread by checking the worker\_queue for available entry. If there is nothing, it goes to sleep for a second and then prints current length of the table. If there is an entry, it puts its name in the status queue, which is used by the main thread to recognize successful completion.

**run ()**

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

**class** `threading_demo.WorkerThread (runtime)`

This thread initializes the simulation (reinit) and then runs the simulation in its run method. It keeps querying moose for running status every second and returns when the simulation is over. It puts its name in the global worker\_queue at the end to signal successful completion.

**run ()**

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

`threading_demo.main ()`

Example of using multithreading to run a MOOSE simulation in parallel with querying MOOSE objects involved. See the documentatin of the classes to get an idea of this demo's function.

## PyMoose

Author: Subhasis Ray

`traub_naf.create_compartment (parent_path, name)`

This shows how to use the prototype channel on a compartment.

`traub_naf.create_naf_proto ()`

Create an NaF channel prototype in /library. You can copy it later into any compartment or load a .p file with this channel using loadModel.

This channel has the conductance form:

$$G_k(v) = G_{bar} * m^3 * h * (V - E_k)$$

We are using all SI units

`traub_naf.do_iclamp (vclamp, iclamp, pid)`

Turn on current clamp and turn off voltage clamp

`traub_naf.do_vclamp` (*vclamp, iclamp, pid*)

Turn on voltage clamp and turn off current clamp

`traub_naf.main` ()

This is an example showing pymoose implementation of the NaF channel in Traub et al 2005

`traub_naf.run_clamp` (*model\_dict, clamp, levels, holding=0.0, simtime=0.1*)

Run either voltage or current clamp for default timing settings with multiple levels of command input.

`model_dict`: dictionary containing the model components -

*vclamp* - the voltage clamp amplifier

*iclamp* - the current clamp amplifier

*model* - the model container

*data* - the data container

*inject\_tab* - table recording membrane

*command\_tab* - table recording command input for voltage or current clamp

*vm\_tab* - table recording membrane potential

`clamp`: string specifying clamp mode, either *voltage* or *current*

**levels**: sequence of values for command input levels to be simulated.

`holding`: holding current or voltage

Returns: a dict containing the following lists of time series:

*command* - list of command input time series *inject* - list of of membrane current (includes injected current)

time series *vm* - list of membrane voltage time series *t* - list of time points for all of the above

`traub_naf.run_sim` (*model, data, simtime=0.1, simdt=1e-06, plotdt=0.0001, solver='ee'*)

Reset and run the simulation.

`model`: model container element

`data`: data container element

`simtime`: simulation run time

`simdt`: simulation timestep

`plotdt`: plotting time step

`solver`: neuronal solver to use.

`traub_naf.setup_electronics` (*model\_container, data\_container, compartment*)

Setup voltage and current clamp circuit using DiffAmp and PID and RC filter

`traub_naf.setup_model` ()

Setup the model and the electronic circuit. Also creates the data container.

## Mathematics with MOOSE

### Computing an arbitrary function

`function.main` ()

Function objects can be used to evaluate expressions with arbitrary number of variables and constants. We can assign expression of the form:

```
f(c0, c1, ..., cM, x0, x1, ..., xN, y0, ..., yP )
```

where  $c_i$ 's are constants and  $x_i$ 's and  $y_i$ 's are variables.

The constants must be defined before setting the expression and variables are connected via messages. The constants can have any name, but the variable names must be of the form  $x\{i\}$  or  $y\{i\}$  where  $i$  is increasing integer starting from 0.

The  $x_i$ 's are field elements and you have to set their number first (`function.x.num = N`). Then you can connect any source field sending out double to the 'input' destination field of the  $x[i]$ .

The  $y_i$ 's are useful when the required variable is a value field and is not available as a source field. In that case you connect the `requestOut` source field of the function element to the `get{Field}` destination field on the target element. The  $y_i$ 's are automatically added on connecting. Thus, if you call:

```
moose.connect(function, 'requestOut', a, 'getSomeField')
moose.connect(function, 'requestOut', b, 'getSomeField')
```

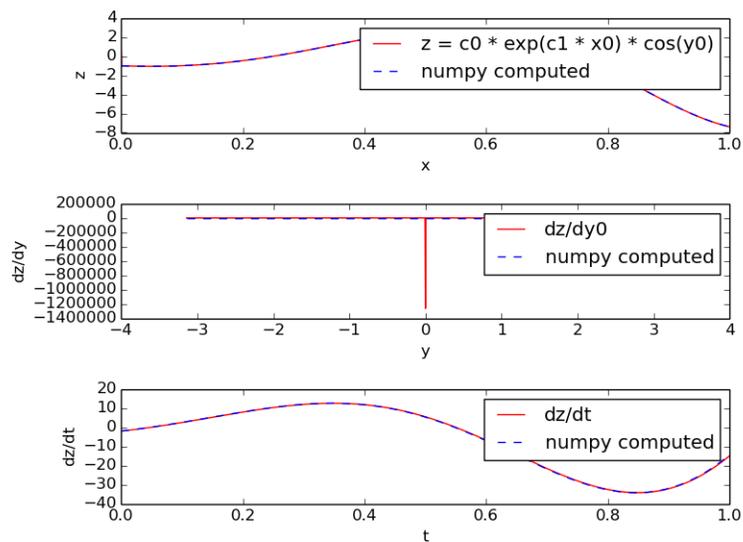
then `a.someField` will be assigned to  $y_0$  and `b.someField` will be assigned to  $y_1$ .

In this example we evaluate the expression:  $z = c_0 * \exp(c_1 * x_0) * \cos(y_0)$

with  $x_0$  ranging from -1 to +1 and  $y_0$  ranging from  $-\pi$  to  $+\pi$ . These values are stored in two stimulus tables called `xtab` and `ytab` respectively, so that at each timestep the next values of  $x_0$  and  $y_0$  are assigned to the function.

Along with the value of the expression itself we also compute its derivative with respect to  $y_0$  and its derivative with respect to time (rate). The former uses a five-point stencil for the numerical differentiation and has a glitch at  $y=0$ . The latter uses backward difference divided by `dt`.

Unlike `Func` class, the number of variables and constants are unlimited in `Function` and you can set all the variables via messages.



## Differential Equation Solving

```
diffEqSolution.main()
```

This snippet illustrates the solution of an arbitrary set of differential equations using the **Func** class and the **Pool**

class. The equations solved here are:

```
tauI.m' = Ca - Ca_tgt
tauG.chan' = m - chan
```

These equations are taken from: O’Leary et al *Neuron* 2014.

**func** evaluates an arbitrary function each timestep. Here this function is the rate of change from the equations above. The rate of change is passed to the *increment* message of the **Pool**. The numerical integration method is the Exponential Euler method but this will work fine if the rates are slow compared to the simulation timestep.

Conceptually, the idea is that if Ca is greater than the target level, then more mRNA is made, which makes more channels. Although the equations have no upper or lower bounds on **m** or **chan**, MOOSE is sensible about preventing the molecular pools from having negative concentrations. This does mean that the solution method employed here won’t work for the general solution of differential equations in non-chemical systems.

## Harmonic Oscillatory Function

funcRateHarmonicOsc.**main**()

funcRateHarmonicOsc illustrates the use of function objects to directly define the rates of change of pool concentration. This example shows how to set up a simple harmonic oscillator system of differential equations using the script. In normal use one would prefer to use SBML.

The equations are

```
p' = v - offset1
v' = -k(p - offset2)
```

where the rates for Pools p and v are computed using Functions. Note the use of offsets. This is because MOOSE chemical systems cannot have negative concentrations.

The model is set up to run using default Exponential Euler integration, and then using the GSL deterministic solver.

## Lotka-Volterra Model

funcReacLotkaVolterra.**main**()

The funcReacLotkaVolterra example shows how to use function objects as part of differential equation systems in the framework of the MOOSE kinetic solvers. Here the system is set up explicitly using the scripting, in normal use one would expect to use SBML.

In this example we set up a Lotka-Volterra system. The equations are readily expressed as a pair of reactions each of whose rate is governed by a function:

```
x' = x( alpha - beta.y )
y' = -y( gamma - delta.x )
```

This translates into two reactions:

```
x ---> z      Kf = beta.y - alpha
y ---> z      Kf = gamma - delta.x
```

Here z is a dummy molecule whose concentration is buffered to zero.

The model first runs using default Exponential Euler integration. This is not particularly accurate even with a small timestep. The model is then converted to use the deterministic Kinetic solver Ksolve. This is accurate

and faster. Note that we cannot use the stochastic GSSA solver for this system, it cannot handle a reaction term whose rate keeps changing.

```
stochasticLotkaVolterra.main()
```

The stochasticLotkaVolterra example is almost identical to the funcReacLotkaVolterra. It shows how to use function objects as part of differential equation systems in the framework of the MOOSE kinetic solvers. Here the difference is that we use a stochastic solver. The system is interesting because it illustrates the instability of Lotka-Volterra systems in stochastic conditions. Here we see extinction of one of the species and runaway buildup of the other. The simulation has to be halted at this point.

Here the system is set up explicitly using the scripting, in normal use one would expect to use SBML.

In this example we set up a Lotka-Volterra system. The equations are readily expressed as a pair of reactions each of whose rate is governed by a function:

```
x' = x( alpha - beta.y )
y' = -y( gamma - delta.x )
```

This translates into two reactions:

```
x ---> z      Kf = beta.y - alpha
y ---> z      Kf = gamma - delta.x
```

Here z is a dummy molecule whose concentration is buffered to zero.

The model first runs using default Exponential Euler integration. This is not particularly accurate even with a small timestep. The model is then converted to use the deterministic Kinetic solver Ksolve. This is accurate and faster.

Note that we cannot use the stochastic GSSA solver for this system, it cannot handle a reaction term whose rate keeps changing.

## Vary Concentration with mathematical function

```
funcInputToPools.main()
```

This example describes the special (and discouraged) use case where functions provide input to a reaction system. Here we have two functions of time which control the pool # and pool rate of change, respectively:

number of molecules of a =  $1 + \sin(t)$  rate of change of number of molecules of b =  $10 * \cos(t)$

In the stochastic case one must set a special flag *useClockedUpdate* in order to achieve clock-triggered updates from the functions. This is needed because the functions do not have reaction events to trigger them, and even if there were reaction events they might not be frequent enough to track the periodic updates. The use of this flag slows down the calculations, so try to use a table to control a pool instead.

To run in stochastic mode:

```
'python funcInputToPools'
```

To run in deterministic mode:

```
'python funcInputToPools false'
```

## 1.3 Cook Book

The MOOSE Cookbook contains recipes showing you, how to do specific tasks in MOOSE.

### 1.3.1 Single Neuron Electrical Aspects (BioPhysics)

#### Neuron Modeling

Neurons are modelled as equivalent electrical circuits. The morphology of a neuron can be broken into isopotential compartments connected by axial resistances  $R_a$  denoting the cytoplasmic resistance. In each compartment, the neuronal membrane is represented as a capacitance  $C_m$  with a shunt leak resistance  $R_m$ . Electrochemical gradient (due to ion pumps) across the leaky membrane causes a voltage drive  $E_m$ , that hyperpolarizes the inside of the cell membrane compared to the outside.

Each voltage dependent ion channel, present on the membrane, is modelled as a voltage dependent conductance  $G_k$  with gating kinetics, in series with an electrochemical voltage drive (battery)  $E_k$ , across the membrane capacitance  $C_m$ , as in the figure below.

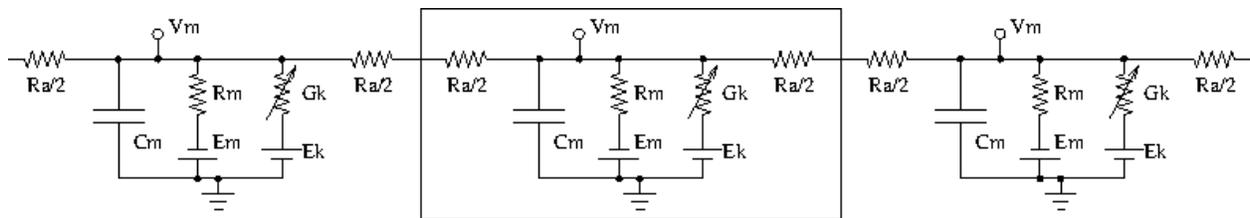


Fig. 6: Equivalent circuit of neuronal compartments

Neurons fire action potentials / spikes (sharp rise and fall of membrane potential  $V_m$ ) due to voltage dependent channels. These result in opening of excitatory / inhibitory synaptic channels (conductances with batteries, similar to voltage gated channels) on other connected neurons in the network.

MOOSE can handle large networks of detailed neurons, each with complicated channel dynamics. Further, MOOSE can integrate chemical signalling with electrical activity. Presently, creating and simulating these requires PyMOOSE scripting, but these will be incorporated into the GUI in the future.

To understand channel kinetics and neuronal action potentials, run the Squid Axon demo installed along with MOOSEGUI and consult its help/tutorial.

Read more about compartmental modelling in the first few chapters of the [Book of Genesis](#).

Models can be defined in [NeuroML](#), an XML format which is mostly supported across simulators. Channels, neuronal morphology (compartments), and networks can be specified using various levels of NeuroML, namely ChannelML, MorphML and NetworkML. Importing of cell models in the [GENESIS .p](#) format is supported for backwards compatibility.

#### Modeling details

Some salient properties of neuronal building blocks in MOOSE are described below. Variables that are updated at every simulation time step are listed **dynamical**. Rest are parameters.

- **Compartment** When you select a compartment, you can view and edit its properties in the right pane.  $V_m$  and  $I_m$  are plot-able.
  - $V_m$  membrane potential (across  $C_m$ ) in Volts. It is a dynamical variable.
  - $C_m$  membrane capacitance in Farads.

- $E_m$  membrane leak potential in Volts due to the electrochemical gradient setup by ion pumps.
  - $I_m$  current in Amperes across the membrane via leak resistance  $R_m$ .
  - **inject** current in Amperes injected externally into the compartment.
  - **initVm** initial  $V_m$  in Volts.
  - $R_m$  membrane leak resistance in Ohms due to leaky channels.
  - **diameter** diameter of the compartment in metres.
  - **length** length of the compartment in metres.
- **HHChannel** Hodgkin-Huxley channel with voltage dependent dynamical gates.
    - **Gbar** peak channel conductance in Siemens.
    - $E_k$  reversal potential of the channel, due to electrochemical gradient of the ion(s) it allows.
    - $G_k$  conductance of the channel in Siemens.  $G_k(t) = Gbar \times X(t)^{Xpower} \times Y(t)^{Ypower} \times Z(t)^{Zpower}$
    - $I_k$ 

**current through the channel into the neuron in Amperes.**  $I_k(t) = G_k(t) \times (E_k - V_m(t))$
    - **X, Y, Z** gating variables (range 0.0 to 1.0) that may turn on or off as voltage increases with different time constants.
 
$$dX(t)/dt = X_{inf}/\tau - X(t)/\tau$$

Here,  $X_{inf}$  and  $\tau$  are typically sigmoidal/linear/linear-sigmoidal functions of membrane potential  $V_m$ , which are described in a ChannelML file and presently not editable from MOOSEGUI. Thus, a gate may open ( $X_{inf}(V_m) \rightarrow 1$ ) or close ( $X_{inf}(V_m) \rightarrow 0$ ) on increasing  $V_m$ , with time constant  $\tau(V_m)$ .
    - **Xpower, Ypower, Zpower** powers to which gates are raised in the  $G_k(t)$  formula above.
  - **HHChannel2D** The Hodgkin-Huxley channel2D can have the usual voltage dependent dynamical gates, and also gates that depend on voltage and an ionic concentration, as for say Ca-dependent K conductance. It has the properties of HHChannel above, and a few more, similar to in the [GENESIS tab2Dchannel reference](#).
  - **CaConc** This is a pool of Ca ions in each compartment, in a shell volume under the cell membrane. The dynamical Ca concentration increases when Ca channels open, and decays back to resting with a specified time constant  $\tau$ . Its concentration controls Ca-dependent K channels, etc.
    - **Ca** Ca concentration in the pool in units mM ( i.e., mol/m<sup>3</sup>).
 
$$d[Ca^{2+}]/dt = B \times I_{Ca} - [Ca^{2+}]/\tau$$
    - **CaBasal/Ca\_base** Base Ca concentration to which the Ca decays
    - **tau** time constant with which the Ca concentration decays to the base Ca level.
    - **B** constant in the  $[Ca^{2+}]$  equation above.
    - **thick** thickness of the Ca shell within the cell membrane which is used to calculate B (see Chapter 19 of [Book of GENESIS](#).)

### Neuronal simulations in MOOSEGUI

Neuronal models in various formats can be loaded and simulated in the **MOOSE Graphical User Interface**. The GUI displays the neurons in 3D, and allows visual selection and editing of neuronal properties. Plotting and visualization of activity proceeds concurrently with the simulation. Support for creating and editing channels, morphology and networks is planned for the future. MOOSEGUI uses SI units throughout.

## moose-examples

- **Cerebellar granule cell**

**File -> Load ->** ~/moose/moose-examples/neuroml/GranuleCell/GranuleCell.net.xml

This is a single compartment Cerebellar granule cell with a variety of channels [Maex, R. and De Schutter, E., 1997](#) (exported from <http://www.neuroconstruct.org/>). Click on its soma, and **See children** for its list of channels. Vary the Gbar of these channels to obtain regular firing, adapting and bursty behaviour (may need to increase tau of the Ca pool).

- **Pyloric rhythm generator in the stomatogastric ganglion of lobster**

**File -> Load ->** ~/moose/moose-examples/neuroml/pyloric/Generated.net.xml

- **Purkinje cell**

**File -> Load ->** ~/moose/moose-examples/neuroml/PurkinjeCell/Purkinje.net.xml

This is a purely passive cell, but with extensive morphology [De Schutter, E. and Bower, J. M., 1994] (exported from <http://www.neuroconstruct.org/>). The channel specifications are in an obsolete ChannelML format which MOOSE does not support.

