

MOOSE User Manual

Release chennapoda (3.2.rc)

**Upinder Bhalla, Niraj Dudani, Subhasis Ray
Aditya Gilra, Harsha Rani, Aviral Goel
Dilawar Singh, Malav Shah, Dhruva Gowda Storz**

November 13, 2018

| | | |
|----------|---|------------|
| 1 | What is MOOSE and what is it good for? | 1 |
| 2 | Installation | 3 |
| 2.1 | Use pre-built packages | 3 |
| 2.2 | Building MOOSE | 4 |
| 2.3 | Graphical User Interface (GUI) | 6 |
| 2.4 | Building moogli | 7 |
| 3 | Quick Start | 9 |
| 3.1 | Interactive Tutorials | 9 |
| 3.2 | MOOSE GUI: Graphical interface for MOOSE | 9 |
| 3.3 | Getting started with python scripting for MOOSE | 15 |
| 3.4 | Demonstration of basic functionalities | 30 |
| 3.5 | MOOSE Classes | 30 |
| 4 | Cook Book | 33 |
| 4.1 | Single Neuron Electrical Aspects (BioPhysics) | 33 |
| 4.2 | Chemical Aspects | 38 |
| 4.3 | Networking | 56 |
| 4.4 | MultiScale Modeling | 59 |
| 5 | Rdesignuer | 61 |
| 5.1 | Rdesignuer: Building multiscale models | 61 |
| 5.2 | Rdesignuer Examples | 151 |
| 6 | Teaching Tutorials | 153 |
| 6.1 | Chemical Bistables | 153 |
| 6.2 | Chemical Oscillators | 186 |
| 6.3 | Squid giant axon | 199 |
| 7 | Graphics | 201 |
| 7.1 | MOOGLI | 201 |
| 7.2 | MatPlotLib | 201 |
| 8 | References | 203 |
| 8.1 | How to use the documentation | 203 |

| | |
|------------------------------------|------------|
| 9 Doxygen | 215 |
| 10 Release Notes | 217 |
| 11 Series <i>chennapoda</i> | 219 |
| 11.1 Version 3.2.0 | 219 |
| 12 Series <i>chamcham</i> | 221 |
| 12.1 Version 3.1.3 | 221 |
| 13 Known issues | 223 |
| 14 Indices and tables | 225 |

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.

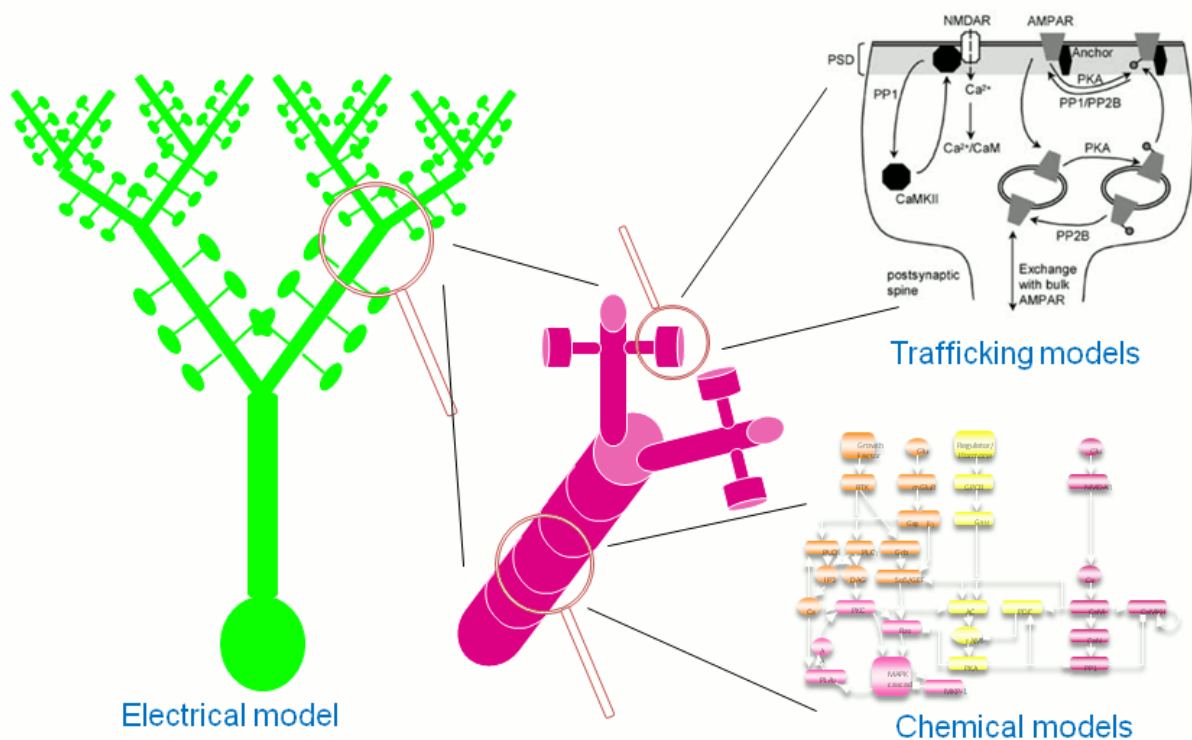


Fig. 1: Multiple scales can be modelled and simulated in MOOSE

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.

2.1 Use pre-built packages

2.1.1 pip

If you only need *python* interface, the recommended way is via *pip*.

```
pip install pymoose --user
```

To install nightly version

```
pip install pymoose --pre --upgrade --user
```

We also have moose package with additional components such as gui and *moogli*.

2.1.2 Linux

We recommend that you use our repositories hosted at [Open Build Service](https://build.opensuse.org/package/show/home:moose/moose) (<https://build.opensuse.org/package/show/home:moose/moose>). Packages for most linux distributions are available. Visit [this page](https://software.opensuse.org/download.html?project=home:moose&package=moose) (<https://software.opensuse.org/download.html?project=home:moose&package=moose>) to pick your distribution and follow instructions.

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

Todo: Packages for gentoo

2.1.3 Mac OSX

MacOSX support for moose-gui is not complete yet because moose-gui depends on PyQt4 but that world has moved onto PyQt5 (See the status here: <https://github.com/BhallaLab/moose-gui/issues/16>). However, the python-scripting interface can be installed on OSX using homebrew

```
$ brew tap BhallaLab/moose
$ brew install moose
```

Or alternatively, via pip

```
$ pip install pymoose --user
```

2.1.4 Docker Images

Docker images of stable version are available from public repository.

```
$ docker pull bhallalab/moose
$ docker run -it --rm -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=
↪$DISPLAY bhallalab/moose
```

This will launch *xterm*; run *moosegui* in terminal to launch the GUI.

2.2 Building MOOSE

In case your distribution is not listed on [our repository page](https://software.opensuse.org/download.html?project=home:moose&package=moose) (<https://software.opensuse.org/download.html?project=home:moose&package=moose>), 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](https://github.com/BhallaLab/moose/archive/master.zip) (<https://github.com/BhallaLab/moose/archive/master.zip>).

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

(This will create folder called “moose”) Or,

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

If you don't want latest snapshot of MOOSE, you can download other released versions from [here](https://github.com/BhallaLab/moose/releases) (<https://github.com/BhallaLab/moose/releases>).

2.2.1 Install dependencies

Next, you need to install required dependencies. Depending on your OS, names of following packages may vary.

Core MOOSE

- **Required:**
 - cmake (version 2.8 or higher)
 - g++ or clang (with *c++11* support).
 - gsl-1.16 or higher.
- **Optional**
 - HDF5 ($\geq 1.8.x$) For reading and writing data into HDF5 based formats. Disabled by default.

Python interface for core MOOSE API (pymoose)

- **Required**
 - python (Both 2.7 and 3.x versions are supported).
 - python-dev. Python development headers and libraries, e.g. *python-dev* or *python-devel*
 - NumPy ($\geq 1.6.x$) For array interface, e.g. *python-numpy* or *numpy*
- **Optional**
 - networkx ($\geq 1.x$) For automatical layout
 - pygraphviz. For automatic layout for chemical models
 - matplotlib ($\geq 2.x$). For plotting simulation results
 - python-libsaml. For reading and writing chemical models from and into SBML format
 - pylibsaml

All of these dependencies can be installed using *pip* or your package manager.

On Debian/Ubuntu

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

On CentOS/Fedora/RHEL/Scientific Linux

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

On OpenSUSE

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

2.2.2 Build moose

```
$ cd /to/moose_directory/moose-core/  
$ mkdir _build  
$ cd _build  
$ cmake ..  
$ make  
$ ctest --output-on-failure # optional  
$ sudo make install
```

This will build pyMOOSE (MOOSE's python extention), *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
```

2.3 Graphical User Interface (GUI)

If you have installed the pre-built package, then you already have the GUI. You can launch it by running *moosegui* command in terminal.

You can get the source of *moose-gui* from [here](https://github.com/BhallaLab/moose-gui) (<https://github.com/BhallaLab/moose-gui>). You can download it either by clicking on [this link](https://github.com/BhallaLab/moose-gui/archive/master.zip) (<https://github.com/BhallaLab/moose-gui/archive/master.zip>) or by using *git*

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

Alternatively the *moose-gui* folder exists within the *moose* folder downloaded and built earlier in the installation process. It can be found under *location_of_moose_folder/moose/moose-gui/*.

Below are packages which you may want to install to use MOOSE Graphical User Interface.

- **Required:**
 - PyQt4 (4.8.x). For Python GUI
 - Matplotlib (>= 2.x). For plotting simulation results

– NetworkX (1.x). For automatical layout

- **Optional:**

– python-libsbnl. For reading and writing signalling models from and into SBML format

On Ubuntu/Debian, these can be installed with

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

On CentOS/Fedora/RHEL

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

Now you can fire up the GUI

```
$ cd /to/moose-gui
$ python mgui.py
```

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

```
$ moosegui
```

2.4 Building moogli

moogli is subproject of MOOSE for visualizing models. More details can be found [here](http://moose.ncbs.res.in/moogli) (<http://moose.ncbs.res.in/moogli>).

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](https://github.com/BhallaLab/moogli/archive/master.zip) (<https://github.com/BhallaLab/moogli/archive/master.zip>).

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 -j3
$ sudo make install
```

If you run into troubles, please report it on our [github repository](https://github.com/BhallaLab/moose) (<https://github.com/BhallaLab/moose/issues>).

3.1 Interactive Tutorials

Some of the pages in the documentation, such as the python scripting page and teaching tutorials are available in interactive form.

These interactive tutorials serve as fully executable python environments that can run moose. Therefore, it is a great place to both learn about and play around with MOOSE.

All the currently available interactive tutorials are available by clicking the link below: (<https://mybinder.org/v2/gh/BhallaLab/moose-binder/master?filepath=home%2Fmooser%2FIndex.ipynb>)

3.2 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.

3.2.1 Contents

- *Introduction* (page 10)
- *Interface* (page 10)
 - *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 14)
 - * *Plot Widget* (page 14)
 - *Toolbar* (page ??)
 - *Context Menu* (page ??)

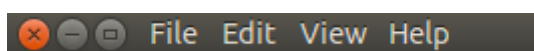
3.2.2 Introduction

The Moose GUI currently allow you work on *chemical* (page 38) and *electrical* (page 35) models using a interface. This document describes the salient features of the GUI.

3.2.3 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.

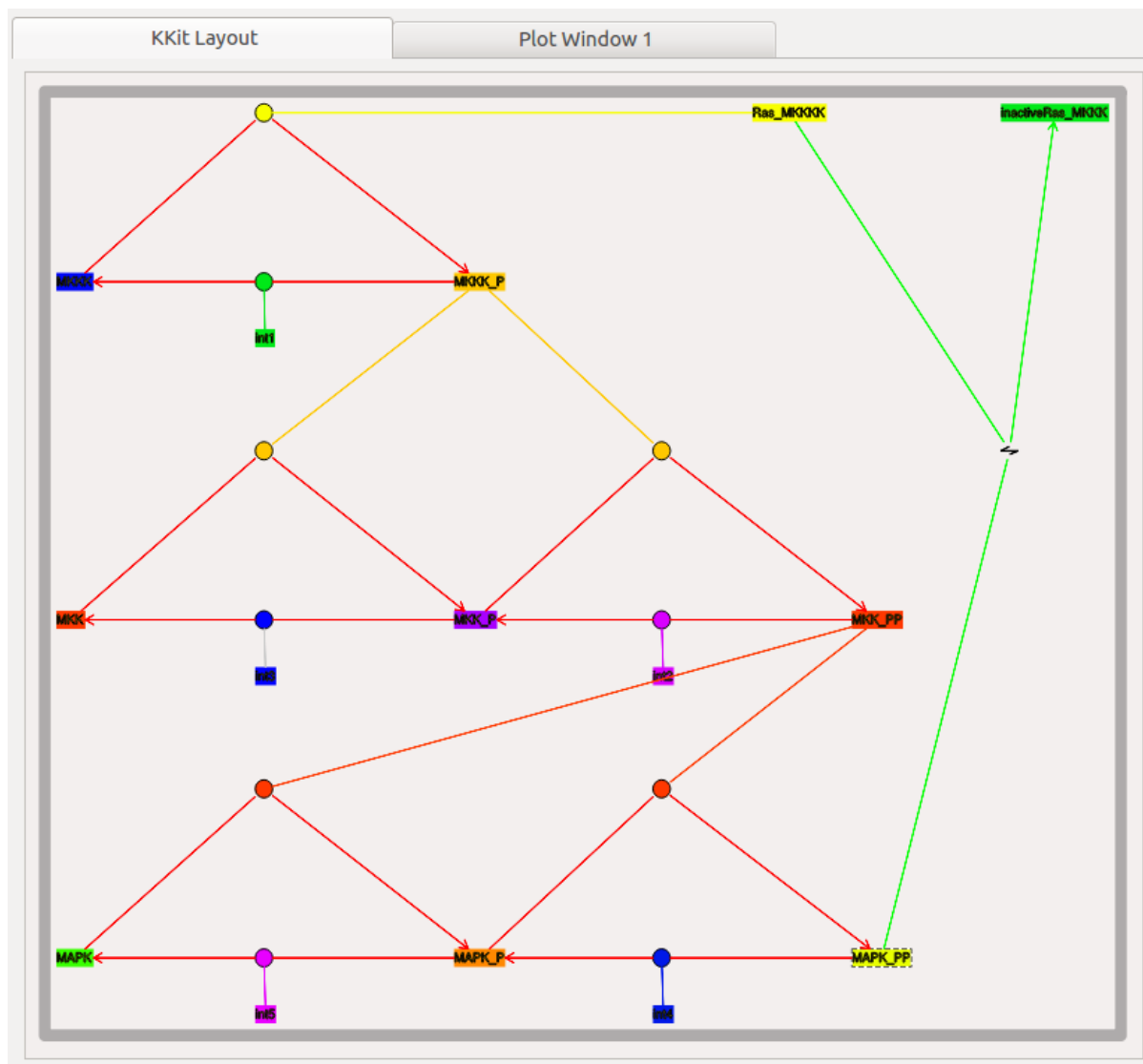





Fig. 1: Chemical Signalling Model Editor

- *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

Edit: /Reaction[0]/model[0]/...  

| Field | Value |
|---------------|--|
| name | B |
| className | Pool |
| n | 20073805.0 |
| nInit | 0.0 |
| conc (mM) | 0.03333333333333 |
| conclnit (mM) | 0.0 |
| volume | 1e-15 |
| Color |  |

...

The total conc. of B is 30uM|

Fig. 2: Property Editor

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 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

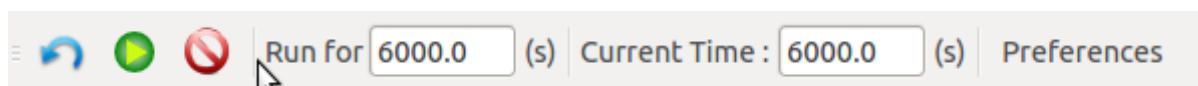


Fig. 3: Simulation Control

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.

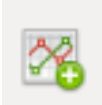
Plot Widget







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.

3.3 Getting started with python scripting for MOOSE

To see an interactive version of this page, click the following link ([https://mybinder.org/v2/gh/BhallaLab/moose-binder/master?filepath=home%2Fmooser%2Fquickstart%](https://mybinder.org/v2/gh/BhallaLab/moose-binder/master?filepath=home%2Fmooser%2Fquickstart%2F)) 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.

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

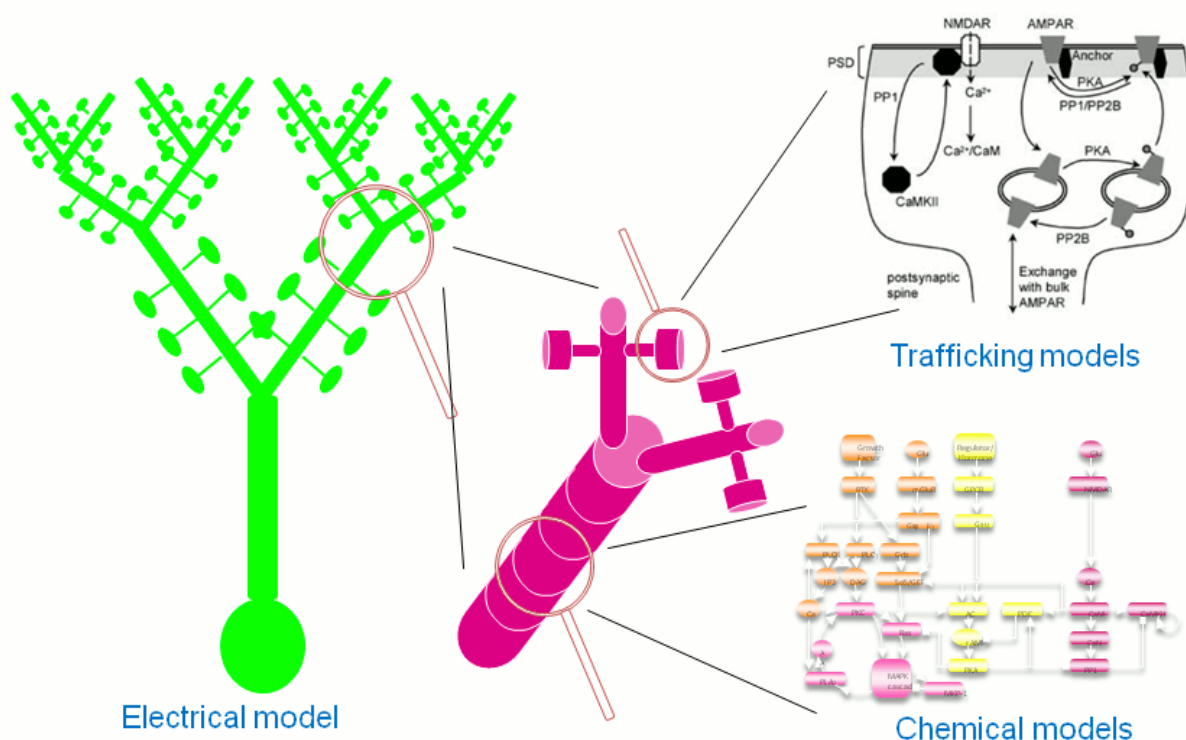


Fig. 4: Multiple scales can be modelled and simulated in MOOSE

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.

Contents:

- *Coding basics and how to use this document* (page 17)
- *Importing moose and accessing documentation* (page 17)
- *Setting the properties of elements: accessing fields* (page 20)
- *Putting them together: setting up connections* (page 22)
- *Scheduling* (page 24)
- *Running the simulation* (page 26)
- *Some more details* (page 27)
- *Moving on* (page 28)

3.3.1 Coding basics and how to use this document

This page acts as the first stepping stone for learning how moose works. The tutorials here are intended to be interactive, and are presented as python commands. Python commands are identifiable by the `>>>` before the command as opposed to `$` which identifies a command-line command.

```
>>> this_is_a_python_command
```

You are encouraged to run a python shell while reading through this documentation and trying out each command for yourself. Python shells are environments within your terminal wherein everything you type is interpreted as a python command. They can be accessed by typing

```
$ python
```

in your command-line terminal.

While individually typing lines of code in a python terminal is useful for practicing using moose and coding in general, note that once you close the python environment all the code you typed is gone and the moose models created are also lost. In order to 'save' models that you create, you would have to type your code in a text file with a `.py` extension. The easiest way to do this is to create a text file in command line, open it with a text editor (for example, gedit), and simply type your code in (make sure you indent correctly).

```
$ touch code.py
$ gedit code.py
```

Once you have written your code in the file, you can run it through your python environment.

```
$ python code.py
```

Note that apart from this section of the quickstart, most of the moose documentation is in the form of snippets. These are basically `.py` files with code that demonstrates a certain functionality in moose. If you see a dialogue box like this one:

You can view the code by clicking the green source button on the left side of the box. Alternatively, the source code for all of the examples in the documentation can be found in `moose/moose-examples/snippets`. Once you run each file in python, it is encouraged that you look through the code to understand how it works.

In the quickstart, most of the snippets demonstrate the functionality of specific classes. However, snippets in later sections such as the cookbook show how to do specific things in moose such as creating networks, chemical models, and synaptic channels.

3.3.2 Importing moose and accessing documentation

In a python script you import modules to access the functionalities they provide. In order to use moose, you need to import it within a python environment or at the beginning of your python script.

```
>>> import moose
```

This make 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` function¹.

```
>>> 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.

Alternatively, if you want to see a full list of classes, functions and their fields, you can browse through the following pages. This is especially helpful when going through snippets.

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

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)

¹ 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.

3.3.3 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.

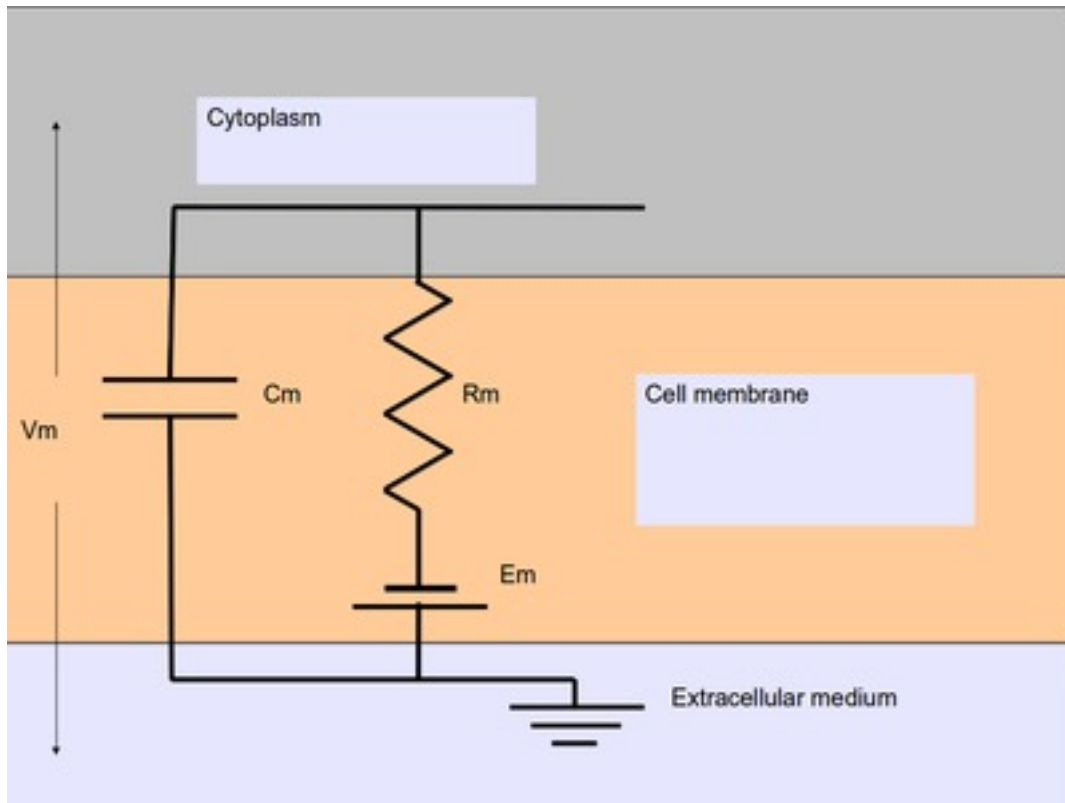


Fig. 5: Passive neuronal compartment

```
>>> 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)².

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

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
```

(continues on next page)

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

(continued from previous page)

```

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.

3.3.4 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 element's `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 27). 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*.³

You can skip the next section on fine control of the timing of updates and read [Running the simulation](#) (page 26).

3.3.5 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: electrical compartments, V and ligand-gated ion channels, Calcium conc and Nernst, stimulus generators and tables, HSolve. | 0-7 | 50 microseconds |
| Table (to plot elec. signals) | 8 | 100 microseconds |
| Diffusion solver | 10 | 0.01 seconds |
| Chemical computations: Pool, Reac, Enz, MMEnz, Func, Function, Gsolve, Ksolve, | 11-17 | 0.1 seconds |

(continues on next page)

³ 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.*

(continued from previous page)

| | | |
|--|----|--------------|
| Stats (to do stats on outputs) | | |
| Table2 (to plot chem. signals) | 18 | 1 second |
| HDF5DataWriter | 30 | 1 second |
| Postmaster (for parallel computations) | 31 | 0.01 seconds |

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 there 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 Vm, 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 order⁴. The `Compartment` class is one such case where a neuronal compartment has to know the Vm of its neighboring compartments before it can calculate its Vm for the next step. This is done with:

```
>>> 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.

3.3.6 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` (<http://matplotlib.org/>) library.

⁴ 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.

```
>>> 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?

3.3.7 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[0]/comp[0] <type 'moose.Compartment'>
/model[0]/comp[1] <type 'moose.Compartment'>
/model[0]/comp[2] <type 'moose.Compartment'>
```

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 to set it: 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`.

3.3.8 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](http://www.genesis-sim.org/GENESIS/Hyperdoc/Manual.html) (<http://www.genesis-sim.org/GENESIS/Hyperdoc/Manual.html>) 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.

3.4 Demonstration of basic functionalities

3.4.1 Load and Run a Model

3.4.2 Start, Stop, and setting clocks

3.4.3 Run Python from MOOSE

3.5 MOOSE Classes

3.5.1 Messages

One-to-one message

Show the message

Single Message Cross

3.5.2 Time

Clocks

Generating Time Data Table

3.5.3 Vectors

3.5.4 Data Entries

3.5.5 Interpolation

1-dimentional Interpolation

2-dimentional interpolation

3.5.6 Function

3.5.7 SymCompartment

3.5.8 Tables

3.5.9 Data Types

HDF DataType

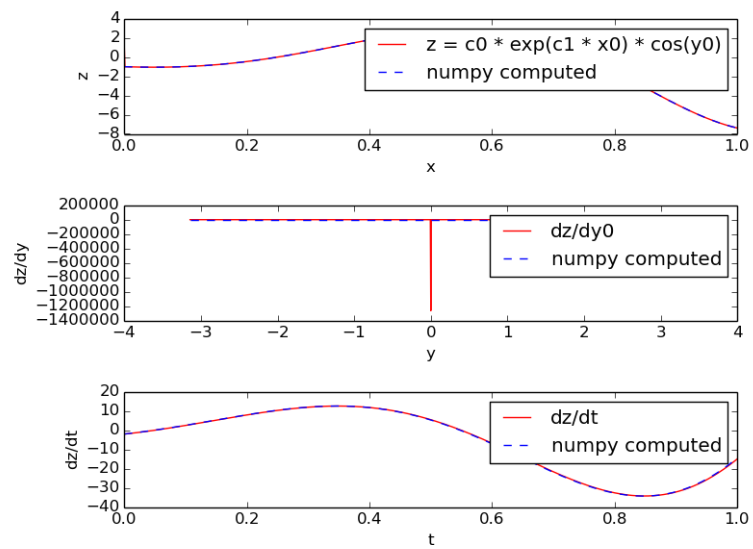
NSDF DataType

3.5.10 Threading

3.5.11 PyMoose

3.5.12 Mathematics with MOOSE

3.5.13 Computing an arbitrary function



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

4.1 Single Neuron Electrical Aspects (BioPhysics)

4.1.1 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.

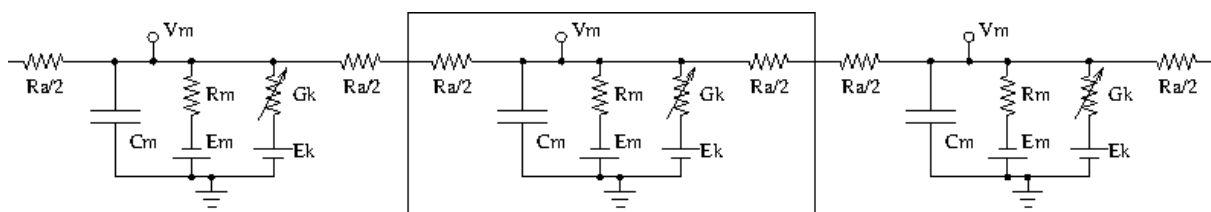


Fig. 1: Equivalent circuit of neuronal compartments

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](http://www.genesis-sim.org/GENESIS/iBoG/iBoGpdf/index.html) (<http://www.genesis-sim.org/GENESIS/iBoG/iBoGpdf/index.html>).

Models can be defined in [NeuroML](http://www.neuroml.org) (<http://www.neuroml.org>), 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](http://www.genesis-sim.org/GENESIS) (<http://www.genesis-sim.org/GENESIS>) .p format is supported for backward 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 **dynamically**. 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. $G_k(t) = Gbar \times X(t)^{X_{power}} \times Y(t)^{Y_{power}} \times Z(t)^{Z_{power}}$
 - 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 τ 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](http://www.genesis-sim.org/GENESIS/Hyperdoc/Manual-26.html#ss26.61) (<http://www.genesis-sim.org/GENESIS/Hyperdoc/Manual-26.html#ss26.61>).
- **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 τ . Its concentration controls Ca-dependent K channels, etc.
 - **Ca** Ca concentration in the pool in units mM (i.e., mol/m³).
 - $d[Ca^{2+}]/dt = B \times I_{Ca} - [Ca^{2+}]/\tau$
 - **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](http://www.genesis-sim.org/GENESIS/iBoG/iBoGpdf/index.html) (<http://www.genesis-sim.org/GENESIS/iBoG/iBoGpdf/index.html>).)

4.1.2 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 proceed 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](http://www.tnb.ua.ac.be/models/network.shtml) (<http://www.tnb.ua.ac.be/models/network.shtml>) (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.net.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.

4.1.3 Load and Run simple models

Each of the following examples can be run by clicking on the green  source button on the right side of each example, and running from within a  python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in  the main moose directory. They can be found under

(...)/moose/moose-examples/snippets

They can be run by typing

```
$ python filename.py
```

(continues on next page)

(continued from previous page)

in your command line, where filename.py is the python file you want
 ↳to run.

All of the following examples show one or more methods within each
 ↳python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.
 ↳main()` methods.

The filename is the bit that comes before the `.` in the blue boxes,
 ↳with `.py` added at the end of it. In this case, the file name would be `cubeMeshSigNeur.py`.

Single Cubicle Compartmental Neuron

Single Neuron Model

Load neuron model from GENESIS

Integrate-and-fire models

4.1.4 Simple Examples

Each of the following examples can be run by clicking on the green
 ↳source button on the right side of each example, and running from within a `.py
 ↳python` file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in
 ↳the main moose directory. They can be found under

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