- **Olfactory bulb subnetwork**

**File -> Load ->** ~/moose/moose-examples/neuroml/OlfactoryBulb/numgloms2\_seed100.0\_decimated.xml

This is a pruned and decimated version of a detailed network model of the Olfactory bulb [Gilra A. and Bhalla U., in preparation] without channels and synaptic connections. We hope to post the ChannelML specifications of the channels and synapses soon.

- **All channels cell**

**File -> Load ->** ~/moose/moose-examples/neuroml/allChannelsCell/allChannelsCell.net.xml

This is the Cerebellar granule cell as above, but with loads of channels from various cell types (exported from <http://www.neuroconstruct.org/>). Play around with the channel properties to see what they do. You can also edit the ChannelML files in ~/moose/moose-examples/neuroml/allChannelsCell/cells\_channels/ to experiment further.

- **NeuroML python scripts** In directory ~/moose/moose-examples/neuroml/GranuleCell, you can run python FvsI\_Granule98.py which plots firing rate vs injected current for the granule cell. Consult this python script to see how to read in a NeuroML model and to set up simulations. There are ample snippets in ~/moose/moose-examples/snippets too.

## Load and Run simple models

### Single Cubicle Compartmental Neuron

```
cubeMeshSigNeur.createSquid()  
    Create a single compartment squid model.
```

```
cubeMeshSigNeur.main()  
    This snippet demonstrates how to model a neuronal compartment, with chemical model in just a cubic volume.  
    The neuron is a squid neuron.
```

## Single Neuron Model

`testSigNeur.createSpine` (*parentCompt, parentObj, index, frac, length, dia, theta*)  
Create spine of specified dimensions and index

`testSigNeur.createSquid` ()  
Create a single compartment squid model.

`testSigNeur.main` ()  
A toy compartmental neuronal + chemical model. The neuronal model geometry sets up the chemical volume to match the parent dendrite and five dendritic spines, each with a shaft and head. This volume mapping uses the `NeuroMesh`, `SpineMesh` and `PsdMesh` classes from MOOSE. There is a 3-compartment chemical model to go with this: one for the dendrite, one for the spine head, and one for the postsynaptic density. Note that the three mesh classes distribute the chemical model appropriately to all the respective spines, and set up the diffusion to the dendrite. The electrical model contributes the incoming calcium flux to the chemical model. This comes from the synaptic channels. The signalling here does two things to the electrical model. First, the amount of receptor in the chemical model controls the amount of glutamate receptor in the PSD. Second, there is a small kinase reaction that phosphorylates and inactivates the dendritic potassium channel.

## Load neuron model from GENESIS

`neuronFromDotp.main` ()  
Demonstrates how to load a simple neuronal model in GENESIS dotp format. The model has branches and a few spines. It is adorned just with classic HH squid channels.

`neuronFromDotp.makeChannelPrototypes` ()  
Create channel prototypes for readcell.

## Integrate-and-fire models

`IntegrateFireZoo.main` ()  
Simulate current injection into various Integrate and Fire neurons.

All integrate and fire (IF) neurons are subclasses of `compartment`, so they have all the fields of a passive compartment. Multicompartmental neurons can be created. Even ion channels and synaptic channels can be added to them, say for sub-threshold behaviour.

The key addition is that they have a reset mechanism when the membrane potential  $V_m$  crosses a threshold. On each reset, a `spikeOut` message is generated, and the membrane potential is reset to  $V_{reset}$ . The threshold may be the spike generation threshold as for `LIF` and `AdThreshIF`, or it may be the peak of the spike as for `QIF`, `ExIF`, `AdExIF`, and `IzhIF`. The adaptive IFs have an extra adapting variable apart from membrane potential  $V_m$ .

Details of the IFs are given below. The fields of the MOOSE objects are named exactly as the parameters in the equations below.

**LIF: Leaky Integrate and Fire:**  $R_m * C_m * dV_m/dt = -(V_m - E_m) + R_m * I$

**QIF: Quadratic LIF:**  $R_m * C_m * dV_m/dt = a_0 * (V_m - E_m) * (V_m - v_{critical}) + R_m * I$

**ExIF: Exponential leaky integrate and fire:**  $R_m * C_m * dV_m/dt = -(V_m - E_m) + \Delta_{thresh} * \exp((V_m - thresh)/\Delta_{thresh}) + R_m * I$

**AdExIF: Adaptive Exponential LIF:**  $R_m * C_m * dV_m/dt = -(V_m - E_m) + \Delta_{thresh} * \exp((V_m - thresh)/\Delta_{thresh}) + R_m * I - w,$

$\tau_w * dw/dt = a_0 * (V_m - E_m) - w,$

At each spike,  $w \rightarrow w + b_0$  “

**AdThreshIF: Adaptive threshold LIF:**  $R_m * C_m * dV_m/dt = -(V_m - E_m) + R_m * I,$

$\tau_{Thresh} * d \text{ threshAdaptive} / dt = a_0 * (V_m - E_m) - \text{threshAdaptive},$

At each spike, `threshAdaptive` is increased by `threshJump` the spiking threshold adapts as `thresh + threshAdaptive`

**IzhIF: Izhikevich:**  $d V_m / dt = a_0 * V_m^2 + b_0 * V_m + c_0 - u + I/C_m,$

$d u / dt = a * ( b * V_m - u ),$

At each spike,  $u \rightarrow u + d,$

By default,  $a_0 = 0.04e6/V/s,$   $b_0 = 5e3/s,$   $c_0 = 140 V/s$  are set to SI units, so use SI consistently, or change  $a_0, b_0, c_0$  also if you wish to use other units.  $R_m$  from `Compartment` is not used here, `vReset` is same as `c` in the usual formalism. At rest,  $u_0 = b V_0,$  and  $V_0 = ( -(-b_0 - b) +/- \sqrt{((b_0 - b)^2 - 4 * a_0 * c_0)}) / (2 * a_0).$

On the command-line, in `moose-examples/snippets` directory, run `python IntegrateFireZoo.py`. The script will ask you which neuron you want to simulate and you can choose and run what you want. Play with the parameters of the IF neurons in the source code.

## Simple Examples

### Create a Leaky Neuron

Demonstrates use of Leaky Integrate and Fire (`LeakyIaf` class) in `moose`.

```
lif.setupmodel (modelpath, iaf_Rm, iaf_Cm, pulse_interval)
```

Create a `LeakyIaF` neuron under `modelpath` and a synaptic channel (`SynChan`) in it. Create a spike generator stimulated by a pulse generator to give input to the synapse.

### Create a Leaky Compartment

```
lifcomp.main ()
```

This is an example of how you can create a Leaky Integrate and Fire compartment using regular compartment and `Func` to check for threshold crossing and resetting the `Vm`.

```
lifcomp.setup_two_cells ()
```

Create two cells with leaky integrate and fire compartments. Each cell is a single compartment `a1` and `b2`. `a1` is stimulated by a step current injection.

The compartment `a1` is connected to the compartment `b2` through a synaptic channel.

### Voltage Clamping

```
vclamp.main ()
```

This snippet is to demonstrate modelling of voltage clamping.

### Generate Pulse

```
pulsegen.main ()
```

Demonstrates a pulsegen with multiple levels, delays and widths.

```
pulsegen2.main()
```

Pulse generator example

This example shows the full range of operations of PulseGen objects with a reimplementaion of corresponding GENESIS demo.

A PulseGen object can be run in three modes: free running (`trigMode=0`), triggered (`trigMode=1`) and gated (`trigMode=2`).

In the free running mode it keeps repeating the pulse series indefinitely.

In triggered mode, it generates a pulse series on the leading edge of the trigger signal coming to its *input* field. The trigger can be the *output* of another PulseGen as in this example.

In gated mode, the PulseGen acts as if it was free-running as long as the *input* remains high.

## Synapse

```
synapse.main()
```

This is an example of event messages from multiple SpikeGen objects into a synchan.

Create a SynChan element with 2 elements in synapse field.

Create 5 SpikeGen elements.

Connect alternet SpikeGen elements to synapse[0] and synapse[1]

... This is a minimal example. In real simulations the SpikeGens will be embedded in compartments representing axon terminals and the SynChans will be embedded in somatic/dendritic compartments.

## Message transmission via synapse

```
intfire.connect_spikegen()
```

Connect a SpikeGen object to an IntFire neuron such that spike events in spikegen get transmitted to the synapse of the IntFire neuron.

```
if3 = moose.IntFire('if3')
```

```
intfire.connect_two_intfires()
```

Connect two IntFire neurons so that spike events in one gets transmitted to synapse of the other.

```
intfire.main()
```

Demonstrates connection between 2 IntFire neurons to observe spike generation.

```
intfire.setup_synapse()
```

Create an intfire object and create two synapses on it.

## Gap Junction

```
gapjunction.main()
```

This example is to demonstrate, how gap junction can be modeled using MOOSE.

## Insert Spine heads

```
insertSpinesWithoutRdesigneur.main()
```

This snippet illustrates how the Neuron class does the spine specification, without the rdesigneur intermediate.

## Multi compartmental Neuron model

`testHsolve.create_spine` (*parentCompt, parentObj, index, frac, length, dia, theta*)  
Create spine of specified dimensions and index

`testHsolve.create_squid` ()  
Create a single compartment squid model.

`testHsolve.main` ()

### A small compartmental model that demonstrates ::

1. how to set up a multicompartmental model using SymCompartments
2. Solving this with the default Exponential Euler (EE) method
3. Solving this with the Hsolver.
4. What happens at different timesteps.

## 1.3.2 Chemical Aspects

### Interface for chemical kinetic models in MOOSEGUI

Upinder Bhalla, Harsha Rani

Nov 8 2016.

---

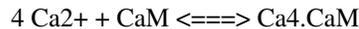
- *Introduction* (page 41)
- **\*\*TODO\*\*** *What are chemical kinetic models?* (page 41)
  - *Levels of model* (page 41)
  - *Numerical methods* (page 41)
- *Using Kinetikit 12* (page 41)
  - *Overview* (page 41)
  - *Model layout and icons* (page 42)
    - \* *Compartment* (page 43)
    - \* *Pool* (page 43)
    - \* *Buffered pools* (page 43)
    - \* *Reaction* (page 44)
    - \* *Mass-action enzymes* (page 45)
    - \* *Michaelis-Menten Enzymes* (page 47)
    - \* *Summation* (page 48)
  - *Model operations* (page 48)
  - *Model Building* (page 48)

## Introduction

Kinetikit 12 is a graphical interface for doing chemical kinetic modeling in MOOSE. It is derived in part from Kinetikit, which was the graphical interface used in GENESIS for similar models. Kinetikit, also known as kkit, was at version 11 with GENESIS. Here we start with Kinetikit 12.

### \*\*TODO\*\* What are chemical kinetic models?

Much of neuronal computation occurs through chemical signaling. For example, many forms of synaptic plasticity begin with calcium influx into the synapse, followed by calcium binding to calmodulin, and then calmodulin activation of numerous enzymes. These events can be represented in chemical terms:



Such chemical equations can be modeled through standard Ordinary Differential Equations, if we ignore space:

$$\begin{aligned} d[\text{Ca}]/dt &= -4K_f [\text{Ca}]^4 [\text{CaM}] + 4K_b [\text{Ca}_4.\text{CaM}] \\ d[\text{CaM}]/dt &= K_f [\text{Ca}]^4 [\text{CaM}] - K_b [\text{Ca}_4.\text{CaM}] \\ d[\text{Ca}_4.\text{CaM}]/dt &= K_b [\text{Ca}_4.\text{CaM}] - K_f [\text{Ca}]^4 [\text{CaM}] \end{aligned}$$

MOOSE models these chemical systems. This help document describes how to do such modelling using the graphical interface, Kinetikit 12.

## Levels of model

Chemical kinetic models can be simple well-stirred (or point) models, or they could have multiple interacting compartments, or they could include space explicitly using reaction-diffusion. In addition such models could be solved either deterministically, or using a stochastic formulation. At present Kinetikit handles compartmental models but does not compute diffusion within the compartments, though MOOSE itself can do this at the script level. Kkit12 will do deterministic as well as stochastic chemical calculations.

## Numerical methods

- **Deterministic:** Adaptive timestep 5th order Runge-Kutta-Fehlberg from the GSL (GNU Scientific Library).
- **Stochastic:** Optimized Gillespie Stochastic Systems Algorithm, custom implementation.

## Using Kinetikit 12

### Overview

- Load models using **'File -> Load model'**. A reaction schematic for the chemical system appears in the **'Editor view'** tab.
- From **'Editor view'** tab
- View parameters by clicking on icons, and looking at entries in **'Properties'** table to the right.
- Edit parameters by changing their values in the **'Properties'** table.
- From **'Run View'**
- Pools can be plotted by clicking on their icons and dragging the icons onto the plot Window. Presently only concentration v/s time is plottable.



## Compartment

The *compartment* in moose is usually a contiguous domain in which a certain set of chemical reactions and molecular species occur. The definition is very closely related to that of a cell-biological compartment. Examples include the extracellular space, the cell membrane, the cytosol, and the nucleus. Compartments can be nested, but of course you cannot put a bigger compartment into a smaller one.

- **Icon:** Grey boundary around a set of reactions.
- **Moving Compartments:** Click and drag on the boundary.
- **Resizing Compartment boundary:** Happens automatically when contents are repositioned, so that the boundary just contains contents.
- **Compartment editable parameters:**
- **‘name’:** The name of the compartment.
- **‘size’:** This is the volume, surface area or length of the compartment, depending on its type.
- **Compartment fixed parameters:**
- **‘numDimensions’:** This specifies whether the compartment is a volume, a 2-D surface, or if it is just being represented as a length.

## Pool

This is the set of molecules of a given species within a compartment. Different chemical states of the same molecule are in different pools.

# Pool

- **Icon:**  Colored rectangle with pool name in it.
- **Moving pools:** Click and drag.
- **Pool editable parameters:**
- **name:** Name of the pool
- **n:** Number of molecules in the pool
- **nInit:** Initial number of molecules in the pool. ‘n’ gets set to this value when the ‘reinit’ operation is done.
- **conc:** Concentration of the molecules in the pool.  $\text{conc} = n * \text{unit\_scale\_factor} / (N_{\text{sub>A</sub>}} * \text{vol})$
- **concInit:** Initial concentration of the molecules in the pool. ‘conc’ is set to this value when the ‘reinit’ operation is done.  
 $\text{concInit} = n\text{Init} * \text{unit\_scale\_factor} / (N_{\text{sub>A</sub>}} * \text{vol})$
- **Pool fixed parameters**
- **size:** Derived from the compartment that holds the pool. Specifies volume, surface area or length of the holding compartment.

## Buffered pools

Some pools are set to a fixed ‘n’, that is number of molecules, and therefore a fixed concentration, throughout a simulation. These are buffered pools.

# Buf- Pool

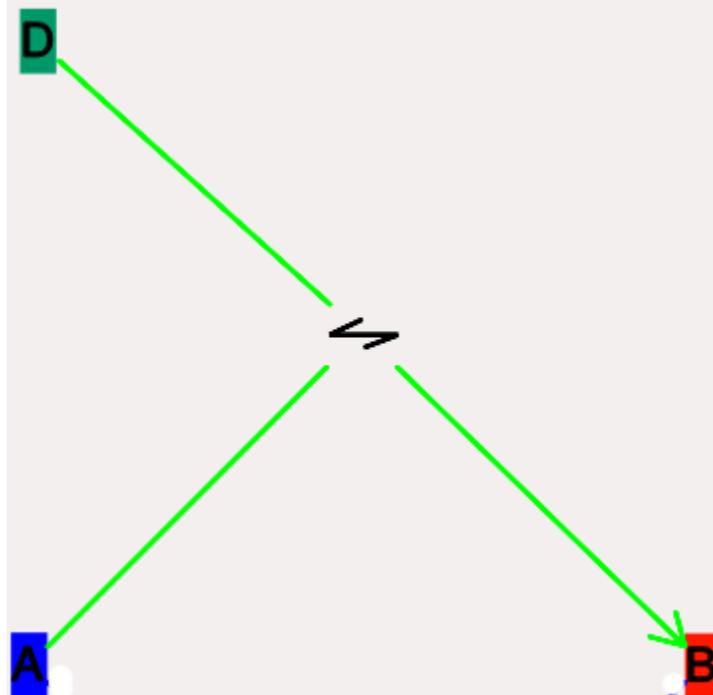
- **Icon:**  Colored rectangle with pool name in it.
- **Moving Buffered pools:** Click and drag.
- **Buffered Pool editable parameters**
- **name:** Name of the pool
- **nInit:** Fixed number of molecules in the pool. 'n' gets set to this value throughout the run.
- **concInit:** Fixed concentration of the molecules in the pool. 'conc' is set to this value throughout the run.  

$$\text{concInit} = \text{nInit} * \text{unit\_scale\_factor} / (\text{N}_{\text{A}} * \text{vol})$$
- **Pool fixed parameters:**
- **n:** Number of molecules in the pool. Derived from 'nInit'.
- **conc:** Concentration of molecules in the pool. Derived from 'concInit'.
- **size:** Derived from the compartment that holds the pool. Specifies volume, surface area or length of the holding compartment.

## Reaction

These are conversion reactions between sets of pools. They are reversible, but you can set either of the rates to zero to get irreversibility. In the illustration below, 'D' and 'A' are substrates, and 'B' is the product of the reaction. This is indicated by the direction of the green arrow.

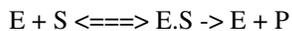
- **Icon:**  Reversible reaction arrow.
- **Moving Reactions:** Click and drag.
- **Reaction editable parameters:**
- **Name :** Name of reaction
- **K:sub:'f' :** 'Forward rate' of reaction, in 'concentration/time' units. This is the normal way to express and manipulate the reaction rate.
- **k:sub:'f' :** Forward rate of reaction, in 'number/time' units. This is used internally for computations, but is volume-dependent and should not be used to manipulate the reaction rate unless you really know what you are doing.
- **K:sub:'b' :** Backward rate' of reaction, in 'concentration/time' units. This is the normal way to express and manipulate the reaction rate.
- **k:sub:'b' :** Backward rate of reaction, in 'number/time' units. This is used internally for computations, but is volume-dependent and should not be used to manipulate the reaction rate unless you really know what you are doing.



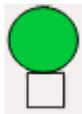
- **Reaction fixed parameters:**
- **numSubstrates:** Number of substrates molecules.
- **numProducts:** Number of product molecules.

### Mass-action enzymes

These are enzymes that model the chemical equation's



Note that the second reaction is irreversible. Note also that mass-action enzymes include a pool to represent the 'E.S' (enzyme-substrate) complex. In the example below, the enzyme pool is named 'MassActionEnz', the substrate is 'C', and the product is 'E'. The direction of the enzyme reaction is indicated by the red arrows.

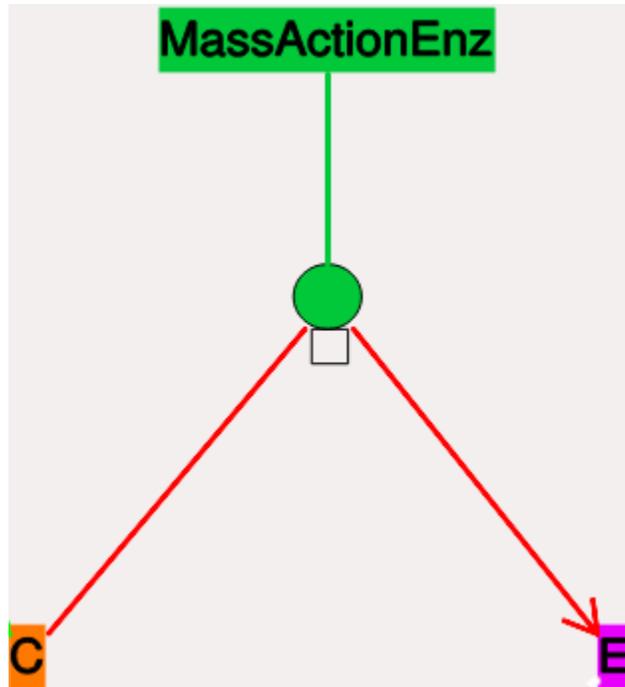


- **Icon:** Colored ellipse atop a small square. The ellipse represents the enzyme. The small square represents 'E.S', the enzyme-substrate complex. The ellipse icon has the same color as the enzyme pool 'E'. It is connected to the enzyme pool 'E' with a straight line of the same color.

The ellipse icon sits on a continuous, typically curved arrow in red, from the substrate to the product.

A given enzyme pool can have any number of enzyme activities, since the same enzyme might catalyze many reactions.

- **Moving Enzymes:** Click and drag on the ellipse.
- **Enzyme editable parameters**
- **name :** Name of enzyme.
- **K:sub:'m' :** Michaelis-Menten value for enzyme, in 'concentration' units.



- **k:sub:'cat'** : Production rate of enzyme, in '1/time' units. Equal to  $k_3$ , the rate of the second, irreversible reaction.
- **k1** : Forward rate of the **E+S** reaction, in number and '1/time' units. This is what is used in the internal calculations.
- **k2** : Backward rate of the **E+S** reaction, in '1/time' units. Used in internal calculations.
- **k3** : Forward rate of the **E.S -> E + P** reaction, in '1/time' units. Equivalent to  $k_{cat}$ . Used in internal calculations.
- **ratio** : This is equal to  $k_2/k_{sub:3}$ . Needed to define the internal rates in terms of  $K_m$  and  $k_{cat}$ . I usually use a value of 4.
- **Enzyme-substrate-complex editable parameters:** These are identical to those of any other pool.
- **name:** Name of the **E.S** complex. Defaults to `**_cplx**`.
- **n:** Number of molecules in the pool
- **nInit:** Initial number of molecules in the complex. 'n' gets set to this value when the 'reinit' operation is done.
- **conc:** Concentration of the molecules in the pool.  

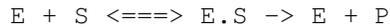
$$\text{conc} = n * \text{unit\_scale\_factor} / (N_{\text{sub>A</sub>}} * \text{vol})$$
- **concInit:** Initial concentration of the molecules in the pool. 'conc' is set to this value when the 'reinit' operation is done.  $\text{concInit} = n\text{Init} * \text{unit\_scale\_factor} / (N_{\text{sub>A</sub>}} * \text{vol})$
- **Enzyme-substrate-complex fixed parameters:**
- **size:** Derived from the compartment that holds the pool. Specifies volume, surface area or length of the holding compartment. Note that the Enzyme-substrate-complex is assumed to be in the same compartment as the enzyme molecule.

## Michaelis-Menten Enzymes

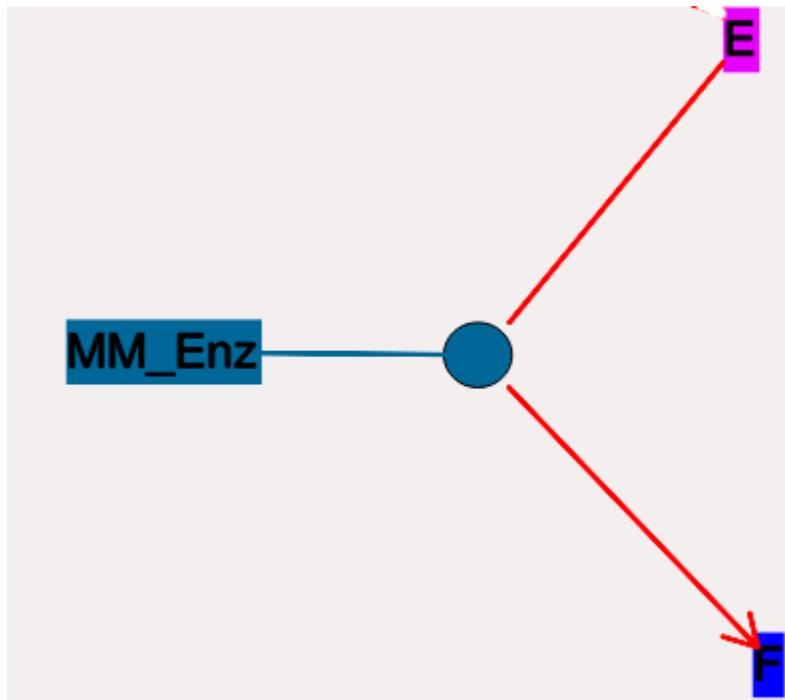
These are enzymes that obey the Michaelis-Menten equation

$V = V_{\text{max}} * [S] / (K_m + [S]) = k_{\text{cat}} * [E_{\text{tot}}] * [S] / (K_m + [S])$  where -  $V_{\text{max}}$  is the maximum rate of the enzyme -  $[E_{\text{tot}}]$  is the total amount of the enzyme -  $K_m$  is the Michaelis-Menten constant -  $S$  is the substrate.

Nominally these enzymes model the same chemical equation as the mass-action enzyme':



but they make the assumption that the  $E.S$  is in a quasi-steady-state with  $E$  and  $S$ , and they also ignore sequestration of the enzyme into the complex. So there is no representation of the  $E.S$  complex. In the example below, the enzyme pool is named **MM\_Enz**, the substrate is **E**, and the product is **P**. The direction of the enzyme reaction is indicated by the red arrows.

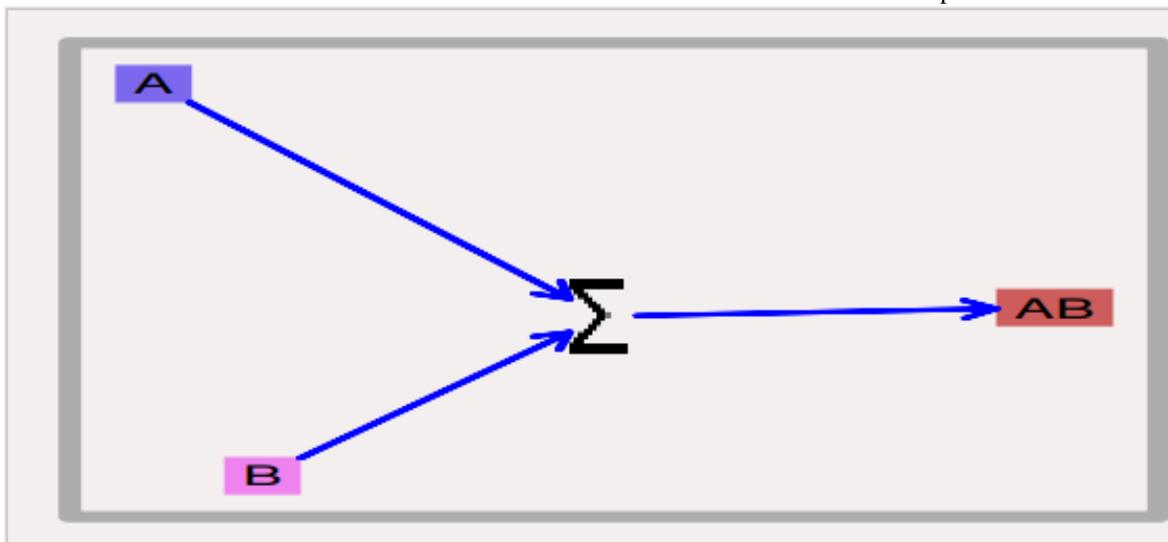


- **Icon:**  Colored ellipse. The ellipse represents the enzyme. The ellipse icon has the same color as the enzyme 'MM\_Enz'. It is connected to the enzyme pool 'MM\_Enz' with a straight line of the same color. The ellipse icon sits on a continuous, typically curved arrow in red, from the substrate to the product. A given enzyme pool can have any number of enzyme activities, since the same enzyme might catalyze many reactions.
- **Moving Enzymes:** Click and drag.
- **Enzyme editable parameters:**
  - **name:** Name of enzyme.
  - $K_m$ : Michaelis-Menten value for enzyme, in 'concentration' units.
  - $k_{\text{cat}}$ : Production rate of enzyme, in '1/time' units. Equal to  $k_3$ , the rate of the second, irreversible reaction.

## Summation

Summation object can be used to add specified variable values. The variables can be input from pool object.

- **Icon:** This is  in the example image below. The input pools 'A' and 'B' connect to the  with blue arrows. The function outputs to BuffPool



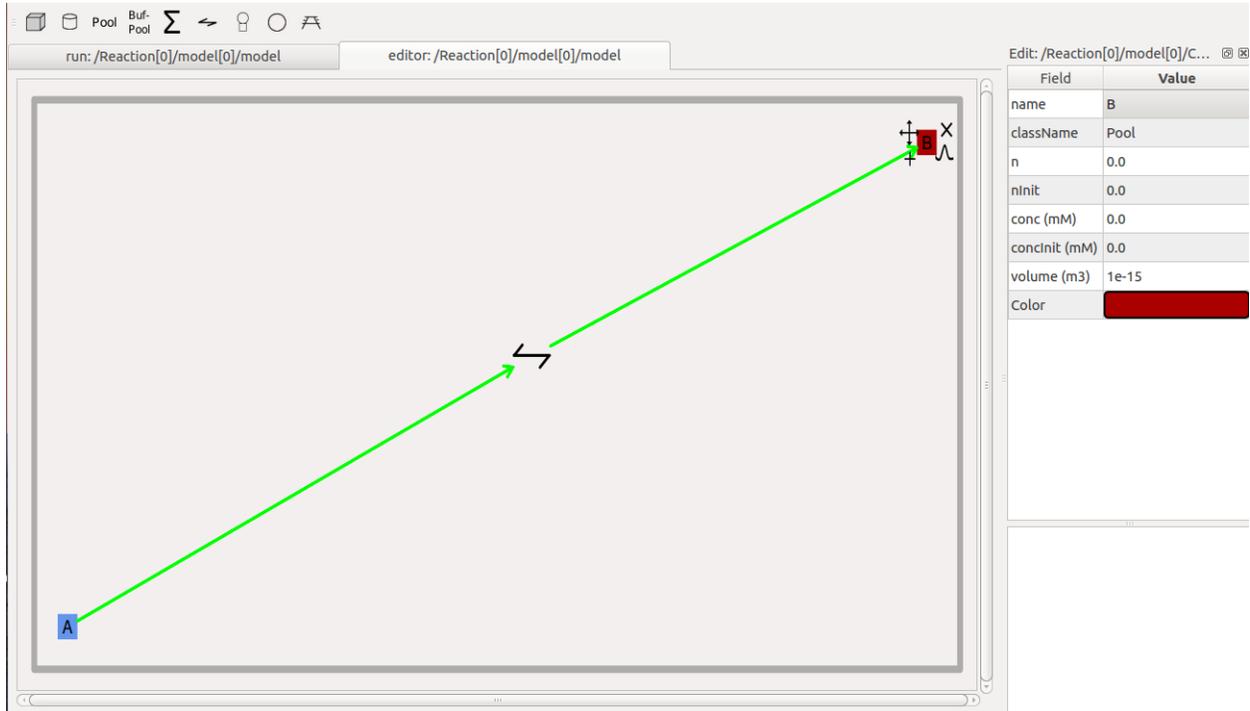
## Model operations

- **Loading models:** File -> Load Model -> select from dialog. This operation makes the previously loaded model disable and loads newly selected models in 'Model View'.
- **New:** File -> New -> Model name. This opens a empty widget for model building
- **Saving models:** File -> Save Model -> select from dialog.
- **Changing numerical methods:** Preference->Chemical tab item from Simulation Control. Currently supports:
  - Runge Kutta: This is the Runge-Kutta-Fehlberg implementation from the GNU Scientific Library (GSL). It is a fifth order variable timestep explicit method. Works well for most reaction systems except if they have very stiff reactions.
  - Gillespie: **Optimized Gillespie stochastic systems algorithm, custom implementation. This uses variable timesteps internal**  
Note that it slows down with increasing numbers of molecules in each pool. It also slows down, but not so badly, if the number of reactions goes up.
  - **Exponential Euler:** This methods computes the solution of partial and ordinary differential equations.

## Model building

- The **Edit Widget** includes various menu options and model icons on the top. Use the mouse button to click and drag icons from toolbar to Edit Widget, two things will happen, **icon** will appear in the editor widget and a **object editor** will pop up with lots of parameters with respect to moose object.

**Rules:**

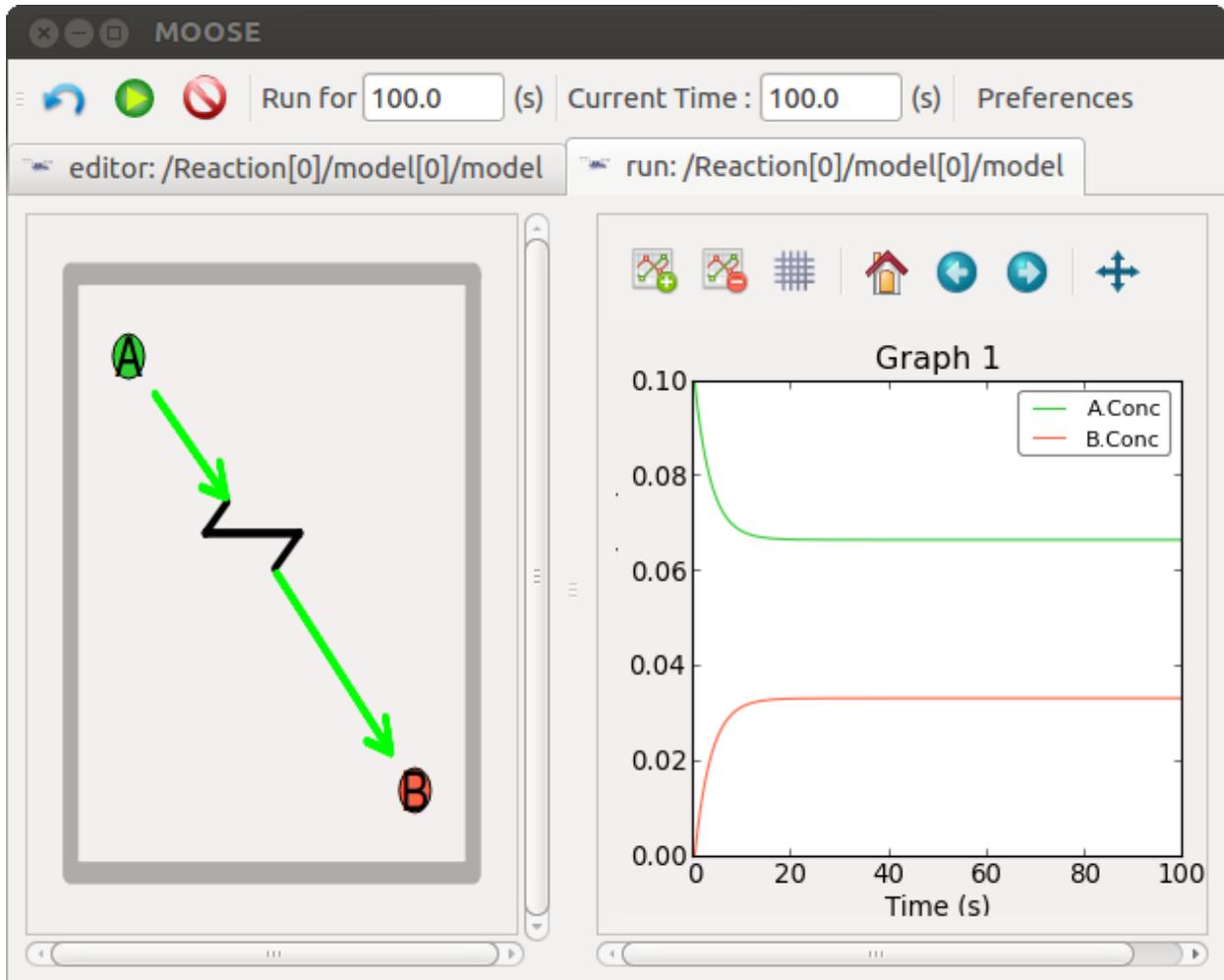


- \* Compartment has to be created firstly (At present only single compartment model **is** allowed)
- \* Enzyme should be dropped on a pool **as** parent
- \* function should be dropped on buffPool **for** output

#### Note:

- \* Drag **in** pool's and reaction on to the editor widget, now one can set up a **reaction**.
- \* Click on mooseObject one can find a little arrow on the top right corner of the **object**, drag **from this** little arrow to any **object for** connection. e.g pool to **reaction and** reaction to pool. Specific connection **type** gets specific colored arrow. e.g. Green color arrow **for** specifying connection between reactant **and** product **for** reaction.
- \* Clicking on the **object** one can rearrange **object for** clean layout.
- \* Second order reaction can also be done by repeating the connection over again
- \* Each connection can be deleted **and** using rubberband selection each moose **object** can be deleted

- From **run widget**, pools are draggable to plot window for plotting. (Currently **conc** is plotted as default field) Plots are color-coded as per in model.
- Model can be run by clicking **start** button. One can stop button in mid-stream and start up again without affecting the calculations. The reset button clears the simulation.