```
$ python filename.py
```

in your command line, where filename.py is the python file you want
 ↳to run.



All of the following examples show one or more methods within each
 ↳python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.
 ↳main()`

(continues on next page)

(continued from previous page)

methods.

The filename is the bit that comes before the . in the blue boxes,  with  with .py added at the end of it. In this case, the file name would be cubeMeshSigNeur.py.

Create a Leaky Neuron

Create a Leaky Compartment

Voltage Clamping

Generate Pulse

Synapse

Message transmission via synapse

Gap Junction

Insert Spine heads

4.2 Chemical Aspects

4.2.1 Interface for chemical kinetic models in MOOSEGUI

Upinder Bhalla, Harsha Rani

Nov 8 2016.

-
- *Introduction* (page 39)
 - ***TODO** What are chemical kinetic models?* (page 39)
 - *Levels of model* (page 39)
 - *Numerical methods* (page 39)
 - *Using Kinetikit 12* (page 40)
 - *Overview* (page 40)
 - *Model layout and icons* (page 40)
 - * *Compartment* (page 41)
 - * *Pool* (page 41)

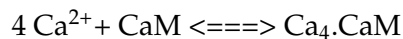
- * *Buffered pools* (page 42)
- * *Reaction* (page 43)
- * *Mass-action enzymes* (page 44)
- * *Michaelis-Menten Enzymes* (page 45)
- * *Summation* (page 46)
- *Model operations* (page 47)
- *Model Building* (page 47)

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:

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

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.
- Select simulation, diffusion dt's along updateInterval for plot and Gui with numerical method using options under **'Preferences'** button in simulation control.
- Run model using **'Run'** button.
- Save plots image using the icons at the top of the **'Plot Window'** or right click on plot to Export to csv.

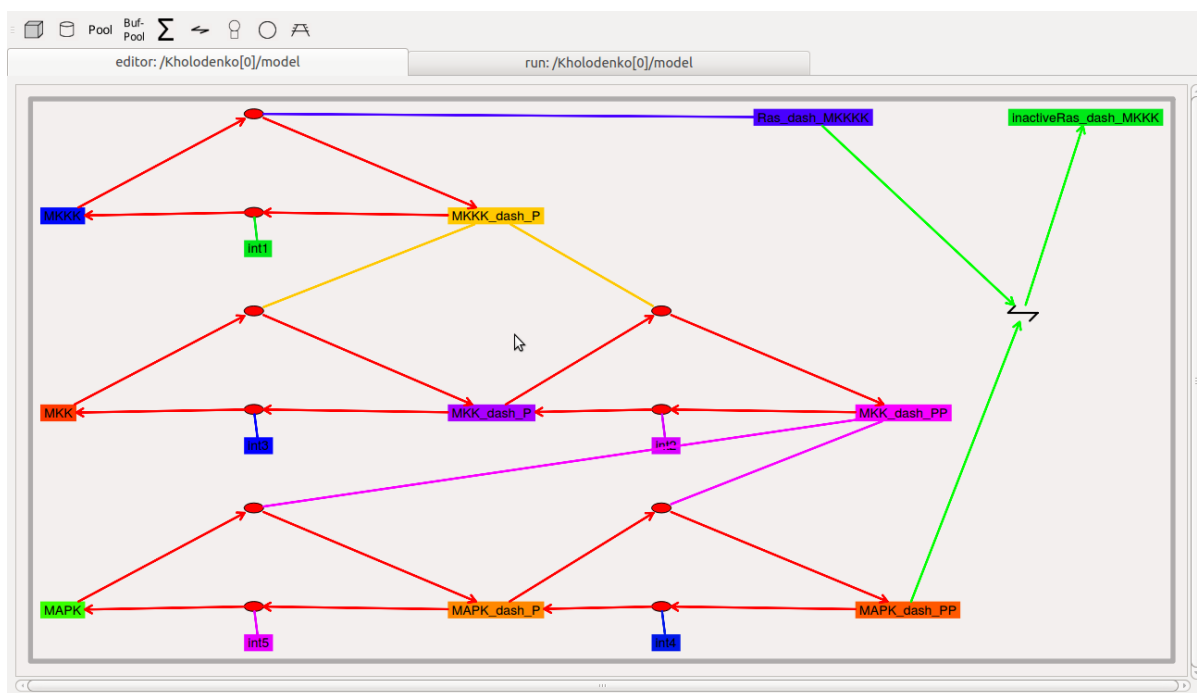
Most of these operations are detailed in other sections, and are shared with other aspects of the MOOSE simulation interface. Here we focus on the Kinetikit-specific items.

Model layout and icons

When you are in the **'Editor View'** tab you will see a collection of icons, arrows, and grey boxes surrounding these. This is a schematic of the reaction scheme being modeled. You can view and change parameters, and change the layout of the model.

Resizing the model layout and icons:

- **Zoom:** Comma and period keys. Alternatively, the mouse scroll wheel or vertical scroll line on the track pad will cause the display to zoom in and out.
- **Pan:** The arrow keys move the display left, right, up, and down.
- **Entire Model View:** Pressing the **'a'** key will fit the entire model into the entire field of view.
- **Resize Icons:** Angle bracket keys, that is, **'<'** and **'>'** or **'+' and '-'**. This resizes the icons while leaving their positions on the screen layout more or less the same.
- **Original Model View:** Pressing the **'A'** key (capital 'A') will revert to the original model view including the original icon scaling.



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. $\text{conc} = n * \text{unit_scale_factor} / (N_A * \text{vol})$
 - **concInit:** Initial concentration of the molecules in the pool. 'conc' is set to this value when the 'reinit' operation is done.
 $\text{concInit} = n\text{Init} * \text{unit_scale_factor} / (N_A * \text{vol})$
- **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-

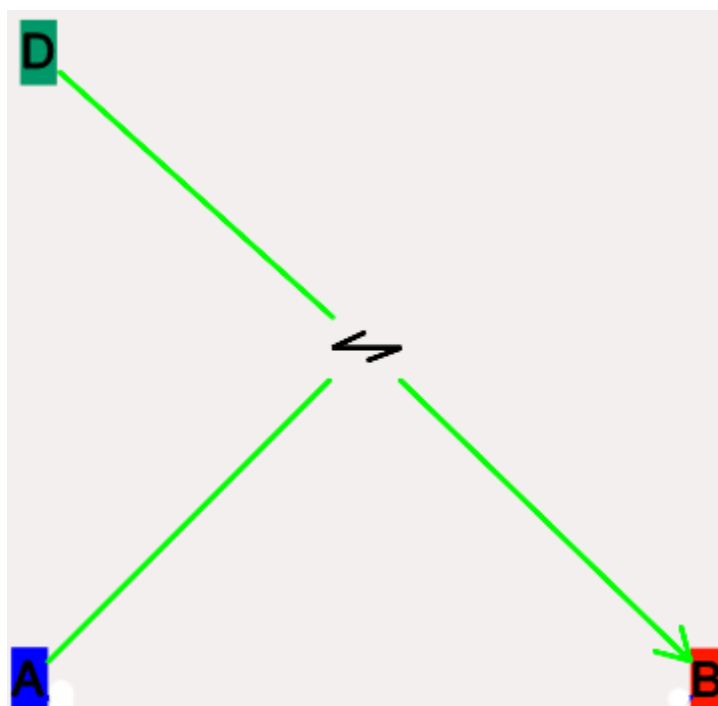
Pool


- **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.
 $\text{concInit} = n\text{Init} * \text{unit_scale_factor} / (N_A * \text{vol})$
- **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.

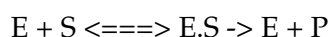


- **Icon:**  Reversible reaction arrow.
- **Moving Reactions:** Click and drag.
- **Reaction editable parameters:**
- **Name :** Name of reaction
- **K_f :** 'Forward rate' of reaction, in 'concentration/time' units. This is the normal way to express and manipulate the reaction rate.
- **k_f :** Forward 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_b :** Backward rate' of reaction, in 'concentration/time' units. This is the normal way to express and manipulate the reaction rate.
- **k_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.

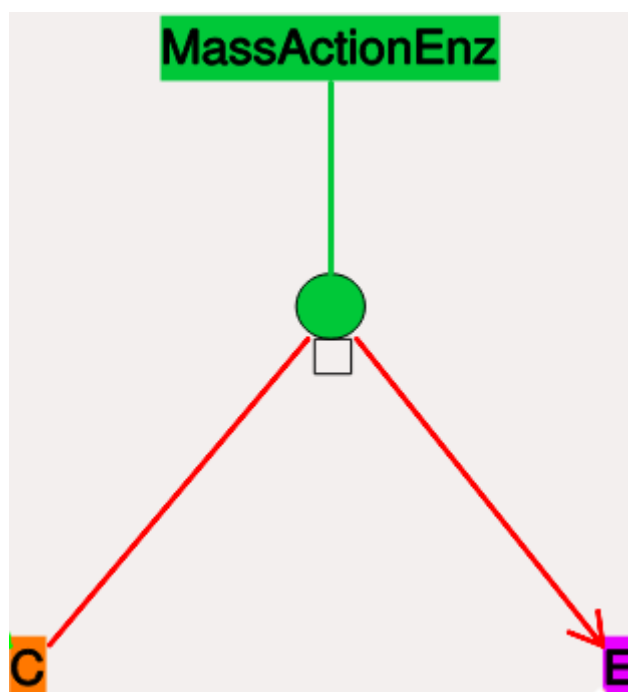
- **Reaction fixed parameters:**
- **numSubstrates:** Number of substrates molecules.
- **numProducts:** Number of product molecules.

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_m** : Michaelis-Menten value for enzyme, in 'concentration' units.
- **k_{cat}** : Production rate of enzyme, in '1/time' units. Equal to k₃, 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 k₂/k₃. 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.

$$\text{conc} = n * \text{unit_scale_factor} / (N_A * \text{vol})$$
- **concInit**: Initial concentration of the molecules in the pool. 'conc' is set to this value when the 'reinit' operation is done. $\text{concInit} = n\text{Init} * \text{unit_scale_factor} / (N_A * \text{vol})$
- **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_{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.

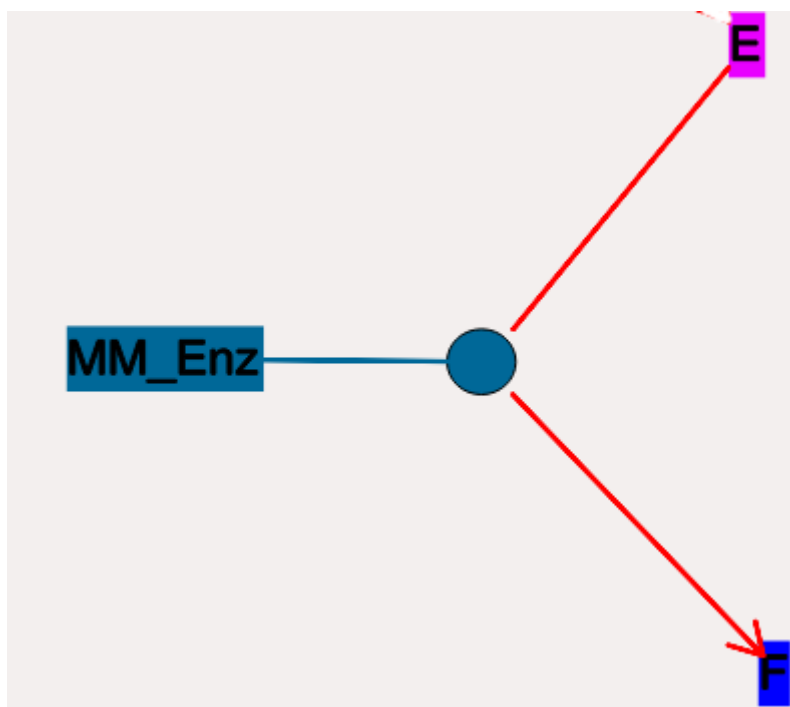
Nominally these enzymes model the same chemical equation as the mass-action enzyme':



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





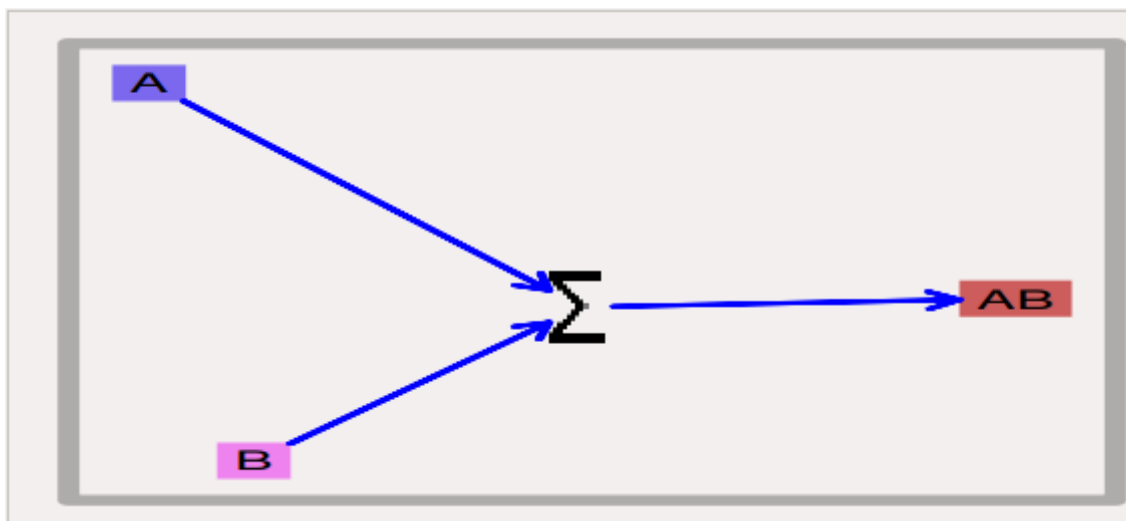
a straight line of the same color. The ellipse icon sits on a continuous, typically curved arrow in red, from the substrate to the product. A given enzyme pool can have any number of enzyme activities, since the same enzyme might catalyze many reactions.

- **Moving Enzymes:** Click and drag.
- **Enzyme editable parameters:**
 - **name:** Name of enzyme.
 - **K_m :** Michaelis-Menten value for enzyme, in 'concentration' units.
 - **k_{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 output's 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:
 1. 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.
 2. Gillespie: Optimized Gillespie stochastic systems algorithm, custom implementation. This uses variable timesteps internally. 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.
 3. Exponential Euler: This method computes the solution of partial and ordinary differential equations.

Model building

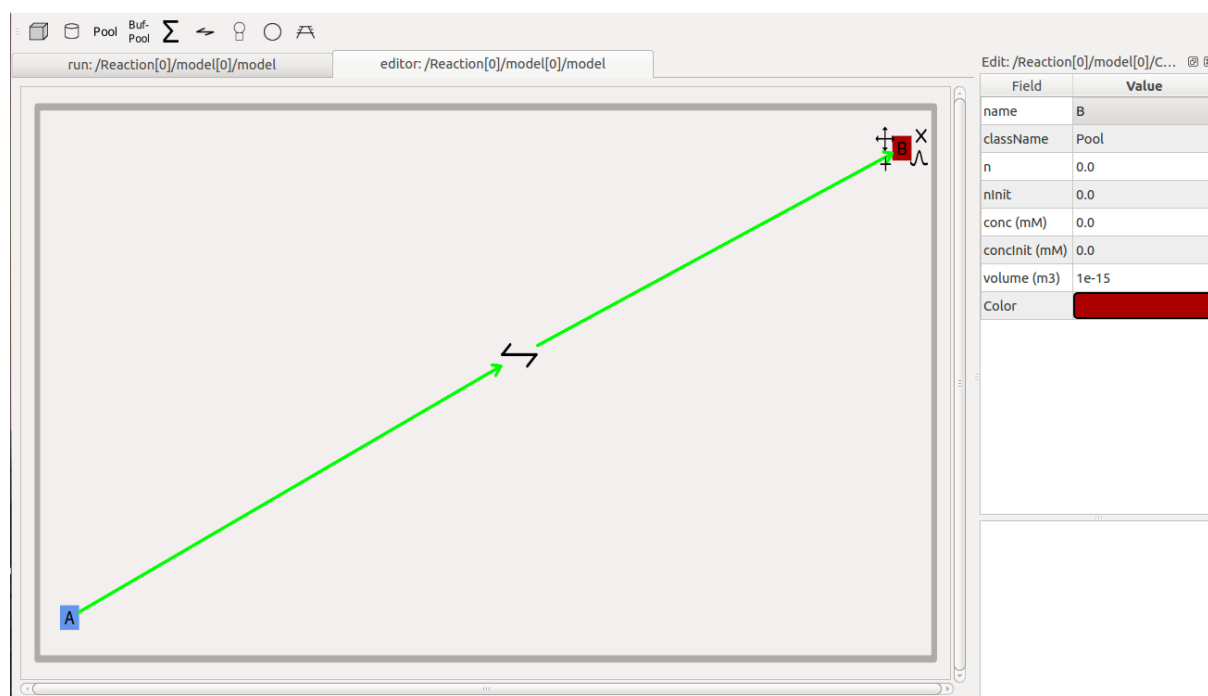
- The **Edit Widget** includes various menu options and model icons on the top. Use the mouse button to click and drag icons from toolbar to Edit Widget, two things will happen, **icon** will appear in the editor widget and an **object editor** will pop up with lots of parameters with respect to moose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

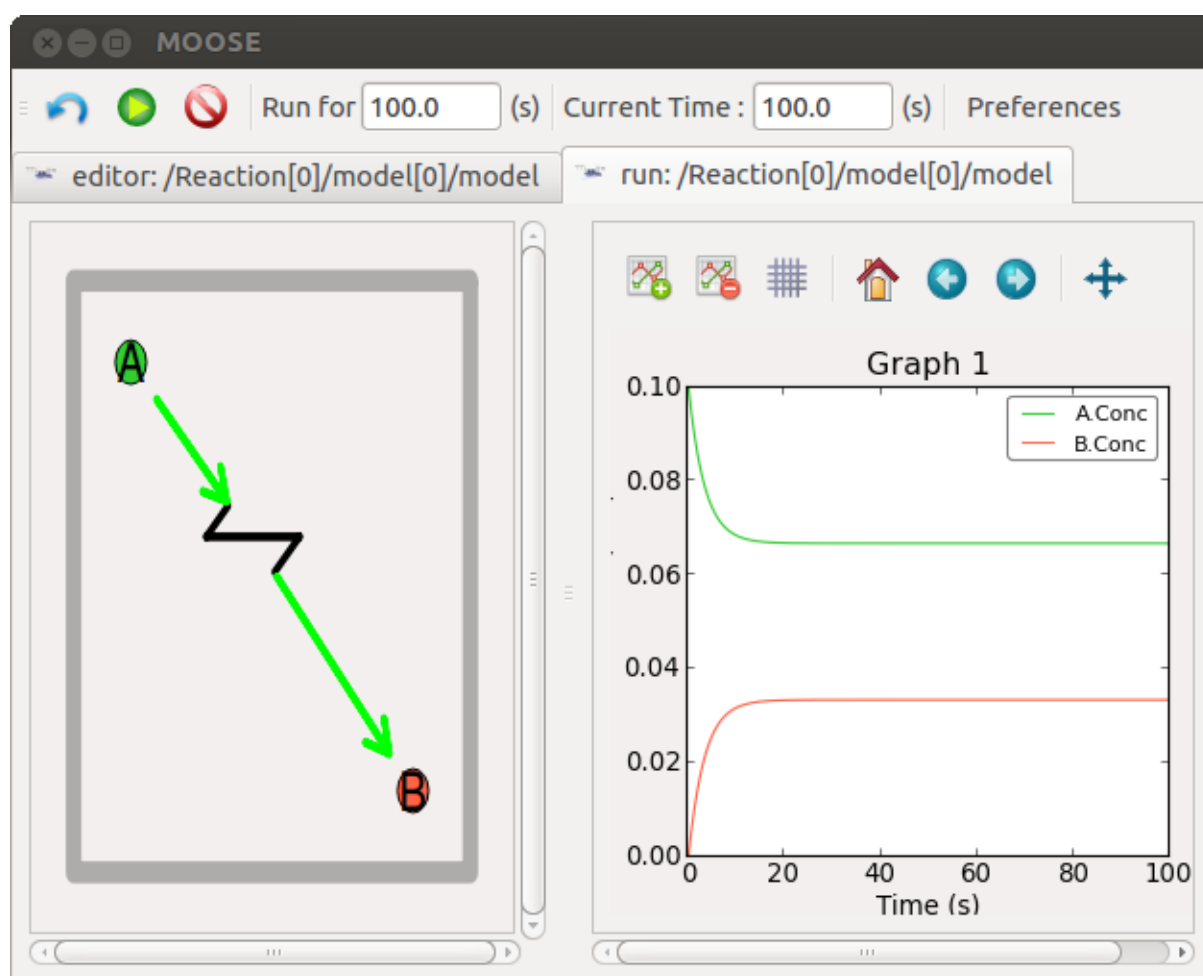
- From **run widget**, pools are draggable to plot window for plotting. (Currently **conc** is plotted as default field) Plots are color-coded as per in model.
- Model can be run by clicking **start** button. One can stop button in mid-stream and start up again without affecting the calculations. The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

4.2.2 Load - Run - Save models

Each of the following examples can be run by clicking on the green source button on the right side of each example, and running from within a **.py** python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in the main moose directory. They can be found under

(continues on next page)



(continued from previous page)

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

```
$ python filename.py
```

in your command line, where filename.py is the python file you want to run.

All of the following examples show one or more methods within each python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.main()` methods.

The filename is the bit that comes before the `.` in the blue boxes, with `.py` added at the end of it. In this case, the file name would be `cubeMeshSigNeur.py`.

Load a Kinetic Model

Load an SBML Model

Load a CSpace Model

Save a model into SBML format

Save a model

4.2.3 Simple Examples

Each of the following examples can be run by clicking on the green source button on the right side of each example, and running from within a python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in the main moose directory. They can be found under

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

(continues on next page)

(continued from previous page)

```
$ python filename.py
```

in your command line, where filename.py is the python file you want to run.

All of the following examples show one or more methods within each python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.main()` methods.

The filename is the bit that comes before the `.` in the blue boxes, with `.py` added at the end of it. In this case, the file name would be `cubeMeshSigNeur.py`.

Set-up a kinetic solver and model

with Scripting

With something else

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: K_f and K_b .
 - **Enz**ymes have two primary parameters k_{cat} and K_m . That is enough for **MMenz**ymes. Regular **Enz**ymes have an additional parameter k_2 which by default is set to 4.0 times k_{cat} , 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.

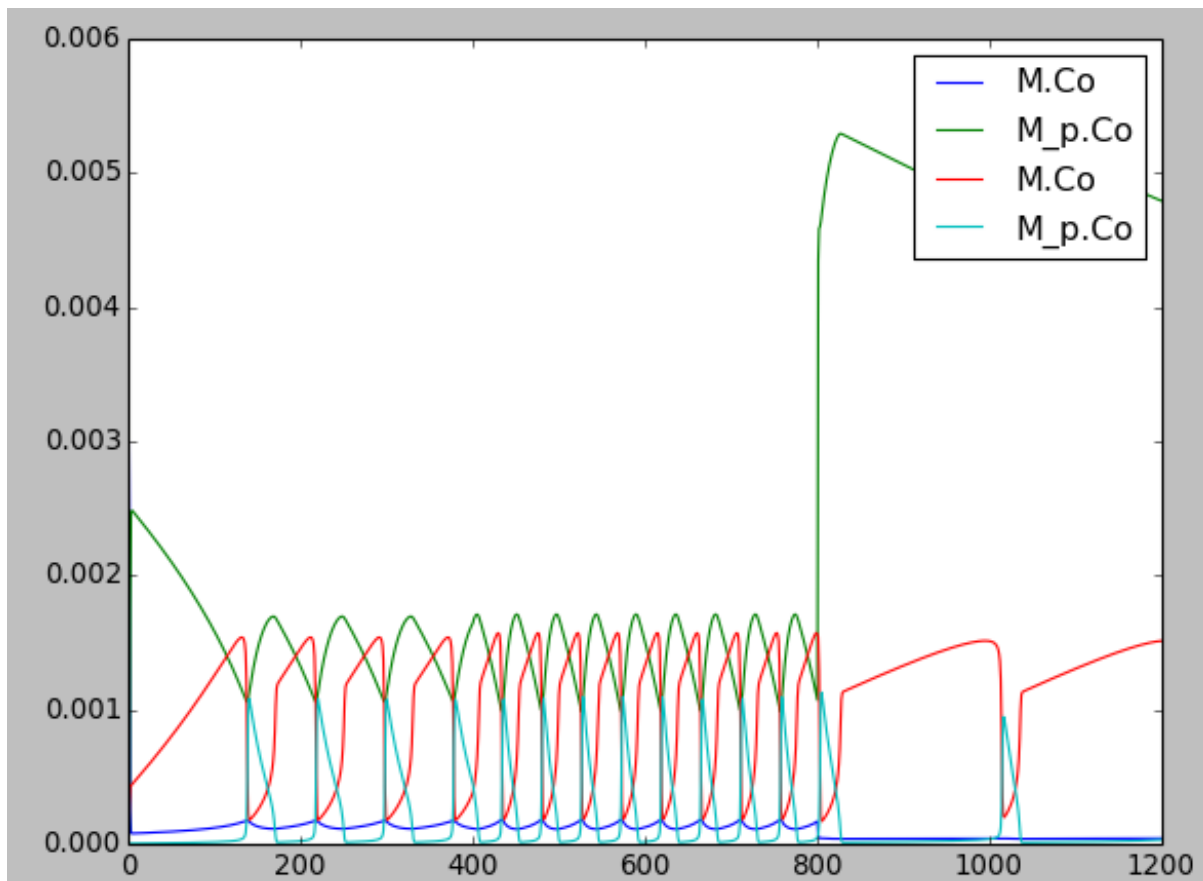
An example of manipulating the models is as following:

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 **ospace** (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.



Tweaking Parameters

Models' Demonstration

Oscillation Model

Bistability Models

MAPK feedback loop model

Simple minimal bistable model

Strongly bistable Model

Model of bidirectional synaptic plasticity

[showing bistable chemical switch]

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]

Neuronal Diffusion Reaction

A Turing Model

Reaction Diffusion in Neurons

Manipulating Chemical Models

Running with different numerical methods

Changing volumes

Feeding tabulated input to a model

Finding steady states

Making a dose-response curve

Transport in branching dendritic tree

4.2.4 Tutorials

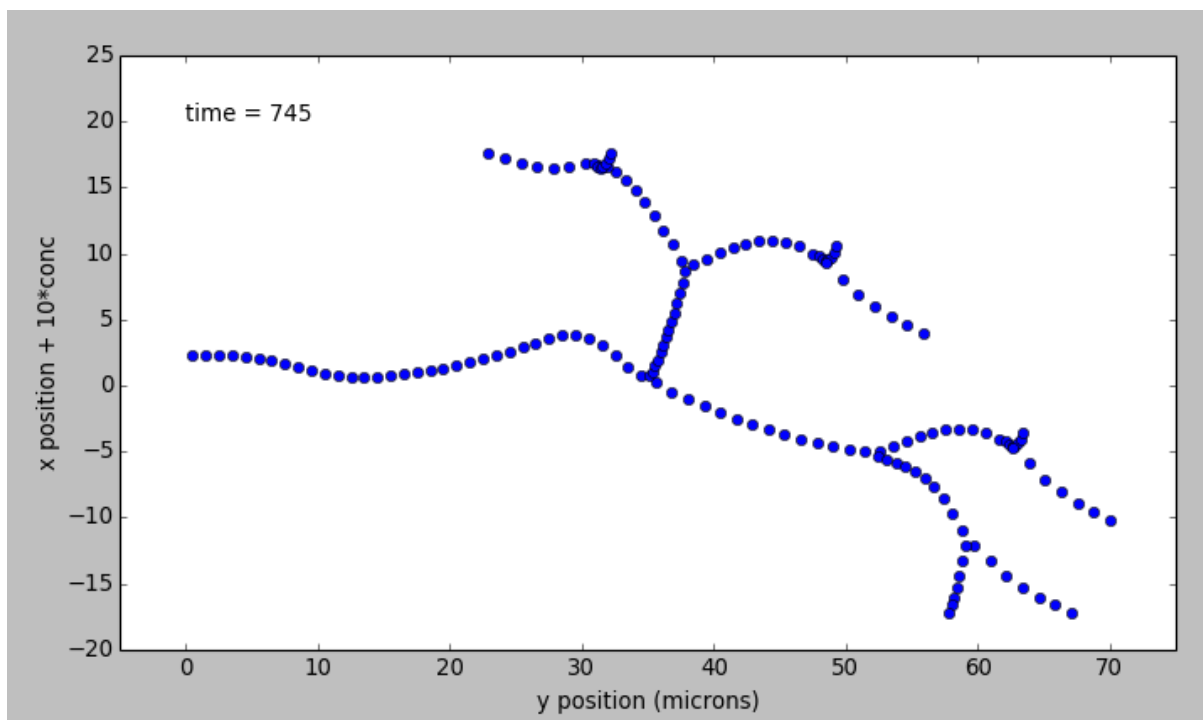
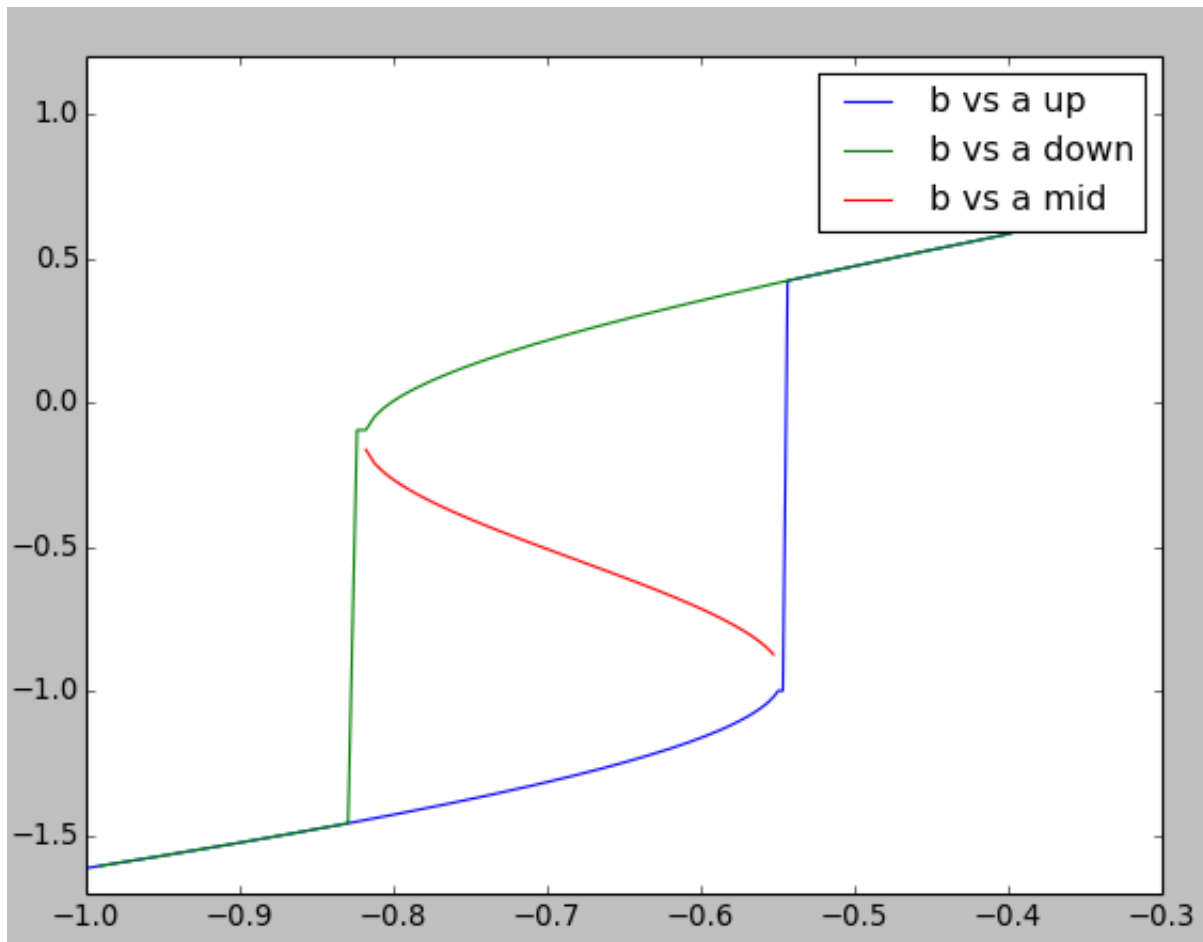
Each of the following examples can be run by clicking on the green ↩source button on the right side of each example, and running from within a .py ↩python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in ↩the main moose directory. They can be found under

(...)/moose/moose-examples/snippets

They can be run by typing

(continues on next page)



(continued from previous page)

```
$ python filename.py
```

in your command line, where filename.py is the python file you want to run.

All of the following examples show one or more methods within each python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.main()` methods.

The filename is the bit that comes before the `.` in the blue boxes, with `.py` added at the end of it. In this case, the file name would be `cubeMeshSigNeur.py`.

Finding Steady State (CSpace)

Define a kinetic model using the scripting

4.3 Networking

4.3.1 Simple Examples

Each of the following examples can be run by clicking on the green source button on the right side of each example, and running from within a python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in the main moose directory. They can be found under

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

```
$ python filename.py
```

in your command line, where filename.py is the python file you want to run.

All of the following examples show one or more methods within each python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs

(continues on next page)

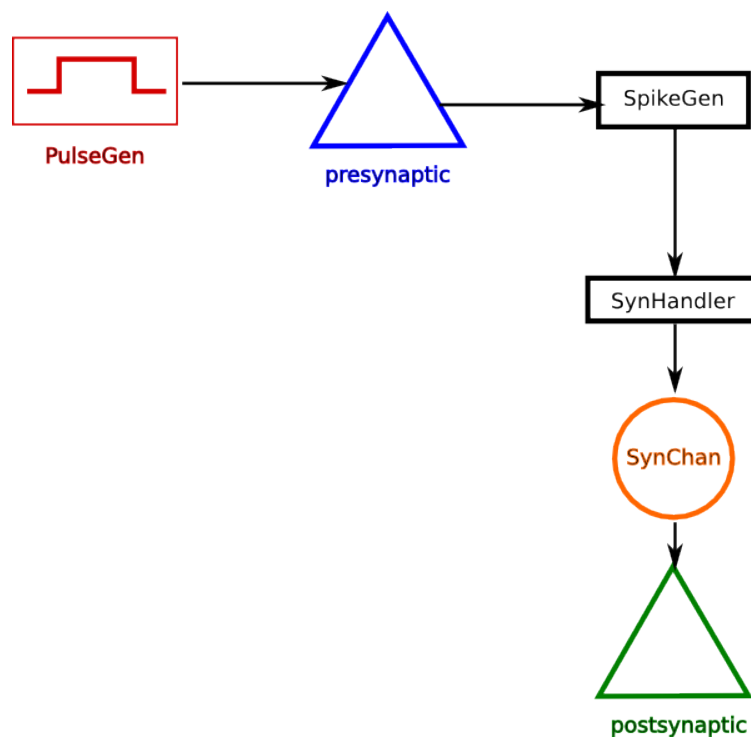
(continued from previous page)

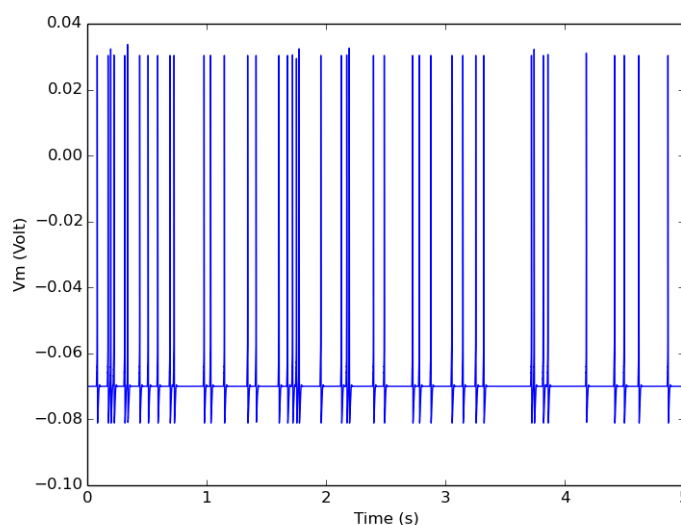
describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.
 ↳main()
 methods.`

The filename is the bit that comes before the `.` in the blue boxes, `↳`
 ↳with
`.py` added at the end of it. In this case, the file name would be
`cubeMeshSigNeur.py`.

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-compartment with Hodgkin-Huxley type Na⁺ and K⁺ channels.





Multi Compartmental Leaky Neurons

Providing random input to a cell

Plastic synapse

Synapse Handler for Spikes



Recurrent integrate-and-fire network


Recurrent integrate-and-fire network with plasticity

Demonstration Models

Building Models

4.3.2 Tutorials

Each of the following examples can be run by clicking on the green  source button on the right side of each example, and running from within a `.py`  python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in  the main moose directory. They can be found under

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

```
$ python filename.py
```

(continues on next page)

(continued from previous page)

in your command line, where filename.py is the python file you want
 ↳to run.

All of the following examples show one or more methods within each
 ↳python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.
 ↳main()` methods.

The filename is the bit that comes before the `.` in the blue boxes,
 ↳with `.py` added at the end of it. In this case, the file name would be `cubeMeshSigNeur.py`.

Network with Ca-based plasticity

4.4 MultiScale Modeling

4.4.1 Simple Examples

Each of the following examples can be run by clicking on the green
 ↳source button on the right side of each example, and running from within a `.py
 ↳python` file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in
 ↳the main moose directory. They can be found under

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

```
$ python filename.py
```

in your command line, where filename.py is the python file you want
 ↳to run.

All of the following examples show one or more methods within each
 ↳python file.

For example, in the `cubeMeshSigNeur` section, there are two blue tabs describing the `cubeMeshSigNeur.createSquid()` and `cubeMeshSigNeur.
 ↳main()` methods.

The filename is the bit that comes before the `.` in the blue boxes,
 ↳with

(continues on next page)

(continued from previous page)

`.py` added at the end of it. In this case, the file name would be `cubeMeshSigNeur.py`.

Single-compartment multiscale model

Multi compartment Single Neuron Model

Modeling chemical reactions in neurons

5.1 Rdesigneur: Building multiscale models

Author: Upi Bhalla

Date: Aug 26 2016,

Last-Updated: Nov 08 2018

By: Upi Bhalla

5.1.1 Contents

Contents

- *Rdesigneur: Building multiscale models* (page 61)
 - *Contents* (page 61)
 - *Introduction* (page 63)
 - *Rdesigneur examples* (page 64)
 - * *Bare Rdesigneur: single passive compartment* (page 64)
 - * *Simulate and display current pulse to soma* (page 65)
 - * *Simulate and display voltage clamp stimulus to soma* (page 66)
 - * *HH Squid model in a single compartment* (page 67)
 - * *HH Squid model with voltage clamp* (page 71)
 - * *HH Squid model in an axon* (page 71)
 - * *Action potential collision in HH Squid axon model* (page 75)

- * *HH Squid model in a myelinated axon* (page 76)
- * *Alternate (non-squid) way to define soma* (page 78)
- * *Ball-and-stick model of a neuron* (page 78)
- * *Benchmarking simulation speed* (page 81)
- * *Synaptic stimulus: random (Poisson)* (page 84)
- * *Synaptic stimulus: periodic* (page 84)
- * *Reaction system in a single compartment* (page 86)
- * *Reaction-diffusion system* (page 87)
- * *Primer on using the 3-D MOGLI display* (page 90)
- * *Diffusion of a single molecule* (page 91)
- * *Calcium-induced calcium release* (page 92)
- * *Intracellular transport* (page 94)
- * *Travelling oscillator* (page 99)
- * *Bidirectional transport* (page 101)
- * *Controlling a reaction by a function* (page 104)
- * *Multiscale models: single compartment* (page 105)
- * *Multiscale model of CICR in dendrite triggered by synaptic input* (page 108)
- * *Multiscale model spanning PSD, spine head and dendrite* (page 115)
- * *Multiscale model in which spine geometry changes due to signaling* (page 121)
- * *Morphology: Load .swc morphology file and view it* (page 127)
- * *Build an active neuron model by putting channels into a morphology file* (page 129)
- * *Build a spiny neuron from a morphology file and put active channels in it.* (page 131)
- * *Place spines in a spiral along a dendrite* (page 134)
- *Rdesigneur command reference* (page 135)
 - * *turnOffElec* (page 136)
 - * *useGssa* (page 136)
 - * *combineSegments* (page 136)
 - * *stealCellFromLibrary* (page 136)
 - * *verbose* (page 136)
 - * *addSomaChemCompt* (page 137)
 - * *addEndoChemCompt* (page 137)
 - * *diffusionLength* (page 137)
 - * *temperature* (page 137)
 - * *chemDt* (page 137)

- * *diffDt* (page 138)
- * *elecDt* (page 138)
- * *chemPlotDt* (page 138)
- * *elecPlotDt* (page 138)
- * *funcDt* (page 138)
- * *cellProto* (page 139)
- * *spineProto* (page 139)
- * *chanProto* (page 140)
- * *chemProto* (page 142)
- * *passiveDistrib* (page 142)
- * *spineDistrib* (page 143)
- * *chanDistrib* (page 145)
- * *chemDistrib* (page 146)
- * *adaptorList* (page 147)
- * *stimList* (page 147)
- * *plotList* (page 148)
- * *moogList* (page 149)

5.1.2 Introduction

Rdesigneur (Reaction Diffusion and Electrical SIGnaling in NEURons) is an interface to the multiscale modeling capabilities in MOOSE. It is designed to build models incorporating biochemical signaling pathways in dendrites and spines, coupled to electrical events in neurons. Rdesigneur assembles models from predefined parts: it delegates the details to specialized model definition formats. Rdesigneur combines one or more of the following cell parts to build models:

- Neuronal morphology
- Dendritic spines
- Ion channels
- Reaction systems
- Adaptors that couple between these for multiscale models

It also folds in simulation input and output

- Time-series stimuli for molecular concentration change and reaction rates
- Current and voltage clamp
- Synaptic input.
- Time-series plots

- File dumps
- 3-D neuronal graphics

Rdesigneur's main role is to specify how these are put together, including assigning parameters for the model. Using Rdesigneur one can compactly and quickly put together quite complex multiscale models.

5.1.3 Rdesigneur examples

Here we provide a few use cases, building up from a minimal model to a reasonably complete multiscale model spanning chemical and electrical signaling. The files for these examples are also available in `moose-examples/tutorials/Rdesigneur`, and the file names are mentioned as we go along.

Bare Rdesigneur: single passive compartment

ex1_minimalModel.py

If we don't provide any arguments at all to the Rdesigneur, it makes a model with a single passive electrical compartment in the MOOSE path `/model/elec/soma`. Here is how to do this:

```
import moose
import rdesigneur as rd
rdes = rd.rdesigneur()
rdes.buildModel()
```

To confirm that it has made a compartment with some default values we can add a line:

```
moose.showfields( rdes.soma )
```

This should produce the output:

```
[ /model[0]/elec[0]/soma[0] ]
diameter      = 0.0005
fieldIndex    = 0
Ra            = 7639437.26841
y0            = 0.0
Rm            = 424413.177334
index         = 0
numData       = 1
inject        = 0.0
initVm        = -0.065
Em            = -0.0544
y             = 0.0
numField      = 1
path          = /model[0]/elec[0]/soma[0]
dt            = 0.0
tick          = -2
z0            = 0.0
name          = soma
Cm            = 7.85398163398e-09
x0            = 0.0
```

(continues on next page)

(continued from previous page)

```

Vm          = -0.06
className   = ZombieCompartment
idValue     = 465
length      = 0.0005
Im          = 1.3194689277e-08
x           = 0.0005
z           = 0.0

```

Simulate and display current pulse to soma

ex2.0_currentPulse.py

A more useful script would run and display the model. Rdesigneur can help with the stimulus and the plotting. This simulation has the same passive compartment, and current is injected as the simulation runs. This script displays the membrane potential of the soma as it charges and discharges.

```

import moose
import rdesigneur as rd
rdes = rd.rdesigneur(
    stimList = [['soma', '1', '.', 'inject', '(t>0.1 && t<0.2) * 2e-8
→']],
    plotList = [['soma', '1', '.', 'Vm', 'Soma membrane potential']],
)
rdes.buildModel()
moose.reinit()
moose.start( 0.3 )
rdes.display()

```

The *stimList* defines a stimulus. Each entry has five arguments:

```

[region_in_cell, region_expression, moose_object, parameter, _
→expression_string]

```

- `region_in_cell` specifies the objects to stimulate. Here it is just the soma.
- `region_expression` specifies a geometry based calculation to decide whether to apply the stimulus. The value must be >0 for the stimulus to be present. Here it is just 1. `moose_object` specifies the simulation object to operate upon during the stimulus. Here the `.` means that it is the soma itself. In other models it might be a channel on the soma, or a synapse, and so on.
- `parameter` specifies the simulation parameter on the moose object that the stimulus will modify. Here it is the injection current to the soma compartment.
- `expression_string` calculates the value of the parameter, typically as a function of time. Here we use the function $(t > 0.1 \ \&\& \ t < 0.2) * 2e-8$ which evaluates as $2e-8$ between the times of 0.1 and 0.2 seconds.

To summarise this, the *stimList* here means *inject a current of 20nA to the soma between the times of 0.1 and 0.2 s.*

The *plotList* defines what to plot. It has a similar set of arguments:

```
[region_in_cell, region_expression, moose_object, parameter, title_of_
  ↪plot]
```

These mean the same thing as for the stimList except for the title of the plot.

The *rdes.display()* function causes the plots to be displayed.

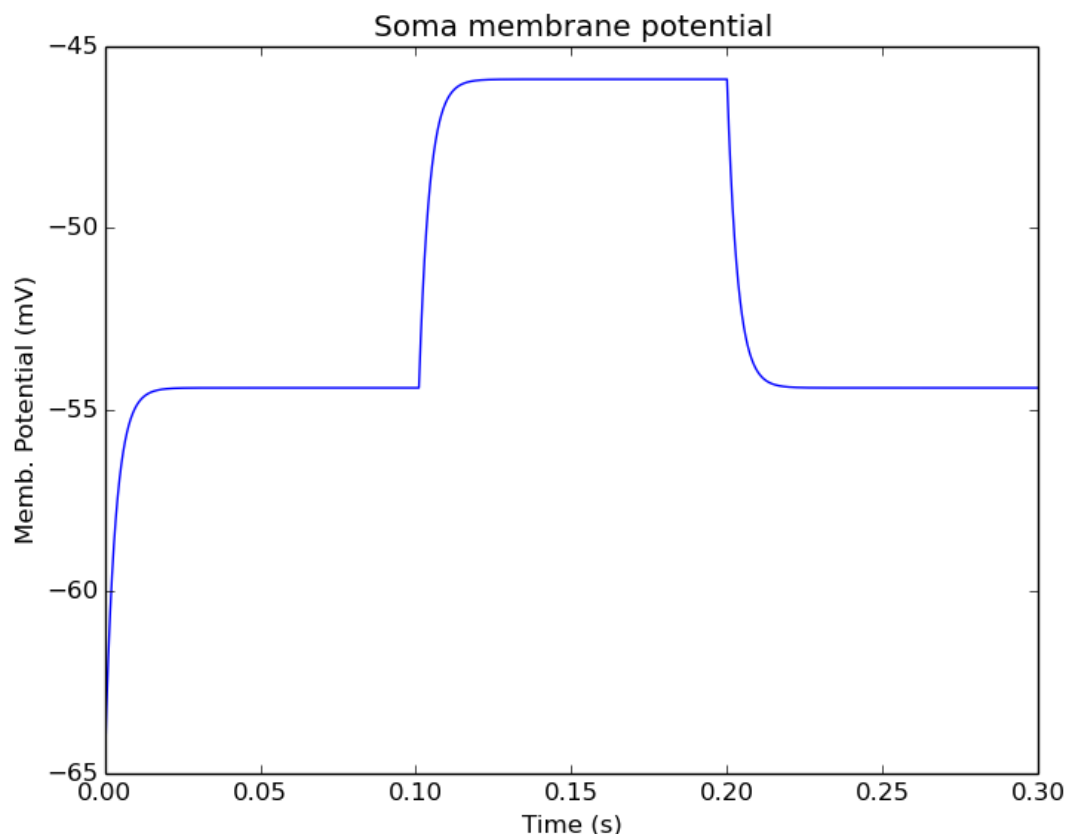


Fig. 1: Plot for current input to passive compartment

When we run this we see an initial depolarization as the soma settles from its initial -65 mV to a resting $E_m = -54.4$ mV. These are the original HH values, see the example above. At $t = 0.1$ seconds there is another depolarization due to the current injection, and at $t = 0.2$ seconds this goes back to the resting potential.

Simulate and display voltage clamp stimulus to soma

ex2.1_vclamp.py

This model introduces the voltage clamp stimulus on a passive compartment. As before, we add a few lines to define the stimulus and plot. This script displays both the membrane potential, and the holding current of the voltage clamp circuit as it charges and discharges the passive compartment model.