## Load - Run - Save models

### Load a Kinetic Model

`loadKineticModel.main()`

This example illustrates loading, running, and saving a kinetic model defined in kkit format. It uses a default kkit model but you can specify another using the command line

```
python filename runtime solver.
```

We use the gsl solver here. The model already defines a couple of plots and sets the runtime 20 secs.

### Load an SBML Model

`loadSbmlmodel.main()`

This example illustrates loading, running of an SBML model defined in XML format. The model 00001-sbml-l3v1.xml is taken from l3v1 SBML testcase. Plots are setup. Model is run for 20sec. As a general rule we created model under '/path/model' and plots under '/path/graphs'.

### Load a CSpace Model

`loadCspaceModel.main()`

This example illustrates loading and running, a kinetic model defined in cspace format. We use the gsl solver here. The model already defines a couple of plots and sets the runtime to 3000 seconds.

### Save a model into SBML format

`convert_Genesis2Sbml.main()`

This example illustrates loading a kinetic model defined in Genesis format into Moose using `loadModel` function and using `writeSBML` function one can save the model into SBML format.

`libsbml` should be installed.

### Save a model

`savemodel.main()`

This example illustrates loading a kinetic model defined in Genesis format into Moose using "loadModel" function and using "saveModel" function one can save the model back to Genesis format

## Simple Examples

### Set-up a kinetic solver and model

#### with Scripting

`scriptGssaSolver.main()`

This example illustrates how to set up a kinetic solver and kinetic model using the scripting interface. Normally this would be done using the `Shell::doLoadModel` command, and normally would be coordinated by the `SimManager` as the base of the entire model. This example creates a bistable model having two enzymes and a reaction. One of the enzymes is autocatalytic. The model is set up to run using Exponential Euler integration.

### With something else

`changeFuncExpression.main()`

This example illustrates how to set up a kinetic solver and kinetic model using the scripting interface. Normally this would be done using the `Shell::doLoadModel` command, and normally would be coordinated by the `SimManager` as the base of the entire model. This example creates a bistable model having two enzymes and a reaction. One of the enzymes is autocatalytic. The model is set up to run using Exponential Euler integration.

### Building a chemical Model from Parts

Disclaimer: Avoid doing this for all but the very simplest models. This is error-prone, tedious, and non-portable. For preference use one of the standard model formats like SBML, which MOOSE and many other tools can read and write.

Nevertheless, it is useful to see how these models are set up. There are several tutorials and snippets that build the entire chemical model system using the basic MOOSE calls. The sequence of steps is typically:

1. Create container (chemical compartment) for model. This is typically a `CubeMesh`, a `CylMesh`, and if you really know what you are doing, a `NeuroMesh`.
2. Create the reaction components: pools of molecules **moose.Pool**; reactions **moose.Reac**; and enzymes **moose.Enz**. Note that when creating an enzyme, one must also create a molecule beneath it to serve as the enzyme-substrate complex. Other less-used components include Michaelis-Menten enzymes **moose.MMenz**, input tables, pulse generators and so on. These are illustrated in other examples. All these reaction components should be child objects of the compartment, since this defines what volume they will occupy. Specifically, a pool or reaction object must be placed somewhere below the compartment in the object tree for the volume to be set correctly and for the solvers to know what to use.
3. Assign parameters for the components.
  - Compartments have a **volume**, and each subtype will have quite elaborate options for partitioning the compartment into voxels.
  - **Pool**s have one key parameter, the initial concentration **concInit**.
  - **Reac**tions have two parameters: **Kf** and **Kb**.
  - **Enz**ymes have two primary parameters **kcat** and **Km**. That is enough for **MMenz**ymes. Regular **Enz**ymes have an additional parameter **k2** which by default is set to 4.0 times **kcat**, but you may also wish to explicitly assign it if you know its value.
4. Connect up the reaction system using moose messaging.
5. Create and connect up input and output tables as needed.
6. Create and connect up the solvers as needed. This has to be done in a specific order. Examples are linked below, but briefly the order is:
  - (a) Make the compartment and reaction system.
  - (b) Make the `Ksolve` or `Gsolve`.
  - (c) Make the `Stoich`.
  - (d) Assign **stoich.compartment** to the compartment
  - (e) Assign **stoich.ksolve** to either the `Ksolve` or `Gsolve`.
  - (f) Assign **stoich.path** to finally fill in the reaction system.

An example of manipulating the models is as following:

```
scriptKineticSolver.main()
```

This example illustrates how to set up a kinetic solver and kinetic model using the scripting interface. Normally this would be done using the `Shell::doLoadModel` command, and normally would be coordinated by the `SimManager` as the base of the entire model. This example creates a bistable model having two enzymes and a reaction. One of the enzymes is autocatalytic. The model is set up to run using Exponential Euler integration.

The recommended way to build a chemical model, of course, is to load it in from a file format specific to such models. MOOSE understands **SBML**, **kkit.g** (a legacy GENESIS format), and **cspace** (a very compact format used in a large study of bistables from Ramakrishnan and Bhalla, PLoS Comp. Biol 2008).

One key concept is that in MOOSE the components, messaging, and access to model components is identical regardless of whether the model was built from parts, or loaded in from a file. All that the file loaders do is to use the file to automate the steps above. Thus the model components and their fields are completely accessible from the script even if the model has been loaded from a file.

## Cross-Compartment Reaction Systems

Frequently reaction systems span cellular (chemical) compartments. For example, a membrane-bound molecule may be phosphorylated by a cytosolic kinase, using soluble ATP as one of the substrates. Here the membrane and the cytosol are different chemical compartments. MOOSE supports such reactions. The following snippets illustrate cross-compartment chemistry. Note that the interpretation of the rates of enzymes and reactions does depend on which compartment they reside in.

```
crossComptSimpleReac.main()
```

This example illustrates a simple cross compartment reaction:

```
a <====> b <====> c
```

Here each molecule is in a different compartment. The initial conditions are such that the end conc on all compartments should be 2.0. The time course depends on which compartment the `Reac` object is embedded in. The cleanest thing numerically and also conceptually is to have both reactions in the same compartment, in this case the middle one (**compt1**). The initial conditions have a lot of **B**. The equilibrium with **C** is fast and so **C** shoots up and passes **B**, peaking at about (2.5,9). This is also just about the crossover point. **A** starts low and slowly climbs up to equilibrate.

If we put **reac0** in **compt0** and **reac1** in **compt1**, it behaves the same qualitatively but now the peak is at around (1, 5.2)

This configuration of reactions makes sense from the viewpoint of having the reactions always in the compartment with the smaller volume, which is important if we need to have junctions where many small voxels talk to one big voxel in another compartment.

Note that putting the reacts in other compartments doesn't work and in some combinations (e.g., **reac0** in **compt0** and **reac1** in **compt2**) give numerical instability.

```
crossComptOscillator.main()
```

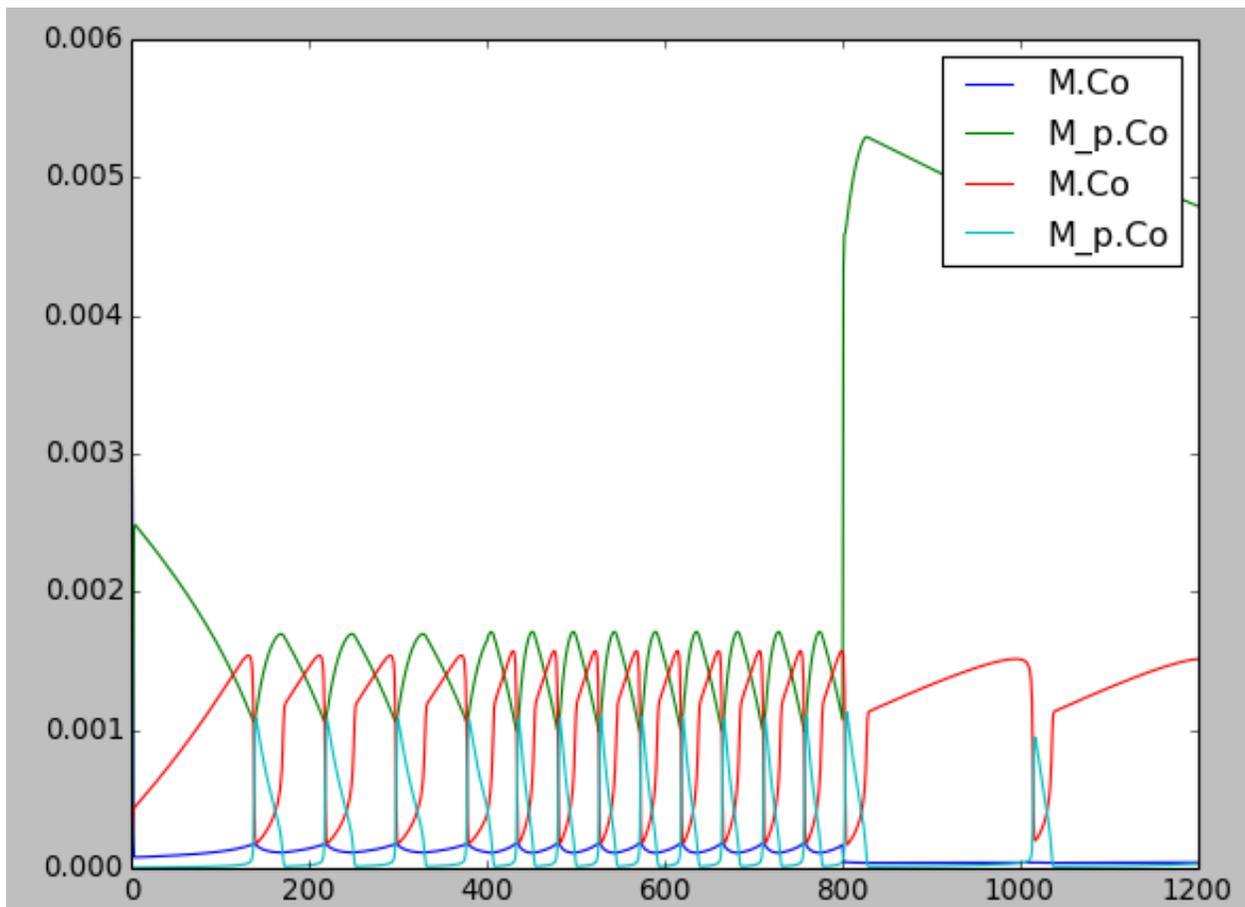
This example illustrates loading and running a reaction system that spans two volumes, that is, is in different compartments. It uses a `kkit` model file. You can tell if it is working if you see nice relaxation oscillations.

```
crossComptNeuroMesh.main()
```

This example illustrates how to define a kinetic model embedded in a `NeuroMesh`, and undergoing cross-compartment reactions. It is completely self-contained and does not use any external model definition files. Normally one uses standard model formats like `SBML` or `kkit` to concisely define kinetic and neuronal models. This example creates a simple reaction:

```
a <==> b <==> c
```





## Models' Demonstration

### Oscillation Model

`slowFbOsc.main()`

This example illustrates loading, and running a kinetic model for a delayed -ve feedback oscillator, defined in kkit format. The model is one by Boris N. Kholodenko from Eur J Biochem. (2000) 267(6):1583-8

This model has a high-gain MAPK stage, whose effects are visible when one looks at the traces from successive stages in the plots. The upstream pools have small early peaks, and the downstream pools have large delayed ones. The negative feedback step is mediated by a simple binding reaction of the end-product of oscillation with an upstream activator.

We use the `gsl` solver here. The model already defines some plots and sets the runtime to 4000 seconds. The model does not really play nicely with the GSSA solver, since it involves some really tiny amounts of the MAPKKK.

Things to do with the model:

- Look at model once it is loaded in:

```
moose.le( '/model' )
moose.showfields( '/model/kinetics/MAPK/MAPK' )
```

- Behold the amplification properties of the cascade. Could do this by blocking the feedback step and giving a small pulse input.
- Suggest which parameters you would alter to change the period of the oscillator:

- Concs of various molecules, for example:

```
ras_MAPKKKK = moose.element( '/model/kinetics/MAPK/Ras_dash_MKKKK' )
moose.showfields( ras_MAPKKKK )
ras_MAPKKKK.concInit = 1e-5
```

- Feedback reaction rates
- Rates of all the enzymes:

```
for i in moose.wildcardFind( '/##[ISA=EnzBase]' ):
    i.kcat *= 10.0
```

`repressillator.main()`

This example illustrates the classic **Repressillator** model, based on Elowitz and Liebler, Nature 2000. The model has the basic architecture:

```
A ---| B---| C
T           |
|           |
|_____|
```

where **A**, **B**, and **C** are genes whose products repress each other. The plunger symbol indicates inhibition. The model uses the Gillespie (stochastic) method by default but you can run it using a deterministic method by saying `python repressillator.py gsl`

Good things to do with this model include:

- Ask what it would take to change period of repressillator:
  - Change inhibitor rates:

```
inhib = moose.element( '/model/kinetics/TetR_gene/inhib_reac' )
moose.showfields( inhib )
inhib.Kf *= 0.1
```

- Change degradation rates:

```
degrade = moose.element( '/model/kinetics/TetR_gene/TetR_degradation' )
degrade.Kf *= 10.0
```

- Run in stochastic mode:

- Change volumes, figure out how many molecules are present:

```
lac = moose.element( '/model/kinetics/lac_gene/lac' )
print lac.n` `
```

- Find when it becomes hopelessly unreliable with small volumes.

relaxationOsc.main()

This example illustrates a **Relaxation Oscillator**. This is an oscillator built around a switching reaction, which tends to flip into one or other state and stay there. The relaxation bit comes in because once it is in state 1, a slow (relaxation) process begins which eventually flips it into state 2, and vice versa.

The model is based on Bhalla, Biophys J. 2011. It is defined in kkit format. It uses the deterministic gsl solver by default. You can specify the stochastic Gillespie solver on the command line `python relaxationOsc.py gssa`

Things to do with the model:

- Figure out what determines its frequency. You could change the initial concentrations of various model entities:

```
ma = moose.element( '/model/kinetics/A/M' )
ma.concInit *= 1.5
```

Alternatively, you could scale the rates of molecular traffic between the compartments:

```
exo = moose.element( '/model/kinetics/exo' )
endo = moose.element( '/model/kinetics/endo' )
exo.Kf *= 1.0
endo.Kf *= 1.0
```

- Play with stochasticity. The standard thing here is to scale the volume up and down:

```
compt.volume = 1e-18
compt.volume = 1e-20
compt.volume = 1e-21
```

## Bistability Models

### MAPK feedback loop model

mapkFB.main()

This example illustrates loading, and running a kinetic model for a bistable positive feedback system, defined in kkit format. This is based on Bhalla, Ram and Iyengar, Science 2002.

The core of this model is a positive feedback loop comprising of the MAPK cascade, PLA2, and PKC. It receives PDGF and Ca<sup>2+</sup> as inputs.

This model is quite a large one and due to some stiffness in its equations, it runs somewhat slowly.

The simulation illustrated here shows how the model starts out in a state of low activity. It is induced to ‘turn on’ when a PDGF stimulus is given for 400 seconds. After it has settled to the new ‘on’ state, model is made to ‘turn off’ by setting the system calcium levels to zero for a while. This is a somewhat unphysiological manipulation!

### Simple minimal bistable model

`scaleVolumes.main()`

This example illustrates how to run a model at different volumes. The key line is just to set the volume of the compartment:

```
compt.volume = vol
```

If everything else is set up correctly, then this change propagates through to all reactions molecules.

For a deterministic reaction one would not see any change in output concentrations. For a stochastic reaction illustrated here, one sees the level of ‘noise’ changing, even though the concentrations are similar up to a point. This example creates a bistable model having two enzymes and a reaction. One of the enzymes is autocatalytic. This model is set up within the script rather than using an external file. The model is set up to run using the GSSA (Gillespie Stochastic systems algorithm) method in MOOSE.

To run the example, run the script

```
python scaleVolumes.py
```

and hit `enter` every cycle to see the outcome of stochastic calculations at ever smaller volumes, keeping concentrations the same.

### Strongly bistable Model

`strongBis.main()`

This example illustrates loading, and running a kinetic model for a bistable system, defined in kkit format. Defaults to the deterministic `gsl` method, you can pick the stochastic one by

```
python filename gssa
```

The model starts out equally poised between sides **b** and **c**. Then there is a small molecular ‘tap’ to push it over to **b**. Then we apply a moderate push to show that it is now very stably in this state. It takes a strong push to take it over to **c**. Then it takes a strong push to take it back to **b**. This is a good model to use as the basis for running stochastically and examining how state stability is affected by changing volume.

### Model of bidirectional synaptic plasticity

[showing bistable chemical switch]

`bidirectionalPlasticity.main()`

This is a toy model of synaptic bidirectional plasticity. The model has a small a bistable chemical switch, and a small set of reactions that decode calcium input. One can turn the switch on with short high calcium pulses (over 2  $\mu\text{M}$  for about 10 sec). One can turn it back off again using a long, lower calcium pulse (0.2  $\mu\text{M}$ , 2000 sec).

## Reaction Diffusion Models

The MOOSE design for reaction-diffusion is to specify one or more cellular ‘compartments’, and embed reaction systems in each of them.

A ‘compartment’, in the context of reaction-diffusion, is used in the cellular sense of a biochemically defined, volume restricted subpart of a cell. Many but not all compartments are bounded by a cell membrane, but biochemically the membrane itself may form a compartment. Note that this interpretation differs from that of a ‘compartment’ in detailed electrical models of neurons.

A reaction system can be loaded in from any of the supported MOOSE formats, or built within Python from MOOSE parts.