```
import moose
import rdesigneur as rd
```

(continues on next page)

(continued from previous page)

```

rdes = rd.rdesigneur(
    stimList = [['soma', '1', '.', 'vclamp', '-0.065 + (t>0.1 && t<0.
→2) * 0.02' ]],
    plotList = [
        ['soma', '1', '.', 'Vm', 'Soma membrane potential'],
        ['soma', '1', 'vclamp', 'current', 'Soma holding current'],
    ]
)
rdes.buildModel()
moose.reinit()
moose.start( 0.3 )
rdes.display()

```

Here the *stimList* line tells the system to deliver a voltage clamp (vclamp) on the soma, starting at -65 mV and jumping up by 20 mV between 0.1 and 0.2 seconds. The *plotList* now includes two entries, and will generate two plots. The first is for plotting the soma membrane potential, just to be sure that the voltage clamp is doing its job.

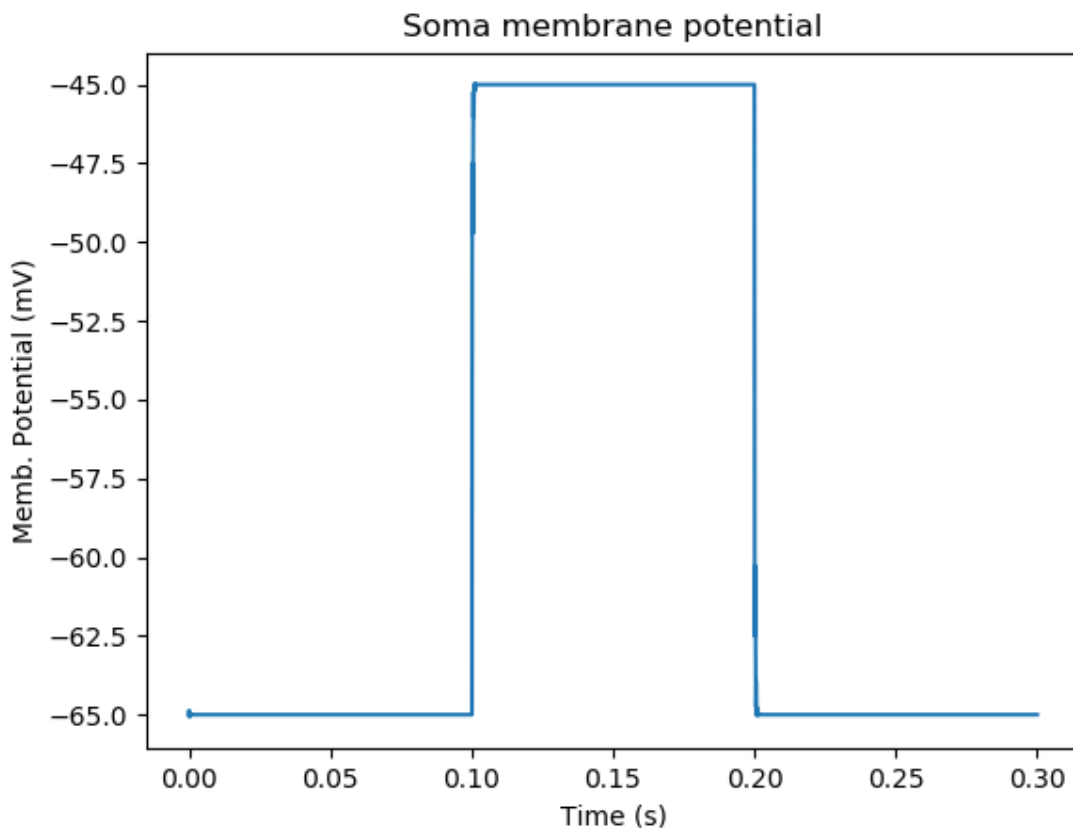


Fig. 2: Plot for membrane potential in voltage clamp

The second graph plots the holding current. Note the capacitive transients.

HH Squid model in a single compartment

ex3.0_squid_currentPulse.py

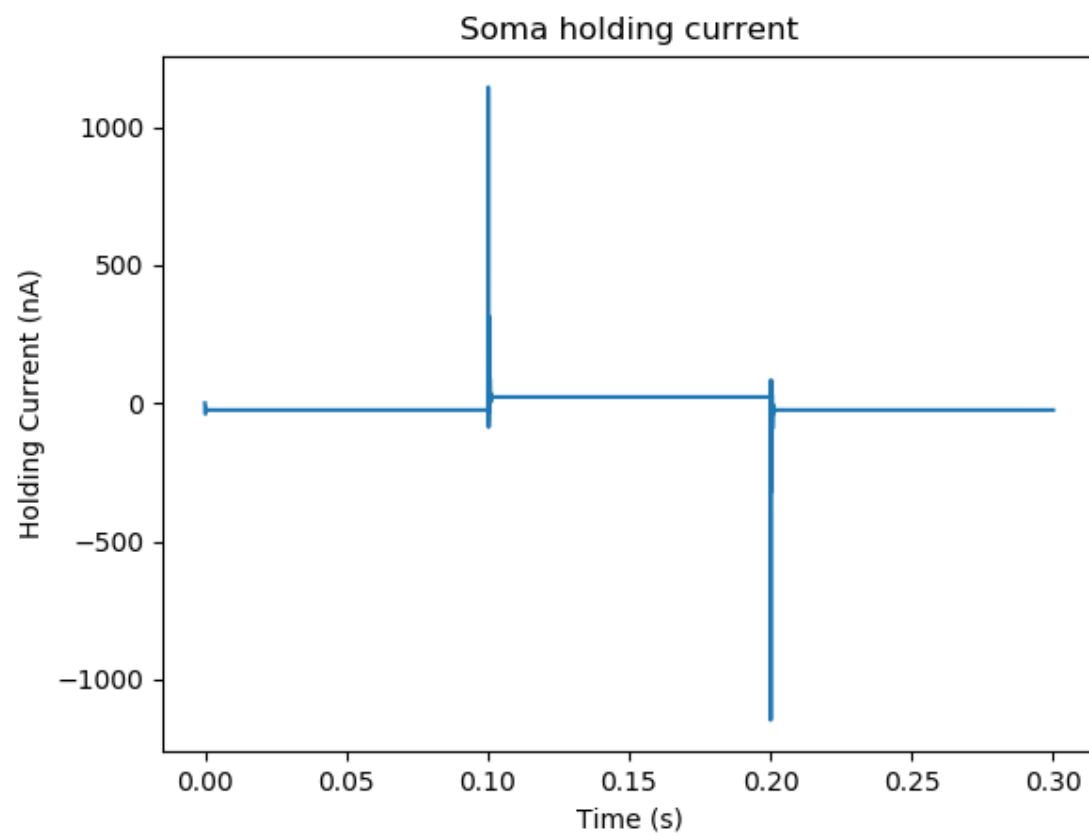


Fig. 3: Plot for holding current for voltage clamp

Here we put the Hodgkin-Huxley squid model channels into a passive compartment. The HH channels are predefined as prototype channels for Rdesigneur,

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '1200' ],
        ['K', 'soma', 'Gbar', '360' ]],
    stimList = [['soma', '1', '.', 'inject', '(t>0.1 && t<0.2) * 1e-8
→ ' ]],
    plotList = [['soma', '1', '.', 'Vm', 'Membrane potential']]
)

rdes.buildModel()
moose.reinit()
moose.start( 0.3 )
rdes.display()
```

Here we introduce two new model specification lines:

- **chanProto:** This specifies which ion channels will be used in the model. Each entry here has two fields: the source of the channel definition, and (optionally) the name of the channel. In this example we specify two channels, an Na and a K channel using the original Hodgkin-Huxley parameters. As the source of the channel definition we use the name of the Python function that builds the channel. The *make_HH_Na()* and *make_HH_K()* functions are predefined but we can also specify our own functions for making prototypes. We could also have specified the channel prototype using the name of a channel definition file in ChannelML (a subset of NeuroML) format.
- **chanDistrib:** This specifies *where* the channels should be placed over the geometry of the cell. Each entry in the chanDistrib list specifies the distribution of parameters for one channel using four entries:

```
[object_name, region_in_cell, parameter, expression_string]
```

In this case the job is almost trivial, since we just have a single compartment named *soma*. So the line

```
['Na', 'soma', 'Gbar', '1200' ]
```

means *Put the Na channel in the soma, and set its maximal conductance density (Gbar) to 1200 Siemens/m².*

As before we apply a somatic current pulse. Since we now have HH channels in the model, this generates action potentials.

There are several interesting things to do with the model by varying stimulus parameters:

- Change injection current.
- Put in a protocol to get rebound action potential.
- Put in a current ramp, and run it for a different duration
- Put in a frequency chirp, and see how the squid model is tuned to a certain frequency range.

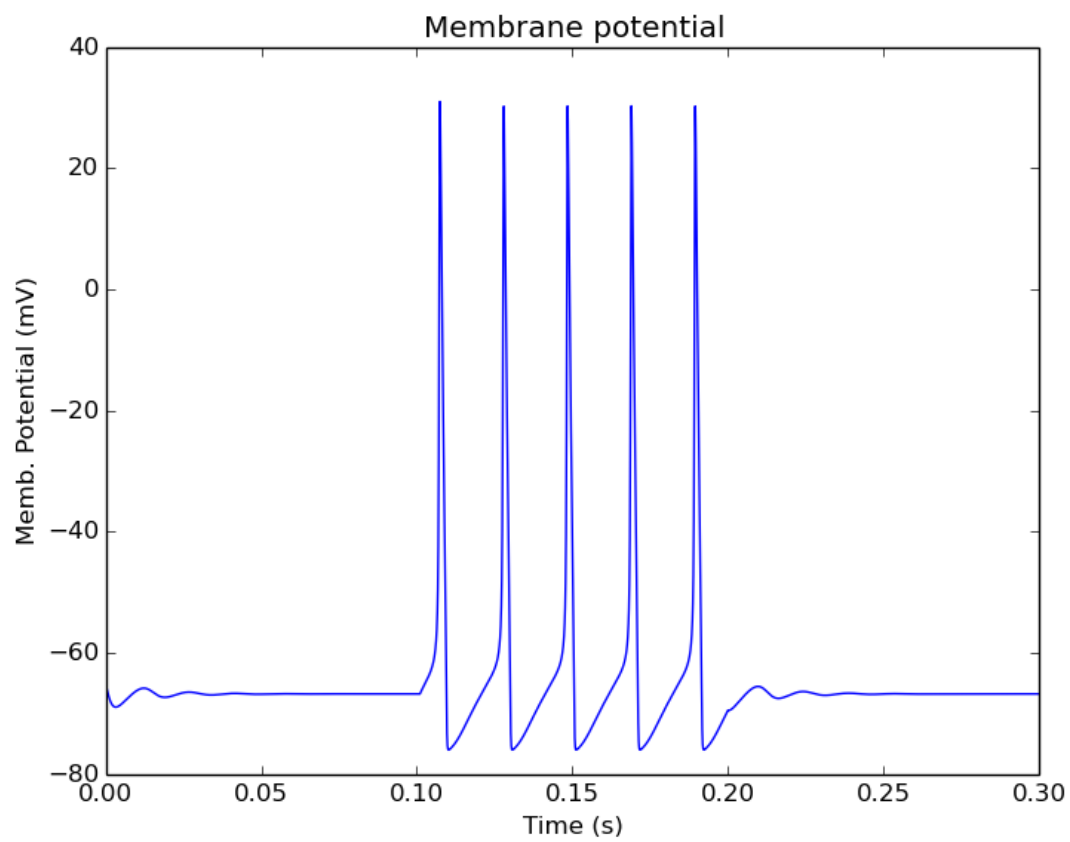


Fig. 4: Plot for HH squid simulation

- Modify channel or passive parameters. See if it still fires.
- Try the frequency chirp on the cell with parameters changed. Does the tuning change?

HH Squid model with voltage clamp

ex3.1_squid_vclamp.py

This is the same squid model, but now we add a voltage clamp to the squid and monitor the holding current. This stimulus line is identical to ex2.1.

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '1200' ],
        ['K', 'soma', 'Gbar', '360' ]],
    stimList = [['soma', '1', '.', 'vclamp', '-0.065 + (t>0.1 && t<0.
↪2) * 0.02' ]],
    plotList = [
        ['soma', '1', '.', 'Vm', 'Membrane potential'],
        ['soma', '1', 'vclamp', 'current', 'Soma holding current']
    ]
)
rdes.buildModel()
moose.reinit()
moose.start( 0.3 )
rdes.display()
```

Here we see the classic HH current response, a downward brief deflection due to the Na channel, and a slower upward sustained current due to the K delayed rectifier.

Here are some suggestions for further exploration:

- Monitor individual channel currents through additional plots.
- Convert this into a voltage clamp series. Easiest way to do this is to complete the `rdes.BuildModel`, then delete the Function object on the `/model/elec/soma/vclamp`. Now you can simply set the 'command' field of the `vclamp` in a for loop, going from -ve to +ve voltages. Remember, SI units. You may wish to capture the plot vectors each cycle. The plot vectors are accessed by something like

```
moose.element( '/model/graphs/plot1' ).vector
```

HH Squid model in an axon

ex3.2_squid_axon_propgn.py

Here we put the Hodgkin-Huxley squid model into a long compartment that is subdivided into many segments, so that we can watch action potentials propagate. Most of this example is boilerplate code to build a spiral axon. There is a short `rdesigneur` segment that takes the spiral axon prototype and populates it with channels, and sets up the display. Later examples will show you how to read morphology files to specify the neuronal geometry.

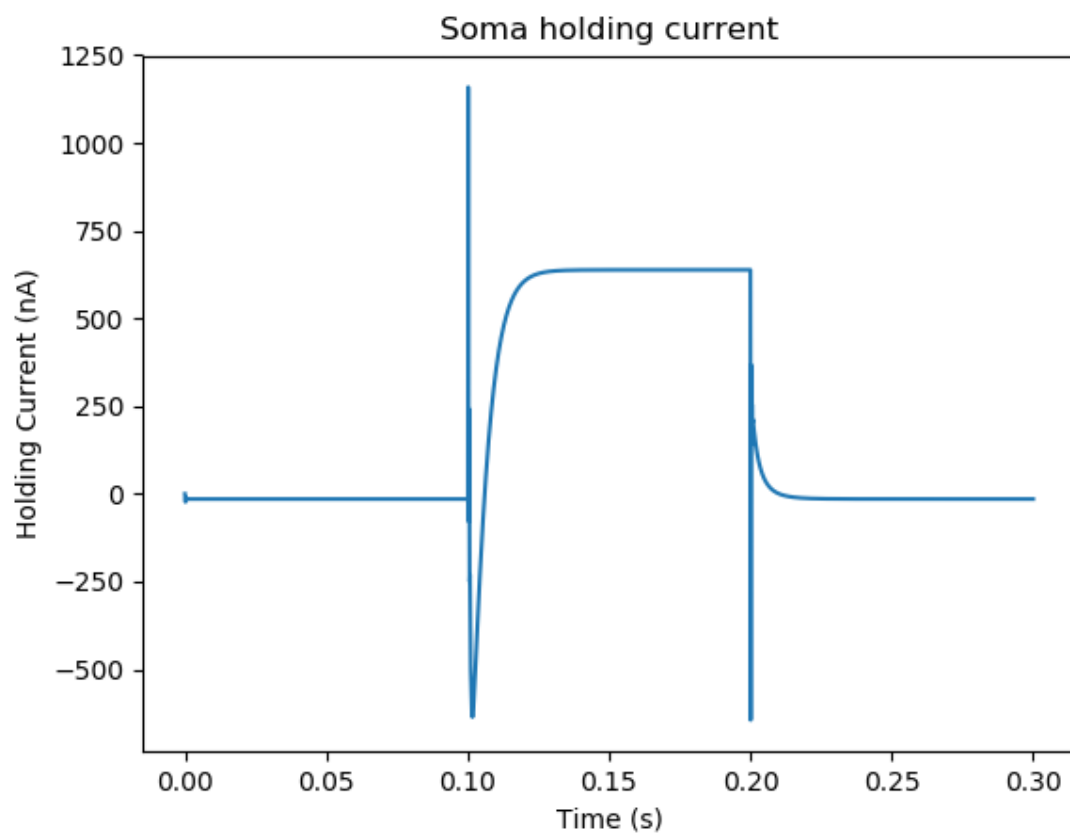


Fig. 5: Plot for HH squid voltage clamp pulse.

```

import numpy as np
import moose
import pylab
import rdesignneur as rd

numAxonSegments = 200
comptLen = 10e-6
comptDia = 1e-6
RM = 1.0
RA = 10.0
CM = 0.01

def makeAxonProto():
    axon = moose.Neuron( '/library/axon' )
    prev = rd.buildCompt( axon, 'soma', RM = RM, RA = RA, CM = CM,
    ↪ dia = 10e-6, x=0, dx=comptLen)
    theta = 0
    x = comptLen
    y = 0.0

    for i in range( numAxonSegments ):
        dx = comptLen * np.cos( theta )
        dy = comptLen * np.sin( theta )
        r = np.sqrt( x * x + y * y )
        theta += comptLen / r
        compt = rd.buildCompt( axon, 'axon' + str(i), RM = RM, RA_
    ↪ RA, CM = CM, x = x, y = y, dx = dx, dy = dy, dia = comptDia )
        moose.connect( prev, 'axial', compt, 'raxial' )
        prev = compt
        x += dx
        y += dy

    return axon

moose.Neutral( '/library' )
makeAxonProto()

rdes = rd.rdesignneur(
    chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
    cellProto = [['elec', 'axon']],
    chanDistrib = [
        ['Na', '#', 'Gbar', '1200' ],
        ['K', '#', 'Gbar', '360' ]],
    stimList = [['soma', '1', '.', 'inject', '(t>0.01 && t<0.2) *
    ↪ 2e-11' ]],
    plotList = [['soma', '1', '.', 'Vm', 'Membrane potential']],
    moogList = [['#', '1', '.', 'Vm', 'Vm (mV)']]
)

rdes.buildModel()
moose.reinit()

rdes.displayMoogli( 0.00005, 0.05, 0.0 )

```

Note how we explicitly create the prototype axon on `'/library'`, and then specify it using the `cellProto` line in the `rdesignneur`. The `moogList` specifies the 3-D display. See below for how to set up and use these displays.

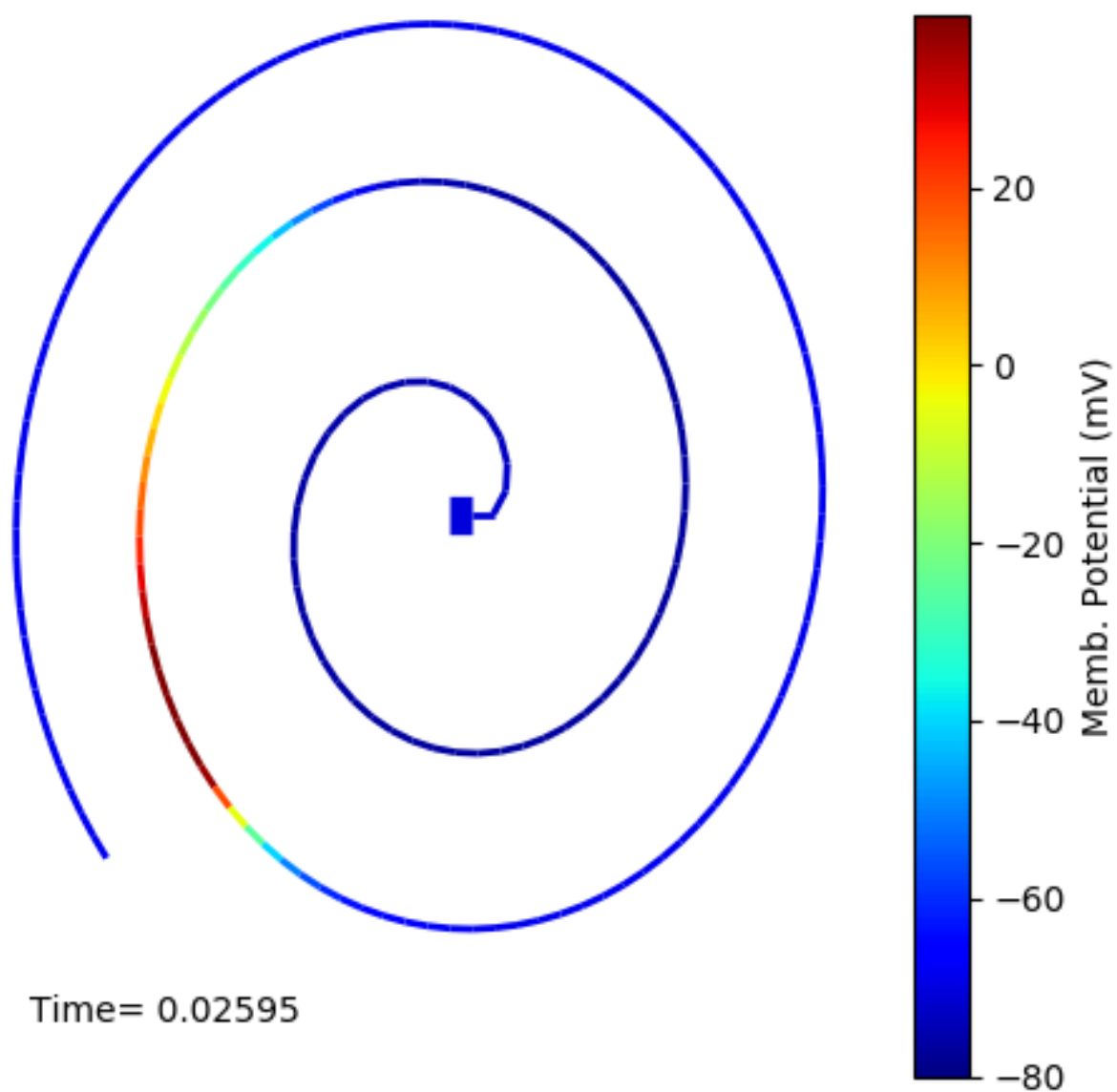


Fig. 6: Axon with propagating action potential

Action potential collision in HH Squid axon model

ex3.3_AP_collision.py

This is identical to the previous example, except that now we deliver current injection at at two points, the soma and a point along the axon. The modified stimulus line is:

```
...
stimList = [['soma', '1', '.', 'inject', '(t>0.01 && t<0.2) * 2e-11' ↵
↵],
['axon100', '1', '.', 'inject', '(t>0.01 && t<0.2) * 3e-11' ]],
...
```

Watch how the AP is triggered bidirectionally from the stimulus point on the 100th segment of the axon, and observe what happens when two action potentials bump into each other.

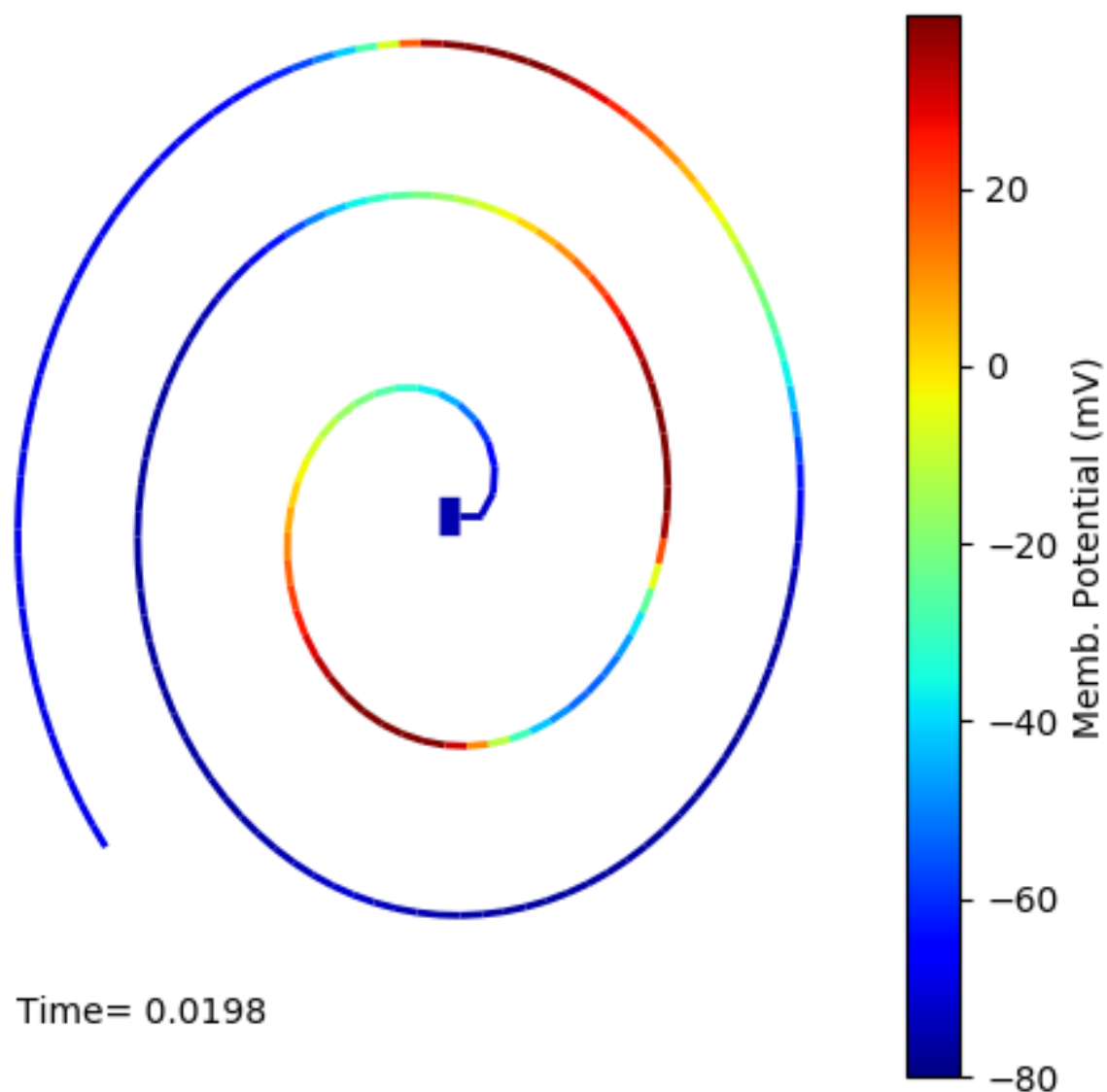


Fig. 7: Colliding action potentials

HH Squid model in a myelinated axon

ex3.4_myelinated_axon.py

This is a curious cross-species chimera model, where we embed the HH equations into a myelinated example model. As for the regular axon above, most of the example is boilerplate setup code. Note how we restrict the HH channels to the nodes of Ranvier using a conditional test for the diameter of the axon segment.

```
import numpy as np
import moose
import pylab
import rdesigneur as rd

numAxonSegments = 405
nodeSpacing = 100
comptLen = 10e-6
comptDia = 2e-6 # 2x usual
RM = 100.0 # 10x usual
RA = 5.0
CM = 0.001 # 0.1x usual

nodeDia = 1e-6
nodeRM = 1.0
nodeCM = 0.01

def makeAxonProto():
    axon = moose.Neuron( '/library/axon' )
    x = 0.0
    y = 0.0
    prev = rd.buildCompt( axon, 'soma', RM = RM, RA = RA, CM = CM,
    ↪ dia = 10e-6, x=0, dx=comptLen)
    theta = 0
    x = comptLen

    for i in range( numAxonSegments ):
        r = comptLen
        dx = comptLen * np.cos( theta )
        dy = comptLen * np.sin( theta )
        r = np.sqrt( x * x + y * y )
        theta += comptLen / r
        if i % nodeSpacing == 0:
            compt = rd.buildCompt( axon, 'axon' + str(i), RM = nodeRM,
    ↪ RA = RA, CM = nodeCM, x = x, y = y, dx = dx, dy = dy, dia =
    ↪ nodeDia )
        else:
            compt = rd.buildCompt( axon, 'axon' + str(i), RM = RM, RA
    ↪ = RA, CM = CM, x = x, y = y, dx = dx, dy = dy, dia = comptDia )
            moose.connect( prev, 'axial', compt, 'raxial' )
            prev = compt
            x += dx
            y += dy

    return axon

moose.Neutral( '/library' )
makeAxonProto()
```

(continues on next page)

(continued from previous page)

```

rdes = rd.rdesigneur(
  chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
  cellProto = [['elec', 'axon']],
  chanDistrib = [
    ['Na', '#', 'Gbar', '12000 * (dia < 1.5e-6)'],
    ['K', '#', 'Gbar', '3600 * (dia < 1.5e-6)']],
  stimList = [['soma', '1', '.', 'inject', '(t>0.01 && t<0.2) * 1e-
→10' ]],
  plotList = [['soma,axon100,axon200,axon300,axon400', '1', '.', 'Vm
→', 'Membrane potential']],
  moogList = [['#', '1', '.', 'Vm', 'Vm (mV)']]
)

rdes.buildModel()

for i in moose.wildcardFind( "/model/elec/#/Na" ):
  print i.parent.name, i.Gbar

moose.reinit()

rdes.displayMoogli( 0.00005, 0.05, 0.0 )

```

When you run the example, keep an eye out for a few things:

- **saltatory conduction:** This is the way the action potential jumps from one node of Ranvier to the next. Between the nodes it is just passive propagation.
- **Failure to propagate:** Observe that the second and fourth action potentials fails to trigger propagation along the axon. Here we have specially tuned the model properties so that this happens. With a larger RA of 10.0, the model will be more reliable.
- **Speed:** Compare the propagation speed with the previous, unmyelinated axon. Note that the current model is larger!

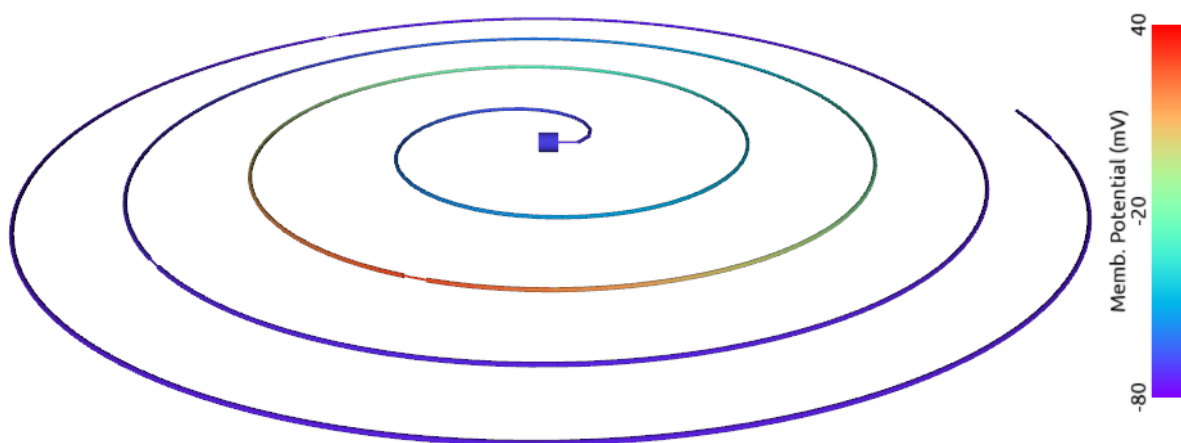


Fig. 8: Myelinated axon with propagating action potential

Alternate (non-squid) way to define soma

ex4.0_scaledSoma.py

The default HH-squid axon is not a very convincing soma. Rdesigneur offers a somewhat more general way to define the soma in the cell prototype line.

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    # cellProto syntax: ['somaProto', 'name', dia, length]
    cellProto = [['somaProto', 'soma', 20e-6, 200e-6]],
    chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '1200' ],
        ['K', 'soma', 'Gbar', '360' ]],
    stimList = [['soma', '1', '.', 'inject', '(t>0.01 && t<0.05) * 1e-
    ↪9' ]],
    plotList = [['soma', '1', '.', 'Vm', 'Membrane potential']],
    moogList = [['#', '1', '.', 'Vm', 'Vm (mV)']]
)

rdes.buildModel()
soma = moose.element( '/model/elec/soma' )
print( 'Soma dia = {}, length = {}'.format( soma.diameter, soma.
    ↪length ) )
moose.reinit()

rdes.displayMoogli( 0.0005, 0.06, 0.0 )
```

Here the crucial line is the *cellProto* line. There are four arguments here:

```
['somaProto', 'name', dia, length]
```

- The first argument tells the system to use a prototype soma, that is a single cylindrical compartment.
- The second argument is the name to give the cell.
- The third argument is the diameter. Note that this is a double, not a string.
- The fourth argument is the length of the cylinder that makes up the soma. This too is a double, not a string. The cylinder is oriented along the x axis, with one end at (0,0,0) and the other end at (length, 0, 0).

This is what the soma looks like:

It a somewhat elongated soma, being a cylinder 10 times as long as it is wide.

Ball-and-stick model of a neuron

ex4.1_ballAndStick.py

A somewhat more electrically reasonable model of a neuron has a soma and a single dendrite, which can itself be subdivided into segments so that it can exhibit voltage gradients, have channel and receptor distributions, and so on. This is accomplished in *rdesigneur* using a variant of the *cellProto* syntax.

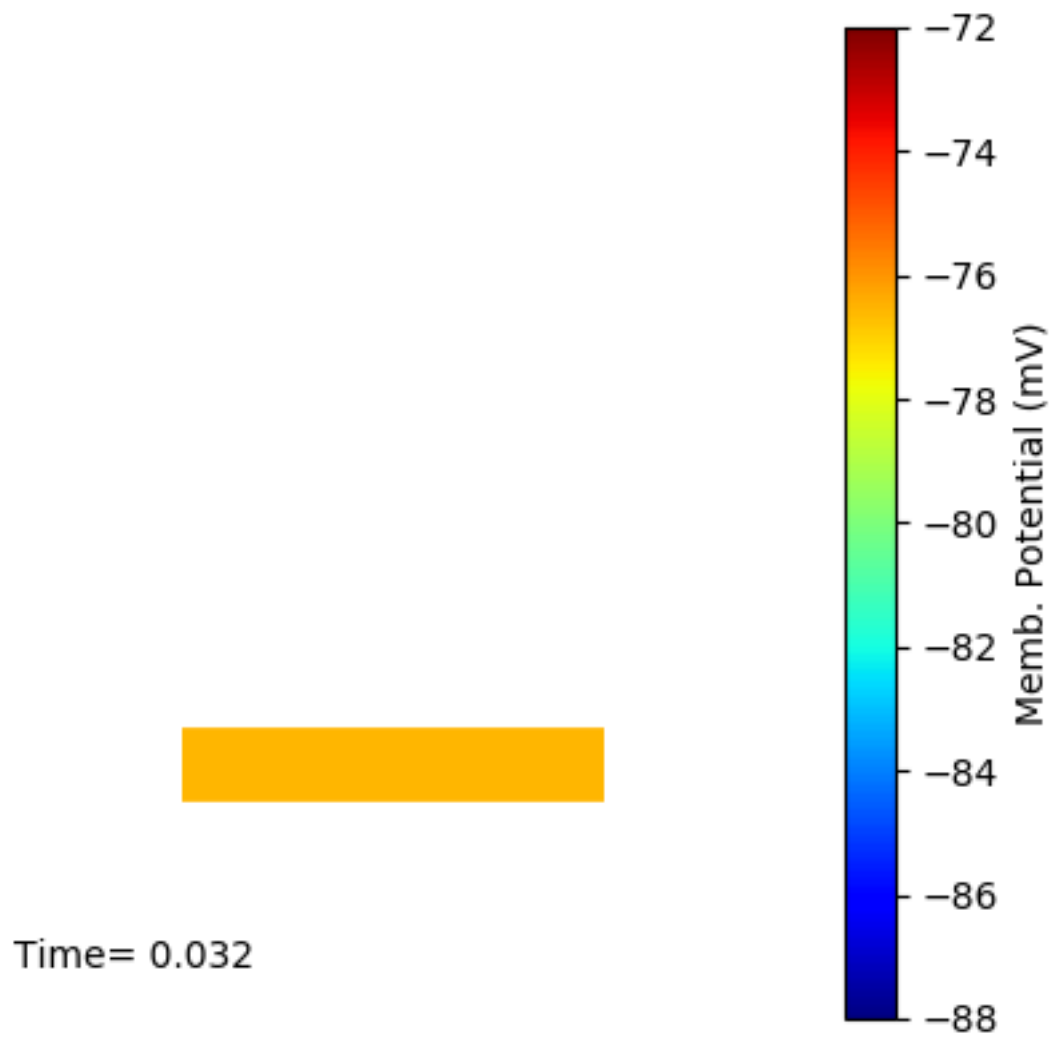


Fig. 9: Image of soma.

```

import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    # cellProto syntax: ['ballAndStick', 'name', somaDia, somaLength,
    ↪dendDia, dendLength, numDendSegments ]
    # The numerical arguments are all optional
    cellProto = [['ballAndStick', 'soma', 20e-6, 20e-6, 4e-6, 500e-6,
    ↪10]],
    chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '1200' ],
        ['K', 'soma', 'Gbar', '360' ],
        ['Na', 'dend#', 'Gbar', '400' ],
        ['K', 'dend#', 'Gbar', '120' ]
    ],
    stimList = [['soma', '1', '.', 'inject', '(t>0.01 && t<0.05) * 1e-
    ↪9' ]],
    plotList = [['soma', '1', '.', 'Vm', 'Membrane potential']],
    moogList = [['#', '1', '.', 'Vm', 'Vm (mV)']]
)
rdes.buildModel()
soma = moose.element( '/model/elec/soma' )
moose.reinit()
rdes.displayMoogli( 0.0005, 0.06, 0.0 )

```

As before, the *cellProto* line plays a key role. Here, because we have a long dendrite, we have a few more numerical arguments. All of the numerical arguments are optional.

```
['ballAndStick', 'name', somaDia, somaLength, dendDia,
dendLength, numDendSegments ]
```

- The first argument specifies a ballAndStick model: soma + dendrite. The length of the dendrite is along the x axis. The soma is a single segment, the dendrite can be more than one.
- The second argument is the name to give the cell.
- Arg 3 is the soma diameter, as a double.
- Arg 4 is the length of the soma, as a double.
- Arg 5 is the diameter of the dendrite, as a double.
- Arg 6 is the length of the dendrite, as a double.
- Arg 7 is the number of segments into which the dendrite should be divided. This is a positive integer greater than 0.

This is what the ball-and-stick cell looks like:

In this version of the 3-D display, the soma is displayed as a bit blocky rather than round. Note that we have populated the dendrite with Na and K channels and it has 10 segments, so it supports action potential propagation. The snapshot illustrates this.

Here are some things to try:

- Change the length of the dendrite

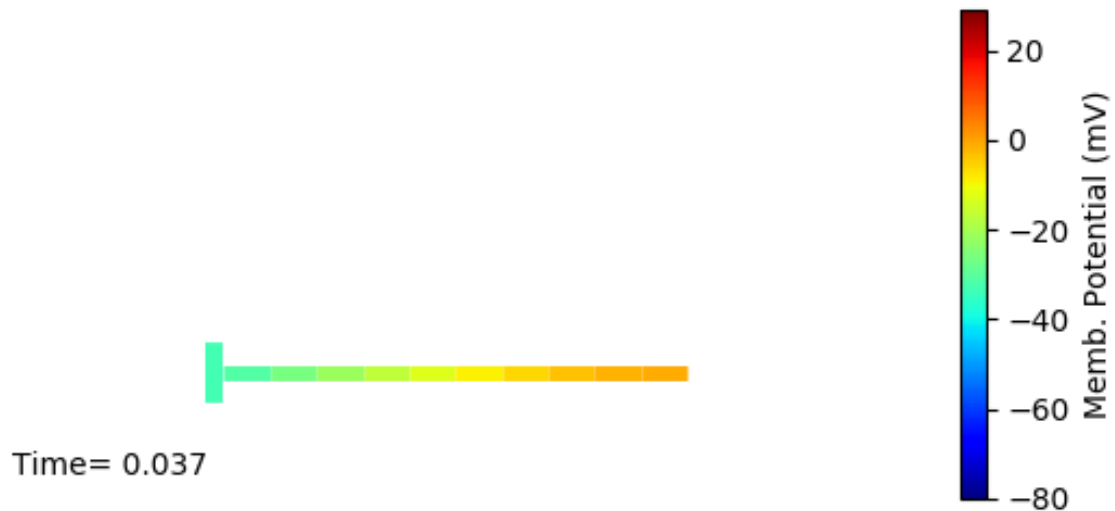


Fig. 10: Image of ball and stick cell.

- Change the number of segments. Explore what it does to accuracy. How will you know that you have an accurate model?

Benchmarking simulation speed

ex4.2_ballAndStickSpeed.py

The ball-and-stick model gives us an opportunity to check out your system and how computation scales with model size. While we're at it we'll deliver a sine-wave stimulus just to see how it can be done. The test model is very similar to the previous one, ex4.1:

```
import moose
import pylab
import rdesigneur as rd
import time
rdes = rd.rdesigneur(
    cellProto = [['ballAndStick', 'soma', 20e-6, 20e-6, 4e-6, 500e-6, ↵
↵10]],
    chanProto = [['make_HH_Na()', 'Na'], ['make_HH_K()', 'K']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '1200' ],
        ['K', 'soma', 'Gbar', '360' ],
        ['Na', 'dend#', 'Gbar', '400' ],
        ['K', 'dend#', 'Gbar', '120' ]
    ],
    stimList = [['soma', '1', '.', 'inject', '(1+cos(t/10))*(t>31.4 &&
↵ t<94) * 0
↵.2e-9' ]],
    plotList = [
        ['soma', '1', '.', 'Vm', 'Membrane potential'],
        ['soma', '1', '.', 'inject', 'Stimulus current']
    ],
)
rdes.buildModel()
```

(continues on next page)

(continued from previous page)

```

runtime = 100
moose.reinit()
t0= time.time()
moose.start( runtime )
print "Real time to run {} simulated seconds = {} seconds".format(_
    runtime, time
    .time() - t0 )

rdes.display()

```

While the real point of this simulation is to check speed, it does illustrate how to deliver a stimulus shaped like a sine wave:

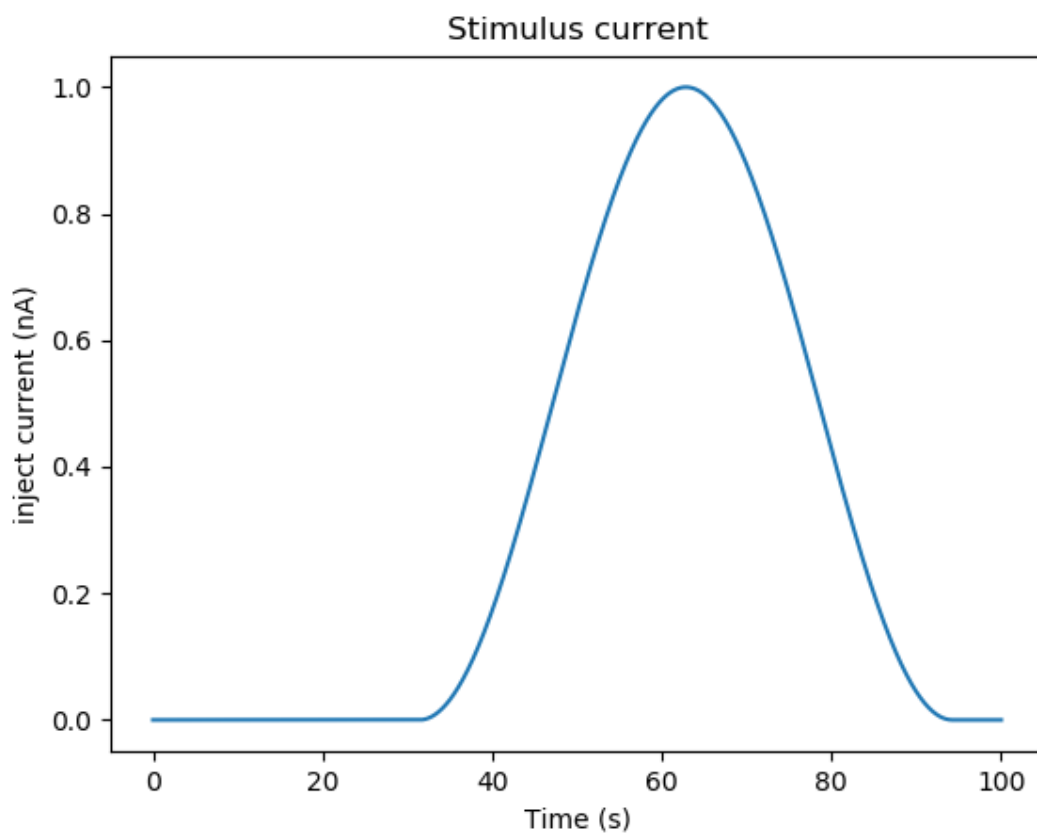


Fig. 11: Sine-wave shaped stimulus.

We can see that the cell has a peculiar response to this. Not surprising, as the cell uses HH channels which are not good at rate coding.

As a reference point, on a fast 2018 laptop this benchmark runs in 5.4 seconds. Some more things to try for benchmarking:

- How slow does it get if you turn on the 3-D moogli display?
- Is it costlier to run 2 compartments for 1000 seconds, or 200 compartments for 10 seconds?

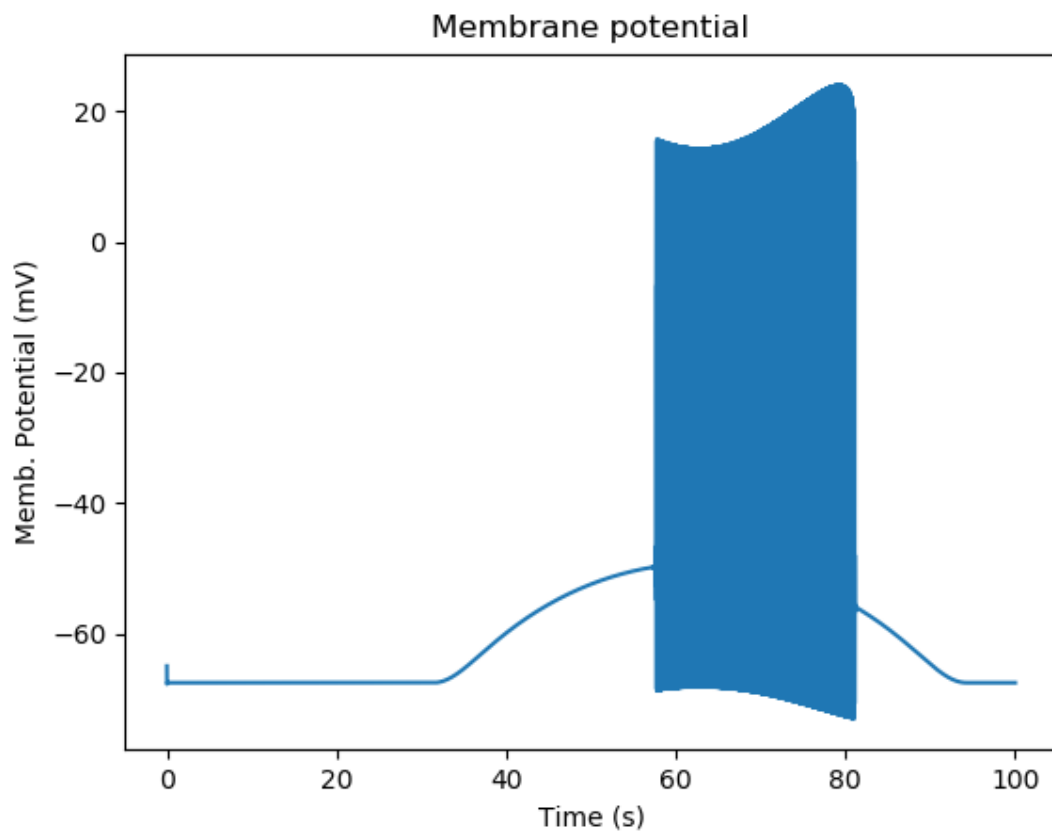


Fig. 12: Spiking response to sine-wave shaped stimulus.

Synaptic stimulus: random (Poisson)

ex5.0_random_syn_input.py

In this example we introduce synaptic inputs: both the receptor channels and a means for stimulating the channels. We do this in a passive model.

```

import moose
import rdesigneur as rd
rdes = rd.rdesigneur(
    cellProto = [['somaProto', 'soma', 20e-6, 200e-6]],
    chanProto = [['make_glu()', 'glu']],
    chanDistrib = [['glu', 'soma', 'Gbar', '1' ]],
    stimList = [['soma', '0.5', 'glu', 'randsyn', '50' ]],
    # Deliver stimulus to glu synapse on soma, at mean 50 Hz Poisson.
    plotList = [['soma', '1', '.', 'Vm', 'Soma membrane potential']]
)
rdes.buildModel()
moose.reinit()
moose.start( 0.3 )
rdes.display()

```

Most of the rdesigneur setup uses familiar syntax.

Novelty 1: we use the default built-in glutamate receptor model, in chanProto. We just put it in the soma at a max conductance of 1 Siemen/sq metre.

Novelty 2: We specify a new kind of stimulus in the stimList:

```
['soma', '0.5', 'glu', 'randsyn', '50' ]
```

Most of this is similar to previous stimLists.

- arg0: 'soma': the named compartments in the cell to populate with the *glu* receptor
- arg1: '0.5': Tell the system to use a uniform synaptic weight of 0.5. This argument could be a more complicated expression incorporating spatial arguments. Here it is just uniform.
- arg2: 'glu': Which receptor to stimulate
- arg3: 'randsyn': Apply random (Poisson) synaptic input.
- arg4: '50': Mean firing rate of the Poisson input. Note that this last argument could be a function of time and hence is quite versatile.

As the model has no voltage-gated channels, we do not see spiking.

Things to try: Vary the rate and the weight of the synaptic input.

Synaptic stimulus: periodic

ex5.1_periodic_syn_input.py

This is almost identical to 5.0, except that the input is now perfectly periodic. The one change is of an argument in the stimList to say *periodicsyn* rather than *randsyn*.

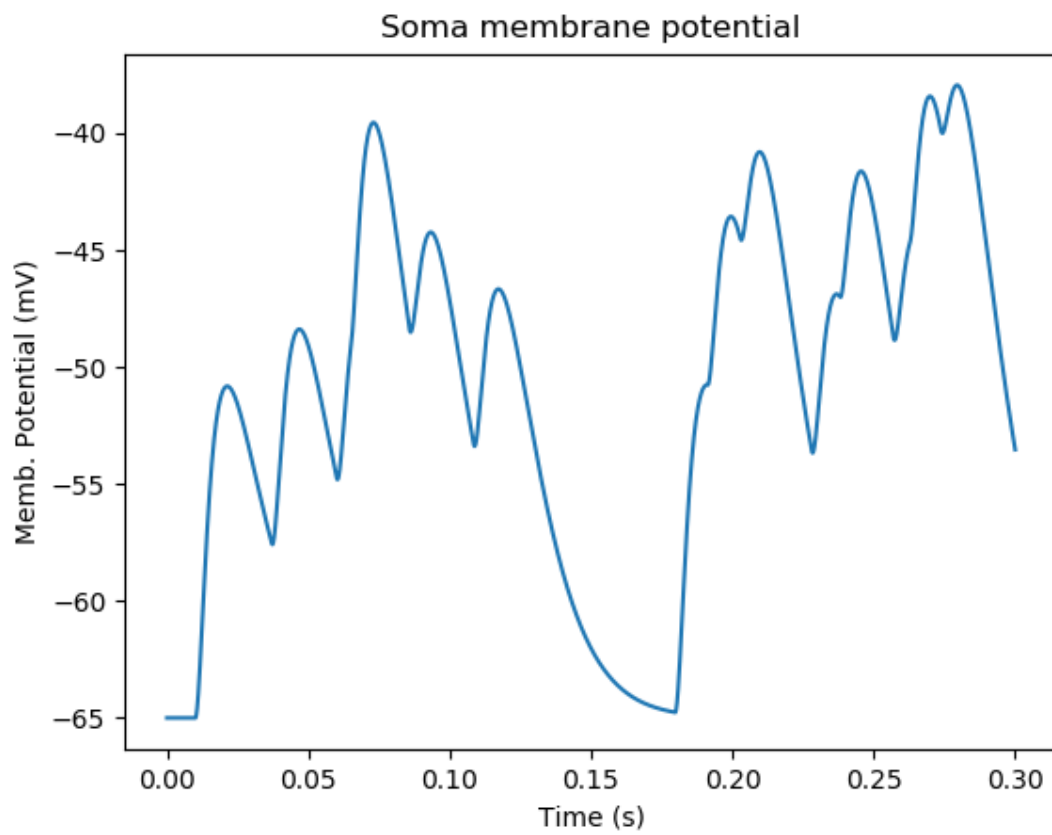


Fig. 13: Random synaptic input with a Poisson distribution.

```

import moose
import rdesigneur as rd
rdes = rd.rdesigneur(
    cellProto = [['somaProto', 'soma', 20e-6, 200e-6]],
    chanProto = [['make_glu()', 'glu']],
    chanDistrib = [['glu', 'soma', 'Gbar', '1' ]],

    # Deliver stimulus to glu synapse on soma, periodically at 50 Hz.
    stimList = [['soma', '0.5', 'glu', 'periodicsyn', '50' ]],
    plotList = [['soma', '1', '.', 'Vm', 'Soma membrane potential']]
)
rdes.buildModel()
moose.reinit()
moose.start( 0.3 )
rdes.display()

```

As designed, we get periodically firing synaptic input.

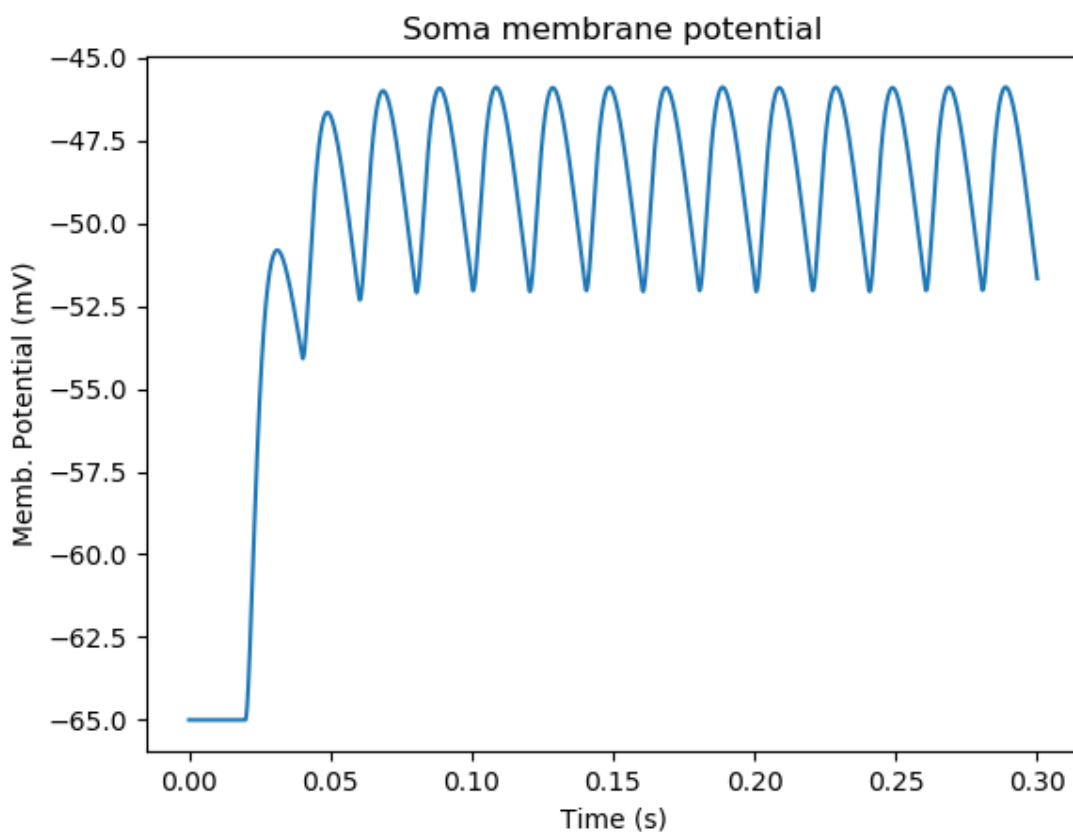


Fig. 14: Periodic synaptic input

Reaction system in a single compartment

ex6_chem_osc.py

Here we use the compartment as a place in which to embed a chemical model. The chemical oscillator model is predefined in the rdesigneur prototypes. Its general form 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
```

Here is the script:

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    turnOffElec = True,
    diffusionLength = 1e-3, # Default diffusion length is 2_
    ↪microns
    chemProto = [['makeChemOscillator()', 'osc']],
    chemDistrib = [['osc', 'soma', 'install', '1']],
    plotList = [['soma', '1', 'dend/a', 'conc', 'a Conc'],
                ['soma', '1', 'dend/b', 'conc', 'b Conc']]
)
rdes.buildModel()
b = moose.element( '/model/chem/dend/b' )
b.concInit *= 5
moose.reinit()
moose.start( 200 )

rdes.display()
```

In this special case we set the `turnOffElec` flag to `True`, so that `Rdesigneur` only sets up chemical and not electrical calculations. This makes the calculations much faster, since we disable electrical calculations and delink chemical calculations from them.

We also have a line which sets the `diffusionLength` to 1 mm, so that it is bigger than the 0.5 mm squid axon segment in the default compartment. If you don't do this the system will subdivide the compartment into the default 2 micron voxels for the purposes of putting in a reaction-diffusion system. We discuss this case below.

Note how the `plotList` is done here. To remind you, each entry has five arguments

```
[region_in_cell, region_expression, moose_object, parameter, title_of_
↪plot]
```

The change from the earlier usage is that the `moose_object` now refers to a chemical entity, in this example the molecule `dend/a`. The simulator builds a default chemical compartment named `dend` to hold the reactions defined in the `chemProto`. What we do in this plot is to select molecule `a` sitting in `dend`, and plot its concentration. Then we do this again for molecule `b`.

After the model is built, we add a couple of lines to change the initial concentration of the molecular pool `b`. Note its full path within MOOSE: `/model/chem/dend/b`. It is scaled up 5x to give rise to slowly decaying oscillations.

Reaction-diffusion system

ex7.0_spatial_chem_osc.py

In order to see what a reaction-diffusion system looks like, we assign the `diffusionLength`

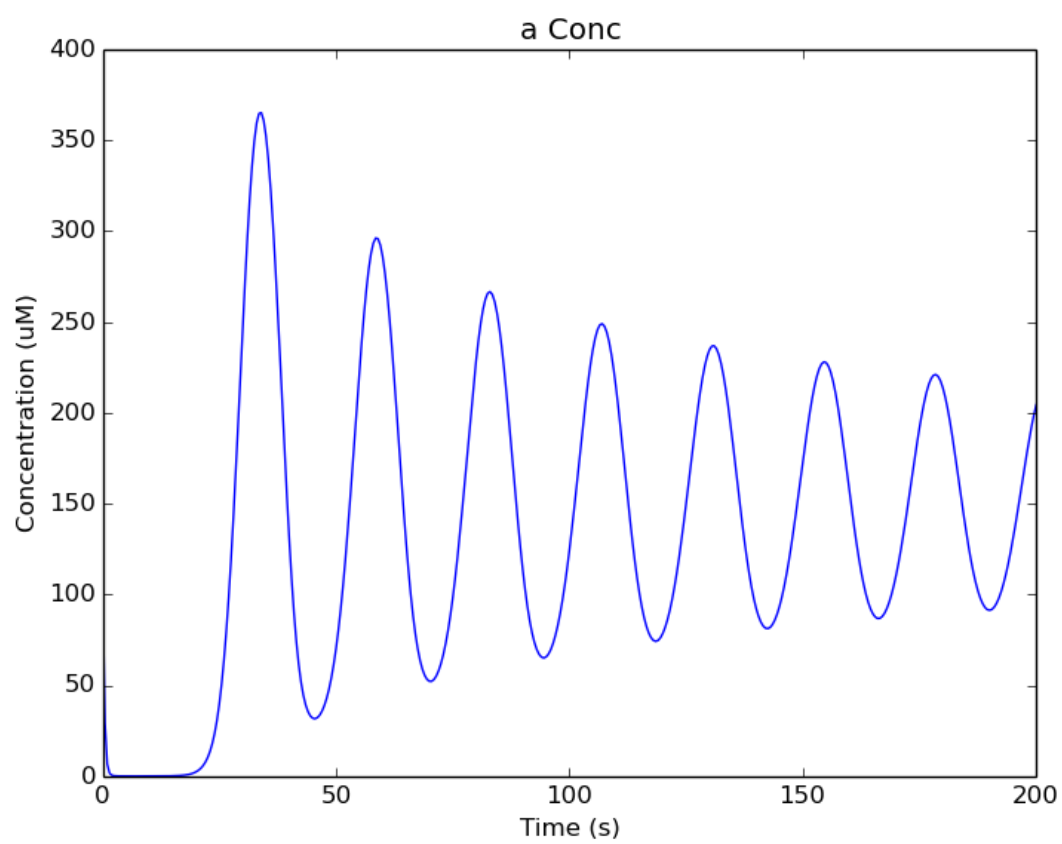


Fig. 15: Plot for single-compartment reaction simulation

expression in the previous example to a much shorter length, and add a couple of lines to set up 3-D graphics for the reaction-diffusion product:

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    turnOffElec = True,
    #This subdivides the length of the soma into 2 micron voxels
    diffusionLength = 2e-6,
    chemProto = [['makeChemOscillator()', 'osc']],
    chemDistrib = [['osc', 'soma', 'install', '1']],
    plotList = [['soma', '1', 'dend/a', 'conc', 'Concentration of a'],
    ['soma', '1', 'dend/b', 'conc', 'Concentration of b']],
    moogList = [['soma', '1', 'dend/a', 'conc', 'a Conc', 0, 360],
    ['soma', '1', 'dend/b', 'conc', 'b Conc', 0, 360]]
)

rdes.buildModel()
bv = moose.vec( '/model/chem/dend/b' )
bv[0].concInit *= 2
bv[-1].concInit *= 2
moose.reinit()

rdes.displayMoogli( 1, 400, rotation = 0, azim = np.pi/2, elev = 0.0 )
```

This is the new value for diffusion length.

```
diffusionLength = 2e-3,
```

With this change we tell *rdesigneur* to use the diffusion length of 2 microns. This happens to be the default too. The 500-micron axon segment is now subdivided into 250 voxels, each of which has a reaction system and diffusing molecules. To make it more picturesque, we have added a line after the `plotList`, to display the outcome in 3-D:

```
moogList = [['soma', '1', 'dend/a', 'conc', 'a Conc', 0, 360 ]]
```

This line says: take the model compartments defined by `soma` as the region to display, do so throughout the the geometry (the 1 signifies this), and over this range find the chemical entity defined by `dend/a`. For each a molecule, find the `conc` and display it. There are two optional arguments, 0 and 360, which specify the low and high value of the displayed variable.

In order to initially break the symmetry of the system, we change the initial concentration of molecule `b` at each end of the cylinder:

```
bv[0].concInit *= 2
bv[-1].concInit *= 2
```

If we didn't do this the entire system would go through a few cycles of decaying oscillation and then reach a boring, spatially uniform, steady state. Try putting an initial symmetry break elsewhere to see what happens.

To display the concentration changes in the 3-D soma as the simulation runs, we use the line

```
rdes.displayMoogli( 1, 400, rotation = 0, azim = np.pi/2, elev = 0.0 )
```

The arguments mean: *displayMoogli(framerate, runtime, rotation, azimuth, elevation)* Here,

```
framerate = time by which simulation advances between display updates
runtime = Total simulated time
rotation = angle by which display rotates in each frame, in radians.
azimuth = Azimuth angle of view point, in radians
elevation = elevation angle of view point, in radians
```

When we run this, we first get a 3-D display with the oscillating reaction-diffusion system making its way inward from the two ends. After the simulation ends the plots for all compartments for the whole run come up.

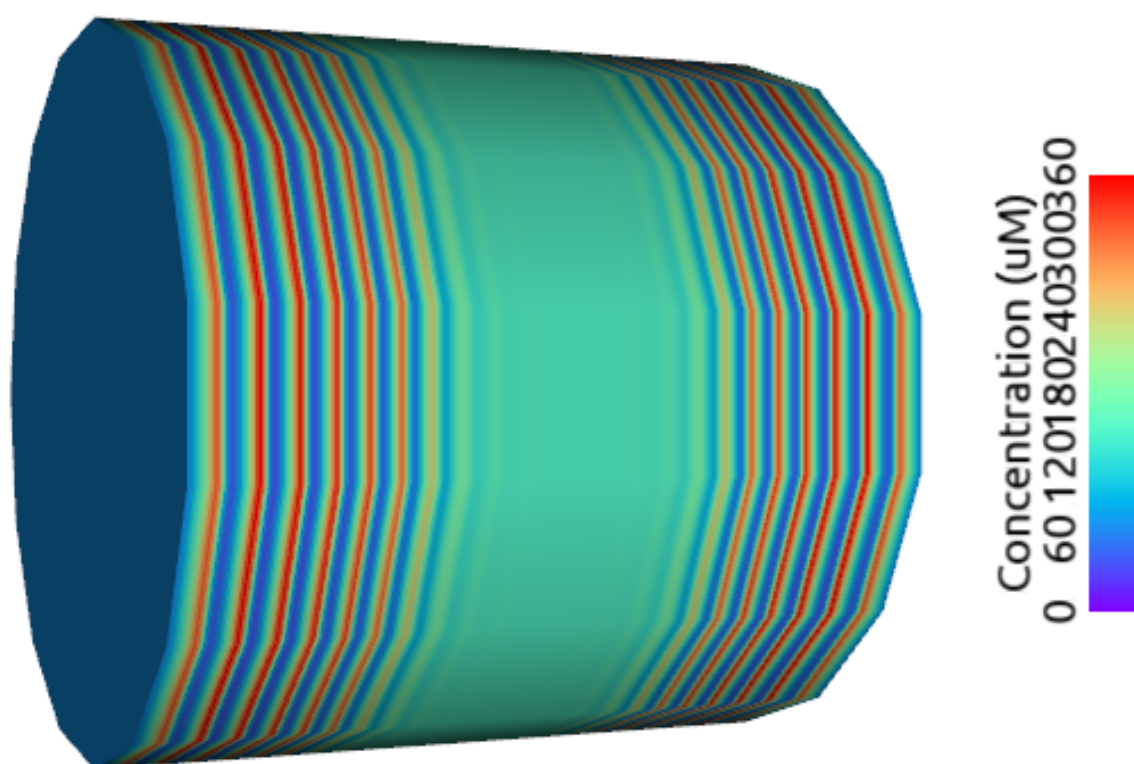


Fig. 16: Display for oscillatory reaction-diffusion simulation

For those who would rather use the much simpler matplotlib 3-D display option, this is what the same simulation looks like:

Primer on using the 3-D MOOGLI display

There are two variants of the MOOGLI display. The first, named Moogli, uses OpenGL and OpenSceneGraph. It is fast to display, slow to load, and difficult to compile. It produces much better looking 3-D graphics. The second is a fallback interface using mplot3d, which is a library of Matplotlib and so should be generally available. It is slower to display, faster to load, but needs no special compilation. It uses stick graphics and though it conveys much the same information, isn't as nice to look at as the original Moogli. Its controls are more or less the same but less smooth than the original Moogli.

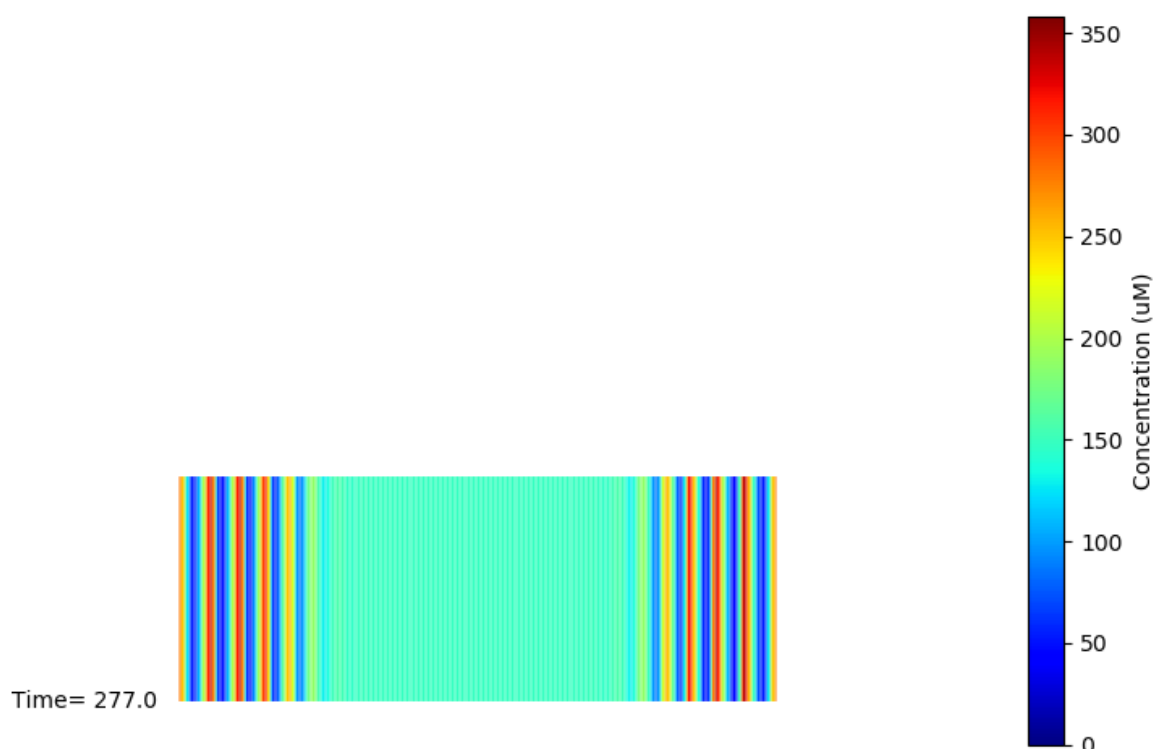


Fig. 17: Display for oscillatory reac-diff simulation using matplotlib

Here is a short primer on the 3-D display controls.

- *Roll, pitch, and yaw*: Use the letters *r*, *p*, and *y*. To rotate backwards, use capitals.
- *Zoom out and in*: Use the , and . keys, or their upper-case equivalents, < and >. Easier to remember if you think in terms of the upper-case.
- *Left/right/up/down*: Arrow keys.
- *Quit*: control-q or control-w.
- You can also use the mouse or trackpad to control most of the above.
- By default rdesigneur gives Moogli a small rotation each frame. It is the *rotation* argument in the line:

```
displayMoogli( frametime, runtime, rotation )
```

These controls operate over and above this rotation, but the rotation continues. If you set the rotation to zero you can, with a suitable flick of the mouse, get the image to rotate in any direction you choose as long as the window is updating.

Diffusion of a single molecule

ex7.1_diffusive_gradient.py

This is simply a test model to confirm that simple diffusion happens as expected. While the model is just that of a single pool, we spend a few lines taking snapshots of the spatial profile of this pool.

```

import moose
import pylab
import re
import rdesigneur as rd
import matplotlib.pyplot as plt
import numpy as np

moose.Neutral( '/library' )
moose.Neutral( '/library/diffn' )
moose.CubeMesh( '/library/diffn/dend' )
A = moose.Pool( '/library/diffn/dend/A' )
A.diffConst = 1e-10

rdes = rd.rdesigneur(
    turnOffElec = True,
    diffusionLength = 1e-6,
    chemProto = [['diffn', 'diffn']],
    chemDistrib = [['diffn', 'soma', 'install', '1' ]],
    moogList = [
        ['soma', '1', 'dend/A', 'conc', 'A Conc', 0, 360 ]
    ]
)
rdes.buildModel()

rdes.displayMoogli( 1, 2, rotation = 0, azim = -np.pi/2, elev = 0.0,
    ↪block = False )
av = moose.vec( '/model/chem/dend/A' )
for i in range(10):
    av[i].concInit = 1
moose.reinit()
plist = []
for i in range( 20 ):
    plist.append( av.conc[:200] )
    moose.start( 2 )
fig = plt.figure( figsize = ( 10, 12 ) )
plist = np.array( plist ).T
plt.plot( range( 0, 200 ), plist )
plt.xlabel( "position ( microns )" )
plt.ylabel( "concentration ( mM )" )
plt.show( block = True )

```

Here are the snapshots, overlaid in a single plot:

Calcium-induced calcium release

ex7.2_CICR.py

This is a somewhat more complex reaction-diffusion system, involving calcium release from intracellular stores that propagates in a wave of activity along a dendrite. This example demonstrates the use of endo compartments.

Endo-compartments, as the name suggests, represent compartments that sit within other cellular compartments. If the surround compartment is subdivided into N voxels, so is the endo-compartment. The *rdesigneur* system looks at the provided model, and if there are 2 compartments and the *addEndoChemCompt* flag is True, then the chemistry contained in the smaller of the two compartments is positioned in an endo compartment surrounded by the first com-

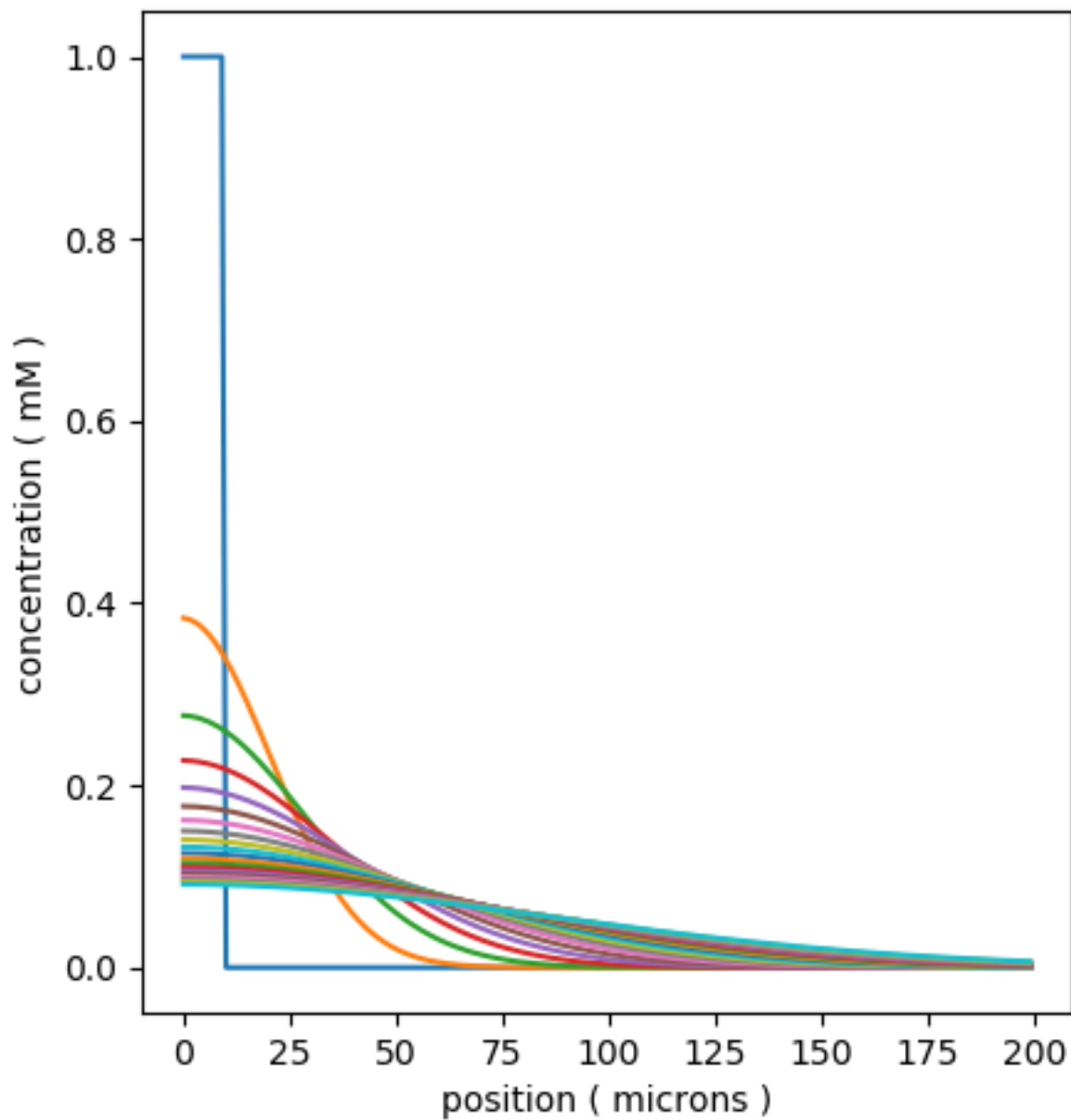


Fig. 18: Display for simple time-series of spread of a diffusing molecule using matplotlib

partment. Here we use the endo-compartment to represent the endoplasmic reticulum sitting inside the dendrite.

In the chemical model, we also introduce a new MOOSE class, ConcChan. These act as membrane pores whose permeability scales with number of channels in the open state. The IP3 receptor in this model is implemented as a ConcChan which opens due to binding to IP3 and Calcium. This leads to the release of more calcium from the ER, and this feedback loop develops into a propagating-wave oscillation.

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    turnOffElec = True,
    chemDt = 0.005,
    chemPlotDt = 0.02,
    diffusionLength = 1e-6,
    useGssa = False,
    addSomaChemCompt = False,
    addEndoChemCompt = True,
    # cellProto syntax: ['somaProto', 'name', dia, length]
    cellProto = [['somaProto', 'soma', 2e-6, 10e-6]],
    chemProto = [['./chem/CICRwithConcChan.g', 'chem']],
    chemDistrib = [['chem', 'soma', 'install', '1']],
    plotList = [
        ['soma', '1', 'dend/CaCyt', 'conc', 'Dendritic Ca'],
        ['soma', '1', 'dend/CaCyt', 'conc', 'Dendritic Ca', 'wave'],
        ['soma', '1', 'dend_endo/CaER', 'conc', 'ER Ca'],
        ['soma', '1', 'dend/ActIP3R', 'conc', 'active IP3R'],
    ],
)
rdes.buildModel()
IP3 = moose.element( '/model/chem/dend/IP3' )
IP3.vec.concInit = 0.004
IP3.vec[0].concInit = 0.02
moose.reinit()
moose.start( 40 )
rdes.display()
```

Note how the dendritic calcium is displayed both as a time-series plot and as a wave plot, which presents the time-evolution of the calcium as a function of position in successive image frames.

Intracellular transport

ex7.3_simple_transport.py

This illustrates how intracellular transport works in MOOSE. We have a an elongated soma in which molecules start out at the left and are transported to the right. Note that they spread out as they go along, This is because the transport is implemented as drift-diffusion, in which a fraction of the molecules move to the next location each timestep. The equation is

$$\text{flux} = \text{motorConst} * \text{conc} / \text{spacing}$$

for a uniform cylinder. MOOSE applies suitable scaling terms if the neuronal geometry is non-uniform.

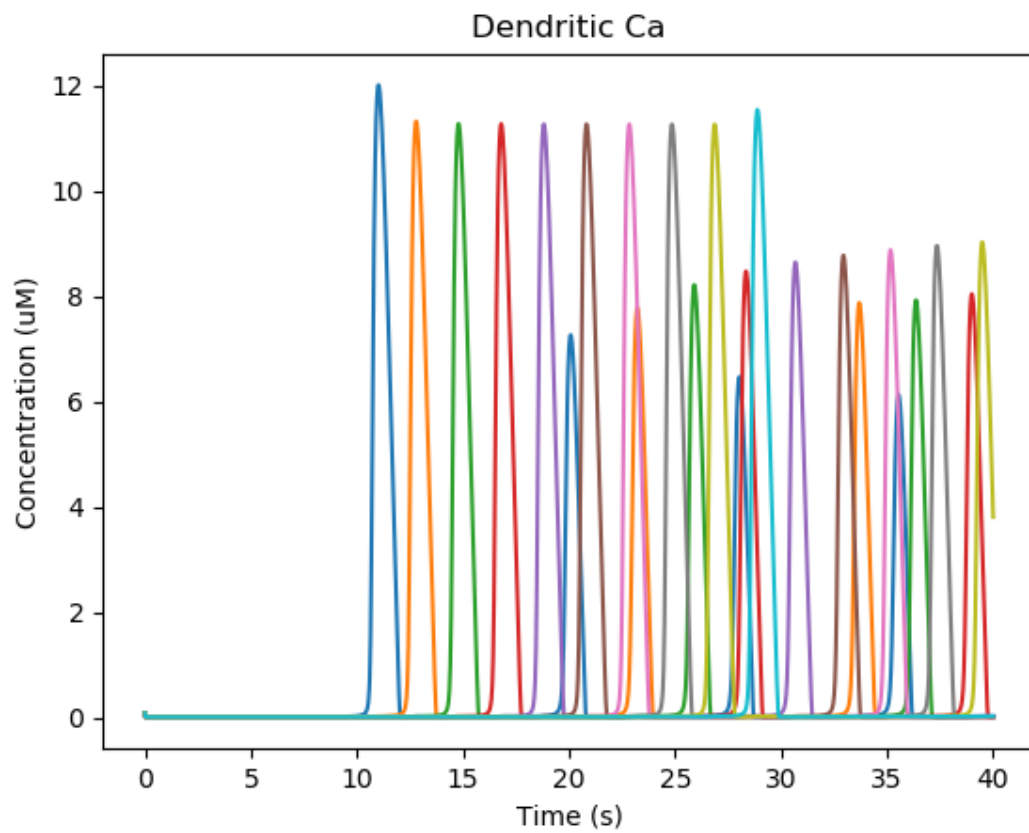


Fig. 19: Time-series plot of dendritic calcium. Different colors represent different voxels in the dendrite.