The computations for such models are done by a set of objects: Stoich, Ksolve and Dsolve. Respectively, these handle the model reactions and stoichiometry matrix, the reaction computations for each voxel, and the diffusion between voxels. The ‘Compartment’ specifies how the model should be spatially discretized.

[Reaction-diffusion + transport in a tapering cylinder]

`cylinderDiffusion.main()`

This example illustrates how to set up a diffusion/transport model with a simple reaction-diffusion system in a tapering cylinder:

Molecule **a** diffuses with `diffConst` of  $10e-12$  m<sup>2</sup>/s.

Molecule **b** diffuses with `diffConst` of  $5e-12$  m<sup>2</sup>/s.

Molecule **b** also undergoes motor transport with a rate of  $10e-6$  m/s

Thus it ‘piles up’ at the end of the cylinder.

Molecule **c** does not move: `diffConst` = 0.0

Molecule **d** does not move: `diffConst` =  $10.0e-12$  but it is buffered.

Because it is buffered, it is treated as non-diffusing.

All molecules other than **d** start out only in the leftmost (first) voxel, with a concentration of 1 mM. **d** is present throughout at 0.2 mM, except in the last voxel, where it is at 1.0 mM.

The cylinder has a starting radius of 2 microns, and end radius of 1 micron. So when the molecule undergoing motor transport gets to the narrower end, its concentration goes up.

There is a little reaction in all compartments:  $b + d \rightleftharpoons c$

As there is a high concentration of **d** in the last compartment, when the molecule **b** reaches the end of the cylinder, the reaction produces lots of **c**.

Note that molecule **a** does not participate in this reaction.

The concentrations of all molecules are displayed in an animation.

`cylinderMotor.main()`

This example illustrates how to set up a transport model with four non-reacting molecules in a cylinder. Molecule **a** and **b** have a positive `motorConst` so they are transported from soma (voxel 0) to the end of the cylinder. Molecules **c** and **d** have a negative `motorConst` so they are transported from the end of the cylinder to the soma. Rate of all motors is  $1e-6$  microns/sec. Pools **a** and **c** start out with all molecules at the soma, **b** and **d** start with all molecules at the end of the cylinder. Net effect is that only molecules **a** and **d** actually move. **B** and **c** stay put as their motors are pushing further toward their respective ends, and I assume all cells have sealed ends.

`gssaCylinderDiffusion.main()`

This example illustrates how to set up a diffusion/transport model with a simple reaction-diffusion system in a tapering cylinder:

Molecule **a** diffuses with diffConst of  $10e-12$  m<sup>2</sup>/s.

Molecule **b** diffuses with diffConst of  $5e-12$  m<sup>2</sup>/s.

Molecule **b** also undergoes motor transport with a rate of  $10e-6$  m/s

Thus it ‘piles up’ at the end of the cylinder.

Molecule **c** does not move: diffConst = 0.0

Molecule **d** does not move: diffConst =  $10.0e-12$  but it is buffered.

Because it is buffered, it is treated as non-diffusing.

All molecules other than **d** start out only in the leftmost (first) voxel, with a concentration of 1 mM. **d** is present throughout at 0.2 mM, except in the last voxel, where it is at 1.0 mM.

The cylinder has a starting radius of 2 microns, and end radius of 1 micron. So when the molecule undergoing motor transport gets to the narrower end, its concentration goes up.

There is a little reaction in all compartments:  $b + d \rightleftharpoons c$

As there is a high concentration of **d** in the last compartment, when the molecule **b** reaches the end of the cylinder, the reaction produces lots of **c**.

Note that molecule **a** does not participate in this reaction.

The concentrations of all molecules are displayed in an animation.

### Neuronal Diffusion Reaction

`rxdFuncDiffusion.main()`

This example implements a reaction-diffusion like system which is bistable and propagates losslessly. It is based on the NEURON example `rxdrun.py`, but incorporates more compartments and runs for a longer time. The system is implemented in a function rather than as a proper system of chemical reactions. Please see `rxdReacDiffusion.py` for a variant that uses a reaction plus a function object to control its rates.

`rxdReacDiffusion.main()`

This example implements a reaction-diffusion like system which is bistable and propagates losslessly. It is based on the NEURON example `rxdrun.py`, but incorporates more compartments and runs for a longer time. The system is implemented as a hybrid of a reaction and a function which sets its rates. Please see `rxdFuncDiffusion.py` for a variant that uses just a function object to set up the system.

`rxdFuncDiffusionStoch.main()`

This example implements a reaction-diffusion like system which is bistable and propagates losslessly. It is based on the NEURON example `rxdrun.py`, but incorporates more compartments and runs for a longer time. The system is implemented in a function rather than as a proper system of chemical reactions. Please see `rxdReacDiffusion.py` for a variant that uses a reaction plus a function object to control its rates.

### A Turing Model

`TuringOneDim.makeModel()`

This example illustrates how to set up a oscillatory Turing pattern in 1-D using reaction diffusion calculations. Reaction system is:

```
s ---a----> a // s goes to a, catalyzed by a.
s ---a----> b // s goes to b, catalyzed by a.
a ---b----> s // a goes to s, catalyzed by b.
b -----> s // b is degraded irreversibly to s.
```

in sum, **a** has a positive feedback onto itself and also forms **b**. **b** has a negative feedback onto **a**. Finally, the diffusion constant for **a** is 1/10 that of **b**.

This chemical system is present in a 1-dimensional (cylindrical) compartment. The entire reaction-diffusion system is set up within the script.

## A Spatial Bistable Model

### Reaction Diffusion in Neurons

`reactDiffConcGradient.main()`

This example shows how to maintain a conc gradient against diffusion

```

compt0      compt1      compt 2
a ..... a ..... a      [Diffusion between compts]
|\          |\          |
↔[Reacs within compts]
\|          \|          \|
b0 <----->b1 <----->b2      [Reacs between compts]
4x          2x          1x      [Ratios of vols of compts]

```

If there is no diffusion then the ratio of concs should be 1:10:100 If there is no x-compt reac, then clearly the concs should all be the same, in this case they should be 2.0. If both are happening then the final concs are 1.4, 2.5, 3.4.

`reactDiffBranchingNeuron.main()`

This example illustrates how to define a kinetic model embedded in the branching pseudo 1-dimensional geometry of a neuron. This means that diffusion only happens along the axis of dendritic segments, not radially from inside to outside a dendrite, nor tangentially around the dendrite circumference. The model oscillates in space and time due to a Turing-like reaction-diffusion mechanism present in all compartments. For the sake of this demo, the initial conditions are set to be slightly different on one of the terminal dendrites, so as to break the symmetry and initiate oscillations. This example uses an external model file to specify a binary branching neuron. This model does not have any spines. The electrical model is used here purely for the geometry and is not part of the computations. In this example we build an identical chemical model throughout the neuronal geometry, using the `makeChemModel` function. The model is set up to run using the `Ksolve` for integration and the `Dsolve` for handling diffusion.

The display has two parts:

1. Animated pseudo-3D plot of neuronal geometry, where each point represents a diffusive voxel and moves in the y-axis to show changes in concentration.
2. Time-series plot that appears after the simulation has ended. The plots are for the first and last diffusive voxel, that is, the soma and the tip of one of the apical dendrites.

`reactDiffBranchingNeuron.makeChemModel (compt)`

This function sets up a simple oscillatory chemical system within the script. The reaction system is:

```

s ---a----> a // s goes to a, catalyzed by a.
s ---a----> b // s goes to b, catalyzed by a.
a ---b----> s // a goes to s, catalyzed by b.
b -----> s // b is degraded irreversibly to s.

```

in sum, **a** has a positive feedback onto itself and also forms **b**. **b** has a negative feedback onto **a**. Finally, the diffusion constant for **a** is 1/10 that of **b**.

```
reacDiffSpinyNeuron.main()
```

This example illustrates how to define a kinetic model embedded in the branching pseudo-1-dimensional geometry of a neuron. The model oscillates in space and time due to a Turing-like reaction-diffusion mechanism present in all compartments. For the sake of this demo, the initial conditions are set up slightly different on the PSD compartments, so as to break the symmetry and initiate oscillations in the spines. This example uses an external electrical model file with basal dendrite and three branches on the apical dendrite. One of those branches has a dozen or so spines. In this example we build an identical model in each compartment, using the `makeChemModel` function. One could readily define a system with distinct reactions in each compartment. The model is set up to run using the `Ksolve` for integration and the `Dsolve` for handling diffusion. The display has four parts:

1. animated line plot of concentration against main compartment#.
2. animated line plot of concentration against spine compartment#.
3. animated line plot of concentration against psd compartment#.
4. time-series plot that appears after the simulation has ended. The plot is for the last (rightmost) compartment.

```
reacDiffSpinyNeuron.makeChemModel(compt)
```

This function setups up a simple oscillatory chemical system within the script. The reaction system is:

```
s ---a----> a // s goes to a, catalyzed by a.
s ---a----> b // s goes to b, catalyzed by a.
a ---b----> s // a goes to s, catalyzed by b.
b -----> s // b is degraded irreversibly to s.
```

in sum, **a** has a positive feedback onto itself and also forms **b**. **b** has a negative feedback onto **a**. Finally, the diffusion constant for **a** is 1/10 that of **b**.

```
diffSpinyNeuron.main()
```

This example illustrates and tests diffusion embedded in the branching pseudo-1-dimensional geometry of a neuron. An input pattern of Ca stimulus is applied in a periodic manner both on the dendrite and on the PSDs of the 13 spines. The Ca levels in each of the dend, the spine head, and the spine PSD are monitored. Since the same molecule name is used for Ca in the three compartments, these are automatically connected up for diffusion. The simulation shows the outcome of this diffusion. This example uses an external electrical model file with basal dendrite and three branches on the apical dendrite. One of those branches has the 13 spines. The model is set up to run using the `Ksolve` for integration and the `Dsolve` for handling diffusion. The timesteps here are not the defaults. It turns out that the chem reactions and diffusion in this example are sufficiently fast that the `chemDt` has to be smaller than default. Note that this example uses rates quite close to those used in production models. The display has four parts:

1. animated line plot of concentration against main compartment#.
2. animated line plot of concentration against spine compartment#.
3. animated line plot of concentration against psd compartment#.
4. time-series plot that appears after the simulation has ended.

```
diffSpinyNeuron.makeChemModel(compt, doInput)
```

This function setups up a simple chemical system in which Ca input comes to the dend and to selected PSDs. There is diffusion between PSD and spine head, and between dend and spine head.

```
:: Ca_input —> Ca // in dend and spine head only.
```

## Manipulating Chemical Models

### Running with different numerical methods

`switchKineticSolvers.main()`

At zero order, you can select the solver you want to use within the function `moose.loadModel( filename, model-path, solver )`. Having loaded in the model, you can change the solver to use on it. This example illustrates how to assign and change solvers for a kinetic model. This process is necessary in two situations:

- If we want to change the numerical method employed, for example, from deterministic to stochastic.
- If we are already using a solver, and we have changed the reaction network by adding or removing molecules or reactions.

Note that we do not have to change the solvers if the volume or reaction rates change. In this example the model is loaded in with a `gsl` solver. The sequence of solver calculations is:

1. `gsl`
2. `ee`
3. `gsl`
4. `gssa`
5. `gsl`

If you're removing the solvers, you just delete the stoichiometry object and the associated `ksolve/gsolve`. Should there be diffusion (a `dsolve`) then you should delete that too. If you're building the solvers up again, then you must do the following steps in order:

1. build up the `ksolve/gsolve` and `stoich` (any order)
2. Assign `stoich.ksolve`
3. Assign `stoich.path`.

See the Reaction-diffusion section should you want to do diffusion as well.

### Changing volumes

`scaleVolumes.main()`

This example illustrates how to run a model at different volumes. The key line is just to set the volume of the compartment:

```
compt.volume = vol
```

If everything else is set up correctly, then this change propagates through to all reactions molecules.

For a deterministic reaction one would not see any change in output concentrations. For a stochastic reaction illustrated here, one sees the level of 'noise' changing, even though the concentrations are similar up to a point. This example creates a bistable model having two enzymes and a reaction. One of the enzymes is autocatalytic. This model is set up within the script rather than using an external file. The model is set up to run using the GSSA (Gillespie Stochastic systems algorithm) method in MOOSE.

To run the example, run the script

```
python scaleVolumes.py
```

and hit `enter` every cycle to see the outcome of stochastic calculations at ever smaller volumes, keeping concentrations the same.

## Feeding tabulated input to a model

`analogStimTable.main()`

Example of using a StimulusTable as an analog signal source in a reaction system. It could be used similarly to give other analog inputs to a model, such as a current or voltage clamp signal.

This demo creates a StimulusTable and assigns it half a sine wave. Then we assign the start time and period over which to emit the wave. The output of the StimTable is sent to a pool **a**, which participates in a trivial reaction:

```
table ----> a <====> b
```

The output of **a** and **b** are recorded in a regular table for plotting.

## Finding steady states

`findChemSteadyState.getState(ksolve, state)`

This function finds a steady state starting from a random initial condition that is consistent with the stoichiometry rules and the original model concentrations.

`findChemSteadyState.main()`

This example sets up the kinetic solver and steady-state finder, on a bistable model of a chemical system. The model is set up within the script. The algorithm calls the steady-state finder 50 times with different (randomized) initial conditions, as follows:

- Set up the random initial condition that fits the conservation laws
- Run for 2 seconds. This should not be mathematically necessary, but for obscure numerical reasons it makes it much more likely that the steady state solver will succeed in finding a state.
- Find the fixed point
- Print out the fixed point vector and various diagnostics.
- Run for 10 seconds. This is completely unnecessary, and is done here just so that the resultant graph will show what kind of state has been found.

After it does all this, the program runs for 100 more seconds on the last found fixed point (which turns out to be a saddle node), then is hard-switched in the script to the first attractor basin from which it runs for another 100 seconds till it settles there, and then is hard-switched yet again to the second attractor and runs for 400 seconds.

Looking at the output you will see many features of note:

- the first attractor (stable point) and the saddle point (unstable fixed point) are both found quite often. But the second attractor is found just once. It has a very small basin of attraction.
- The values found for each of the fixed points match well with the values found by running the system to steady-state at the end.
- There are a large number of failures to find a fixed point. These are found and reported in the diagnostics. They show up on the plot as cases where the 10-second runs are not flat.

If you wanted to find fixed points in a production model, you would not need to do the 10-second runs, and you would need to eliminate the cases where the state-finder failed. Then you could identify the good points and keep track of how many of each were found.

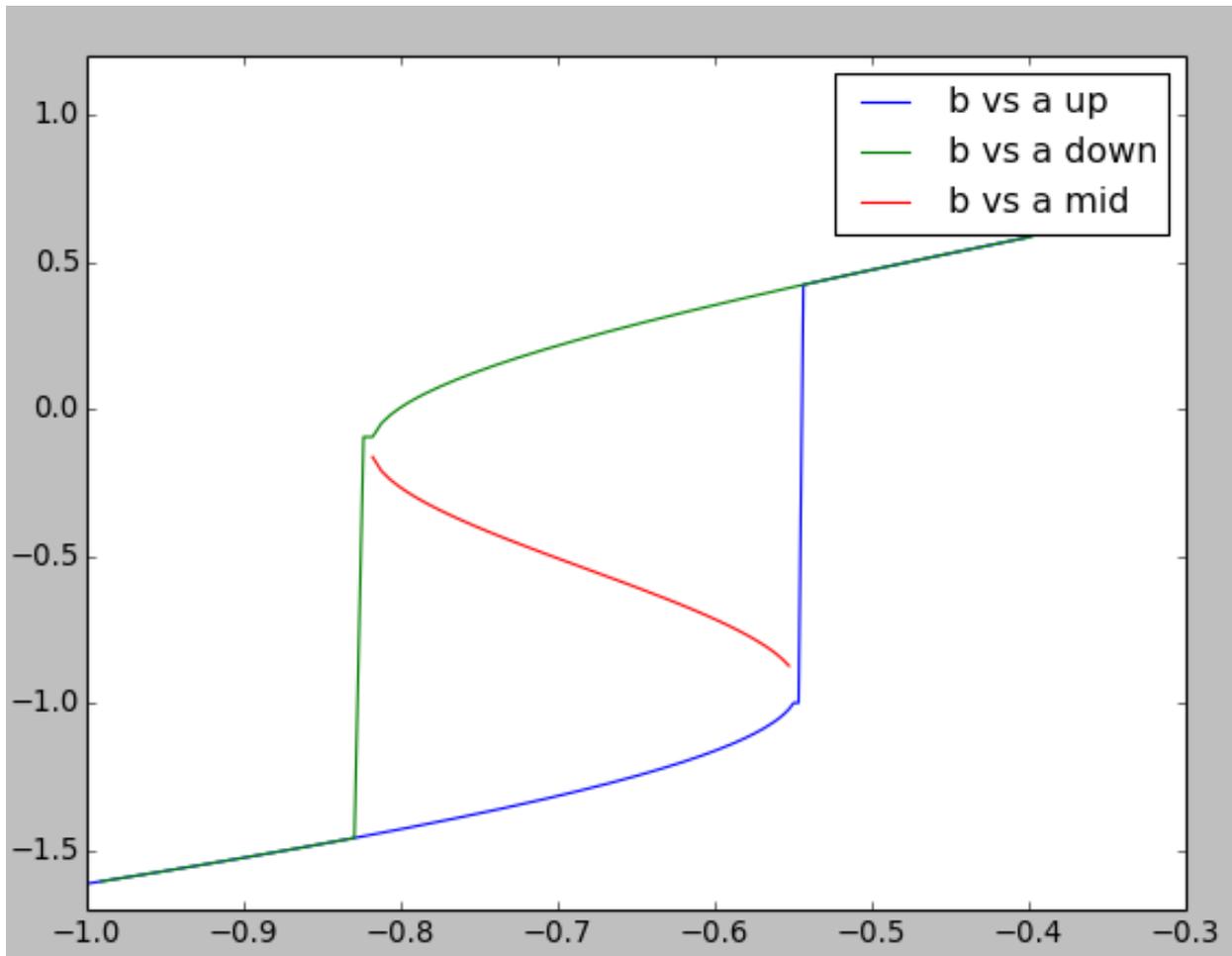
There is no way to guarantee that all fixed points have been found using this algorithm! If there are points in an obscure corner of state space (as for the singleton second attractor convergence in this example) you may have to iterate very many times to find them.