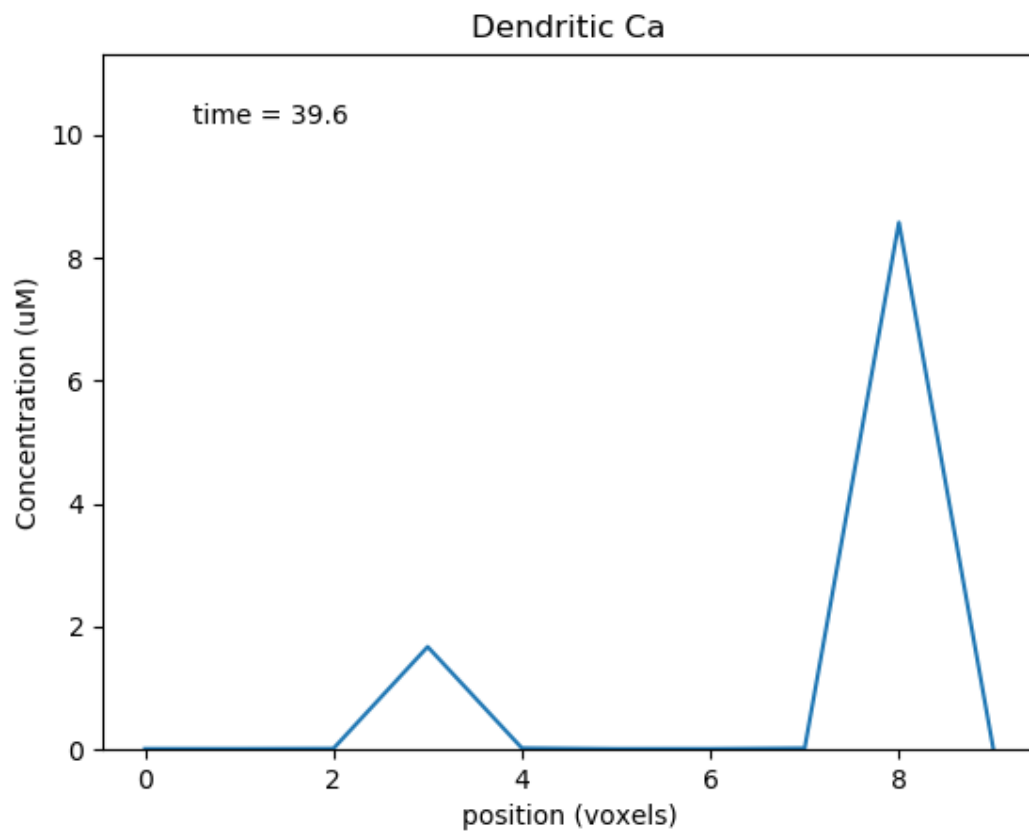


Fig. 20: Place holder for time-evolving movie of dendritic calcium as a function of position along the dendrite.

```

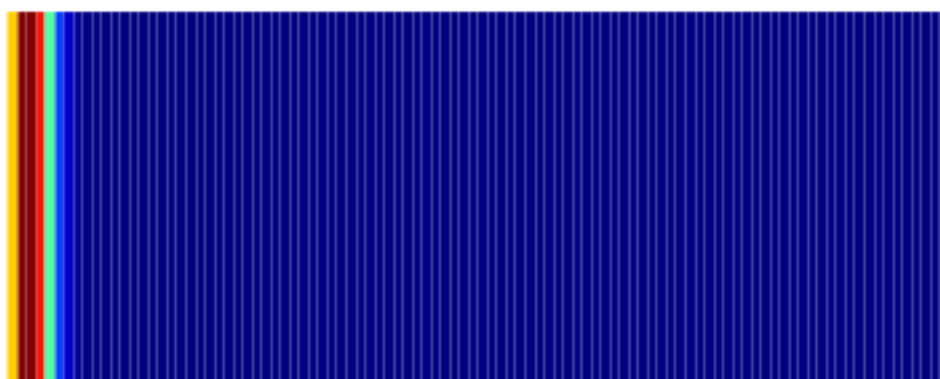
import moose
import numpy as np
import pylab
import rdesigneur as rd

moose.Neutral( '/library' )
moose.Neutral( '/library/transp' )
moose.CubeMesh( '/library/transp/dend' )
A = moose.Pool( '/library/transp/dend/A' )
A.diffConst = 0
A.motorConst = 1e-6      # Metres/sec

rdes = rd.rdesigneur(
    turnOffElec = True,
    #This subdivides the length of the soma into 0.5 micron voxels
    diffusionLength = 0.5e-6,
    cellProto = [['somaProto', 'soma', 2e-6, 50e-6]],
    chemProto = [['transp', 'transp']],
    chemDistrib = [['transp', 'soma', 'install', '1' ]],
    plotList = [
        ['soma', '1', 'dend/A', 'conc', 'Concentration of A'],
        ['soma', '1', 'dend/A', 'conc', 'Concentration of A', 'wave'],
    ],
    moogList = [['soma', '1', 'dend/A', 'conc', 'A Conc', 0, 20 ]]
)
rdes.buildModel()
moose.element( '/model/chem/dend/A[0]' ).concInit = 0.1
moose.reinit()
rdes.displayMoogli( 1, 80, rotation = 0, azim = -np.pi/2, elev = 0.0 )

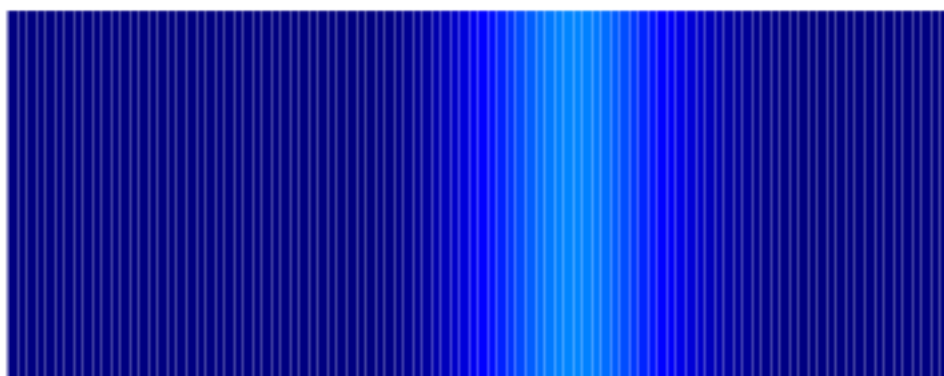
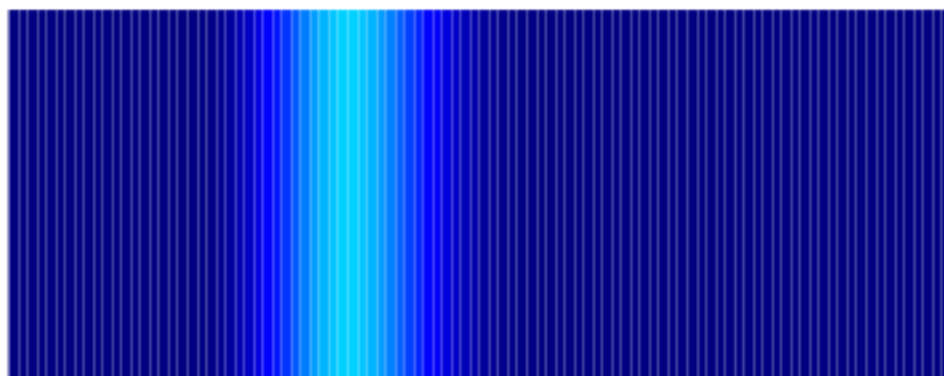
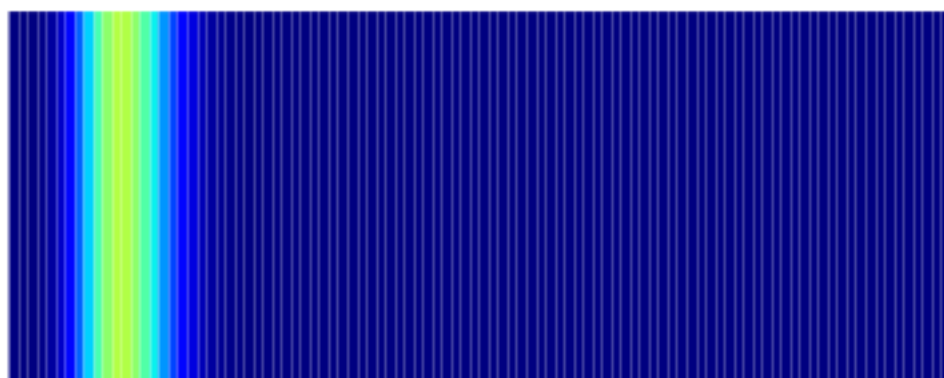
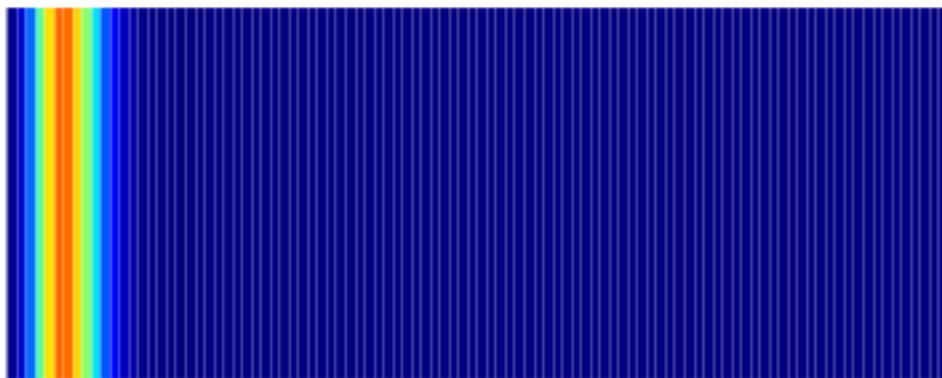
```

In this example we explicitly create the single-molecule reaction system, and assign a motor-Const of 1 micron/sec to the molecule A. We start off with all the molecules in a single voxel on the left of the cylinder, and then watch the molecules move. Once the molecules reach the end of the cylindrical soma, they have nowhere further to go so they pile up.



Suggestions:

- Play with different motor rates.
- The motor constant sign determines the direction of transport. See what happens if you get it going in the opposite direction.
- Consider how you could avoid the buildup in the last voxel.



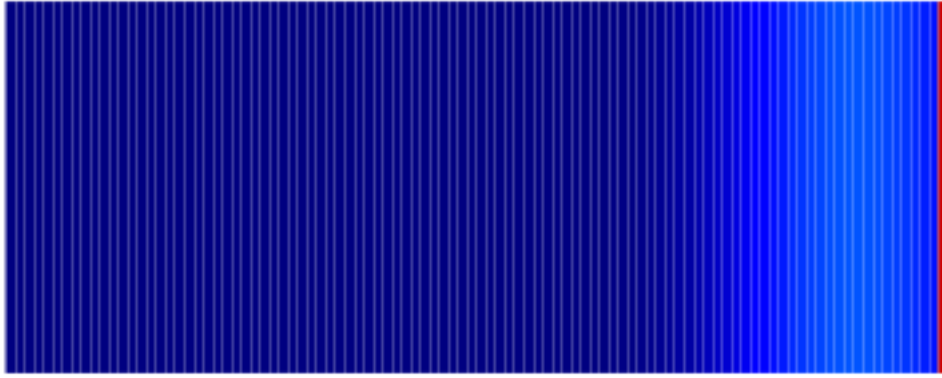


Fig. 21: Frames at increasing intervals from the transport simulation showing spreading and piling up of the molecule at the right end of the cylinder.

- Consider how to achieve a nice exponential falloff over a much longer range than possible with diffusion.

Travelling oscillator

ex7.4_travelling_osc.py

Here we put a chemical oscillator into a cylinder, and activate motor transport in one of the molecules. The oscillatory zone slowly moves to the right, with an amplification in the last compartment due to end-effects.

```
import moose
import numpy as np
import pylab
import rdesignneur as rd
rdes = rd.rdesignneur(
    turnOffElec = True,
    diffusionLength = 2e-6,
    chemProto = [['makeChemOscillator()', 'osc']],
    chemDistrib = [['osc', 'soma', 'install', '1']],
    plotList = [
        ['soma', '1', 'dend/a', 'conc', 'Concentration of a'],
        ['soma', '1', 'dend/b', 'conc', 'Concentration of b'],
        ['soma', '1', 'dend/a', 'conc', 'Concentration of a', 'wave'],
    ],
    moogList = [['soma', '1', 'dend/a', 'conc', 'a Conc', 0, 360]]
)
a = moose.element( '/library/osc/kinetics/a' )
b = moose.element( '/library/osc/kinetics/b' )
s = moose.element( '/library/osc/kinetics/s' )
a.diffConst = 0
b.diffConst = 0
a.motorConst = 1e-6

rdes.buildModel()
moose.reinit()

rdes.displayMoogli( 1, 400, rotation = 0, azim = -np.pi/2, elev = 0.0,
    ↪ )
```

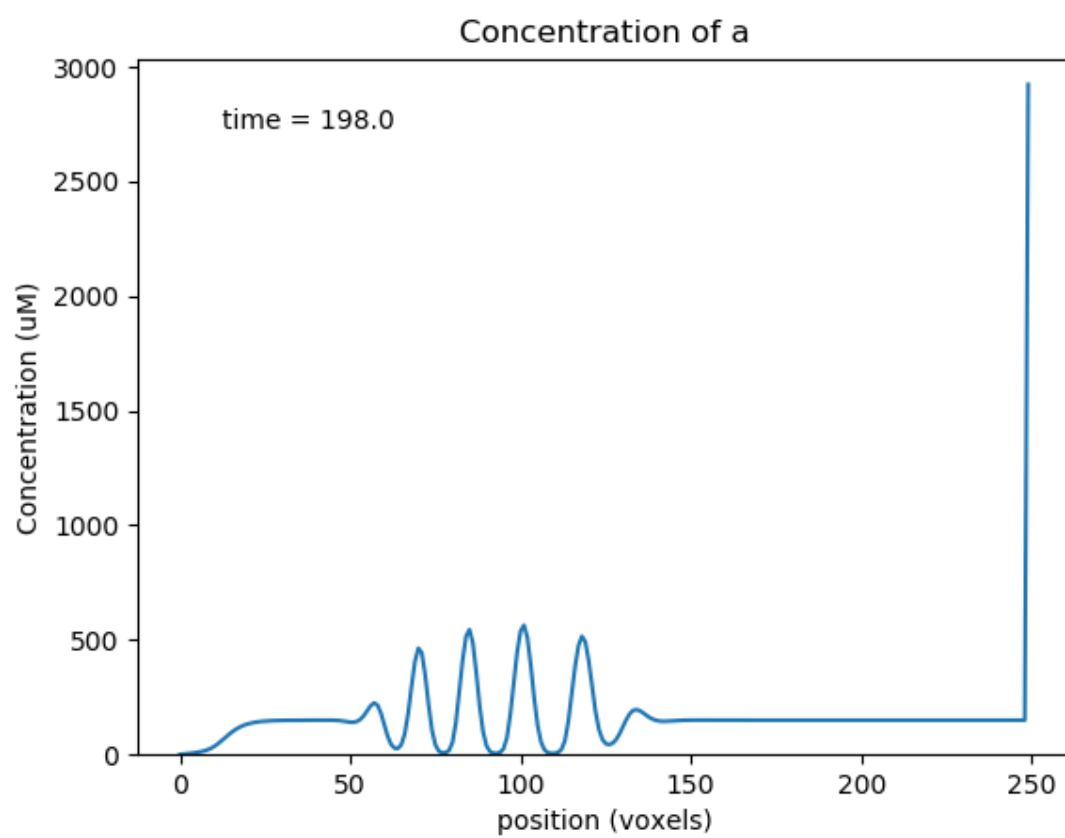


Fig. 22: Snapshot of travelling oscillator waveform at $t = 198$.

Suggestions:

- What happens if all molecules undergo transport?
- What happens if *b* is transported opposite to *a*?
- What happens if there is also diffusion?

Bidirectional transport

ex7.5_bidirectional_transport.py

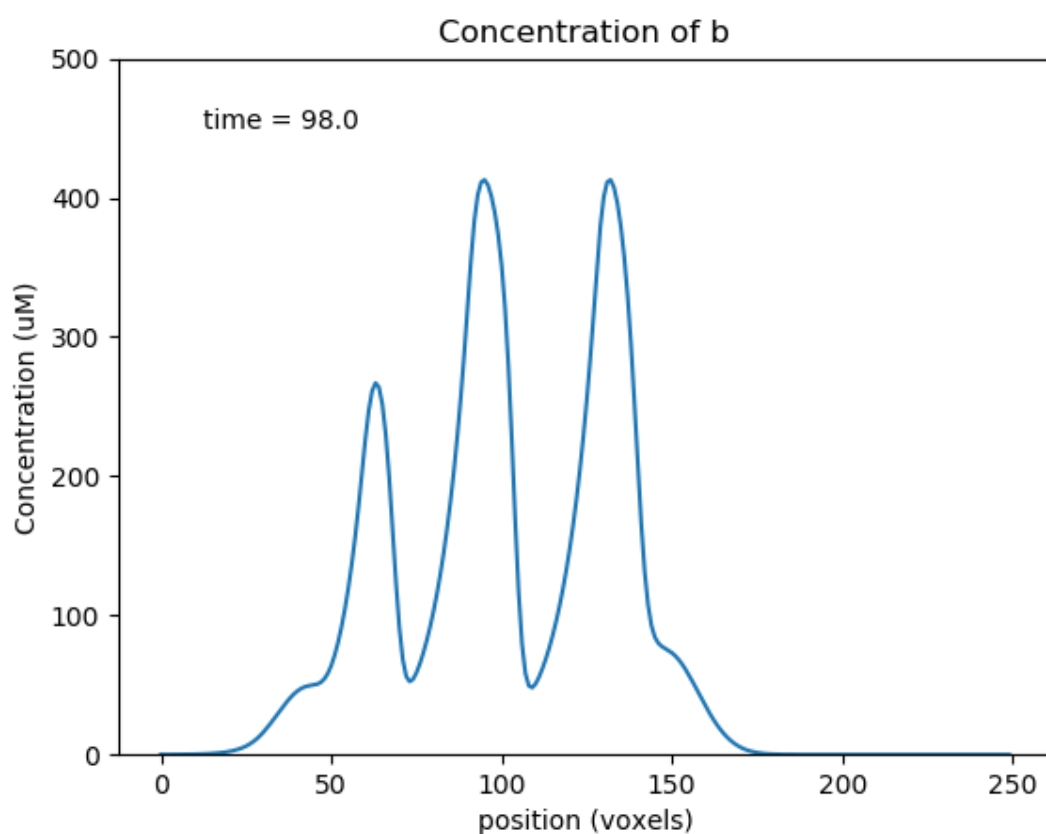
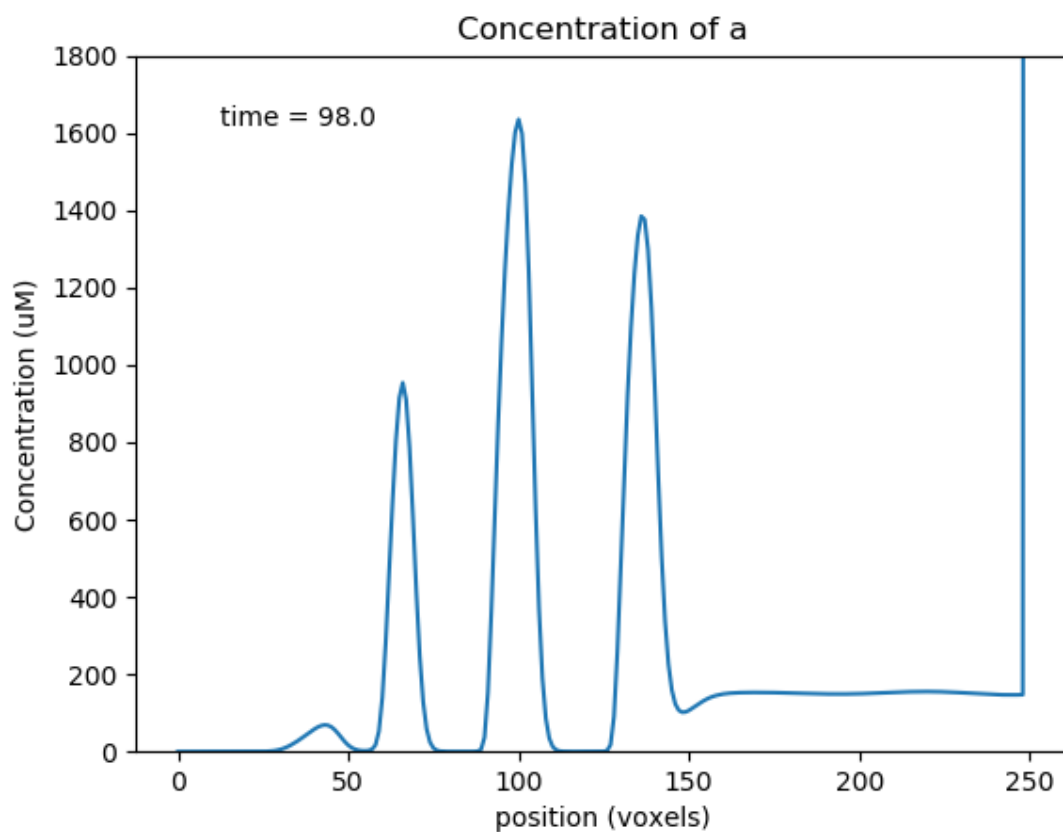
This is almost identical to ex7.4, except that we implement bidirectional transport. Molecule *a* goes from left to right, and *b* and *s* go from right to left. Here we see that the system builds up with large oscillations in the middle as the molecules converge, then the peaks collapse when the molecules go away.

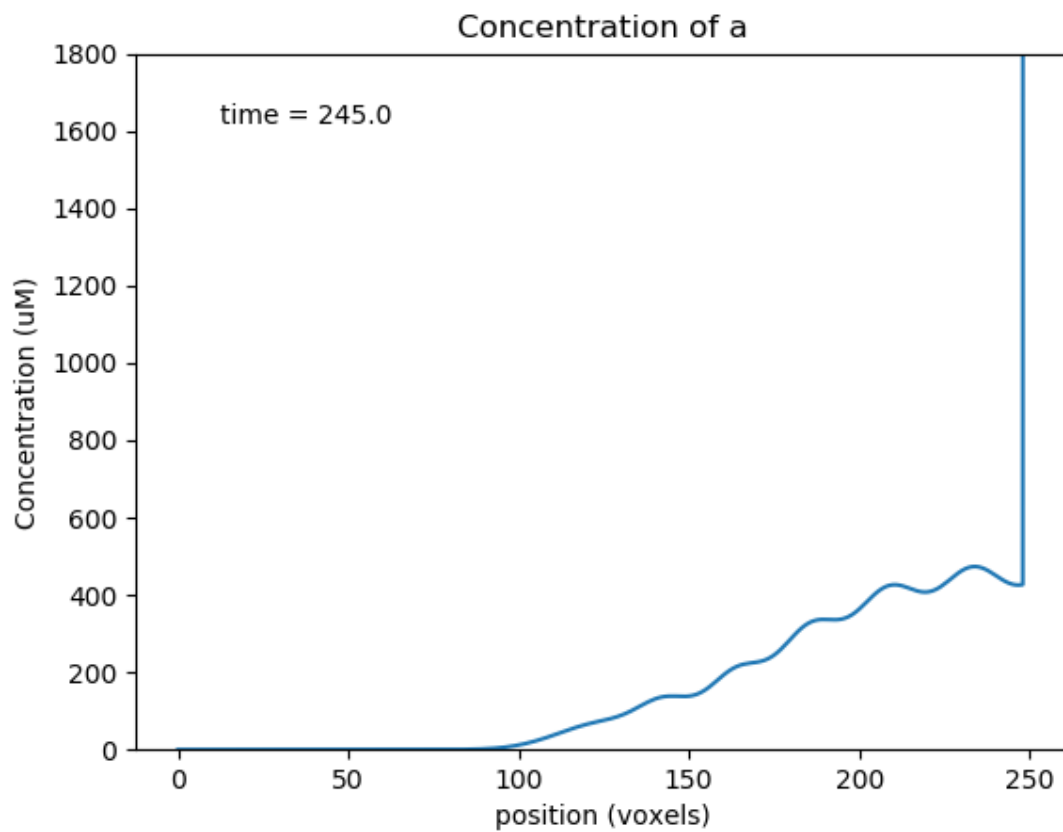
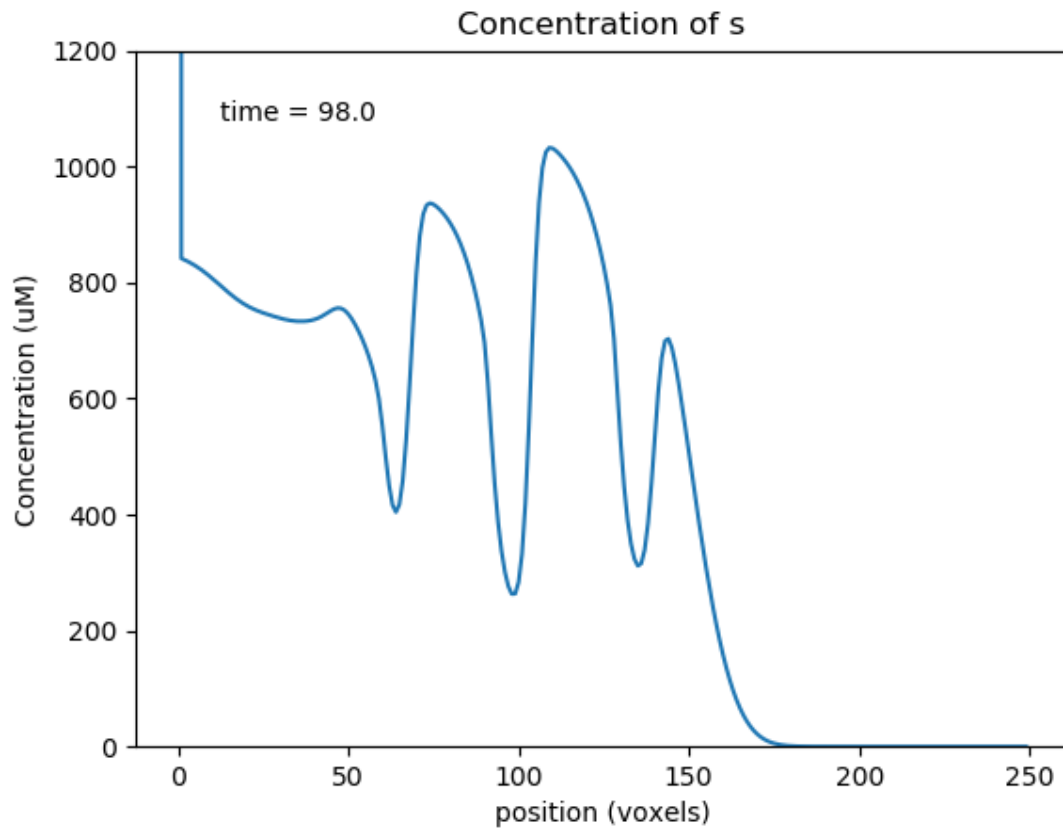
```
import moose
import numpy as np
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    turnOffElec = True,
    diffusionLength = 2e-6,
    numWaveFrames = 50,
    chemProto = [['makeChemOscillator()', 'osc']],
    chemDistrib = [['osc', 'soma', 'install', '1' ]],
    plotList = [
        ['soma', '1', 'dend/a', 'conc', 'Concentration of a', 'wave', 1800],
        ['soma', '1', 'dend/b', 'conc', 'Concentration of b', 'wave', 500],
        ['soma', '1', 'dend/s', 'conc', 'Concentration of s', 'wave', 1200],
    ],
    moogList = [['soma', '1', 'dend/a', 'conc', 'a Conc', 0, 600 ]]
)
a = moose.element( '/library/osc/kinetics/a' )
b = moose.element( '/library/osc/kinetics/b' )
s = moose.element( '/library/osc/kinetics/s' )
a.diffConst = 0
b.diffConst = 0
a.motorConst = 2e-6
b.motorConst = -2e-6
s.motorConst = -2e-6

rdes.buildModel()
moose.reinit()

rdes.displayMoogli( 1, 250, rotation = 0, azim = -np.pi/2, elev = 0.0 )
```

Above we see *a*, *b*, *s* at a point where the transport has collected the molecules toward the middle of the cylinder, so the oscillations are large. Below we see molecule *a* later, when it has gone past the *b* and *s* pools and so the reaction system is depleted and does not oscillate.





Controlling a reaction by a function

ex7.6_func_controls_reac_rate.py

This example illustrates how a function can be used to control a reaction rate. This kind of calculation is appropriate when we need to link different kinds of physical processes with chemical reactions, for example, membrane curvature with molecule accumulation. The use of functions to modify reaction rates should be avoided in purely chemical systems since they obscure the underlying chemistry, and do not map cleanly to stochastic calculations.

In this example we simply have a molecule C that controls the forward rate of a reaction that converts A to B. C is a function of location on the cylinder, and is fixed. In more elaborate computations we could have a function of multiple molecules, some of which could be changing and others could be buffered.

```
import numpy as np
import moose
import pylab
import rdesignneur as rd

def makeFuncRate():
    model = moose.Neutral( '/library' )
    model = moose.Neutral( '/library/chem' )
    compt = moose.CubeMesh( '/library/chem/compt' )
    compt.volume = 1e-15
    A = moose.Pool( '/library/chem/compt/A' )
    B = moose.Pool( '/library/chem/compt/B' )
    C = moose.Pool( '/library/chem/compt/C' )
    reac = moose.Reac( '/library/chem/compt/reac' )
    func = moose.Function( '/library/chem/compt/reac/func' )
    func.x.num = 1
    func.expr = "(x0/1e8)^2"
    moose.connect( C, 'nOut', func.x[0], 'input' )
    moose.connect( func, 'valueOut', reac, 'setNumKf' )
    moose.connect( reac, 'sub', A, 'reac' )
    moose.connect( reac, 'prd', B, 'reac' )

    A.concInit = 1
    B.concInit = 0
    C.concInit = 0
    reac.Kb = 1

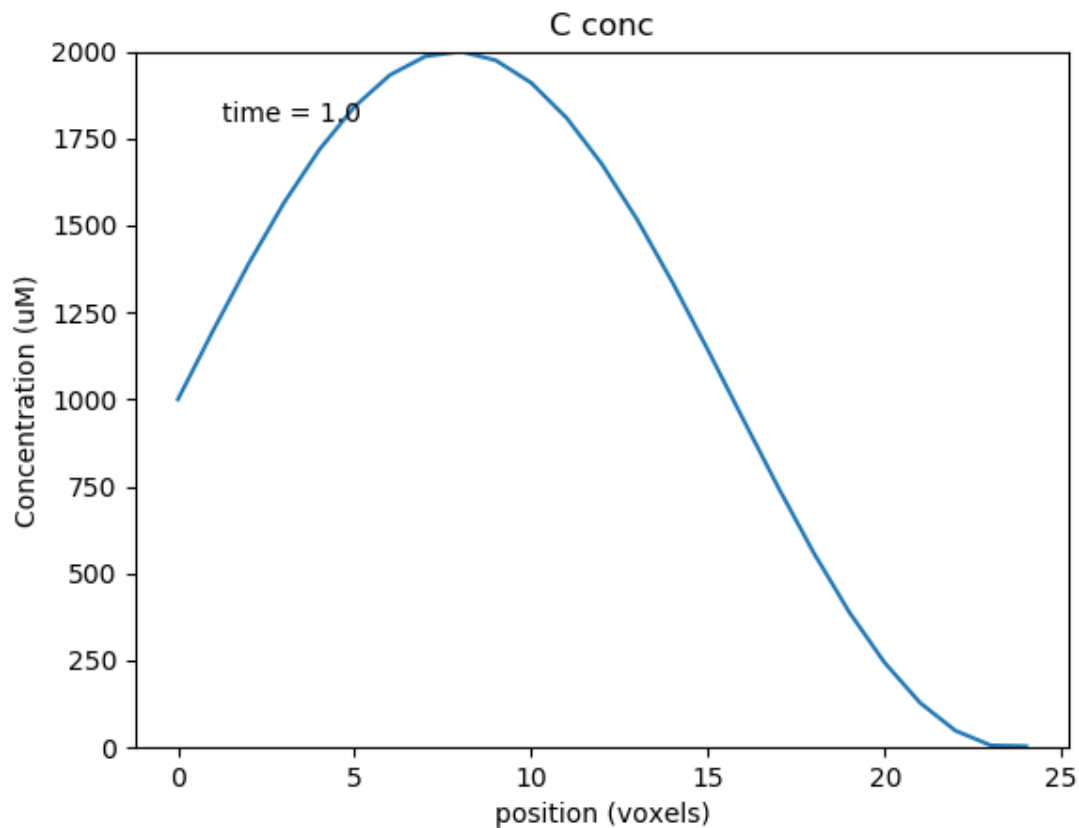
makeFuncRate()
rdes = rd.rdesignneur(
    turnOffElec = True,
    #This subdivides the 50-micron cylinder into 2 micron voxels
    diffusionLength = 2e-6,
    cellProto = [['somaProto', 'soma', 5e-6, 50e-6]],
    chemProto = [['chem', 'chem']],
    chemDistrib = [['chem', 'soma', 'install', '1' ]],
    plotList = [['soma', '1', 'dend/A', 'conc', 'A conc', 'wave'],
                ['soma', '1', 'dend/C', 'conc', 'C conc', 'wave']],
)
rdes.buildModel()
C = moose.element( '/model/chem/dend/C' )
```

(continues on next page)

(continued from previous page)

```
C.vec.concInit = [ 1+np.sin(x/5.0) for x in range( len(C.vec) ) ]
moose.reinit()
moose.start(10)
rdes.display()
```

We plot the controlling molecule C and the substrate molecule A as functions of position, using a waveplot. C remains fixed, and A decreases with time and space. A is smallest at about voxel 8, where the reaction rate, as controlled by C, is highest.



Multiscale models: single compartment

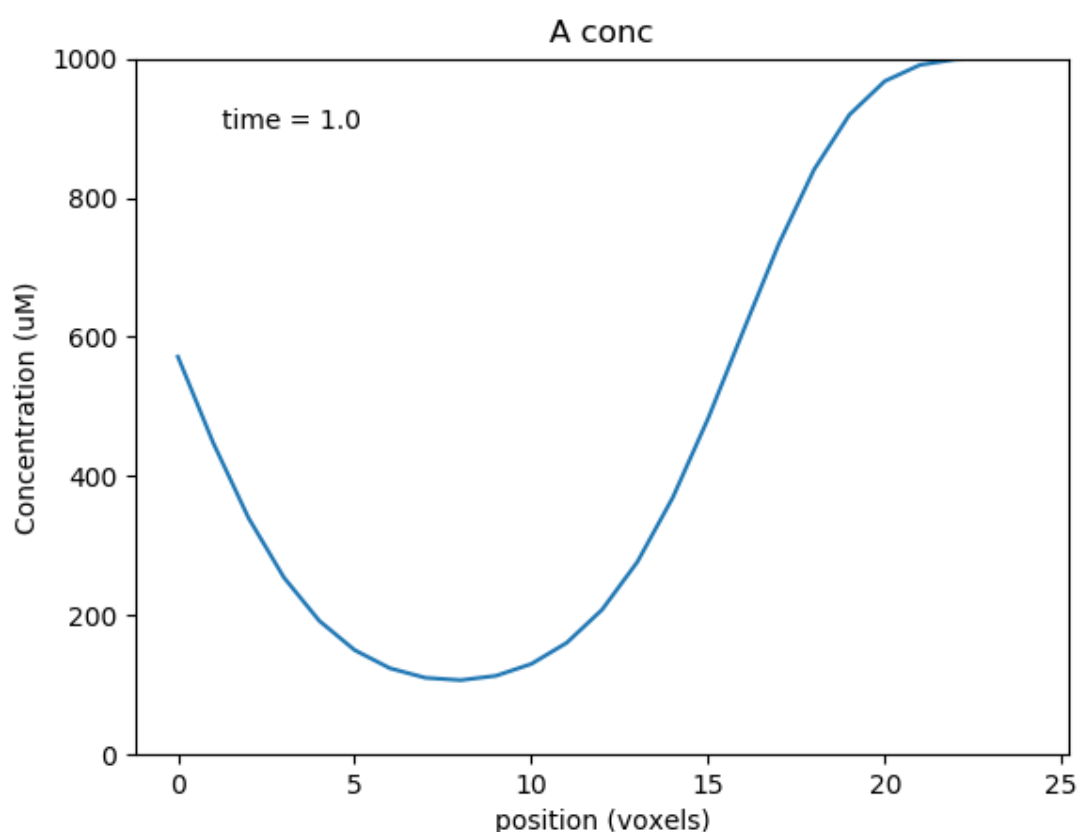
ex8.0_multiscale_KA_phosph.py

The next few examples are for the multiscale modeling that is the main purpose of rdesigneur and MOOSE as a whole. These are 'toy' examples in that the chemical and electrical signaling is simplified, but they exhibit dynamics that are of real interest.

The first example is of a bistable system where the feedback loop comprises of

calcium influx -> chemical activity -> channel modulation -> electrical activity -> calcium influx.

Calcium enters through voltage gated calcium channels, leads to enzyme activation and phosphorylation of a KA channel, which depolarizes the cell, so it spikes more, so more calcium enters.



```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    elecDt = 50e-6,
    chemDt = 0.002,
    chemPlotDt = 0.002,
    # cellProto syntax: ['somaProto', 'name', 'dia', 'length']
    cellProto = [['somaProto', 'soma', 12e-6, 12e-6]],
    chemProto = [['./chem/chanPhosphByCaMKII.g', 'chem']],
    chanProto = [
        ['make_Na()', 'Na'],
        ['make_K_DR()', 'K_DR'],
        ['make_K_A()', 'K_A'],
        ['make_Ca()', 'Ca'],
        ['make_Ca_conc()', 'Ca_conc']
    ],
    # Some changes to the default passive properties of the cell.
    passiveDistrib = [['soma', 'CM', '0.03', 'Em', '-0.06']],
    chemDistrib = [['chem', 'soma', 'install', '1']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '300'],
        ['K_DR', 'soma', 'Gbar', '250'],
        ['K_A', 'soma', 'Gbar', '200'],
        ['Ca_conc', 'soma', 'tau', '0.0333'],
        ['Ca', 'soma', 'Gbar', '40']
    ]
)
```

(continues on next page)

(continued from previous page)

```

    ],
    adaptorList = [
        [ 'dend/chan', 'conc', 'K_A', 'modulation', 0.0, 70 ],
        [ 'Ca_conc', 'Ca', 'dend/Ca', 'conc', 0.00008, 2 ]
    ],
    # Give a + pulse from 5 to 7s, and a - pulse from 20 to 21.
    stimList = [[ 'soma', '1', '.', 'inject', '((t>5 && t<7) - (t>20 &&
→ t<21)) * 1.0e-12' ]],
    plotList = [
        [ 'soma', '1', '.', 'Vm', 'Membrane potential'],
        [ 'soma', '1', '.', 'inject', 'current inj'],
        [ 'soma', '1', 'K_A', 'Ik', 'K_A current'],
        [ 'soma', '1', 'dend/chan', 'conc', 'Unphosph K_A conc'],
        [ 'soma', '1', 'dend/Ca', 'conc', 'Chem Ca'],
    ],
)

rdes.buildModel()
moose.reinit()
moose.start( 30 )

rdes.display()

```

There is only one fundamentally new element in this script:

adaptor List: [*source, sourceField, dest, destField, offset, scale*] The adaptor list maps between molecular, electrical or even structural quantities in the simulation. At present it is linear mapping, in due course it may evolve to an arbitrary function.

The two adaptorLists in the above script do the following:

```
[ 'dend/chan', 'conc', 'K_A', 'modulation', 0.0, 70 ]:
```

Use the concentration of the 'chan' molecule in the 'dend' compartment, to modulate the conductance of the 'K_A' channel such that the basal conductance is zero and 1 millimolar of 'chan' results in a conductance that is 70 times greater than the baseline conductance of the channel, *Gbar*.

It is advisable to use the field '*modulation*' on channels undergoing scaling, rather than to directly assign the conductance '*Gbar*'. This is because *Gbar* is an absolute conductance, and therefore it is scaled to the area of the electrical segment. This makes it difficult to keep track of. *Modulation* is a simple multiplier term onto *Gbar*, and is therefore easier to work with.

```
[ 'Ca_conc', 'Ca', 'dend/Ca', 'conc', 0.00008, 2 ]:
```

Use the concentration of *Ca* as computed in the electrical model, to assign the concentration of molecule *Ca* on the dendrite compartment. There is a basal level of 80 nanomolar, and every unit of electrical *Ca* maps to 2 millimolar of chemical *Ca*.

The arguments in the adaptorList are:

- **Source and Dest:** Strings. These can be either a molecular or an electrical object. To identify a molecular object, it should be prefixed with the name of the chemical compartment, which is one of *dend*, *spine*, *psd*. Thus *dend/chan* specifies a molecule named 'chan' sitting in the 'dend' compartment.

To identify an electrical object, just pass in its path, such as '.' or 'Ca_conc'.

Note that the adaptors do **not** need to know anything about the location. It is assumed that the adaptors do their job wherever the specified source and dest coexist. There is a subtlety here due to the different length and time scales. The rule of thumb is that the adaptor averages whichever one is subdivided more finely.

- Example 1: Molecules are typically spatially partitioned into short voxels (micron-scale) compared to typical 100-micron electrical segments. So an adaptor going from molecules to, say, channel conductance, would average all the molecular voxels that fit in the electrical segment.
- Example 2: Electrical activity is typically much faster than chemical. So an adaptor going from an electrical entity (Ca computed from channel opening) to molecules (Chemical Ca concentration) would average all the time-steps between updates to the molecule.
- **Fields:** Strings. These are simply the field names on the objects coupled by the adaptors.
- **offset and scale:** Doubles. At present the adaptor is just a straight-line conversion, obeying $y = mx + c$. The computed output is y , averaged input is x , offset is c and scale is m .

There is a handy new line to specify cellular passive properties:

passiveDistrib: *[path, field, value, field, value, ...]*,

- **path:** String. Specifies the object whose parameters are to be changed.
- **field:** String. Name of the field on the object.
- **value:** String, that is the value has to be enclosed in quotes. The value to be assigned to the object.

With these in place, the model behavior is rather neat. It starts out silent, then we apply 2 seconds of +ve current injection.

The cell fires briskly, and keeps firing even when the current injection drops to zero.

The firing of the neuron leads to Ca influx.

The chemical reactions downstream of Ca lead to phosphorylation of the K_A channel. Only the unphosphorylated K_A channel is active, so the net effect is to reduce K_A conductance while the Ca influx persists.

Since the phosphorylated form has low conductance, the cell becomes more excitable and keeps firing even when the current injection is stopped. It takes a later, -ve current injection to turn the firing off again.

Suggestions for things to do with the model:

- Vary the adaptor settings, which couple electrical to chemical signaling and vice versa.
- Play with the channel densities
- Open the chem model in moosegui and vary its parameters too.

Multiscale model of CICR in dendrite triggered by synaptic input

ex8.1_synTrigCICR.py

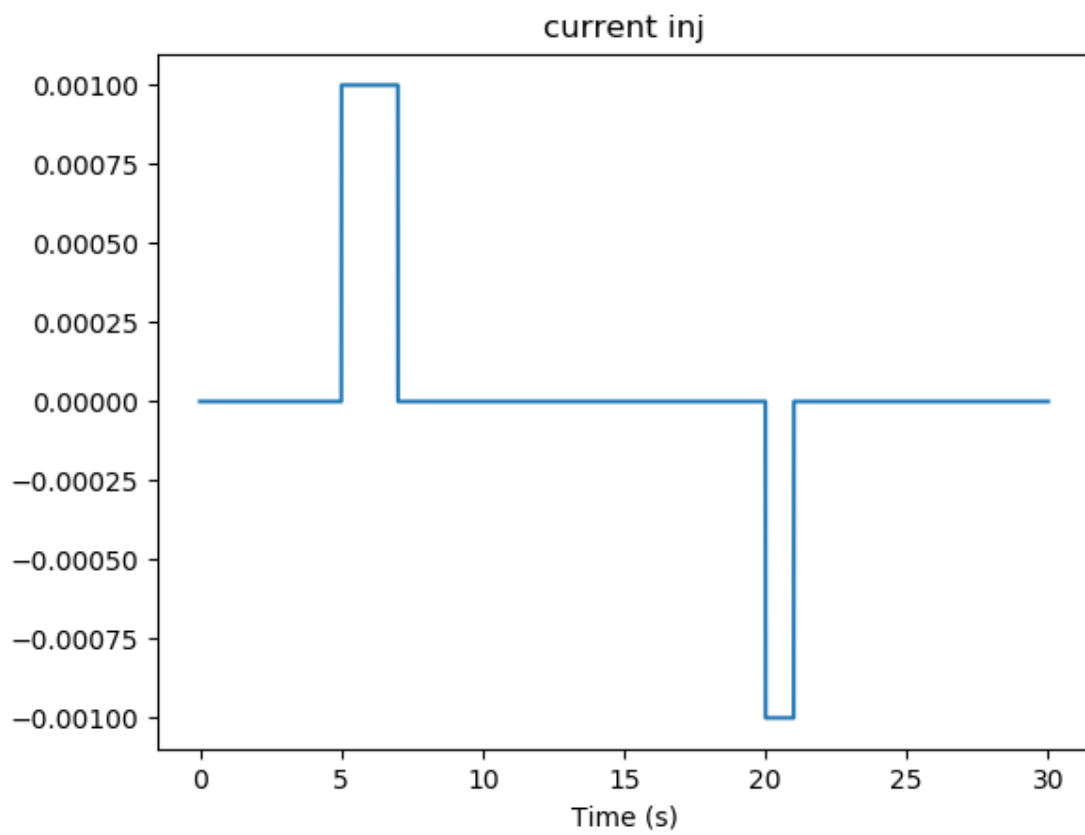


Fig. 23: Current injection stimuli for multiscale model.

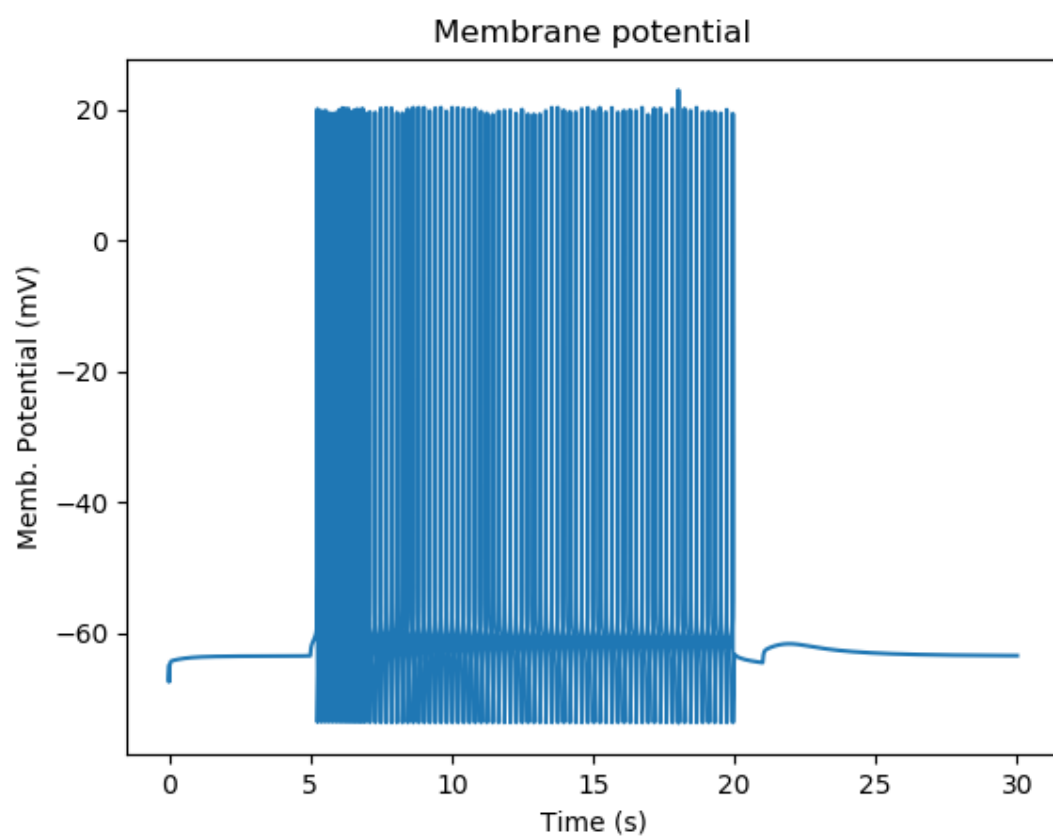


Fig. 24: Firing responses of cell with multiscale signaling.

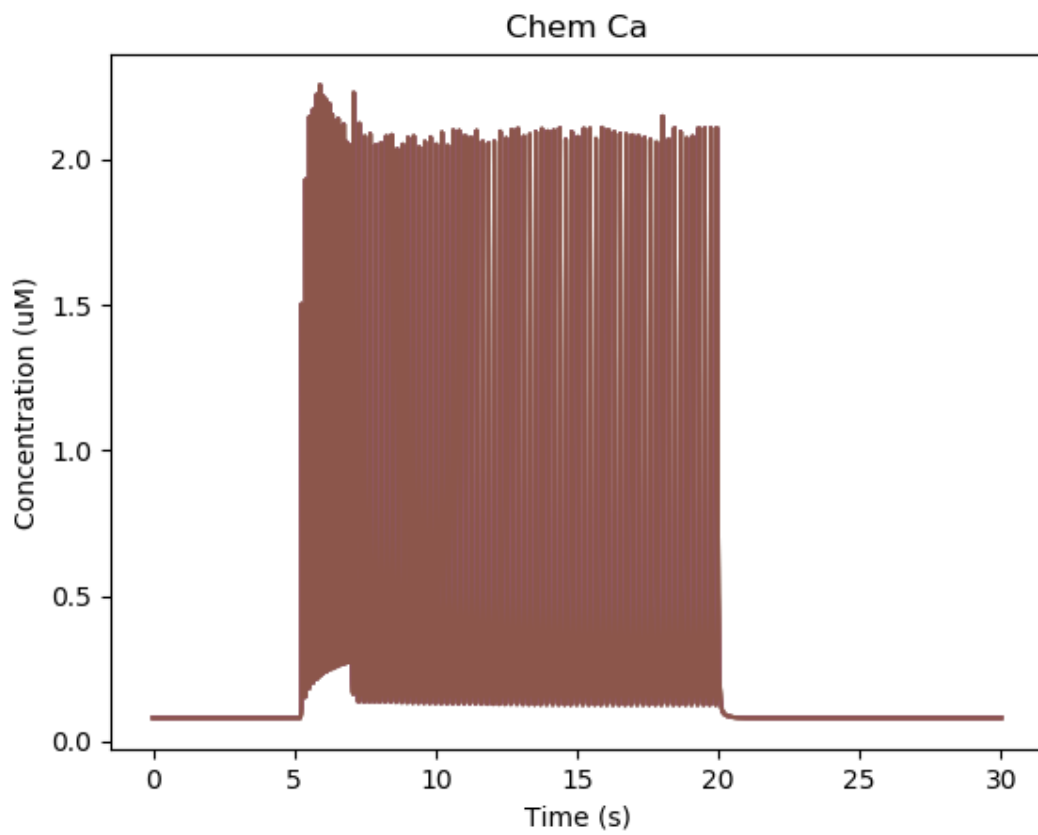


Fig. 25: Calcium buildup in cell due to firing.

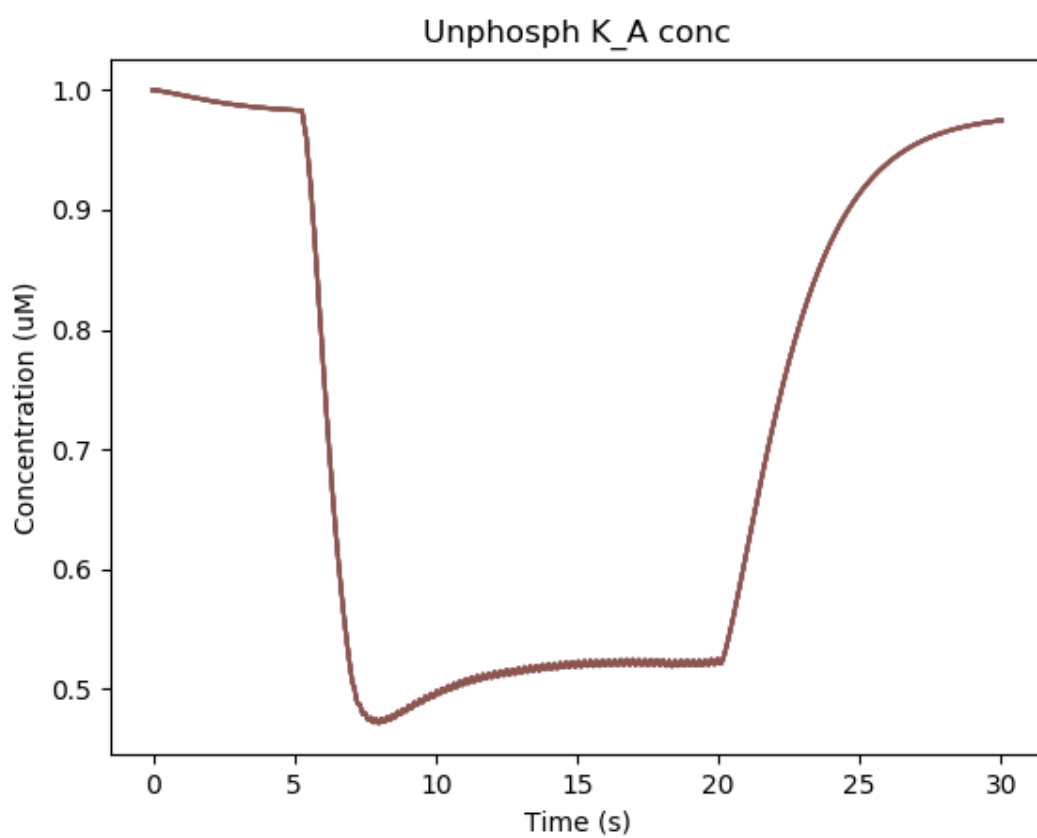


Fig. 26: Removal of KA channel due to phosphorylation.

In this model synaptic input arrives at a dendritic spine, leading to calcium influx through the NMDA receptor. An adaptor converts this influx to the concentration of a chemical species, and this then diffuses into the dendrite and sets off the CICR.

This example models Calcium events in three compartments: dendrite, ER inside dendrite, and spine. The signaling is a slight change from the toy model used in *ex7.2_CICR.py*. Note how the range of CICR wave propagation is limited by a domain of the dendrite in which the level of IP3 is elevated.

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    turnOffElec = False,
    chemDt = 0.002,
    chemPlotDt = 0.02,
    diffusionLength = 1e-6,
    numWaveFrames = 50,
    useGssa = False,
    addSomaChemComp = False,
    addEndoChemComp = True,
    # cellProto syntax: ['ballAndStick', 'name', somaDia, somaLength,
    ↪dendDia, dendLength, numDendSeg]
    cellProto = [['ballAndStick', 'soma', 10e-6, 10e-6, 2e-6, 40e-6,
    ↪4]],
    spineProto = [['makeActiveSpine()', 'spine']],
    chemProto = [['./chem/CICRspineDend.g', 'chem']],
    spineDistrib = [['spine', '#dend#', '10e-6', '0.1e-6']],
    chemDistrib = [['chem', 'dend#', 'spine#', 'head#', 'install', '1' ]],
    adaptorList = [
        [ 'Ca_conc', 'Ca', 'spine/Ca', 'conc', 0.00008, 8 ]
    ],
    stimList = [
        ['head0', '0.5', 'glu', 'periodicsyn', '1 + 40*(t>5 && t<6)'],
        ['head0', '0.5', 'NMDA', 'periodicsyn', '1 + 40*(t>5 && t<6)
    ↪'],
        ['dend#', 'g>10e-6 && g<=31e-6', 'dend/IP3', 'conc', '0.0006
    ↪' ],
    ],
    plotList = [
        ['head#', '1', 'spine/Ca', 'conc', 'Spine Ca conc'],
        ['dend#', '1', 'dend/Ca', 'conc', 'Dend Ca conc'],
        ['dend#', '1', 'dend/Ca', 'conc', 'Dend Ca conc', 'wave'],
        ['dend#', '1', 'dend_endo/CaER', 'conc', 'ER Ca conc', 'wave
    ↪'],
        ['soma', '1', '.', 'Vm', 'Memb pot1'],
    ],
)
moose.seed( 1234 )
rdes.buildModel()
moose.reinit()
moose.start( 16 )
rdes.display()
```

The demo illustrates how to specify the range of elevated IP3 in the *stimList* using the second argument, which selects a geometric range of electrical compartments.

```
[ 'dend#', 'g>10e-6 && g<=31e-6', 'dend/IP3', 'conc', '0.0006' ]
```

This means to look at all dendrite compartments (first argument), and select those which are between a geometrical distance g of 10 to 31 microns from the soma (second argument). The system then sets the IP3 concentration (third and fourth arguments) to 0.6 μM (last argument) for all the chemical voxels embedded in these dendrite compartments.

A note on defining the endo compartments: In cases like this, where the compartment identity isn't built into the chemical model definition, we need a heuristic to decide which compartment is which. The heuristic used in *rdesigner* goes like this:

- Sort chemical compartments in decreasing order by volume
- If the `addSomaChemCompt` flag is **true**, they are assigned to *soma*, *dendrite*, *spine-head*, *spine-psd*, depending on how many compartments are specified. If the flag is **false**, the soma is omitted.
- If the `addEndoChemCompt` is **true**, then alternate compartments are assigned to the `endo_compartment`. Here it is *dend*, *dend_endo*, *spine-head*. If we had six compartments defined (no soma) it would have been: *dend*, *dend_endo*, *spine-head*, *spine-endo*, *psd*, *psd_endo*. The *psd-endo* doesn't make a lot of biological sense, though.

When we run this model, we trigger a propagating Ca wave from about voxel number 16 of 40. It spreads in both directions, and comes to a halt at voxels 10 and 30, which mark the limits of the IP3 elevation zone.

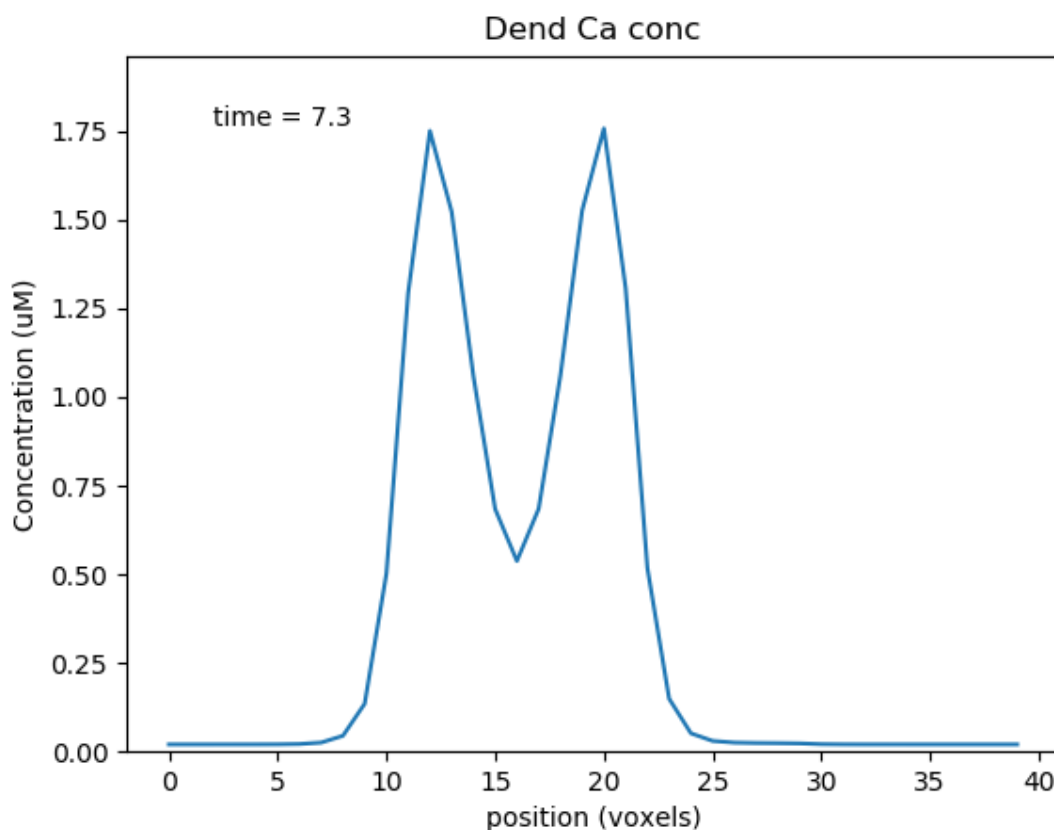


Fig. 27: Calcium wave propagation along the dendrite

Note two subtle effects on the ER Ca concentration: first, there is a periodic small influx of calcium at voxel 16 due to synaptic input. Second, there is a slow restoration of the ER Ca level toward baseline due to diffusion in the dendrite and the action of pumps to within the ER, and out of the cell. Note also that the gradient within the ER is actually quite small, being about a 12% deviation from the resting calcium.

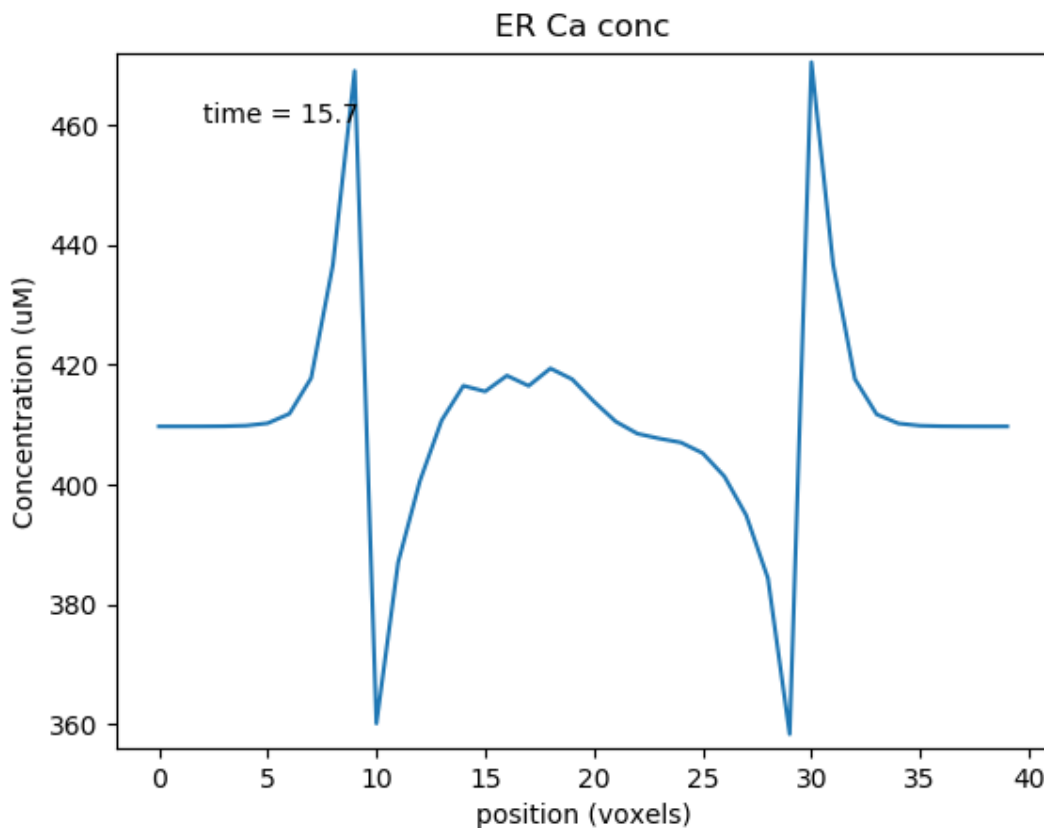


Fig. 28: Calcium depletion and buildup in the ER due to CICR wave.

Multiscale model spanning PSD, spine head and dendrite

ex8.2_multiscale_glurR_phosph_3compt.py

This is another multiscale model on similar lines to 8.0. It is structurally and computationally more complicated, because the action is distributed between spines and dendrites, but formally it does the same thing: it turns on and stays on after a strong stimulus, due to phosphorylation of a (receptor) channel leading to greater excitability.

calcium influx -> chemical activity -> channel modulation -> electrical activity -> calcium influx.

The model is bistable as long as synaptic input keeps coming along at a basal rate, in this case 1 Hz.

Here we have two new lines, to do with addition of spines. These are discussed in detail in a later example. For now it is enough to know that the **spineProto** line defines one of the prototype spines to be used to put into the model, and the **spineDistrib** line tells the system where to put them, and how widely to space them.

```

import moose
import rdesigneur as rd
rdes = rd.rdesigneur(
    elecDt = 50e-6,
    chemDt = 0.002,
    diffDt = 0.002,
    chemPlotDt = 0.02,
    useGssa = False,
    # cellProto syntax: ['ballAndStick', 'name', somaDia, somaLength,
    ↪dendDia, d
endLength, numDendSegments ]
    cellProto = [['ballAndStick', 'soma', 12e-6, 12e-6, 4e-6, 100e-6,
    ↪2 ]],
    chemProto = [['./chem/chanPhosph3compt.g', 'chem']],
    spineProto = [['makeActiveSpine()', 'spine']],
    chanProto = [
        ['make_Na()', 'Na'],
        ['make_K_DR()', 'K_DR'],
        ['make_K_A()', 'K_A' ],
        ['make_Ca()', 'Ca' ],
        ['make_Ca_conc()', 'Ca_conc' ]
    ],
    passiveDistrib = [['soma', 'CM', '0.01', 'Em', '-0.06']],
    spineDistrib = [['spine', '#dend#', '50e-6', '1e-6']],
    chemDistrib = [['chem', '#', 'install', '1' ]],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '300' ],
        ['K_DR', 'soma', 'Gbar', '250' ],
        ['K_A', 'soma', 'Gbar', '200' ],
        ['Ca_conc', 'soma', 'tau', '0.0333' ],
        ['Ca', 'soma', 'Gbar', '40' ]
    ],
    adaptorList = [
        [ 'psd/chan_p', 'n', 'glu', 'modulation', 0.1, 1.0 ],
        [ 'Ca_conc', 'Ca', 'spine/Ca', 'conc', 0.00008, 8 ]
    ],
    # Syn input baseline 1 Hz, and 40Hz burst for 1 sec at t=20. Syn_
    ↪weight
    # is 0.5, specified in 2nd argument as a special case stimLists.
    stimList = [['head#', '0.5', 'glu', 'periodicsyn', '1 + 40*(t>10 &&
    ↪ t<11)']],
    plotList = [
        ['soma', '1', '.', 'Vm', 'Membrane potential'],
        ['#', '1', 'spine/Ca', 'conc', 'Ca in Spine'],
        ['#', '1', 'dend/DEND/Ca', 'conc', 'Ca in Dend'],
        ['#', '1', 'spine/Ca_CaM', 'conc', 'Ca_CaM'],
        ['head#', '1', 'psd/chan_p', 'conc', 'Phosph gluR'],
        ['head#', '1', 'psd/Ca_CaM_CaMKII', 'conc', 'Active CaMKII'],
    ]
)
moose.seed(123)
rdes.buildModel()
moose.reinit()
moose.start( 25 )
rdes.display()

```

This is how it works:

This is a ball-and-stick model with a couple of spines sitting on the dendrite. The spines get synaptic input onto NMDARs and gluRs. There is a baseline input rate of 1 Hz throughout, and there is a burst at 40 Hz for 1 second at $t = 10$ s.

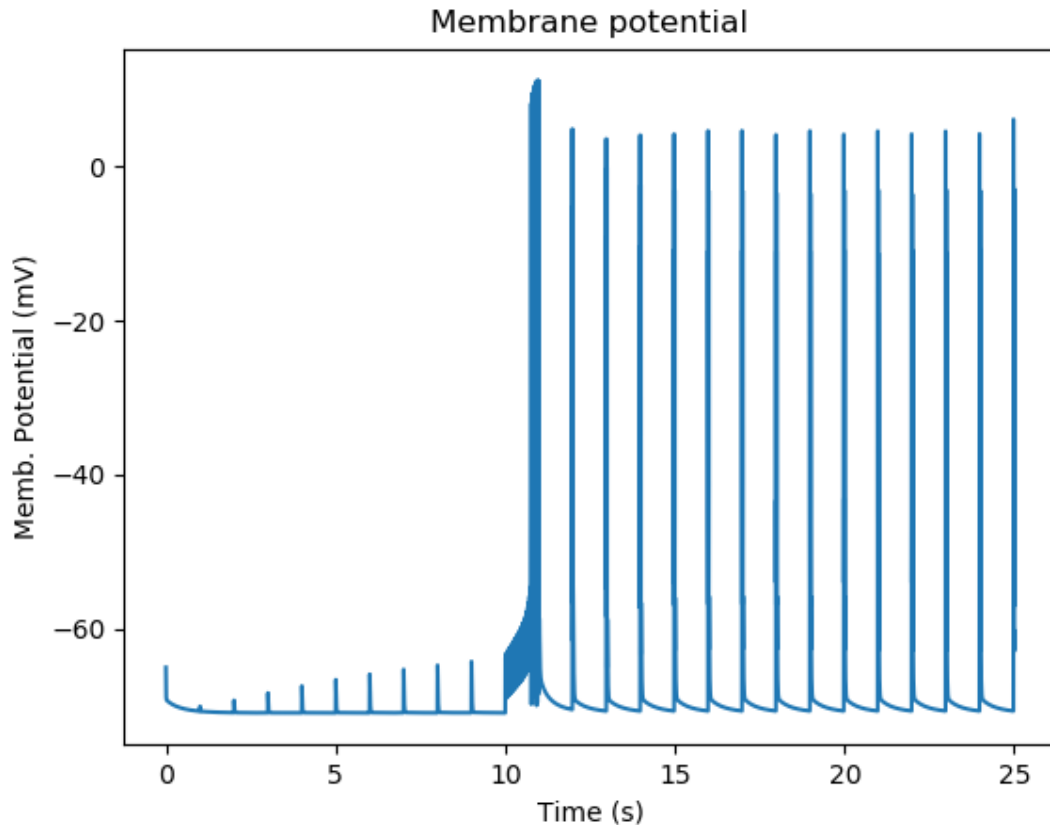


Fig. 29: Membrane potential responses of cell with synaptic input and multiscale signaling

At baseline, we just have small EPSPs and little Ca influx. A burst of strong synaptic input causes Ca entry into the spine via NMDAR.

Ca diffuses from the spine into the dendrite and spreads. In the graph below we see how Calcium goes into the 50-odd voxels of the dendrite.

The Ca influx into the spine triggers activation of CaMKII and its translocation to the PSD, where it phosphorylates and increases the conductance of gluR. We have two spines with slightly different geometry, so the CaMKII activity differs slightly.

Now that gluR has a greater weight, the baseline synaptic input keeps Ca trickling in enough to keep the CaMKII active.

Here are the reactions:

```
Ca+CaM <==> Ca_CaM;      Ca_CaM + CaMKII <==> Ca_CaM_CaMKII (all in
spine head, except that the Ca_CaM_CaMKII translocates to the PSD)

chan -----Ca_CaM_CaMKII-----> chan_p; chan_p -----> chan (all in_
->PSD)
```

Suggestions:

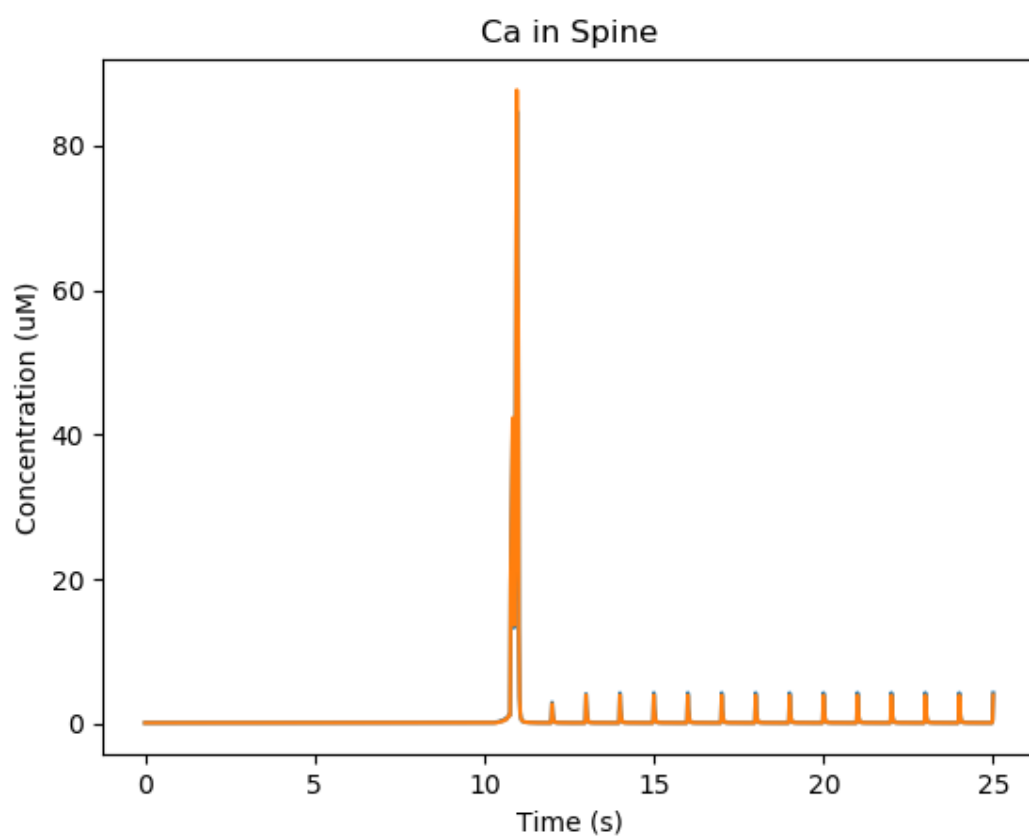


Fig. 30: Calcium influx into spine.

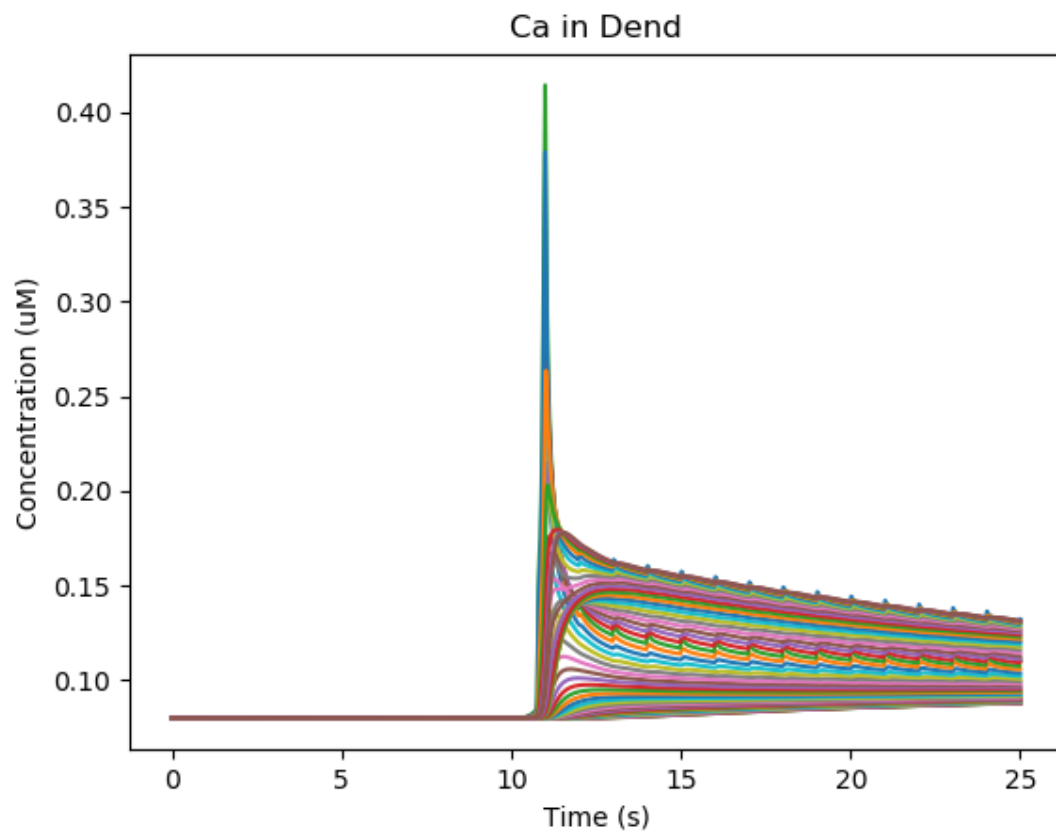


Fig. 31: Calcium influx and diffusion in dendrite.

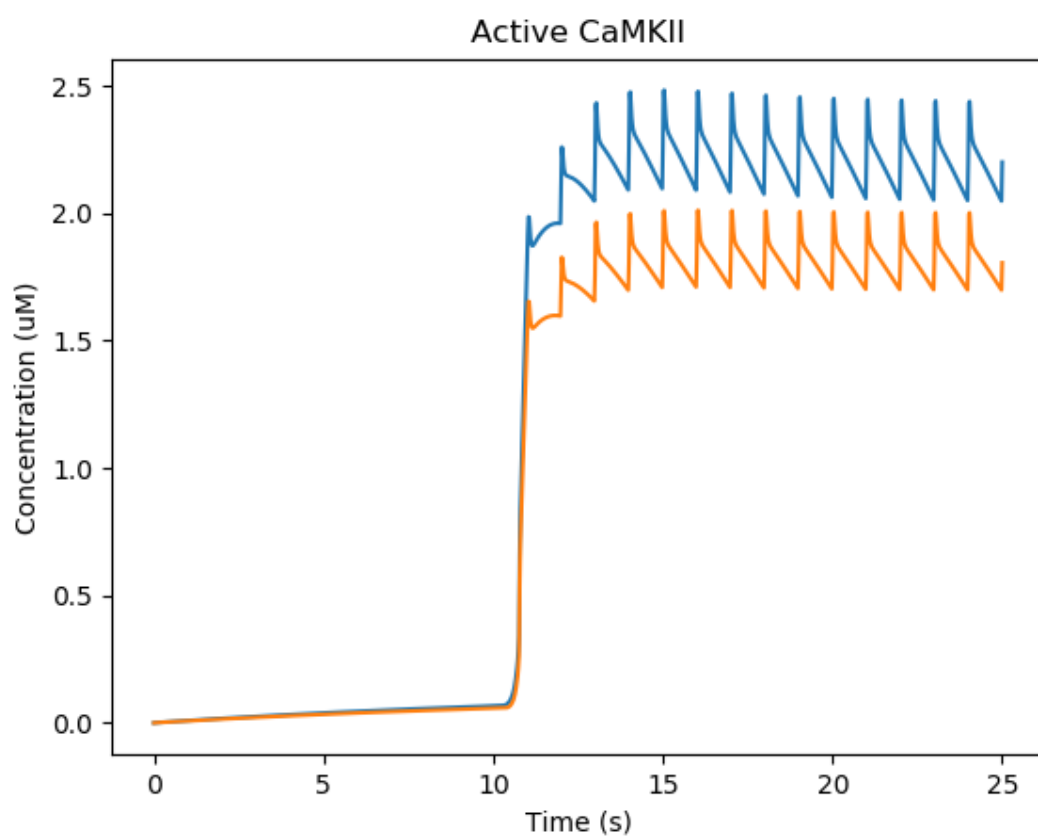


Fig. 32: Activation of CaMKII and translocation to PSD

- Add GABAR using `make_GABA()`, put it on soma or dendrite. Stimulate it after 20 s to see if you can turn off the sustained activation
- Replace the 'periodicsyn' in `stimList` with 'randsyn'. This gives Poisson activity at the specified mean frequency. Does the switch remain reliable?
- What are the limits of various parameters for this switching? You could try basal synaptic rate, burst rate, the various scaling factors for the adaptors, the densities of various channels, synaptic weight, and so on.
- In real life an individual synaptic EPSP is tiny, under a millivolt. How many synapses would you need to achieve this kind of switching? You can play with # of synapses by altering the spacing between spines as the third argument of `spineDistrib`.

Multiscale model in which spine geometry changes due to signaling

ex8.3_spine_vol_change.py

This model is very similar to 8.2. The main design difference is that *adaptor*, instead of just modulating the `gluR` conductance, scales the entire spine cross-section area, with all sorts of electrical and chemical ramifications. There are a lot of plots, to illustrate some of these outcomes.