You may wish to sample concentration space logarithmically rather than linearly.

```
findChemSteadyState.makeModel()
```

This function creates a bistable reaction system using explicit MOOSE calls rather than load from a file

### Making a dose-response curve



```
chemDoseResponse.main()
```

This example builds a dose-response of a bistable model of a chemical system. It uses the kinetic solver *Ksolve* and the steady-state finder *SteadyState*. The model is set up within the script.

The basic approach is to increment the control variable, **a** in this case, while monitoring **b**. The algorithm marches through a series of values of the buffered pool **a** and measures resultant values of pool **b**. At each cycle the algorithm calls the steady-state finder. Since **a** is incremented only a small amount on each step, each new steady state is (usually) quite close to the previous one. The exception is when there is a state transition.

Here we plot three dose-response curves to illustrate the bistable nature of the system.

On the upward going curve in blue, **a** starts low. Here, **b** follows the low arm of the curve and then jumps up to the high value at roughly  $\log([a]) = -0.55$ .

On the downward going curve in green, **b** follows the high arm of the curve forming a nice hysteretic loop. Eventually **b** has to fall to the low state at about  $\log([a]) = -0.83$

Through nasty concentration manipulations, we find the third arm of the curve, which tracks the unstable fixed point. This is in red. We find this arm by setting an initial point close to the unstable fixed point, which the

steady-state finder duly locates. We then follow a dose-response curve as with the other arms of the curve.

Note that the steady-state solver doesn't always succeed in finding a good solution, despite moving only in small steps. Nevertheless the resultant curves are smooth because it gives up pretty close to the correct value, simply because the successive points are close together. Overall, the system is pretty robust despite the core root-finder computations in GSL being temperamental.

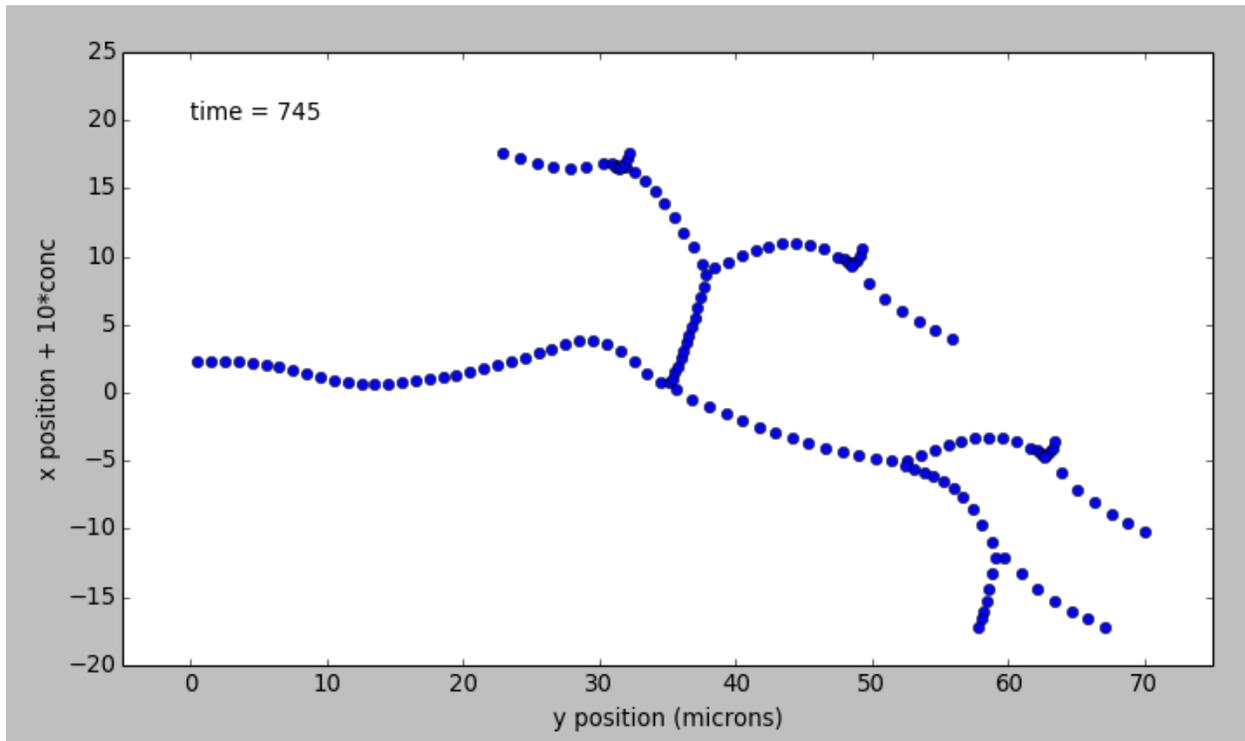
In doing a production dose-response series you may wish to sample concentration space logarithmically rather than linearly.

`chemDoseResponse.makeModel()`

This function creates a bistable reaction system using explicit MOOSE calls rather than load from a file. The reaction is:

```
a ---b----> 2b    # b catalyzes a to form more of b.
2b ---c----> a    # c catalyzes b to form a.
a <=====> 2b     # a interconverts to b.
```

### Transport in branching dendritic tree



`transportBranchingNeuron.main()`

`transportBranchingNeuron`: This example illustrates bidirectional transport embedded in the branching pseudo 1-dimensional geometry of a neuron. This means that diffusion and transport only happen along the axis of dendritic segments, not radially from inside to outside a dendrite, nor tangentially around the dendrite circumference. In this model there is a molecule **a** starting at the soma, which is transported out to the dendrites. There is another molecule, **b**, which is initially present at the dendrite tips, and is transported toward the soma. This example uses an external model file to specify a binary branching neuron. This model does not have any spines. The electrical model is used here purely for the geometry and is not part of the computations. In this example we build trival chemical model just having molecules **a** and **b** throughout the neuronal geometry, using

the `makeChemModel` function. The model is set up to run using the `Ksolve` for integration and the `Dsolve` for handling diffusion.

The display has three parts:

1. Animated pseudo-3D plot of neuronal geometry, where each point represents a diffusive voxel and moves in the y-axis to show changes in concentration of molecule a.
2. Similar animated pseudo-3D plot for molecule b.
3. Time-series plot that appears after the simulation has ended. The plots are for the first and last diffusive voxel, that is, the soma and the tip of one of the apical dendrites.

## Tutorials

### Finding Steady State (Cspace)

`cspaceSteadyState.main()`

This example sets up the kinetic solver and steady-state finder, on a bistable model. It looks for the fixed points 100 times, as follows: - Set up the random initial condition that fits the conservation laws - Run for 2 seconds. This should not be mathematically necessary, but

for obscure numerical reasons it makes it much more likely that the steady state solver will succeed in finding a state.

- Find the fixed point
- Print out the fixed point vector and various diagnostics.
- **Run for 10 seconds. This is completely unnecessary, and is done here**

just so that the resultant graph will show what kind of state has been found.

After it does all this, the program runs for 100 more seconds on the last found fixed point (which turns out to be a saddle node), then is hard-switched in the script to the first attractor basin from which it runs for another 100 seconds till it settles there, and then is hard-switched yet again to the second attractor and runs for 100 seconds. Looking at the output you will see many features of note: - the first attractor (stable point) and the saddle point

(unstable fixed point) are both found quite often. But the second attractor is found just once. Has a very small basin of attraction.

- **The values found for each of the fixed points match well with the** values found by running the system to steady-state at the end.
- **There are a large number of failures to find a fixed point. These are** found and reported in the diagnostics. They show up on the plot as cases where the 10-second runs are not flat.

If you wanted to find fixed points in a production model, you would not need to do the 10-second runs, and you would need to eliminate the cases where the state-finder failed. Then you could identify the good points and keep track of how many of each were found. There is no way to guarantee that all fixed points have been found using this algorithm! You may wish to sample concentration space logarithmically rather than linearly.

## Building Simple Reaction Model

### Define a kinetic model using the scripting

```
scriptKineticModel.main()
```

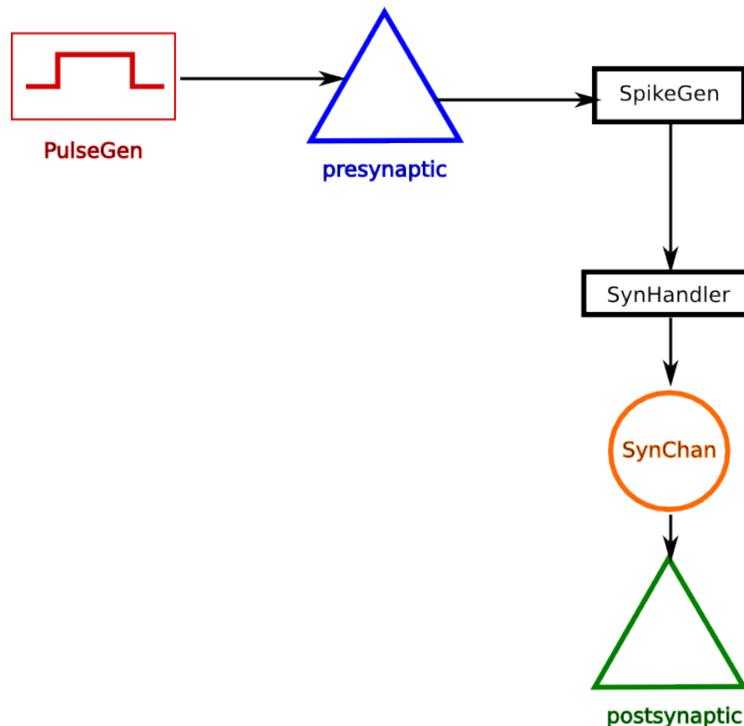
This example illustrates how to define a kinetic model using the scripting interface. Normally one uses standard model formats like SBML or kkit to concisely define kinetic models, but in some cases one would like to modify the model through the script. This example creates a bistable model having two enzymes and a reaction. One of the enzymes is autocatalytic. The model is set up to run using default Exponential Euler integration. The snippet `scriptKineticSolver.py` uses the much better GSL Runge-Kutta-Fehlberg integration scheme on this same model.

## 1.3.3 Networking

### Simple Examples

#### Connecting two cells via a synapse

Below is the connectivity diagram for setting up a synaptic connection from one neuron to another. The `PulseGen` object is there for stimulating the presynaptic cell as part of experimental setup. The cells are defined as single-compartments with Hodgkin-Huxley type Na<sup>+</sup> and K<sup>+</sup> channels.



```
twocells.create_model()
```

Create two single compartmental neurons, `neuron_A` is the presynaptic neuron and `neuron_B` is the postsynaptic neuron.

1. The presynaptic cell's Vm is monitored by a `SpikeGen` object. Whenever the Vm crosses the threshold of the spikegen, it sends out a spike event message.
2. This event message is received by a `SynHandler`, which passes the event as activation parameter to a `SynChan` object.

3. The SynChan, which is connected to the postsynaptic neuron as a channel, updates its conductance based on the activation parameter.

4. The change in conductance due to a spike may evoke an action potential in the post synaptic neuron.

`twocells.main()`

A demo to create a network of single compartmental neurons connected via alpha synapses. Here SynChan class is used to setup synaptic connection between two single-compartmental Hodgkin-Huxley type neurons.

`twocells.setup_experiment(presynaptic, postsynaptic, synchan)`

Setup step current stimulation of presynaptic neuron. Also setup recording of pre and postsynaptic Vm, Gk of synchan.

## Multi Compartmental Leaky Neurons

`multicomp_lif.main()`

This is an example of how you can create a Leaky Integrate and Fire compartment using regular compartment and Func to check for threshold crossing and resetting the Vm.

`multicomp_lif.setup_two_cells()`

Create two cells with leaky integrate and fire compartments. The first cell is composed of two compartments a1 and a2 and the second cell is composed of compartments b1 and b2. Each pair is connected via raxial message so that the voltage of one compartment influences the other through axial resistance Ra.

The compartment a1 of the first neuron is connected to the compartment b2 of the second neuron through a synaptic channel.

## Providing random input to a cell

`randomspike.create_cell()`

Create a single-compartment Hodgkin-Huxley neuron with a synaptic channel.

This uses the `ionchannel.create_1comp_neuron()` function for model creation.

Returns a dict containing the neuron, the synchan and the synhandler for accessing the synapse,

`randomspike.example()`

The RandSpike class generates spike events from a Poisson process and sends out a trigger via its *spikeOut* message. It is very common to approximate the spiking in many neurons as a Poisson process, i.e., the probability of  $k$  spikes in any interval  $t$  is given by the Poisson distribution:

$$\exp(-ut)(ut)^k/k!$$

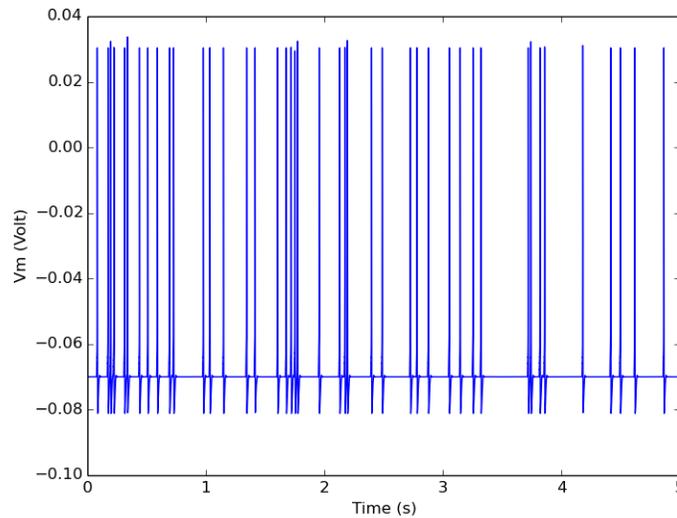
for  $k = 0, 1, 2, \dots$   $u$  is the rate of spiking (the mean of the Poisson distribution). See [wikipedia](#) for details.

Many cortical neuron types spontaneously fire action potentials. These are called ectopic spikes. In this example we simulate this with a RandSpike object with rate 10 spikes/s and send this to a single compartmental neuron via a synapse.

In this model the synaptic conductance is set so high that each incoming spike evokes an action potential.

`randomspike.main()`

This is an example of simulating random events from a Poisson process and applying the event as spike input to a single-compartmental Hodgkin-Huxley type neuron model.



## Plastic synapse

`STDP.main()`

Connect two cells via a plastic synapse (STDPSynHandler). Induce spikes separated by varying intervals, in the pre and post synaptic cells. Plot the synaptic weight change for different intervals between the spike-pairs. This is a pseudo-STDP protocol and we get the STDP rule.

`STDP.make_neuron_spike(nrnidx, I=1e-07, duration=0.001)`

Inject a brief current pulse to make a neuron spike

`STDP.reset_settle()`

Call this between every pre-post pair to reset the neurons and make them settle to rest.

`STDP.setupModel()`

Set up two LIF neurons and connect them by an STDPSynHandler. Set up some tables, and reinit MOOSE before simulation.

## Synapse Handler for Spikes

`RandSpikeStats.main()`

This snippet shows the use of several objects. This snippet sets up a StimulusTable to control a RandSpike which sends its outputs to two places: to a SimpleSynHandler on an IntFire, which is used to monitor spike arrival, and to various Stats objects. Each of these are recorded and plotted. The StimulusTable has a sine-wave waveform.

## Recurrent integrate-and-fire network

`recurrentIntFire.main()`

This snippet sets up a recurrent network of IntFire objects, using SimpleSynHandlers to deal with spiking events. It isn't very satisfactory as activity runs down after a while. It is a good example for using the IntFire, setting up random connectivity, and using SynHandlers.

## Recurrent integrate-and-fire network with plasticity

`recurrentLIF.main()`

This snippet sets up a recurrent network of LIF objects, using SimpleSynHandlers to deal with spiking events. It isn't very satisfactory as activity runs down after a while. It is a good example for using the LIF, setting up random connectivity, and using SynHandlers.

## Demonstration Models

`compartment_net.create_network(size=2)`

Create a network containing two neuronal populations, `pop_A` and `pop_B` and connect them up.

`compartment_net.create_population(container, size)`

Create a population of `size` single compartmental neurons with Na+ and K+ channels. Also create SpikeGen objects and SynChan objects connected to these which can act as plug points for setting up synapses later.

This uses `..ref::ionchannel.create_1comp_neuron`.

`compartment_net.main()`

This example illustrates how to create a network of single compartmental neurons connected via alpha synapses. It also shows the use of SynChan class to create a network of single-compartment neurons connected by synapse.

`compartment_net.make_synapses(spikegen, synhandler, connprob=1.0, delay=0.005)`

Create synapses from spikegen array to synchan array.

**spikegen** [vec of SpikGen elements] Spike generators from neurons.

**synhandler** [vec of SynHandler elements] Handles presynaptic spike event inputs to synchans.

**connprob: float in range (0, 1]** connection probability between any two neurons

**delay: float** mean delay of synaptic transmission. Individual delays are normally distributed with  $sd=0.1*mean$ .

`compartment_net_no_array.assign_clocks(model_container_list, simdt, plotdt)`

Assign clocks to elements under the listed paths.

This should be called only after all model components have been created. Anything created after this will not be scheduled.

`compartment_net_no_array.create_population(container, size)`

Create a population of `size` single compartmental neurons with Na and K channels. Also create SpikeGen objects and SynChan objects connected to these which can act as plug points for setting up synapses later.

`compartment_net_no_array.main()`

A demo to create a network of single compartmental neurons connected via alpha synapses. This is same as `compartment_net.py` except that we avoid ematrix and use single melements.

`compartment_net_no_array.make_synapses(spikegen, synchan, delay=0.005)`

Create synapses from spikegens to synchans in a manner similar to OneToAll connection.

**spikegen:** list of spikegen objects - these are sources of synaptic event messages.

**synchan:** list of synchan objects - these are the targets of the synaptic event messages.

**delay:** mean delay of synaptic transmission. Individual delays are normally distributed with  $sd=0.1*mean$ .

`compartment_net_no_array.single_population(size=2)`

Example of a single population where each cell is connected to every other cell.