```
import moose
import rdesigneur as rd
rdes = rd.rdesigneur(
    elecDt = 50e-6,
    chemDt = 0.002,
    diffDt = 0.002,
    chemPlotDt = 0.02,
    useGssa = False,
    stealCellFromLibrary = True, # Simply move library model to use_
    for sim
    cellProto = [['ballAndStick', 'soma', 12e-6, 12e-6, 4e-6, 100e-6,
    2 ]],
    chemProto = [['./chem/chanPhosph3compt.g', 'chem']],
    spineProto = [['makeActiveSpine()', 'spine']],
    chanProto = [
        ['make_Na()', 'Na'],
        ['make_K_DR()', 'K_DR'],
        ['make_K_A()', 'K_A'],
        ['make_Ca()', 'Ca'],
        ['make_Ca_conc()', 'Ca_conc']
    ],
    passiveDistrib = [['soma', 'CM', '0.01', 'Em', '-0.06']],
    spineDistrib = [['spine', '#dend#', '50e-6', '1e-6']],
    chemDistrib = [['chem', '#', 'install', '1']],
    chanDistrib = [
        ['Na', 'soma', 'Gbar', '300'],
        ['K_DR', 'soma', 'Gbar', '250'],
        ['K_A', 'soma', 'Gbar', '200'],
        ['Ca_conc', 'soma', 'tau', '0.0333'],
        ['Ca', 'soma', 'Gbar', '40']
    ],
    adaptorList = [
        # This scales the psdArea of the spine by # of chan_p. Note_
    that
```

(continues on next page)

(continued from previous page)

```

# the cross-section area of the spine head is identical to
→psdArea.
[ 'psd/chan_p', 'n', 'spine', 'psdArea', 0.1e-12, 0.01e-12 ],
[ 'Ca_conc', 'Ca', 'spine/Ca', 'conc', 0.00008, 8 ]
],
# Syn input baseline 1 Hz, and 40Hz burst for 1 sec at t=20. Syn
→wt=10
stimList = [['head#', '10', 'glu', 'periodicsyn', '1 + 40*(t>10 &&
→t<11)']],
plotList = [
[ 'soma', '1', '.', 'Vm', 'Membrane potential'],
[ '#', '1', 'spine/Ca', 'conc', 'Ca in Spine'],
[ '#', '1', 'dend/DEND/Ca', 'conc', 'Ca in Dend'],
[ 'head#', '1', 'psd/chan_p', 'n', 'Amount of Phospho-chan'],
[ 'head#', '1', 'spine/CaMKII', 'conc', 'Conc of CaMKII in
→spine'],
[ 'head#', '1', '.', 'Cm', 'Capacitance of spine head'],
[ 'head#', '1', '.', 'Rm', 'Membrane res of spine head'],
[ 'head#', '1', '.', 'Ra', 'Axial res of spine head'],
[ 'head#', '1', 'glu', 'Gbar', 'Conductance of gluR'],
[ 'head#', '1', 'NMDA', 'Gbar', 'Conductance of NMDAR'],
]
)
moose.seed(123)
rdes.buildModel()
moose.reinit()
moose.start( 25 )
rdes.display()

```

The key *adaptor* line is as follows:

```
[ 'psd/chan_p', 'n', 'spine', 'psdArea', 0.1e-12, 0.01e-12 ]
```

Here, we use the phosphorylated *chan_p* molecule in the PSD as a proxy for processes that control spine size. We operate on a special object called *spine* which manages many aspects of spines in the model (see below). Here we control the *psdArea*, which defines the cross-section area of the spine head and by extension of the PSD too. We keep a minimum spine area of 0.1 μm^2 , and a scaling factor of 0.01 μm^2 per phosphorylated molecule.

The reaction system is identical to the one in *ex8.2*:

```

Ca+CaM <==> Ca_CaM;      Ca_CaM + CaMKII <==> Ca_CaM_CaMKII (all in
spine head, except that the Ca_CaM_CaMKII translocates to the PSD)

chan -----Ca_CaM_CaMKII-----> chan_p; chan_p -----> chan (all in
→PSD)

```

Rather than list all the 10 plots, here are a few to show what is going on.

First, just the spiking activity of the cell. Here the burst of activity is followed by a few seconds of enhanced synaptic weight, followed by subthreshold EPSPs:

Then, we fast-forward to the amount of *chan_p* which is the molecule that controls spine size scaling:

This causes some obvious outcomes. One of them is to increase the synaptic conductance of the glutamate receptor. The system assumes that the conductance of all channels in the PSD

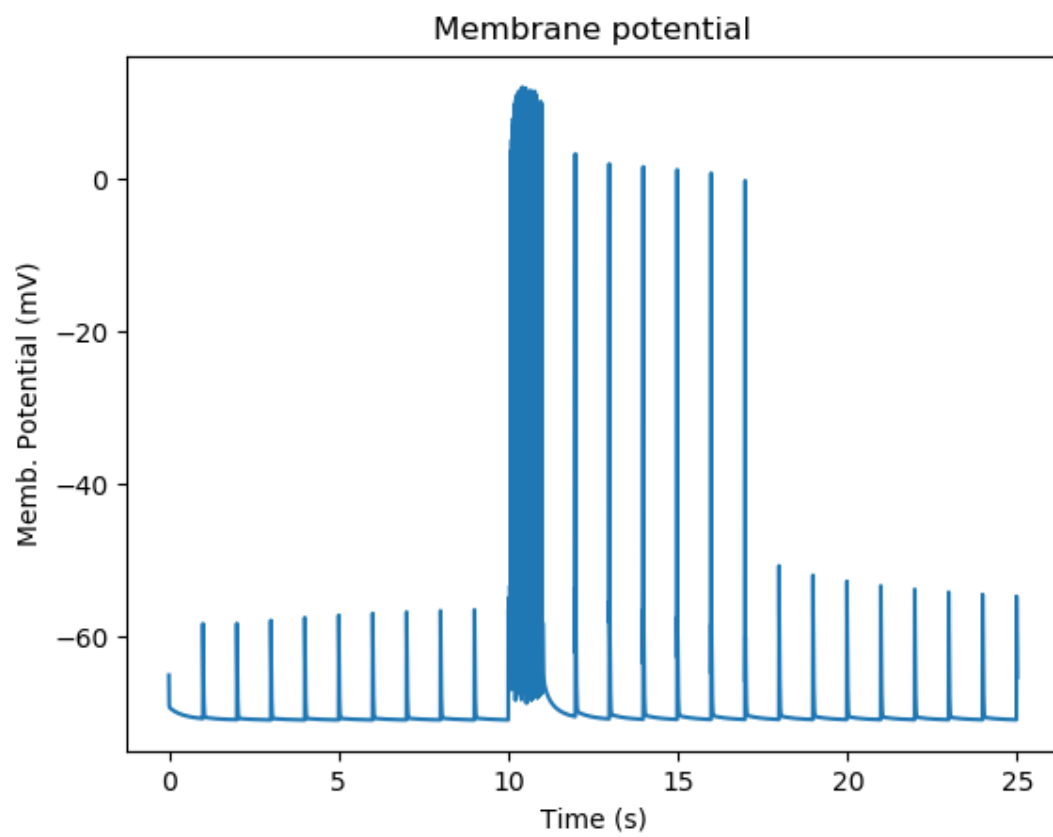


Fig. 33: Membrane potential and spiking.

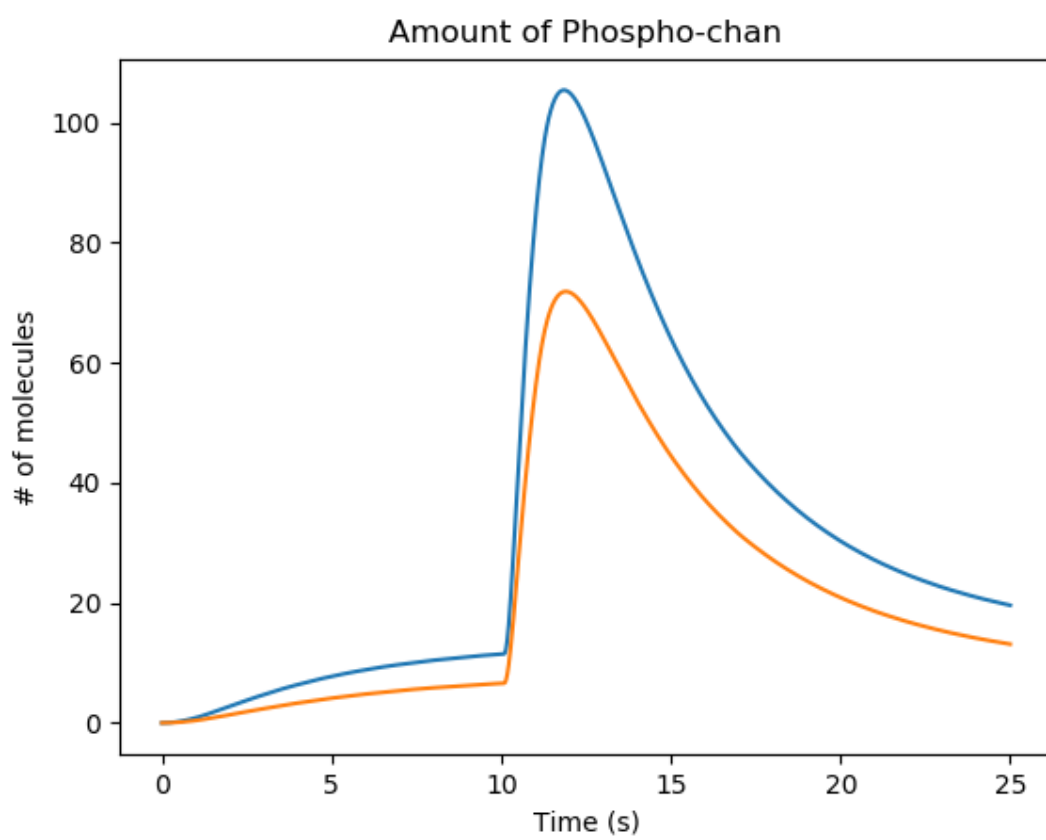


Fig. 34: Molecule that controles spine size

scales linearly with the `psdArea`.

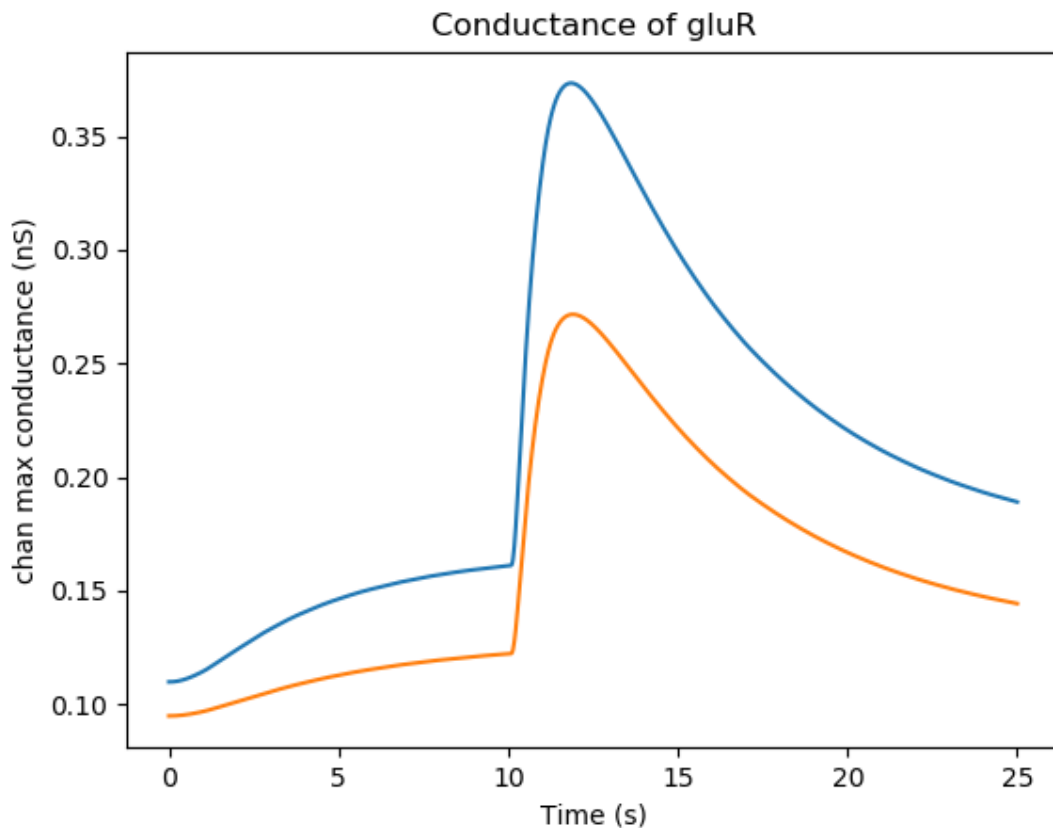


Fig. 35: Conductance of glutamate receptor

Here is one of several non-intuitive outcomes. Because the spine volume has increased, the concentration of molecules in the spine is diluted out. So the concentration of active CaMKII actually falls when the spine gets bigger. In a more detailed model, this would be a race between the increase in spine size and the time taken for diffusion and further reactions to replenish CaMKII. In the current model we don't have a diffusive coupling of CaMKII to the dendrite, so this replenishment doesn't happen.

In the simulation we display several other electrical and chemical properties that change with spine size. The diffusion properties also change since the cross-section areas are altered. This is harder to visualize but has large effects on coupling to the dendrite, especially if the *shaftDiameter* is the parameter scaled by the signaling.

Suggestions:

- The Spine class (instance: spine) manages several possible scaling targets on the spine geometry: `shaftLength`, `shaftDiameter`, `headLength`, `headDiameter`, `psdArea`, `headVolume`, `totalLength`. Try them out. Think about mechanisms by which molecular concentrations might affect each.
- When volume changes, we assume that the molecular numbers stay fixed, so concentration changes. Except for buffered molecules, where we assume concentration remains fixed. Use this to design a bistable simply relying on molecules and spine geometry terms.

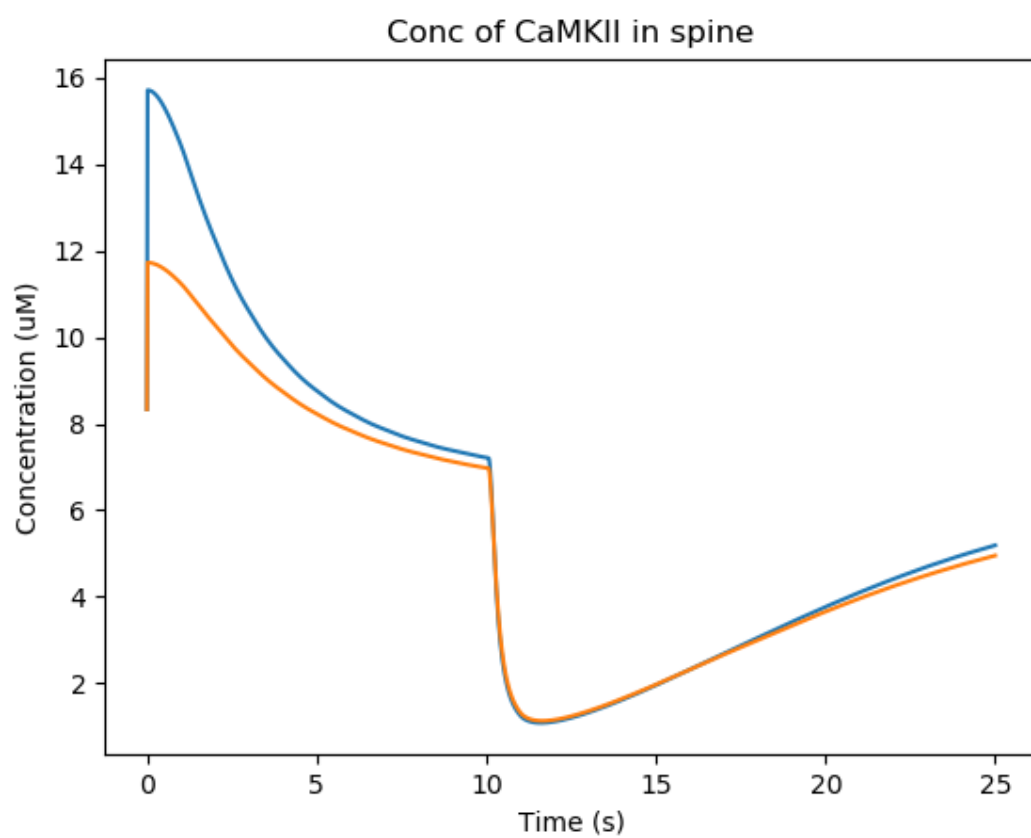


Fig. 36: Concentration of CaMKII in the spine

- Even more interesting, use it to design an oscillator. You could look at Bhalla, BiophysJ 2011 for some ideas.

Morphology: Load .swc morphology file and view it

ex9.0_load_neuronal_morphology_file.py

Here we build a passive model using a morphology file in the .swc file format (as used by NeuroMorpho.org). The morphology file is predefined for Rdesigneur and resides in the directory ./cells. We apply a somatic current pulse, and view the somatic membrane potential in a plot, as before. To make things interesting we display the morphology in 3-D upon which we represent the membrane potential as colors.

```
import sys
import moose
import rdesigneur as rd

if len( sys.argv ) > 1:
    fname = sys.argv[1]
else:
    fname = './cells/h10.CNG.swc'
rdes = rd.rdesigneur(
    cellProto = [[fname, 'elec']],
    stimList = [['soma', '1', '.', 'inject', 't * 25e-9']],
    plotList = [['#', '1', '.', 'Vm', 'Membrane potential'],
                ['#', '1', 'Ca_conc', 'Ca', 'Ca conc (uM)']],
    moogList = [['#', '1', '.', 'Vm', 'Soma potential']]
)
rdes.buildModel()
moose.reinit()
rdes.displayMoogli( 0.001, 0.1, rotation = 0.02 )
```

Here the new concept is the cellProto line, which loads in the specified cell model:

```
[ filename, cellname ]
```

The system recognizes the filename extension and builds a model from the swc file. It uses the cellname **elec** in this example.

We use a similar line as in the reaction-diffusion example, to build up a Moogli display of the cell model:

```
moogList = [['#', '1', '.', 'Vm', 'Soma potential']]
```

Here we have:

```
# : the path to use for selecting the compartments to display.
This wildcard means use all compartments.
1 : The expression to use for the compartments. Again, 1 means use
all of them.
. : Which object in the compartment to display. Here we are using the
compartment itself, so it is just a dot.
Vm : Field to display
Soma potential : Title for display.
```

Suggestions:

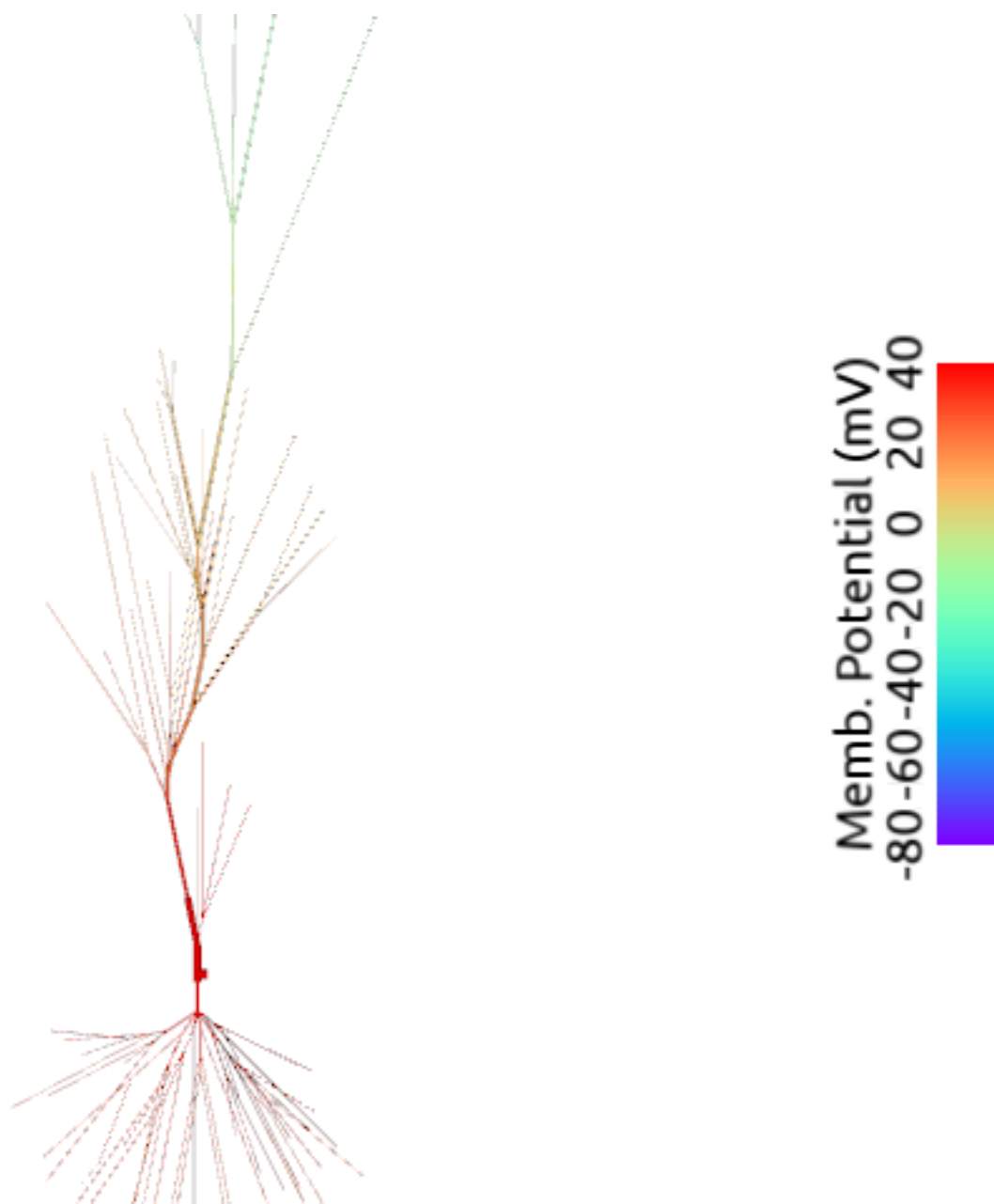


Fig. 37: 3-D display for passive neuron

- The tutorial directory already has a number of pre-loaded files from NeuroMorpho. Pass them in to ex9.0 on the command line:

```
python ex9.0_load_neuronal_morphology_file.py <morpho.swc>
```

- Grab other morphology files from NeuroMorpho.org, try them out.

Build an active neuron model by putting channels into a morphology file

ex9.1_chans_in_neuronal_morph.py

Here we load in a morphology file and distribute voltage-gated ion channels over the neuron. The voltage-gated channels are obtained from a number of channelML files, located in the ./channels subdirectory. Since we have a spatially extended neuron, we need to specify the spatial distribution of channel densities too.

```
import moose
import rdesigneur as rd
rdes = rd.rdesigneur(
    chanProto = [
        ['./chans/hd.xml'],
        ['./chans/kap.xml'],
        ['./chans/kad.xml'],
        ['./chans/kdr.xml'],
        ['./chans/na3.xml'],
        ['./chans/nax.xml'],
        ['./chans/CaConc.xml'],
        ['./chans/Ca.xml']
    ],
    cellProto = [['./cells/h10.CNG.swc', 'elec']],
    chanDistrib = [ \
        ["hd", "#dend#,#apical#", "Gbar", "50e-2*(1+(p*3e4))" ],
        ["kdr", "#", "Gbar", "p < 50e-6 ? 500 : 100" ],
        ["na3", "#soma#,#dend#,#apical#", "Gbar", "850" ],
        ["nax", "#soma#,#axon#", "Gbar", "1250" ],
        ["kap", "#axon#,#soma#", "Gbar", "300" ],
        ["kap", "#dend#,#apical#", "Gbar",
            "300*(H(100-p*1e6)) * (1+(p*1e4))" ],
        ["Ca_conc", "#", "tau", "0.0133" ],
        ["kad", "#soma#,#dend#,#apical#", "Gbar", "50" ],
        ["Ca", "#", "Gbar", "50" ]
    ],
    stimList = [['soma', '1', '.', 'inject', '(t>0.02) * 1e-9' ]],
    plotList = [['#', '1', '.', 'Vm', 'Membrane potential'],
        ['#', '1', 'Ca_conc', 'Ca', 'Ca conc (uM)']],
    moogList = [['#', '1', 'Ca_conc', 'Ca', 'Calcium conc (uM)', 0, ↵
        ↵120],
        ['#', '1', '.', 'Vm', 'Soma potential']]
)

rdes.buildModel()

moose.reinit()
rdes.displayMoogli( 0.0002, 0.052 )
```

Here we make more extensive use of two concepts which we've already seen from the single compartment squid model:

1. *chanProto*: This defines numerous channels, each of which is of the form:

```
[ filename ]
```

or

```
[ filename, channelname ]
```

or

```
[ channelFunction(), channelname ]
```

If the *channelname* is not specified the system uses the last part of the channel name, before the filetype suffix.

2. *chanDistrib*: This defines the spatial distribution of each channel type. Each line is of a form that should be familiar now:

```
[channelname, region_in_cell, parameter, expression_string]
```

- The *channelname* is the name of the prototype from *chanproto*. This is usually an ion channel, but in the example above you can also see a calcium concentration pool defined.
- The *region_in_cell* is typically defined using wildcards, so that it generalizes to any cell morphology. For example, the plain wildcard *#* means to consider all cell compartments. The wildcard *#dend#* means to consider all compartments with the string *dend* somewhere in the name. Wildcards can be comma-separated, so *#soma#, #dend#* means consider all compartments with either *soma* or *dend* in their name. The naming in MOOSE is defined by the model file. Importantly, in *.swc* files MOOSE generates names that respect the classification of compartments into axon, soma, dendrite, and apical dendrite compartments respectively. SWC files generate compartment names such as:

```
soma_<number>
dend_<number>
apical_<number>
axon_<number>
```

where the number is automatically assigned by the reader. In order to select all dendritic compartments, for example, one would use *"#dend#"* where the *"#"* acts as a wildcard to accept any string. - The *parameter* is usually *Gbar*, the channel conductance density in S/m^2 . If *Gbar* is zero or less, then the system economizes by not incorporating this channel mechanism in this part of the cell. Similarly, for calcium pools, if the *tau* is below zero then the calcium pool object is simply not inserted into this part of the cell. - The *expression_string* defines the value of the parameter, such as *Gbar*. This is typically a function of position in the cell. The expression evaluator knows about several parameters of cell geometry. All units are in metres:

- *x*, *y* and *z* coordinates.
- *g*, the geometrical distance from the soma
- *p*, the path length from the soma, measured along the dendrites.
- *dia*, the diameter of the dendrite.
- *L*, The electrotonic length from the soma (no units).

Along with these geometrical arguments, we make liberal use of the ternary expressions like *p < 50e-6 ? 500 : 100* or multiplying a channel density with a logical function or Heaviside function *H(x)* to set up the channel distributions. The expression evaluator also knows about

pretty much all common algebraic, trigonometric, and logarithmic functions, should you wish to use these.

Also note the two Moogli displays. The first is the calcium concentration. The second is the membrane potential in each compartment. Easy!

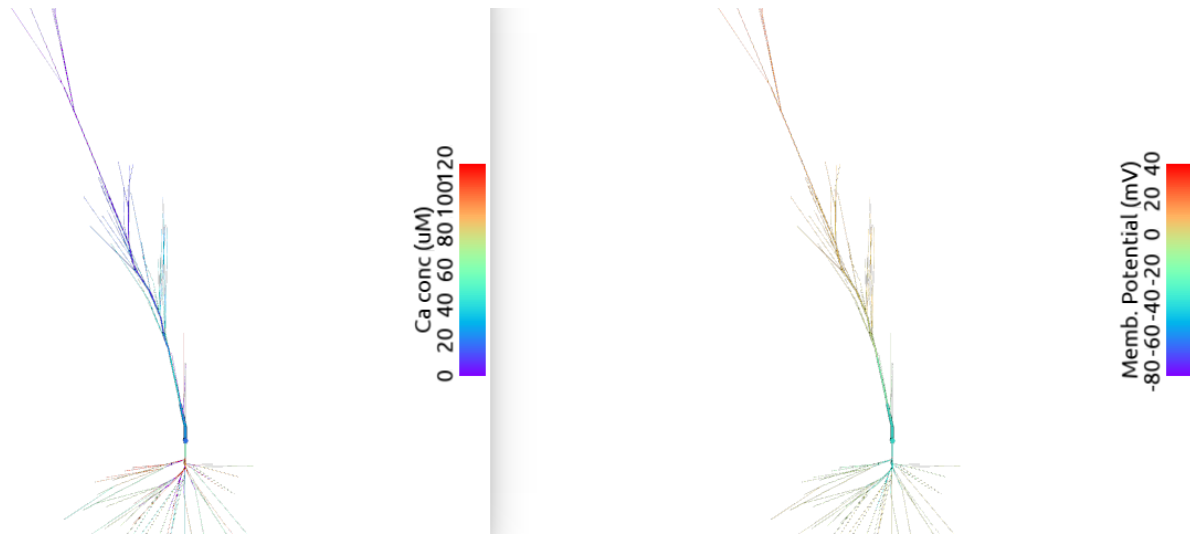


Fig. 38: 3-D display for active neuron

Suggestions:

- Try another morphology file.
- Try different channel distributions by editing the chanDistrib lines.
- There are numerous predefined channels available within Rdesigneur. These can be defined using the following chanProto options:

```
[ 'make_HH_Na()', 'HH_Na' ]
[ 'make_HH_K_DR()', 'HH_K' ]
[ 'make_Na()', 'Na' ]
[ 'make_K_DR()', 'K_DR' ]
[ 'make_K_A()', 'K_A' ]
[ 'make_K_AHP()', 'K_AHP' ]
[ 'make_K_C()', 'K_C' ]
[ 'make_Ca()', 'Ca' ]
[ 'make_Ca_conc()', 'Ca_conc' ]
[ 'make_glu()', 'glu' ]
[ 'make_GABA()', 'GABA' ]
```

Then the chanDistrib can refer to these channels instead.

- Deliver stimuli on the dendrites rather than the soma.

Build a spiny neuron from a morphology file and put active channels in it.

ex9.2_spines_in_neuronal_morpho.py

This model is one step elaborated from the previous one, in that we now also have dendritic spines. MOOSE lets one decorate a bare neuronal morphology file with dendritic spines, spec-

ifying various geometric parameters of their location. As before, we use an swc file for the morphology, and the same ion channels and distribution.

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    chanProto = [
        ['./chans/hd.xml'],
        ['./chans/kap.xml'],
        ['./chans/kad.xml'],
        ['./chans/kdr.xml'],
        ['./chans/na3.xml'],
        ['./chans/nax.xml'],
        ['./chans/CaConc.xml'],
        ['./chans/Ca.xml']
    ],
    cellProto = [['./cells/h10.CNG.swc', 'elec']],
    spineProto = [['makeActiveSpine()', 'spine']],
    chanDistrib = [
        ["hd", "#dend#,#apical#", "Gbar", "50e-2*(1+(p*3e4))" ],
        ["kdr", "#", "Gbar", "p < 50e-6 ? 500 : 100" ],
        ["na3", "#soma#,#dend#,#apical#", "Gbar", "850" ],
        ["nax", "#soma#,#axon#", "Gbar", "1250" ],
        ["kap", "#axon#,#soma#", "Gbar", "300" ],
        ["kap", "#dend#,#apical#", "Gbar",
            "300*(H(100-p*1e6)) * (1+(p*1e4))" ],
        ["Ca_conc", "#", "tau", "0.0133" ],
        ["kad", "#soma#,#dend#,#apical#", "Gbar", "50" ],
        ["Ca", "#", "Gbar", "50" ]
    ],
    spineDistrib = [['spine', '#dend#,#apical#', '20e-6', '1e-6']],
    stimList = [['soma', '1', '.', 'inject', '(t>0.02) * 1e-9' ]],
    plotList = [['#', '1', '.', 'Vm', 'Membrane potential'],
        ['#', '1', 'Ca_conc', 'Ca', 'Ca conc (uM)']],
    moogList = [['#', '1', 'Ca_conc', 'Ca', 'Calcium conc (uM)', 0,
        ↪120],
        ['#', '1', '.', 'Vm', 'Soma potential']]
)

rdes.buildModel()

moose.reinit()
rdes.displayMoogli( 0.0002, 0.023 )
```

Spines are set up in a familiar way: we first define one (or more) prototype spines, and then distribute these around the cell. Here is the prototype string:

```
[spine_proto, spinenam]
```

spineProto: This is typically a function. One can define one's own, but there are several predefined ones in *rdesigneur*. All these define a spine with the following parameters:

- head diameter 0.5 microns
- head length 0.5 microns
- shaft length 1 micron

- shaft diameter of 0.2 microns
- $RM = 1.0$ ohm-metre square
- $RA = 1.0$ ohm-meter
- $CM = 0.01$ Farads per square metre.

Here are the predefined spine prototypes:

- *makePassiveSpine()*: This just makes a passive spine with the default parameters
- *makeExcSpine()*: This makes a spine with NMDA and glu receptors, and also a calcium pool. The NMDA channel feeds the Ca pool.
- *makeActiveSpine()*: This adds a Ca channel to the exc_spine. and also a calcium pool.

The spine distributions are specified in a familiar way for the first few arguments, and then there are multiple (optional) spine-specific parameters:

[*spine*name, *region_in_cell*, *spacing*, *spacing_distrib*, *size*, *size_distrib*, *angle*, *angle_distrib*]

Only the first two arguments are mandatory.

- *spine*name: The prototype name
- *region_in_cell*: Usual wildcard specification of names of compartments in which to put the spines.
- *spacing*: Math expression to define spacing between spines. In the current implementation this evaluates to $1/\text{probability_of_spine_per_unit_length}$. Defaults to 10 microns. Thus, there is a 10% probability of a spine insertion in every micron. This evaluation method has the drawback that it is possible to space spines rather too close to each other. If spacing is zero or less, no spines are inserted.
- *spacing_distrib*: Math expression for distribution of spacing. In the current implementation, this specifies the interval at which the system samples from the spacing probability above. Defaults to 1 micron.
- *size*: Linear scale factor for size of spine. All dimensions are scaled by this factor. The default spine head here is 0.5 microns in diameter and length. If the scale factor were to be 2, the volume would be 8 times as large. Defaults to 1.0.
- *size_distrib*: Range for size of spine. A random number R is computed in the range 0 to 1, and the final size used is $\text{size} + (R - 0.5) * \text{size_distrib}$. Defaults to 0.5
- *angle*: This specifies the initial angle at which the spine sticks out of the dendrite. If all angles were zero, they would all point away from the soma. Defaults to 0 radians.
- *angle_distrib*: Specifies a random number to add to the initial angle. Defaults to 2 PI radians, so the spines come out in any direction.

Suggestions:

- Try different spine settings. Warning: if you put in too many spines it will take much longer to load and run!
- Try different spine geometry layouts.
- See if you can deliver the current injection to the spine. Hint: the name of the spine compartments is 'head#' where # is the index of the spine.

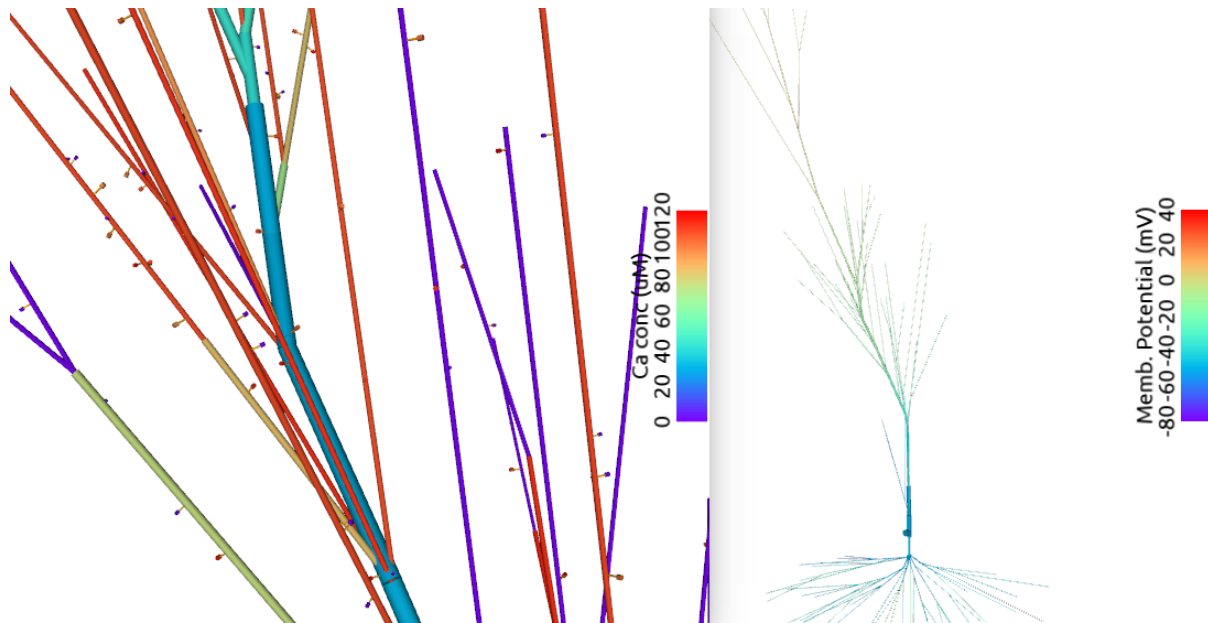


Fig. 39: 3-D display for spiny active neuron

Place spines in a spiral along a dendrite

ex9.3_spiral_spines.py

Just for fun. Illustrates how to place spines in a spiral around the dendrite. For good measure the spines get bigger the further they are from the soma.

Note that the uniform spacing of spines is signified by the negative minSpacing term, the fourth argument to spineDistrib.

```
import moose
import pylab
import rdesigneur as rd
rdes = rd.rdesigneur(
    cellProto = [['ballAndStick', 'elec', 10e-6, 10e-6, 2e-6, 300e-6, 50]],
    spineProto = [['makePassiveSpine()', 'spine']],
    spineDistrib = [['spine', '#dend#', '3e-6', '-1e-6', '1+p*2e4', '0', 'p*6.28e7', '0']],
    stimList = [['soma', '1', '.', 'inject', '(t>0.02) * 1e-9']],
    moogList = [['#', '1', '.', 'Vm', 'Soma potential']]
)
rdes.buildModel()
moose.reinit()
rdes.displayMoogli( 0.0002, 0.025, 0.02 )
```

Note that the uniform spacing of spines is signified by the negative minSpacing term, the fourth argument to spineDistrib.

```
spineDistrib = [['spine', '#dend#', '3e-6', '-1e-6', '1+p*2e4', '0', 'p*6.28e7', '0']]
```

Suggestions:

- Play with expressions for spine size and angular placement.
- See what happens if the segment size gets smaller than the spine spacing.

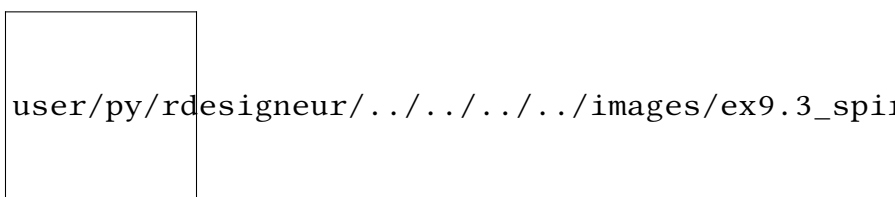


Fig. 40: 3-D display of spines in a spiral

5.1.4 Rdesigneur command reference

Rdesigneur is a Python class used to build multiscale neuronal models involving Reaction-Diffusion and Electrical SIGnaling in NEURons. The stages in its use are illustrated in the following dummy code snippet:

```
# 1. Load in the libraries
import moose
import rdesigneur as rd

# 2. Define the arguments. This does most of the model setup
rdes = rd.rdesigneur( args )

# 3. Tweak parameters of model building-blocks, for example:
a = moose.element( '/library/chem/kinetics/a' )
a.diffConst = 0

# 4. Build the model
rdes.buildModel()

# 5. Tweak values in the constructed model, for example
bv = moose.vec( '/model/chem/dend/b' )
bv[0].concInit *= 2

# 6. Run the model
moose.reinit()
moose.start( runtime )

# 7. Display and/or save model output
rdes.dispay()
```

The rdesigneur arguments are provided in the standard Python keyword-argument format. For example:

```
rdes = rd.rdesigneur(
    turnOffElec = True,
    chemDt = 0.05,
    ...
    chemProto = [ ['makeChemOscillator()', 'osc'] ],
    ...
    plotList = [ rd.rplot( relpath = 'dend/a', field = 'conc', title_
    => '[a] (uM)' ) ],
    ...
)
```

Each argument has a default, hence even *building rdesigneur without arguments* (page 64) will produce a correct, if not very interesting model.

Rdesigneur and Prototypes: Rdesigneur assembles models by taking prototype objects and replicating them into the model. These prototypes can be chemical reaction systems, ion channels, spines, or entire neurons. All the prototypes are placed under the MOOSE object */library*. When building the model, it looks up prototypes by name and places them into the resulting model. The rdesigneur constructor (step 2 above) builds all these prototypes. Once they are in place, the *BuildModel()* method (step 4 above) performs the assembly.