Creates a network of single compartmental cells under `/network1` and a pulse generaor

`compartment_net_no_array.two_populations (size=2)`  
An example with two population connected via synapses.

### Building Models

`synapse_tutorial.main()`

In this example we walk through creation of a vector of IntFire elements and setting up synaptic connection between them. Synapse on IntFire elements is an example of ElementField - elements that do not exist on their own, but only as part of another element. This example also illustrates various operations on *vec* objects and ElementFields.

### Tutorials

#### Network with Ca-based plasticity

`GraupnerBrunel2012_STDPfromCaPlasticity.main()`

Simulate a pseudo-STDP protocol and plot the STDP kernel that emerges from Ca plasticity of Graupner and Brunel 2012. Author: Aditya Gilra, NCBS, Bangalore, October, 2014.

`GraupnerBrunel2012_STDPfromCaPlasticity.make_neuron_spike (nrnid, I=1e-07, duration=0.001)`

Inject a brief current pulse to make a neuron spike

`GraupnerBrunel2012_STDPfromCaPlasticity.reset_settle()`

Call this between every pre-post pair to reset the neurons and make them settle to rest.

`HigginsGraupnerBrunel2014_LifetimeCaPlasticity.main()`

Simulate pre and post Poisson firing for a synapse with Ca plasticity of Graupner and Brunel 2012. See the trace over time (lifetime) for the synaptic efficacy, similar to figure 2A of Higgins, Graupner, Brunel, 2014.

Author: Aditya Gilra, NCBS, Bangalore, October, 2014.

## 1.3.4 MultiScale Modeling

### Simple Examples

#### Single-compartment multiscale model

`multiscaleOneCompt.main()`

This example builds a simple multiscale model involving electrical and chemical signaling, but without spatial dimensions. The electrical cell model is in a single compartment and has voltage-gated channels, including a voltage-gated Ca channel for Ca influx, and a K\_A channel which is regulated by the chemical pathways.

The chemical model has calcium activating Calmodulin which activates CaM-Kinase II. The kinase phosphorylates the K\_A channel to inactivate it.

The net effect of the multiscale activity is a positive feedback loop where activity increases Ca, which activates the kinase, which reduces K\_A, leading to increased excitability of the cell.

In this example this results in a bistable neuron. In the resting state the cell does not fire, but if it is activated by a current pulse the cell will continue to fire even after the current is turned off. Application of an inhibitory current restores the cell to its silent state.

Both the electrical and chemical models are loaded in from model description files, and these files could be replaced if one wished to define different models. However, there are model-specific Adaptor objects needed to

map activity between the models of the two kinds. The Adaptors connect specific model entities between the two models. Here one Adaptor connects the electrical `Ca_conc` object to the chemical `Ca` pool. The other Adaptor connects the chemical pool representing the `K_A` channel to its conductance term in the electrical model.

## Multi compartment Single Neuron Model

`multiComptSigNeur.createSpine` (*parentCompt, parentObj, index, frac, length, dia, theta*)

Create spine of specified dimensions and index

`multiComptSigNeur.createSquid` ()

Create a single compartment squid model.

`multiComptSigNeur.main` ()

A toy compartmental neuronal + chemical model. The neuronal model is in a dendrite and five dendritic spines. The chemical model is in three compartments: one for the dendrite, one for the spine head, and one for the postsynaptic density. However, the spatial geometry of the neuronal model is ignored and the chemical model just has three cubic volumes for each compartment. So there is a functional mapping but spatial considerations are lost. The electrical model contributes the incoming calcium flux to the chemical model. This comes from the synaptic channels. The signalling here does two things to the electrical model. First, the amount of receptor in the chemical model controls the amount of glutamate receptor in the PSD. Second, there is a small kinase reaction that phosphorylates and inactivates the dendritic potassium channel.

## Multi-compartment multiscale model

### Modeling chemical reactions in neurons

`gssaRDspiny.main` ()

This example illustrates how to define a kinetic model embedded in the branching pseudo-1-dimensional geometry of a neuron. The model oscillates in space and time due to a Turing-like reaction-diffusion mechanism present in all compartments. For the sake of this demo, the initial conditions are set up slightly different on the PSD compartments, so as to break the symmetry and initiate oscillations in the spines. This example uses an external electrical model file with basal dendrite and three branches on the apical dendrite. One of those branches has a dozen or so spines. In this example we build an identical model in each compartment, using the `makeChemModel` function. One could readily define a system with distinct reactions in each compartment. The model is set up to run using the `Ksolve` for integration and the `Dsolve` for handling diffusion. The display has four parts:

1. animated line plot of concentration against main compartment#.
2. animated line plot of concentration against spine compartment#.
3. animated line plot of concentration against psd compartment#.
4. time-series plot that appears after the simulation has ended. The plot is for the last (rightmost) compartment.

`gssaRDspiny.makeChemModel` (*compt*)

This function setus up a simple oscillatory chemical system within the script. The reaction system is:

```
s ---a----> a // s goes to a, catalyzed by a.
s ---a----> b // s goes to b, catalyzed by a.
a ---b----> s // a goes to s, catalyzed by b.
b -----> s // b is degraded irreversibly to s.
```

in sum, **a** has a positive feedback onto itself and also forms **b**. **b** has a negative feedback onto **a**. Finally, the diffusion constant for **a** is 1/10 that of **b**.

## RDesignneur

### Building Chemical-Electrical Signalling Models using Rdesignneur

#### Building a compartment

#### Inserting Spines and viewing

```
insertSpines.main()
```

This example illustrates loading a model from an SWC file, inserting spines, and viewing it.

#### Proceeding with Spines

## 1.4 Graphics

### 1.4.1 MOGLI

#### Use Moogli for plotting

### 1.4.2 Matplotlib

#### Displaying time-series plots

```
crossComptNeuroMesh.main()
```

This example illustrates how to define a kinetic model embedded in a NeuroMesh, and undergoing cross-compartment reactions. It is completely self-contained and does not use any external model definition files. Normally one uses standard model formats like SBML or kkit to concisely define kinetic and neuronal models. This example creates a simple reaction:

```
a <==> b <==> c
```

in which

**a**, **b**, and **c** are in the dendrite, spine head, and PSD respectively. The model is set up to run using the Ksolve for integration. Although a diffusion solver is set up, the diff consts here are set to zero. The display has two parts: Above is a line plot of concentration against compartment#. Below is a time-series plot that appears after # the simulation has ended. The plot is for the last (rightmost) compartment. Concs of **a**, **b**, **c** are plotted for both graphs.

#### Animation of values along axis

```
diffSpinyNeuron.main()
```

This example illustrates and tests diffusion embedded in the branching pseudo-1-dimensional geometry of a neuron. An input pattern of Ca stimulus is applied in a periodic manner both on the dendrite and on the PSDs of the 13 spines. The Ca levels in each of the dend, the spine head, and the spine PSD are monitored. Since the same molecule name is used for Ca in the three compartments, these are automagially connected up for diffusion. The simulation shows the outcome of this diffusion. This example uses an external electrical model file with basal dendrite and three branches on the apical dendrite. One of those branches has the 13 spines. The model is set up to run using the Ksolve for integration and the Dsolve for handling diffusion. The timesteps here are not the defaults. It turns out that the chem reactions and diffusion in this example are sufficiently fast that the

chemDt has to be smaller than default. Note that this example uses rates quite close to those used in production models. The display has four parts:

1. animated line plot of concentration against main compartment#.
2. animated line plot of concentration against spine compartment#.
3. animated line plot of concentration against psd compartment#.
4. time-series plot that appears after the simulation has ended.

`diffSpinyNeuron.makeChemModel (compt, doInput)`

This function setups up a simple chemical system in which Ca input comes to the dend and to selected PSDs. There is diffusion between PSD and spine head, and between dend and spine head.

:: Ca\_input —> Ca // in dend and spine head only.

`reacDiffBranchingNeuron.main ()`

This example illustrates how to define a kinetic model embedded in the branching pseudo 1-dimensional geometry of a neuron. This means that diffusion only happens along the axis of dendritic segments, not radially from inside to outside a dendrite, nor tangentially around the dendrite circumference. The model oscillates in space and time due to a Turing-like reaction-diffusion mechanism present in all compartments. For the sake of this demo, the initial conditions are set to be slightly different on one of the terminal dendrites, so as to break the symmetry and initiate oscillations. This example uses an external model file to specify a binary branching neuron. This model does not have any spines. The electrical model is used here purely for the geometry and is not part of the computations. In this example we build an identical chemical model throughout the neuronal geometry, using the `makeChemModel` function. The model is set up to run using the `Ksolve` for integration and the `Dsolve` for handling diffusion.

The display has two parts:

1. Animated pseudo-3D plot of neuronal geometry, where each point represents a diffusive voxel and moves in the y-axis to show changes in concentration.
2. Time-series plot that appears after the simulation has ended. The plots are for the first and last diffusive voxel, that is, the soma and the tip of one of the apical dendrites.

`reacDiffBranchingNeuron.makeChemModel (compt)`

This function sets up a simple oscillatory chemical system within the script. The reaction system is:

```
s ---a----> a // s goes to a, catalyzed by a.
s ---a----> b // s goes to b, catalyzed by a.
a ---b----> s // a goes to s, catalyzed by b.
b -----> s // b is degraded irreversibly to s.
```

in sum, **a** has a positive feedback onto itself and also forms **b**. **b** has a negative feedback onto **a**. Finally, the diffusion constant for **a** is 1/10 that of **b**.

## 1.5 References

### 1.5.1 How to use the documentation

MOOSE documentation is split into Python documentation and builtin documentation. The functions and classes that are only part of the Python interface can be viewed via Python's builtin `help` function:

```
>>> help(moose.connect)
```

The documentation built into main C++ code of MOOSE can be accessed via the module function `doc`:

```
>>> moose.doc('Neutral')
```

To get documentation about a particular field:

```
>>> moose.doc('Neutral.childMsg')
```

## MOOSE Functions

### element

`moose.element(arg)` -> moose object

Convert a path or an object to the appropriate builtin moose class instance.

**arg** [str/vec/moose object] path of the moose element to be converted or another element (possibly available as a superclass instance).

**Returns - melement** MOOSE element (object) corresponding to the *arg* converted to write subclass.

### getFieldNames

`moose.getFieldNames(className, finfoType='valueFinfo')` -> tuple

Get a tuple containing the name of all the fields of *finfoType* kind.

**className** [string] Name of the class to look up.

**finfoType** [string] The kind of field - *valueFinfo* - *srcFinfo* - *destFinfo* - *lookupFinfo*- *fieldElementFinfo* -

**Returns - tuple** Names of the fields of type *finfoType* in class *className*.

### copy

`moose.copy(src, dest, name, n, toGlobal, copyExtMsg)` -> bool

Make copies of a moose object.

**src** [vec, element or str] source object.

**dest** [vec, element or str] Destination object to copy into.

**name** [str] Name of the new object. If omitted, name of the original will be used.

**n** [int] Number of copies to make.

**toGlobal** [int] Relevant for parallel environments only. If false, the copies will reside on local node, otherwise all nodes get the copies.

**copyExtMsg** [int] If true, messages to/from external objects are also copied.

**Returns - vec** newly copied vec

### move

`moose.move(...)` Move a vec object to a destination.

## delete

**moose.delete(...)** delete(obj)->None

Delete the underlying moose object. This does not delete any of the Python objects referring to this vec but does invalidate them. Any attempt to access them will raise a ValueError.

*id* [vec] vec of the object to be deleted.

Returns - None

## useClock

**moose.useClock(tick, path, fn)**

schedule *fn* function of every object that matches *path* on tick no. *tick*.

Most commonly the function is 'process'. NOTE: unlike earlier versions, now autoschedule is not available. You have to call useClock for every element that should be updated during the simulation.

The sequence of clockticks with the same dt is according to their number. This is utilized for controlling the order of updates in various objects where it matters. The following convention should be observed when assigning clockticks to various components of a model:

Clock ticks 0-3 are for electrical (biophysical) components, 4 and 5 are for chemical kinetics, 6 and 7 are for lookup tables and stimulus, 8 and 9 are for recording tables.

Generally, *process* is the method to be assigned a clock tick. Notable exception is *init* method of Compartment class which is assigned tick 0.

- 0 : Compartment: *init*
- 1 : Compartment: *process*
- 2 : HHChannel and other channels: *process*
- 3 : CaConc : *process*
- 4,5 : Elements for chemical kinetics : *process*
- 6,7 : Lookup (tables), stimulus : *process*
- 8,9 : Tables for plotting : *process*

*tick* [int] tick number on which the targets should be scheduled.

*path* [str] path of the target element(s). This can be a wildcard also.

*fn* [str] name of the function to be called on each tick. Commonly *process*.

Examples -

In multi-compartmental neuron model a compartment's membrane potential (Vm) is dependent on its neighbours' membrane potential. Thus it must get the neighbour's present Vm before computing its own Vm in next time step. This ordering is achieved by scheduling the *init* function, which communicates membrane potential, on tick 0 and *process* function on tick 1.:

```
>>> moose.useClock(0, '/model/compartment_1', 'init')
>>> moose.useClock(1, '/model/compartment_1', 'process')
```

### setClock

`moose.setClock(tick, dt)`

set the ticking interval of *tick* to *dt*.

A tick with interval *dt* will call the functions scheduled on that tick every *dt* timestep.

*tick* [int] tick number

*dt* [double] ticking interval

### start

`moose.start(time, notify = False) -> None`

Run simulation for *t* time. Advances the simulator clock by *t* time. If 'notify = True', a message is written to terminal whenever 10% of simulation time is over.

After setting up a simulation, YOU MUST CALL MOOSE.REINIT() before CALLING MOOSE.START() TO EXECUTE THE SIMULATION. Otherwise, the simulator behaviour will be undefined. Once `moose.reinit()` has been called, you can call `moose.start(t)` as many time as you like. This will continue the simulation from the last state for *t* time.

*t* [float] duration of simulation.

*notify* [bool] default False. If True, notify user whenever 10% of simulation is over.

Returns - None

### reinit

`moose.reinit() -> None`

Reinitialize simulation.

This function (re)initializes moose simulation. It must be called before you start the simulation (see `moose.start`). If you want to continue simulation after you have called `moose.reinit()` and `moose.start()`, you must NOT call `moose.reinit()` again. Calling `moose.reinit()` again will take the system back to initial setting (like clear out all data recording tables, set state variables to their initial values, etc).

### stop

`moose.stop(...)` Stop simulation

### isRunning

`moose.isRunning(...)` True if the simulation is currently running.

### exists

`moose.exists(...)` True if there is an object with specified path.

## loadModel

**moose.loadModel(...)** loadModel(filename, modelpath, solverclass) -> vec

Load model from a file to a specified path.

**filename** [str] model description file.

**modelpath** [str] moose path for the top level element of the model to be created.

**solverclass** [str, optional] solver type to be used for simulating the model.

**Returns - vec** loaded model container vec.

## connect

**moose.connect(src, srcfield, destobj, destfield[,msgtype])** -> bool

Create a message between *src\_field* on *src* object to *dest\_field* on *dest* object. This function is used mainly, to say, connect two entities, and to denote what kind of give-and-take relationship they share. It enables the 'destfield' (of the 'destobj') to acquire the data, from 'srcfield' (of the 'src').

**src** [element/vec/string] the source object (or its path) (the one that provides information)

**srcfield** [str] source field on self. (type of the information)

**destobj** [element] Destination object to connect to. (The one that need to get information)

**destfield** [str] field to connect to on *destobj*.

**msgtype** [str] type of the message. Can be *Single* - *OneToAll* - *AllToOne* - *OneToOne* - *Reduce* - *Sparse* - Default: *Single*.

**Returns - msgmanager** [melement] message-manager for the newly created message.

Examples - Connect the output of a pulse generator to the input of a spike generator:

```
>>> pulsegen = moose.PulseGen('pulsegen')
>>> spikegen = moose.SpikeGen('spikegen')
>>> pulsegen.connect('output', spikegen, 'Vm')
```

## getCwe

**moose.getCwe(...)** Get the current working element. 'pwe' is an alias of this function.

## setCwe

**moose.setCwe(...)** Set the current working element. 'ce' is an alias of this function

## getFieldDict

**moose.getFieldDict(className, finfoType)** -> dict

Get dictionary of field names and types for specified class.

**className** [str] MOOSE class to find the fields of.

**finfoType** [str (optional)] Finfo type of the fields to find. If empty or not specified, all fields will be retrieved.

**Returns - dict** field names and their types.

**Notes** - This behaviour is different from `getFieldNames` where only *valueInfo*'s are returned when *infoType* remains unspecified.

**Examples** - List all the source fields on class Neutral:

```
>>> moose.getFieldDict('Neutral', 'srcFinfo')
>>> {'childMsg': 'int'}
```

### getField

**moose.getField(...)** `getField(element, field, fieldtype)` – Get specified field of specified type from object vec.

### seed

**moose.seed(...)** `moose.seed(seedvalue)` -> seed

Reseed MOOSE random number generator.

**seed** [int] Value to use for seeding. All RNGs in moose except `rand` functions in `moose.Function` expression use this seed. By default (when this function is not called) seed is initialized to some random value using system random device (if available).

default: random number generated using system random device

Returns - None

### rand

**moose.rand(...)** `moose.rand()` -> [0,1)

Returns - float in [0, 1) real interval generated by MT19937.

Notes - MOOSE does not automatically seed the random number generator. You must explicitly call `moose.seed()` to create a new sequence of random numbers each time.

### wildcardFind

`moose.wildcardFind(expression)` -> tuple of melements.

Find an object by wildcard.

**expression** [str] MOOSE allows wildcard expressions of the form:

```
{PATH} / {WILDCARD} [ {CONDITION} ]
```

where {PATH} is valid path in the element tree. {WILDCARD} can be # or ##.

# causes the search to be restricted to the children of the element specified by {PATH}.

## makes the search to recursively go through all the descendants of the {PATH} element. {CONDITION} can be:

```

TYPE={CLASSNAME} : an element satisfies this condition if it is of
class {CLASSNAME}.
ISA={CLASSNAME} : alias for TYPE={CLASSNAME}
CLASS={CLASSNAME} : alias for TYPE={CLASSNAME}
FIELD({FIELDNAME}){OPERATOR}{VALUE} : compare field {FIELDNAME} with
{VALUE} by {OPERATOR} where {OPERATOR} is a comparison operator (=,
!=", >, <, >=, <=).

```

For example, /mymodel/##[FIELD(Vm)>=-65] will return a list of all the objects under /mymodel whose Vm field is >= -65.

**Returns - tuple** all elements that match the wildcard.

## quit

Finalize MOOSE threads and quit MOOSE. This is made available for debugging purpose only. It will automatically get called when moose module is unloaded. End user should not use this function.

**moose.quit(...)** Finalize MOOSE threads and quit MOOSE. This is made available for debugging purpose only. It will automatically get called when moose module is unloaded. End user should not use this function.

## Class Hierarchy

- `__builtin__.object` - Melement
  - **Neutral**
    - \* Adaptor
    - \* Annotator
    - \* Arith
    - \* **CaConeBase**
      - CaCone
      - ZombieCaCone
    - \* **ChanBase**
      - HHChannel2D
      - **HHChannelBase**
        - HHChannel
        - ZombieHChannel
      - Leakage
      - MarkovChannel
      - MgBlock
      - **SynChan**
        - NMDAChan
    - \* **ChemCompt**
      - CubeMesh

- CylMesh
- NeuroMesh
- PsdMesh
- SpineMesh
- \* Cinfo
- \* Clock
- \* **CompartmentBase**
  - **Compartment**
    - IntFireBase**
      - AdThreshIF**
        - ExIF**
          - AdExIF
          - IzhIF
          - LIF
          - QIF
        - SymCompartment
        - ZombieCompartment
  - **DifBufferBase**
    - DifBuffer
  - **DifShellBase**
    - DifShell
  - DiffAmp
  - Dsolve
  - **EnzBase**
    - CplxEnzBase**
      - Enz
      - ZombieEnz
    - MMenz
    - ZombieMMenz
  - Finfo
  - Func
  - **Function**
    - ZombieFunction
  - GapJunction
  - Group
  - Gsolve

- **HDF5WriterBase**
  - HDF5DataWriter**
    - NSDFWriter
- HHGate
- HHGate2D
- HSolve
- IntFire
- Interpol2D
- IzhikevichNrn
- Ksolve
- MMPump
- MarkovGslSolver
- MarkovRateTable
- **MarkovSolverBase**
  - MarkovSolver
- MeshEntry
- **Msg**
  - DiagonalMsg
  - OneToAllMsg
  - OneToOneDataIndexMsg
  - OneToOneMsg
  - SingleMsg
  - SparseMsg
- Mstring
- Nernst
- Neuron
- PIDController
- **PoolBase**
  - Pool**
    - BufPool
  - ZombiePool**
    - ZombieBufPool
- PostMaster
- PulseGen
- PyRun
- RC

- **RandGenerator**
  - BinomialRng
  - ExponentialRng
  - GammaRng
  - NormalRng
  - PoissonRng
  - UniformRng
- RandSpike
- **ReacBase**
  - Reac
  - ZombieReac
- Shell
- Species
- SpikeGen
- Spine
- **Stats**
  - Spike
- SteadyState
- Stoich
- **SynHandlerBase**
  - GraupnerBrunel2012CaPlasticitySynHandler
  - STDPSynHandler
  - SeqSynHandler
  - SimpleSynHandler
- **Synapse**
  - STDPSynapse
- **TableBase**
  - Interpol
  - StimulusTable
  - Streamer
  - Table
  - Table2
  - TimeTable
- VClamp
- **Variable**
  - InputVariable

```
· VectorTable
  * testSched
  - vec
  - Moose_BuiltIn
```

## 1.6 Doxygen

Here you can find all the references necessary for MOOSE.

## 1.7 Release Notes

---

**Todo:** Collect release notes from github.

---

## 1.8 Changes

---

**Todo:** collect changes from OBS.

---

## 1.9 Known issues

Full report can be found at the following places

- Related to build, packages and documentation <https://github.com/BhallaLab/moose/issues>
- Related to python interface of MOOSE <https://github.com/BhallaLab/moose-core/issues>
- Related to MOOSE GUI <https://github.com/BhallaLab/moose-gui/issues>
- Related to moogli <https://github.com/BhallaLab/moogli/issues>



## CHAPTER 2

---

### Indices and tables

---

- [genindex](#)
- [modindex](#)
- [search](#)



**a**

analogStimTable, 64

**b**

bidirectionalPlasticity, 58

**c**

changeFuncExpression, 52  
chemDoseResponse, 65  
compartment\_net, 71  
compartment\_net\_no\_array, 71  
convert\_Genesis2Sbml, 51  
crossComptNeuroMesh, 53  
crossComptOscillator, 53  
crossComptSimpleReac, 53  
crossComptSimpleReacGSSA, 54  
cspaceSteadyState, 67  
cubeMeshSigNeur, 36  
cylinderDiffusion, 59  
cylinderMotor, 59

**d**

diffEqSolution, 31  
diffSpinyNeuron, 62

**f**

findChemSteadyState, 64  
func, 26  
funcInputToPools, 33  
funcRateHarmonicOsc, 32  
funcReacLotkaVolterra, 32  
function, 30

**g**

gapjunction, 39  
GraupnerBrunel2012\_STDPfromCaPlasticity,  
72  
gssaCylinderDiffusion, 59  
gssaRDspiny, 73

**h**

hdfdemo, 27  
helloMoose, 22  
HigginsGraupnerBrunel2014\_LifetimeCaPlasticity,  
72

**i**

insertSpines, 74  
insertSpinesWithoutRdesigneur, 39  
IntegrateFireZoo, 37  
interpol, 26  
interpol2d, 26  
intfire, 39

**l**

lif, 38  
lifcomp, 38  
loadCspaceModel, 51  
loadKineticModel, 51  
loadSbmlmodel, 51

**m**

mapkFB, 57  
multicomp\_lif, 69  
multiComptSigNeur, 73  
multiscaleOneCompt, 72

**n**

neuronFromDotp, 37  
nsdf, 28  
nsdf\_vec, 28

**o**

onetonemsg, 25

**p**

pulsegen, 38  
pulsegen2, 38  
pyrun, 22

pyrun1, 24

## r

randomspike, 69  
RandSpikeStats, 70  
reacDiffBranchingNeuron, 61  
reacDiffConcGradient, 61  
reacDiffSpinyNeuron, 61  
recurrentIntFire, 70  
recurrentLIF, 71  
relaxationOsc, 57  
repressillator, 56  
rxdFuncDiffusion, 60  
rxdFuncDiffusionStoch, 60  
rxdReacDiffusion, 60

## S

savemodel, 51  
scaleVolumes, 58  
scriptGssaSolver, 51  
scriptKineticModel, 68  
scriptKineticSolver, 52  
showclocks, 25  
showmsg, 25  
singlemsgcross, 25  
slowFbOsc, 56  
startstop, 22  
STDP, 70  
stimtable, 22  
stochasticLotkaVolterra, 33  
strongBis, 58  
switchKineticSolvers, 63  
symcompartment, 27  
synapse, 39  
synapse\_tutorial, 72

## t

tabledemo, 27  
testHsolve, 40  
testSigNeur, 37  
threading\_demo, 29  
timetable, 26  
transportBranchingNeuron, 66  
traub\_naf, 29  
TuringOneDim, 60  
tweakingParameters, 54  
twocells, 68

## V

vclamp, 38  
vectors, 26

## W

wildcard, 26

**A**

analogStimTable (module), 64  
assign\_clocks() (in module compartment\_net\_no\_array),  
71

**B**

bidirectionalPlasticity (module), 58

**C**

changeFuncExpression (module), 52  
chemDoseResponse (module), 65  
compartment\_net (module), 71  
compartment\_net\_no\_array (module), 71  
connect\_spikegen() (in module intfire), 39  
connect\_two\_intfires() (in module intfire), 39  
convert\_Genesis2Sbml (module), 51  
create\_cell() (in module randomspike), 69  
create\_compartment() (in module traub\_naf), 29  
create\_model() (in module twocells), 68  
create\_naf\_proto() (in module traub\_naf), 29  
create\_network() (in module compartment\_net), 71  
create\_population() (in module compartment\_net), 71  
create\_population() (in module compartment\_net\_no\_array), 71  
create\_spine() (in module testHsolve), 40  
create\_squid() (in module testHsolve), 40  
createSpine() (in module multiComptSigNeur), 73  
createSpine() (in module testSigNeur), 37  
createSquid() (in module cubeMeshSigNeur), 36  
createSquid() (in module multiComptSigNeur), 73  
createSquid() (in module testSigNeur), 37  
crossComptNeuroMesh (module), 53  
crossComptOscillator (module), 53  
crossComptSimpleReac (module), 53  
crossComptSimpleReacGSSA (module), 54  
cspaceSteadyState (module), 67  
cubeMeshSigNeur (module), 36  
cylinderDiffusion (module), 59  
cylinderMotor (module), 59

**D**

diffEqSolution (module), 31  
diffSpinyNeuron (module), 62  
do\_iclamp() (in module traub\_naf), 29  
do\_vclamp() (in module traub\_naf), 29

**E**

example() (in module randomspike), 69

**F**

findChemSteadyState (module), 64  
func (module), 26  
funcInputToPools (module), 33  
funcRateHarmonicOsc (module), 32  
funcReacLotkaVolterra (module), 32  
function (module), 30

**G**

gapjunction (module), 39  
generate\_poisson\_times() (in module timetable), 26  
getState() (in module findChemSteadyState), 64  
GraupnerBrunel2012\_STDPfromCaPlasticity (module),  
72  
gssaCylinderDiffusion (module), 59  
gssaRDspiny (module), 73

**H**

hdfdemo (module), 27  
helloMoose (module), 22  
HigginsGraupnerBrunel2014\_LifetimeCaPlasticity  
(module), 72

**I**

input\_output() (in module pyrun), 22  
insertSpines (module), 74  
insertSpinesWithoutRdesigneur (module), 39  
IntegrateFireZoo (module), 37  
interpol (module), 26  
interpol2d (module), 26

intfire (module), 39

## L

lif (module), 38

lifcomp (module), 38

loadCspaceModel (module), 51

loadKineticModel (module), 51

loadSbmlmodel (module), 51

## M

main() (in module analogStimTable), 64

main() (in module bidirectionalPlasticity), 58

main() (in module changeFuncExpression), 52

main() (in module chemDoseResponse), 65

main() (in module compartment\_net), 71

main() (in module compartment\_net\_no\_array), 71

main() (in module convert\_Genesis2Sbml), 51

main() (in module crossComptNeuroMesh), 53

main() (in module crossComptOscillator), 53

main() (in module crossComptSimpleReac), 53

main() (in module crossComptSimpleReacGSSA), 54

main() (in module cspaceSteadyState), 67

main() (in module cubeMeshSigNeur), 36

main() (in module cylinderDiffusion), 59

main() (in module cylinderMotor), 59

main() (in module diffEqSolution), 31

main() (in module diffSpinyNeuron), 62

main() (in module findChemSteadyState), 64

main() (in module func), 26

main() (in module funcInputToPools), 33

main() (in module funcRateHarmonicOsc), 32

main() (in module funcReacLotkaVolterra), 32

main() (in module function), 30

main() (in module gapjunction), 39

main() (in module Graupner-Brunel2012\_STDPfromCaPlasticity), 72

main() (in module gssaCylinderDiffusion), 59

main() (in module gssaRDspiny), 73

main() (in module hdfdemo), 27

main() (in module helloMoose), 22

main() (in module HigginsGraupner-Brunel2014\_LifetimeCaPlasticity), 72

main() (in module insertSpines), 74

main() (in module insertSpinesWithoutRdesigneur), 39

main() (in module IntegrateFireZoo), 37

main() (in module interpol), 26

main() (in module intfire), 39

main() (in module lifcomp), 38

main() (in module loadCspaceModel), 51

main() (in module loadKineticModel), 51

main() (in module loadSbmlmodel), 51

main() (in module mapkFB), 57

main() (in module multicomp\_lif), 69

main() (in module multiComptSigNeur), 73

main() (in module multiscaleOneCompt), 72

main() (in module neuronFromDotp), 37

main() (in module nsdf\_vec), 28

main() (in module onetoonemsg), 25

main() (in module pulsegen), 38

main() (in module pulsegen2), 38

main() (in module pyrun), 23

main() (in module pyrun1), 24

main() (in module randomspike), 69

main() (in module RandSpikeStats), 70

main() (in module reacDiffBranchingNeuron), 61

main() (in module reacDiffConcGradient), 61

main() (in module reacDiffSpinyNeuron), 61

main() (in module recurrentIntFire), 70

main() (in module recurrentLIF), 71

main() (in module relaxationOsc), 57

main() (in module repressillator), 56

main() (in module rxdFuncDiffusion), 60

main() (in module rxdFuncDiffusionStoch), 60

main() (in module rxdReacDiffusion), 60

main() (in module savemodel), 51

main() (in module scaleVolumes), 58

main() (in module scriptGssaSolver), 51

main() (in module scriptKineticModel), 68

main() (in module scriptKineticSolver), 52

main() (in module showclocks), 25

main() (in module showmsg), 25

main() (in module singlemsgcross), 25

main() (in module slowFbOsc), 56

main() (in module startstop), 22

main() (in module STDP), 70

main() (in module stimtable), 22

main() (in module stochasticLotkaVolterra), 33

main() (in module strongBis), 58

main() (in module switchKineticSolvers), 63

main() (in module symcompartment), 27

main() (in module synapse), 39

main() (in module synapse\_tutorial), 72

main() (in module tabledemo), 27

main() (in module testHsolve), 40

main() (in module testSigNeur), 37

main() (in module threading\_demo), 29

main() (in module timetable), 26

main() (in module transportBranchingNeuron), 66

main() (in module traub\_naf), 30

main() (in module tweakingParameters), 54

main() (in module twocells), 69

main() (in module vclamp), 38

main() (in module vectors), 26

main() (in module wildcard), 26

make\_neuron\_spike() (in module Graupner-Brunel2012\_STDPfromCaPlasticity), 72

make\_neuron\_spike() (in module STDP), 70

make\_synapses() (in module compartment\_net), 71

- make\_synapses() (in module compartment\_net\_no\_array), 71
- makeChannelPrototypes() (in module neuronFromDotp), 37
- makeChemModel() (in module diffSpinyNeuron), 62
- makeChemModel() (in module gssaRDspiny), 73
- makeChemModel() (in module reacDiffBranchingNeuron), 61
- makeChemModel() (in module reacDiffSpinyNeuron), 62
- makeModel() (in module chemDoseResponse), 66
- makeModel() (in module findChemSteadyState), 64
- makeModel() (in module TuringOneDim), 60
- mapkFB (module), 57
- multicomp\_lif (module), 69
- multiComptSigNeur (module), 73
- multiscaleOneCompt (module), 72
- ## N
- neuronFromDotp (module), 37
- nsdf (module), 28
- nsdf\_vec (module), 28
- ## O
- onetonemsg (module), 25
- ## P
- pulsegen (module), 38
- pulsegen2 (module), 28
- pyrun (module), 22
- pyrun1 (module), 24
- ## R
- randomspike (module), 69
- RandSpikeStats (module), 70
- reacDiffBranchingNeuron (module), 61
- reacDiffConcGradient (module), 61
- reacDiffSpinyNeuron (module), 61
- read\_nsdf() (in module nsdf\_vec), 28
- recurrentIntFire (module), 70
- recurrentLIF (module), 71
- relaxationOsc (module), 57
- repressillator (module), 56
- reset\_settle() (in module GraupnerBrunel2012\_STDPfromCaPlasticity), 72
- reset\_settle() (in module STDP), 70
- run() (threading\_demo.StatusThread method), 29
- run() (threading\_demo.WorkerThread method), 29
- run\_clamp() (in module traub\_naf), 30
- run\_sequence() (in module pyrun), 23
- run\_sequence() (in module pyrun1), 24
- run\_sim() (in module traub\_naf), 30
- rxdFuncDiffusion (module), 60
- rxdFuncDiffusionStoch (module), 60
- rxdReacDiffusion (module), 60
- ## S
- savemodel (module), 51
- scaleVolumes (module), 58
- scriptGssaSolver (module), 51
- scriptKineticModel (module), 68
- scriptKineticSolver (module), 52
- setup\_electronics() (in module traub\_naf), 30
- setup\_experiment() (in module twocells), 69
- setup\_model() (in module nsdf), 28
- setup\_model() (in module traub\_naf), 30
- setup\_synapse() (in module intfire), 39
- setup\_two\_cells() (in module lifcomp), 38
- setup\_two\_cells() (in module multicomp\_lif), 69
- setupmodel() (in module lif), 38
- setupModel() (in module STDP), 70
- showclocks (module), 25
- showmsg (module), 25
- single\_population() (in module compartment\_net\_no\_array), 71
- singlemsgcross (module), 25
- slowFbOsc (module), 56
- startstop (module), 22
- StatusThread (class in threading\_demo), 29
- STDP (module), 70
- stimtable (module), 22
- stochasticLotkaVolterra (module), 33
- strongBis (module), 58
- switchKineticSolvers (module), 63
- symcompartment (module), 27
- synapse (module), 39
- synapse\_tutorial (module), 72
- ## T
- tabledemo (module), 27
- test\_crossing\_single() (in module singlemsgcross), 25
- test\_func() (in module func), 26
- test\_func\_nosim() (in module func), 26
- testHsolve (module), 40
- testSigNeur (module), 37
- threading\_demo (module), 29
- timetable (module), 26
- timetable\_file() (in module timetable), 26
- timetable\_npararray() (in module timetable), 26
- transportBranchingNeuron (module), 66
- traub\_naf (module), 29
- TuringOneDim (module), 60
- tweakingParameters (module), 54
- two\_populations() (in module compartment\_net\_no\_array), 71
- twocells (module), 68

## V

vclamp (module), 38  
vectors (module), 26

## W

wildcard (module), 26  
WorkerThread (class in threading\_demo), 29  
write\_nsdf() (in module nsdf\_vec), 29