Below we provide the usage of the argument list to rdesigneur, which does most of the model specification.

turnOffElec

Type: bool

Default: False

Use: Turns off electrical calculations. It is a good idea to set this flag **True** if the model doesn't use electrical calculations, it can make the calculations many times faster.

useGssa

Type: bool

Default: True

Use: Turns on the use of the Gillespie Stochastic Simulation Algorithm (GSSA) in dendritic spines. Advisable in models where you worry about stochasticity. Also it typically makes the simulations run faster.

combineSegments

Type: bool

Default: True

Use: Flag to pass on to the NeuroML loader to tell it to combine segments.

stealCellFromLibrary

Type: bool

Default: False

Use: Use the prototype loaded-in neuron itself for the main simulation run, removing it from the available prototypes. It is advisable to set this to *True* if the model is large and complicated. It saves memory and in some cases runs more reliably.

verbose

Type: bool

Default: True

Use: Tell rdesigneur to be garrulous when loading and reporting status and errors.

addSomaChemCompt

Type: bool

Default: False

Use: Specify that the largest chemical compartment (by volume) should be assigned to the cell soma. Most multiscale models don't bother with a soma chemical compartment, and are happy with dendrite and possibly spines, so this defaults to False.

addEndoChemCompt

Type: bool

Default: False

Use: Specify that each of the chemical compartments should contain an internal *endo*-compartment. This is typically used for the endoplasmic reticulum in *models of calcium-induced calcium release* (page 92) (CICR), however, the EndoCompartments are quite general and can be used for defining chemistry and transport involving any membrane-bound organelle. In MOOSE, when you create an EndoCompartment it must be surrounded by a regular compartment, and a voxel of the EndoCompartment appears within every voxel of the surrounding compartment.

diffusionLength

Type: double

Default: 2e-6 (2 microns)

Use: This sets the spatial discretization length of reaction-diffusion models. If the diffusion constant is D (in $\mu\text{m}^2/\text{sec}$), then the *diffusionLength* should be less than D microns for signaling events that take 1 second. If the signaling is faster, *diffusionLength* should be smaller.

temperature

Type: double

Default: 32 degrees Celsius

Use: ChannelML definitions of ion channels use this value to modulate their kinetics.

chemDt

Type: double

Default: 0.1 s

Use: Specify timestep for chemical computations. Note that internally the MOOSE solver will probably use finer or adaptive timesteps. The *chemDt* just ensures that all the chemical values in different solvers will be synchronized at this interval. You will want to make this somewhat smaller (0.01 to 0.001 s) in the case of multiscale simulations with tight coupling between electrical and signaling events.

diffDt

Type: double

Default: 0.01 s

Use: Specify timestep for diffusion computations, as well as cross-compartment reactions and molecular transport across membrane pores. This timestep does not apply to voltage-gated and synaptic channels handled by the electrical solver, for that use *elecDt*. You will want to make this somewhat smaller (0.01 to 0.001 s) in the case of multiscale simulations with tight coupling between electrical and signaling events.

elecDt

Type: double

Default: 50e-6 s

Use: Specify timestep for electrical calculations, used by the HSolver in MOOSE to carry out calculations using Gaussian Elimination and the Crank- Nicolson method for ion channels. This works well for slower channels, but if you have particularly fast channel kinetics you may wish to use *elecDt* of 10 to 20 us.

chemPlotDt

Type: double

Default: 1 s

Use: Timestep for storing and plotting chemical values.

elecPlotDt

Type: double

Default: 100e-6 s

Use: Timestep for storing and plotting electrical values.

funcDt

Type: double

Default: 100e-6 s

Use: Timestep for performing Function calculations for inputs and stimuli, for electrical models. Only used for electrical models, i.e., when *turnOffElec* (page 136) is False. Otherwise the system uses a *funcDt* equal to the *chemDt*.

cellProto

Type: List of lists

Default: [] (empty list). This generates the Hodgkin-Huxley configuration where length and diameter are 500 microns, $RM = 0.333$, $RA = 3000$, and $CM = 0.01 \text{ F/m}^2$, but no active channels.

Use: This defines which neuronal model specification to use. There are many options here:

1. zero args: make standard soma corresponding to the Hodgkin-Huxley model. length and diameter are both 500 μm .
2. [name, library_proto_name]: uses library prototype object.
3. [fname.suffix, cellname]: Loads cell from file. The file type is identified by the suffix, and can be :
 - .nml: NeuroML
 - .xml: NeuroML
 - .swc: NeuroMorpho.org format for cellular morphology
 - .p: Genesis format
4. [moose<Classname>, cellname]: Makes prototype from MOOSE class.
5. [funcname, cellname]: Calls named function with specified name of cell to be made.
6. [path, cellname]: Copies path to library as proto
7. [libraryName, cellname]: Renames library entry as prototype.
8. [somaProto, name, somaDia=5e-4, somaLen=5e-4] Creates a soma with optional specified diameter and length. Defaults as shown.
9. [ballAndStick,name, somaDia=10e-6, somaLen=10e-6, dendDia=4e-6, dendLen=200e-6, numDendSeg=1] Creates a ball-and-stick with required type and name arguments. The remaining arguments are optional. Defaults as shown.

spineProto

Type: List of lists

Default: [] (empty list). This does not define any spines.

Use: Each list entry should be a list containing two strings: *source* and *destination*. The *source* defines how to build the prototype. The *destination* specifies its name. If the requested *destination* is an object that already exists in the library, the system doesn't do anything.

The *source* can be any of:

- functionName(): Call specified Python function, with the *destination* as the argument. The function is expected to build a prototype of the requested name on '/library'. The following utility functions are built-in:
 - makePassiveSpine(): Makes a 2-compartment spine with the following parameters:
 - * shaft name: shaft
 - * shaft length = 1 micron

- * shaft diameter = 0.2 micron
- * head name: head
- * head length = 0.5 micron
- * head diameter = 0.5 micron
- * RM = 1.0
- * RA = 1.0
- * CM = 0.01
- makeExcSpine(): Same as above but adds in glutamate and NMDA receptors and a calcium pool. The calcium pool has a pumping tau of 13.333 ms, and is present in the volume of the spine head. Both receptors have conductances in the form of dual-exponential alpha functions, with a separate opening and closing tau. The glutamate receptor has the following parameters:
 - * name: glu
 - * opening tau: 2 ms
 - * closing tau: 9 ms
 - * Gbar, ie, conductance per unit area: 200 Siemens/m²

The NMDA receptor has the following parameters:

- * name: NMDA
- * opening tau: 20 ms
- * closing tau: 20 ms
- * Gbar, ie, conductance per unit area: 80 Siemens/m²
- makeActiveSpine(): Same as above, but also adds in a voltage-gated calcium channel with $Gbar = 10 \text{ Siemens/m}^2$ into the spine head.
- Path of existing object in memory, such as */library/source*. In this case rdesigner renames the object to */library/destination*.
- A filename, with any of the suffices:
 - *.nml*: NeuroML
 - *.xml*: NeuroML
 - *.swc*: NeuroMorpho.org format for cellular morphology
 - *.p*: Genesis format
- moose::SymCompartment: Make a SymCompartment for the spine. Deprecated.
- moose::Compartment: Make a Compartment for the spine. Deprecated.

chanProto

Type: List of lists

Default: [] (empty list). The empty list does not define any channels.

Use: Each list entry must have a string for the *source*. It can optionally have a second string for the *destination*, which is the name to give to the *source* channel when it is constructed on */library*.

The following options are available for specifying the *source* for making channel prototypes:

- Filepath. This is relative to the working directory. The following file types are known:
 - xml: ChannelML, which is a subset of NeuroML
 - nml: ChannelML, which is a subset of NeuroML

Channels in these formats are available from [Open Source Brain](http://www.opensourcebrain.org/) (<http://www.opensourcebrain.org/>),

- Predefined channel prototypes, available as functions within rdesigneur. This is indicated by the use of braces after the name. The following prototypes are currently available:
 - make_HH_Na(): Make the classical Hodgkin-Huxley Na channel, with kinetics scaled to SI units.
 - make_HH_K(): Classical HH delayed rectifier K channel.
 - make_Na(): Hippocampal pyramidal Na channel from Traub 1991.
 - make_K_DR(): Hippocampal pyramidal K delayed rectifier channel from Traub 1991.
 - make_K_A(): Hippocampal pyramidal A-type K channel from Traub 1991.
 - make_Ca_conc(): A calcium pool with tau 13.333 ms. This is required for the calcium dynamics of several channels.
 - make_Ca(): Voltage-gated Calcium channel, based on Traub 1991. It requires the Ca_conc.
 - make_K_AHP: Voltage and calcium-gated afterhyperpolarization-activated K channel, from Traub 1991. Note that this channel requires the presence of the Ca_conc.
 - make_K_C: Voltage and calcium-dependent K channel from Traub 1991. This channel requires the presence of the Ca_conc.
 - make_glu(): Glutamate receptor in the form of dual-exponential alpha functions, with a separate opening (2ms) and closing (9ms) tau. Reversal potential = 0 mV.
 - make_GABA(): GABA receptor in the form of dual-exponential alpha functions, with a separate opening (4ms) and closing (9ms) tau. Reversal potential = -65 mV.
- User-defined channel definition functions. These can be from external Python files, using the full path to the file name minus the suffix. The specific function within it is then specified. For example,

```
chanProto = [
    [ '/home/user/models/channelProtos.make_K_AHP()', 'K_
    ↪AHP' ]
]
```

chemProto

Type: List of lists

Default: [] (empty list). The empty list does not define any chemical systems.

Use: Each list entry must have a string for the *source*. It can optionally have a second string for the *destination*, which is the name to give to the *source* chemical system when it is constructed on *library*.

The following options are available for specifying the *source* for making channel prototypes:

- Filepath. This is relative to the working directory. The following file types are known:
 - xml: SBML
 - sbml: SBML
 - .g: GENESIS Kinetikit (kkit.g) format.

Channels in these formats are available from the [DOQCS](https://doqcs.ncbs.res.in/) (<https://doqcs.ncbs.res.in/>) database, and from the [BioModels](https://www.ebi.ac.uk/biomodels-main/) (<https://www.ebi.ac.uk/biomodels-main/>) database,

- Predefined functions. At present only one such function is available, **makeChemOscillator()**
- **User-defined functions.** These can be from external Python files, using the full path to the file name minus the suffix. The specific function within it is then specified. For example,

```
chemProto = [
    [ '/home/user/models/chemProtos.make_Osc()', 'osc' ]
]
```

- Pool objects. These are created on the fly using the form

```
chemProto = [['moose:Pool', 'a']]
```

passiveDistrib

Type: List of lists

Default: [] (empty list). Does nothing.

Use; This is for adjusting the passive properties of the neuron. Each list entry is a list of strings, of the form:

```
[path, field, expr, [field, expr]...]
```

Here the *path* is a MOOSE wildcard path, which defines one or more objects. Briefly, the '#' character specifies any string, and the double '##' specifies any string at any level in the tree.

For example, to specify any compartment with the string 'dend' you would use '#dend#' and to specify any object anywhere in the tree you would use '##'.

The *field* can be any one of the following:

- RM: Membrane resistivity, in ohms.m²
- RA: Axial resistivity, in ohms.m
- CM: Membrane specific capacitance, in Farads/m²
- Rm: Absolute membrane resistance of that segment, in ohms.
- Ra: Absolute axial resistance of that segment, in ohms.
- Cm: Absolute membrane capacitance of that segment, in Farads.
- Em: Membrane resting potential, in Volts.
- initVm: Initial value to set the membrane potential, in Volts.

The *expr* is an expression string that is evaluated to give the desired value for the field. This can be as simple as the value itself, but can be a much more interesting function of geometrical properties of the cell. The geometry arguments available to the *expr* include:

- p: Path length in metres from the soma, measured along the dendrite.
- g: Geometrical distance from the soma.
- L: Number of electronic length constants from the soma
- len: length of the segment of dendrite
- dia: diameter of the segment of dendrite
- maxP: Maximum path length of any dendrite in the cell.
- maxG: Maximum geometrical distance of any dendrite from soma
- maxL: Maximum electrotonic distance of any dendrite from the soma

Putting these together, here is an example of using the passiveDistrib:

```
passiveDistrib = [
  [ 'soma', 'RM', '1.0', 'CM', '0.02' ],
  [ '#dend#', 'RM', '1.5 + 0.5*(p>200e-6)', 'CM', '0.01' ],
]
```

This means set the soma *RM* to 1.0, and *CM* to 0.02, leaving the *RA* as the default. The scaled value for *Rm*, *Ra*, and *Cm* are computed by scaling these terms according to the soma dimensions. For all dendrite compartments, set the *RM* to 1.5 provided it is closer than 200 microns dendritic path length from the soma, and set the *RM* to 2.0 for all dendritic compartments further than this. Finally, for all dendrite compartments, set *CM* to 0.01. Note that again the absolute *Rm* and *Cm* will be scaled according to the local compartment dimensions.

spineDistrib

Type: List of lists

Default: [] (empty list). Does nothing.

Use: This is for inserting dendritic spines onto the neuron. Each entry is a list of strings, of the form:

```
[proto, path, [spacing, minSpacing, size, sizeDistrib, angle, ↵
↵angleDistrib]]
```

Of these, the *name* and the *path* are required entries, and the remainder can be provided in pairs. The defaults for these entries are:

```
['spine', '#dend#,#apical#', '10e-6', '1e-6', '1', '0.5', '0', '6.2832
↵']
```

The interpretation of the arguments is as follows:

- name: This is the name of the *spine prototype* (page 139).
- path: The wildcard path of compartments on which to insert the spines. In the example above, *'#dend#,#apical#'* means all compartments with the strings *dend* or *apical* in their names.
- spacing: The mean spacing between spines. At present the spines are placed with a Poisson distribution. This is a math expression with the same terms as used for the passive distribution, so that the spine spacing can be a function of spine position along the dendritic tree. The form of this expression is shown again below.
- minSpacing: The minimum spacing, and the increment along which the Poisson samples are taken to decide if a spine should be added. In case *minSpacing* is negative, the system places spines with uniform *spacing* along the dendritic segment. If $\text{segment length} < 0.5 * \text{spacing}$ then the system falls back onto Poisson samples so that finely subdivided dendrites don't miss out on spines altogether.
- size: Scale factor for size from the prototype spine. All dimension of the spine are scaled by this number: shaft length, shaft diameter, head length and head diameter. This is a math expression, as shown below.
- sizeDistrib: The range of distribution of sizes. This is a linear distribution centered around the defined size.
- angle: The initial angle of the first spine on each dendrite compartment, in radians. This is a math expression, as shown below.
- angleDistrib: The range of of angles around this initial angle. The angle will be chosen from a linear distribution centered around the centre angle, $\pm \text{angleDistrib}$.

The expression used for spacing, size, and angle is of the form of an expression string that is evaluated to give the desired value for the field. This can be as simple as the value itself, but can be a much more interesting function of geometrical properties of the cell. The geometry arguments available to the *expr* include:

- p: Path length in metres from the soma, measured along the dendrite.
- g: Geometrical distance from the soma.
- L: Number of electronic length constants from the soma
- len: length of the segment of dendrite
- dia: diameter of the segment of dendrite
- maxP: Maximum path length of any dendrite in the cell.

- maxG: Maximum geometrical distance of any dendrite from soma
- maxL: Maximum electrotonic distance of any dendrite from the soma

For example:

```
[ 'spine', '#dend#', '1e-6 + (dia<2e-6)*10', '1e-7', '1', '0.5', '6.
↪28*p/maxP', '0' ]
```

proto: The prototype spine by the name of *spine* is used.

path: All compartments with the string *dend* in their name are used.

Spacing: The spines are only placed on branches smaller than 2 microns (otherwise the spine spacing is 10 metres). On these small branches the spacing is, on average, 1 micron.

Size: The size is anything from 50% to 150% of the prototype spine size.

Angle: The angle is proportional to the distance from the soma, such that the spines make a complete spiral (2π) around the dendrite over its length.

chanDistrib

Type: List of lists

Default: [] (empty list). Does nothing.

Use: This is for inserting ion channels onto the neuron. Each entry is a list of strings, of the form:

```
[proto, path, field, expr, [field, expr]...]
```

The entries here are of the form:

- proto: Specifies the name of the prototype channel to insert
- path: Wildcard path of compartments in which to insert the channel
- field: Field to assign to channel, almost always **Gbar**, to set its channel density.
- expr: Expression evaluated to obtain value to assign to field. This is a mathematical expression of various geometrical properties of the cell, as listed below.

The *expr* can be as simple as the value itself, but can be a much more interesting function of geometrical properties of the cell. The geometry arguments available to the *expr* include:

- p: Path length in metres from the soma, measured along the dendrite.
- g: Geometrical distance from the soma.
- L: Number of electronic length constants from the soma
- len: length of the segment of dendrite
- dia: diameter of the segment of dendrite
- maxP: Maximum path length of any dendrite in the cell.
- maxG: Maximum geometrical distance of any dendrite from soma
- maxL: Maximum electrotonic distance of any dendrite from the soma

A typical channel distribution entry is:

```
[ "kdr", "#", "Gbar", "p < 50e-6 ? 500 : 100" ]
```

Here the *kdr* channel is inserted throughout the cell, and its conductance is at 500 Siemens/m² for all regions closer than 50 microns, and 100 S/m² for the rest of the cell. Basically there is lots of the channel on and near the soma.

chemDistrib

Type: List of lists

Default: [] (empty list). Does nothing.

Use: This is for inserting a chemical system into the neuron. Each entry is a list of strings, of the form:

```
[proto, path, 'install', expr]
```

The entries here are of the form:

- **proto**: Specifies the name of the prototype chemical system to insert
- **path**: Wildcard path of compartments in which to insert the channel
- **'install'**: Default string.
- **expr**: Expression evaluated to decide whether to install the chemical system. This is the *usual function* (page 145) of geometrical properties of the cell. It is usually '1', to tell the system to install throughout the *path*.

The chemical distribution is handled specially for assignment to the neuronal morphology. This is because a given chemical system will have reactions between dendrite, ER, spines and PSD, as well as diffusion between these zones. Thus, though it would be convenient, we cannot simply define separate chemical systems for each cellular compartment. Instead we use one of two conventions for doing the assignment.

1. Volume based. If the model format does not permit explicit naming of the chemical compartments in the model, then the assignment is inferred from the volume of each compartment. This limitation applies for the legacy Genesis/kkit .g format. It may also apply to SBML models that do not assign suitable names for their chemical compartments. In this case the largest chemical compartment is assigned to the dendrite, the next (if present) to the spine head, and the smallest (if present) to the spine PSD.

This is modified in one of two ways by the flags *addSomaChemCompt* (page 137) and *addEndoChemCompt* (page 137).

addSomaChemCompt instructs rdesigneur to use the largest compartment for the soma. The remaining compartments follow in the usual order.

addEndoChemCompt instructs rdesigneur to insert an EndoCompartment in each neuronal compartment. The volume order is now dend, dend_endo, spine-head, spine-head-endo and so on.

2. Name based. This works for recent SBML models, which can assign a compartment name to each of the chemical compartments. Here the expectation is that the names are

one of *soma*, *soma_endo*, *dend*, *dend_endo*, *spine*, *spine_endo*, *psd*, *psd_endo*. Note that the last one, though permitted, doesn't make much biological sense.

adaptorList

Type: List of lists

Default: [] (empty list). Does nothing.

Use: This is for implementing an adaptor between chemical and electrical, or chemical and structural quantities. Adaptors handle the conversion between distinct concepts in chemical and electrical models. For example, Calcium concentration as computed electrically in the *Ca_conc* objects, can map to the calcium concentration of the ion as a molecule, where it can react, diffuse, and undergo other calcium dynamics. Another common use is to map the concentration of the molecular state of an ion channel, to its conductance. The adaptor applies the conversion equation $y = mx + c$ where y is the target value, x is the source value, m is the slope of the conversion, and c is the offset.

Adaptors automatically average over multiple inputs if the mapping requires. Typically electrical segments each contain many chemical voxels, so the adaptor averages all the source chemical quantities to apply to the corresponding electrical quantity. Similarly, each chemical timestep is typically much longer than the electrical timestep, so the adaptor averages the electrical quantity over the entire duration of the chemical timestep.

Each entry is a list of strings, of the form:

```
[source, source_field, dest, dest_field, offset, scaling]
```

The entries here are:

- **source:** Specifies the path of the objects whose quantities need to be converted. In the case of chemical quantities, the path starts with the compartment name, one of *dend*, *spine*, or *psd*. So the molecule Ca in the dendrite would be identified as *dend/Ca*.
- **source_field:** The field on the source object whose value is to be used.
- **dest:** Path of destination object, whose quantities will be assigned. As above, chemical quantities are prefixed by their compartment name.
- **dest_field:** Field to be assigned on the destination object.
- **offset:** Double. In the conversion, what is the value of the *dest_field* when the source value is zero?. In other words, the quantity c in the conversion equation $y = mx + c$
- **scaling:** Double. The slope m .

stimList

Type: List of lists

Default: [] (empty list). Does nothing.

Use: Each entry is a list of strings, as follows:

```
[path, geometry_expr, dest, dest_field, time_expr]
```

The entries here are:

- `path`: The usual MOOSE wildcard path to identify electrical compartments over which the stimulus will extend. Note that the stimulus may be to a chemical entity, but the spatial location is specified in terms of the electrical compartments in which the chemical system is embedded.
- `geometry_expr`. This is the *usual function* (page 145) of geometrical properties of the cell. If it is non-zero, then the stimulus will apply. There is a special case for synaptic inputs in which the `geometry_expr` is replaced with the synaptic weight, recorded as a string.
- `dest`. This is the destination object for the stimulus.
- `dest_field`. This is the field on the destination object to be assigned. There is a special case for synaptic inputs, where the field can be **periodicsyn** or **randsyn**, representing periodic and random synaptic input respectively.
- `time_expr`: This is the time expression of the value of the stimulus. Unlike the `geometry_expr`, the `time_expr` can take the predefined variable `t` which is the current simulation time. The `time_expr` does not have access to the geometry arguments.

Example 1:

```
[ 'head#', '0.5', 'glu', 'periodicsyn', '1 + 40*(t>10 && t<11)']
```

This acts on all glutamate receptors on the spine *heads*. It delivers periodic synaptic input with weight 0.5 at a basal rate of 1 Hz, rising by 40Hz in the interval between 10 and 11 seconds.

Example 2:

```
[ 'soma', '1', '.', 'inject', '(1+cos(t/10))*(t>31.4 && t<94)* 0.5e-9' ]
```

This acts to deliver a current injection on the soma. It delivers cosine input of angular frequency 1/10 radians/s, between times 31.4 and 94 seconds, with a peak amplitude of 0.5 nA.

Rdesigner also supports keyword-based argument lists for the `stimList`. Here each entry is an `rstim` function as follows:

```
rd.rstim( elecpath, geom_expr, relpath, field, expr )
```

The default values of the arguments are

```
rd.rstim(elecpath='soma', geom_expr='1', relpath='.', field='inject',  
        expr='0')
```

Example 3: To get the same outcome as example 2, one could use:

```
rd.rstim( expr=(1+cos(t/10))*(t>31.4 && t<94)* 0.5e-9 )
```

because most of the arguments are the same as the defaults.

plotList

Type: List of lists

Default: [] (empty list). Does nothing.

Use: This displays a line plot of cellular activity. Each entry is a list as follows:

```
[path, geom_expr, relpath, field, title,
  [mode, ymin, ymax, saveFile, saveResolution, showFlag ]
]
```

The entries here are:

- path: string. The usual MOOSE wildcard path to identify electrical compartments over which the plots will be sampled. Note that the stimulus may be to a chemical entity, but the spatial location is specified in terms of the electrical compartments in which the chemical system is embedded.
- geom_expr: string. This is the *usual function* (page 145) of geometrical properties of the cell. If it is non-zero, then the stimulus will apply. There is a special case for synaptic inputs in which the *geometry_expr* is repaced with the synaptic weight, recorded as a string.
- relpath: string. Relative path to object whose value is being monitored.
- field: string. The field to monitor on the source object.
- title: Title string for the generated plot.
- mode: Optional. String to decide what kind of plot to make. Options are:
 - ‘time’: Default. Plot time-series
 - ‘wave’: Generate wave-plot with compartment/voxel number as x axis, value as y axis, and run through a series of frames for different time-points during simulation.
- ymin: Double. Optional. Minimum value for y axis. Default = 0.
- ymax: Double. Optional. Maximum value for y axis. Default = 0. If ymin==ymax then the plot autoscales.
- saveFile: string. Optional. File in which to save plot contents. Default = “”, to indicate that the file is not saved. Currently it can save in *csv* and *xml* formats. *nsdf* will be implemented soon.
- show: Bool. Optional. Flag to decide if the plot should be displayed. Default=True.

Rdesigneur also supports keyword-based argument lists for the plotList, having the same entries as above. Here are two plotList entries with identical outcomes.

```
['soma', '1', '.', 'Vm', 'Soma membrane potential'],
[rd.rplot( field='Vm', title= 'Soma membrane potential')],
```

moogList

Type: List of lists

Default: [] (empty list). Does nothing.

Use: This displays a 3-D plot of cellular activity. Each entry is a list as follows:

```
[path, geom_expr, relpath, field, title, [ymin, ymax]]
```

The entries here are:

- path: string. The usual MOOSE wildcard path to identify electrical compartments over which the display will be sampled. Note that the stimulus may be to a chemical entity, but the spatial location is specified in terms of the electrical compartments in which the chemical system is embedded.
- geom_expr: string. This is the *usual function* (page 145) of geometrical properties of the cell. If it is non-zero, then the stimulus will apply. There is a special case for synaptic inputs in which the *geometry_expr* is replaced with the synaptic weight, recorded as a string.
- replath: string. Relative path to object whose value is being monitored.
- field: string. The field to monitor on the source object.
- title: Title string for the generated display.
- ymin: Double. Minimum value for y axis. Default = 0.
- ymax: Double. Maximum value for y axis. Default = 0. If ymin==ymax then the plot autoscales.
- show: Bool. Flag to decide if it should be displayed. Default=True.

Rdesigner also supports keyword-based argument lists for the moogList, having the same entries as above. Here are two moogList entries with identical outcomes.

```
[ 'soma', '1', 'dend/a', 'conc', 'a Conc', 0, 600 ],  
[rd.rmog(repath='dend/a', field='conc', title = 'a Conc', ymax=600)]
```

To run and display moogli, one replaces the *moose.start()* and the *rdes.display()* functions with the line:

```
rdes.displayMoogli(dt, runtime, rotation, fullscreen, block, azim,   
↪elev)
```

in which the first two arguments are required and the rest are optional and can be assigned by keywords.

The arguments are as follows:

- dt: double. Time interval between frames on the moogli display
- runtime: double. Simulation runtime.
- rotation: double. How much to rotate the display per frame. Defaults to pi/500.
- fullscreen: bool. Flag to do display on the full screen. Defaults to False.
- azim: double. Azimuth setting. Defaults to 0.0
- elev: double. Elevation setting. Defaults to 0.0

The *moogli primer* (page 90) explains how to use the 3-D display.

5.2 Rdesigneur Examples

Each of the following examples can be run by clicking on the green [source](#) button on the right side of each example, and running from within a [.py](#) python file on a computer where moose is installed.

Alternatively, all the files mentioned on this page can be found in [the main](#) moose directory. They can be found under

```
(...)/moose/moose-examples/snippets
```

They can be run by typing

```
$ python filename.py
```

in your command line, where filename.py is the python file you want [to run](#).

All of the following examples show one or more methods within each [python](#) file.

For example, in the [cubeMeshSigNeur](#) section, there are two blue tabs describing the [cubeMeshSigNeur.createSquid\(\)](#) and [cubeMeshSigNeur.main\(\)](#) methods.

The filename is the bit that comes before the [.](#) in the blue boxes, [with](#) [.py](#) added at the end of it. In this case, the file name would be [cubeMeshSigNeur.py](#).

5.2.1 Building Chemical-Electrical Signalling Models

Building a compartment

Inserting Spines and viewing

Proceeding with Spines

To see interactive and executable versions of ChemicalBistables and ChemicalOscillators, please click the following link: (<https://mybinder.org/v2/gh/BhallaLab/moose-binder/master?filepath=home%2Fmooser%2FIndex.ipynb>) These tutorials use the moose simulation environment to illustrate various scientific concepts, phenomena, and research. As such the materials in this section are useful for both teaching and learning about these topics.

6.1 Chemical Bistables

A **bistable system** (<https://en.wikipedia.org/wiki/Bistability>) is a dynamic system that has two stable equilibrium states. The following examples can be used to teach and demonstrate different aspects of bistable systems or to learn how to model them using moose. Each example contains a short description, the model's code, and the output with default settings.

Each example can be found as a python file within the main moose folder under

```
(...)/moose/moose-examples/tutorials/ChemicalBistables
```

In order to run the example, run the script

```
python filename.py
```

in command line, where `filename.py` is the name of the python file you would like to run. The filenames of each example are written in **bold** at the beginning of their respective sections, and the files themselves can be found in the aforementioned directory.

In chemical bistable models that use solvers, there are optional arguments that allow you to specify which solver you would like to use.

```
python filename.py [gsl | gssa | ee]
```

Where:

- **gsl**: 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.
- **gssl**: Optimized Gillespie stochastic systems algorithm, custom implementation. This uses variable timesteps internally. 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 method computes the solution of partial and ordinary differential equations.

All the following examples can be run with either of the three solvers, each of which has different advantages and disadvantages and each of which might produce a slightly different outcome.

Simply running the file without the optional argument will by default use the **gsl** solver. These **gsl** outputs are the ones shown below.

6.1.1 Simple Bistables

Filename: **simpleBis.py**

This example shows the key property of a chemical bistable system: it has two stable states. Here we start out with the system settling rather quickly to the first stable state, where molecule A is high (blue) and the complementary molecule B (green) is low. At $t = 100\text{s}$, we deliver a perturbation, which is to move 90% of the A molecules into B. This triggers a state flip, which settles into a distinct stable state where there is more of B than of A. At $t = 200\text{s}$ we reverse the flip by moving 99% of B molecules back to A.

If we run the simulation with the **gssa** option `python simpleBis.py gssa`

we see exactly the same sequence of events, except now the switch is noisy. The calculations are now run with the Gillespie Stochastic Systems Algorithm (**gssa**) which incorporates probabilistic reaction events. The switch still switches but one can see that it might flip spontaneously once in a while.

Things to do:

1. Open a copy of the script file in an editor, and around line 124 and 129 you will see how the state flip is implemented while maintaining mass conservation. What happens if you flip over fewer molecules? What is the threshold for a successful flip? Why are these thresholds different for the different states?
2. Try different volumes in line 31, and rerun using the **gssa**. Will you see more or less noise if you increase the volume to $1\text{e-}20\text{ m}^3$?

Code:

```

1 #####
2 ↪###
3 ## This program is part of 'MOOSE', the
4 ## Messaging Object Oriented Simulation Environment.
5 ## Copyright (C) 2013 Upinder S. Bhalla. and NCBS
6 ## It is made available under the terms of the
7 ## GNU Lesser General Public License version 2.1
8 ## See the file COPYING.LIB for the full notice.
9 #####
10 ↪###
11
12 # This example illustrates how to set up a kinetic solver and kinetic_
13 ↪model
14 # using the scripting interface. Normally this would be done using the
15 # Shell::doLoadModel command, and normally would be coordinated by the
16 # SimManager as the base of the entire model.
17 # This example creates a bistable model having two enzymes and a_
18 ↪reaction.
19 # One of the enzymes is autocatalytic.
20 # The model is set up to run using deterministic integration.
21 # If you pass in the argument 'gssa' it will run with the stochastic
22 # solver instead
23 # You may also find it interesting to change the volume.
24
25 import math
26 import pylab
27 import numpy
28 import moose
29 import sys
30
31 def makeModel():
32     # create container for model
33     model = moose.Neutral( 'model' )
34     compartment = moose.CubeMesh( '/model/compartment' )
35     compartment.volume = 1e-21 # m^3
36     # the mesh is created automatically by the compartment
37     mesh = moose.element( '/model/compartment/mesh' )
38
39     # create molecules and reactions
40     a = moose.Pool( '/model/compartment/a' )
41     b = moose.Pool( '/model/compartment/b' )
42     c = moose.Pool( '/model/compartment/c' )
43     enz1 = moose.Enz( '/model/compartment/b/enz1' )
44     enz2 = moose.Enz( '/model/compartment/c/enz2' )
45     cplx1 = moose.Pool( '/model/compartment/b/enz1/cplx' )
46     cplx2 = moose.Pool( '/model/compartment/c/enz2/cplx' )
47     reac = moose.Reac( '/model/compartment/reac' )
48
49     # connect them up for reactions
50     moose.connect( enz1, 'sub', a, 'reac' )
51     moose.connect( enz1, 'prd', b, 'reac' )
52     moose.connect( enz1, 'enz', b, 'reac' )
53     moose.connect( enz1, 'cplx', cplx1, 'reac' )
54
55     moose.connect( enz2, 'sub', b, 'reac' )
56     moose.connect( enz2, 'prd', a, 'reac' )

```

(continues on next page)

(continued from previous page)

```

53         moose.connect( enz2, 'enz', c, 'reac' )
54         moose.connect( enz2, 'cplx', cplx2, 'reac' )
55
56         moose.connect( reac, 'sub', a, 'reac' )
57         moose.connect( reac, 'prd', b, 'reac' )
58
59         # connect them up to the compartment for volumes
60         #for x in ( a, b, c, cplx1, cplx2 ):
61         #                               moose.connect( x, 'mesh',
↪mesh, 'mesh' )
62
63         # Assign parameters
64         a.concInit = 1
65         b.concInit = 0
66         c.concInit = 0.01
67         enz1.kcat = 0.4
68         enz1.Km = 4
69         enz2.kcat = 0.6
70         enz2.Km = 0.01
71         reac.Kf = 0.001
72         reac.Kb = 0.01
73
74         # Create the output tables
75         graphs = moose.Neutral( '/model/graphs' )
76         outputA = moose.Table ( '/model/graphs/concA' )
77         outputB = moose.Table ( '/model/graphs/concB' )
78
79         # connect up the tables
80         moose.connect( outputA, 'requestOut', a, 'getConc' );
81         moose.connect( outputB, 'requestOut', b, 'getConc' );
82
83         # Schedule the whole lot
84         moose.setClock( 4, 0.01 ) # for the computational
↪objects
85
86         moose.setClock( 8, 1.0 ) # for the plots
87         # The wildcard uses # for single level, and ## for
↪recursive.
88         moose.useClock( 4, '/model/compartment/##', 'process'
↪)
89         moose.useClock( 8, '/model/graphs/#', 'process' )
90
91 def displayPlots():
92         for x in moose.wildcardFind( '/model/graphs/conc#' ):
93             t = numpy.arange( 0, x.vector.size, 1
↪) #sec
94             pylab.plot( t, x.vector, label=x.name
↪)
95         pylab.legend()
96         pylab.show()
97
98 def main():
99         solver = "gsl"
100         makeModel()
101         if ( len ( sys.argv ) == 2 ):
102             solver = sys.argv[1]
103             stoich = moose.Stoich( '/model/compartment/stoich' )

```

(continues on next page)

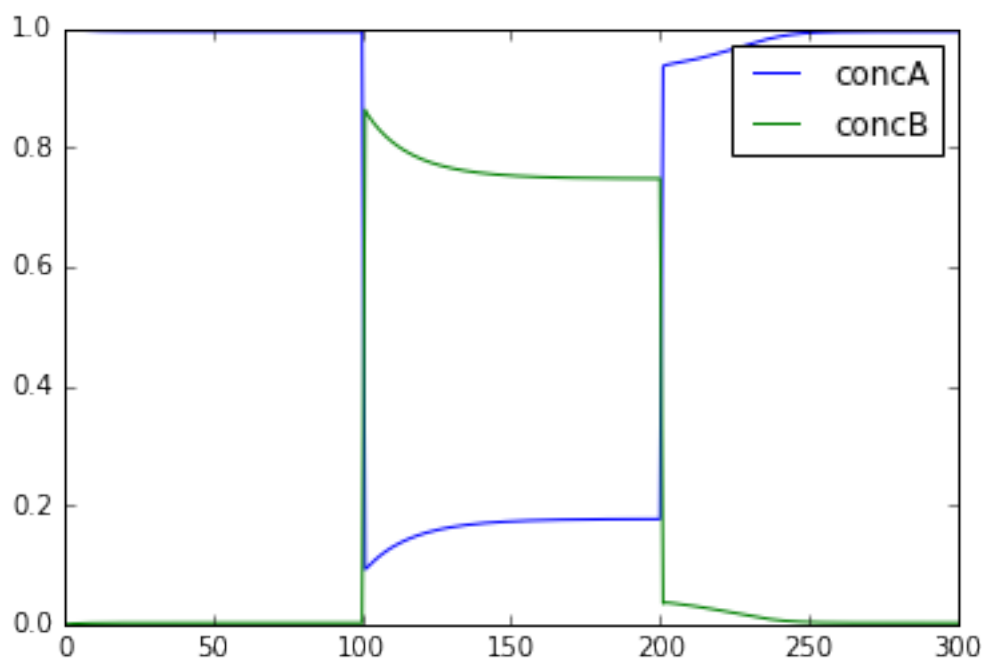
(continued from previous page)

```

103         stoich.compartment = moose.element( '/model/
↪compartment' )
104         if ( solver == 'gssa' ):
105             gsolve = moose.Gsolve( '/model/compartment/ksolve
↪' )
106             stoich.ksolve = gsolve
107         else:
108             ksolve = moose.Ksolve( '/model/compartment/ksolve
↪' )
109             stoich.ksolve = ksolve
110         stoich.path = "/model/compartment/##"
111         #solver.method = "rk5"
112         #mesh = moose.element( "/model/compartment/mesh" )
113         #moose.connect( mesh, "remesh", solver, "remesh" )
114         moose.setClock( 5, 1.0 ) # clock for the solver
115         moose.useClock( 5, '/model/compartment/ksolve',
↪'process' )
116
117         moose.reinit()
118         moose.start( 100.0 ) # Run the model for 100 seconds.
119
120         a = moose.element( '/model/compartment/a' )
121         b = moose.element( '/model/compartment/b' )
122
123         # move most molecules over to b
124         b.conc = b.conc + a.conc * 0.9
125         a.conc = a.conc * 0.1
126         moose.start( 100.0 ) # Run the model for 100 seconds.
127
128         # move most molecules back to a
129         a.conc = a.conc + b.conc * 0.99
130         b.conc = b.conc * 0.01
131         moose.start( 100.0 ) # Run the model for 100 seconds.
132
133         # Iterate through all plots, dump their contents to_
↪data.plot.
134         displayPlots()
135
136         quit()
137
138         # Run the 'main' if this script is executed standalone.
139         if __name__ == '__main__':
140             main()

```

Output:



6.1.2 Scale Volumes

File name: **scaleVolumes.py**

This script runs exactly the same model as in `simpleBis.py`, but it automatically scales the volumes from $1e-19$ down to smaller values.

Note how the simulation successively becomes noisier, until at very small volumes there are spontaneous state transitions.

Code:

```

1 #####
  ↳###
2 ## This program is part of 'MOOSE', the
3 ## Messaging Object Oriented Simulation Environment.
4 ##
5 ## Copyright (C) 2013 Upinder S. Bhalla. and NCBS
6 ## It is made available under the terms of the
7 ## GNU Lesser General Public License version 2.1
8 ## See the file COPYING.LIB for the full notice.
9 #####
  ↳###
10
11 import math
12 import pylab
13 import numpy
14 import moose
15
16 def makeModel():
17     # create container for model
18     model = moose.Neutral( 'model' )
19     compartment = moose.CubeMesh( '/model/compartment' )
20     compartment.volume = 1e-20
    # the mesh is created automatically by the compartment

```

(continues on next page)

(continued from previous page)

```

21 mesh = moose.element( '/model/compartment/mesh' )
22
23 # create molecules and reactions
24 a = moose.Pool( '/model/compartment/a' )
25 b = moose.Pool( '/model/compartment/b' )
26 c = moose.Pool( '/model/compartment/c' )
27 enz1 = moose.Enz( '/model/compartment/b/enz1' )
28 enz2 = moose.Enz( '/model/compartment/c/enz2' )
29 cplx1 = moose.Pool( '/model/compartment/b/enz1/cplx' )
30 cplx2 = moose.Pool( '/model/compartment/c/enz2/cplx' )
31 reac = moose.Reac( '/model/compartment/reac' )
32
33 # connect them up for reactions
34 moose.connect( enz1, 'sub', a, 'reac' )
35 moose.connect( enz1, 'prd', b, 'reac' )
36 moose.connect( enz1, 'enz', b, 'reac' )
37 moose.connect( enz1, 'cplx', cplx1, 'reac' )
38
39 moose.connect( enz2, 'sub', b, 'reac' )
40 moose.connect( enz2, 'prd', a, 'reac' )
41 moose.connect( enz2, 'enz', c, 'reac' )
42 moose.connect( enz2, 'cplx', cplx2, 'reac' )
43
44 moose.connect( reac, 'sub', a, 'reac' )
45 moose.connect( reac, 'prd', b, 'reac' )
46
47 # connect them up to the compartment for volumes
48 #for x in ( a, b, c, cplx1, cplx2 ):
49 #    moose.connect( x, 'mesh', mesh,
↳ 'mesh' )
50
51 # Assign parameters
52 a.concInit = 1
53 b.concInit = 0
54 c.concInit = 0.01
55 enz1.kcat = 0.4
56 enz1.Km = 4
57 enz2.kcat = 0.6
58 enz2.Km = 0.01
59 reac.Kf = 0.001
60 reac.Kb = 0.01
61
62 # Create the output tables
63 graphs = moose.Neutral( '/model/graphs' )
64 outputA = moose.Table( '/model/graphs/concA' )
65 outputB = moose.Table( '/model/graphs/concB' )
66
67 # connect up the tables
68 moose.connect( outputA, 'requestOut', a, 'getConc' );
69 moose.connect( outputB, 'requestOut', b, 'getConc' );
70
71 # Schedule the whole lot
72 moose.setClock( 4, 0.01 ) # for the computational objects
73 moose.setClock( 8, 1.0 ) # for the plots
74 # The wildcard uses # for single level, and ## for_
↳ recursive.

```

(continues on next page)

(continued from previous page)

```

75         moose.useClock( 4, '/model/compartment/##', 'process' )
76         moose.useClock( 8, '/model/graphs/#', 'process' )
77
78     def displayPlots():
79         for x in moose.wildcardFind( '/model/graphs/conc#' ):
80             t = numpy.arange( 0, x.vector.size, 1 )
81             pylab.plot( t, x.vector, label=x.name )
82
83     def main():
84
85         """
86         This example illustrates how to run a model at different volumes.
87         The key line is just to set the volume of the compartment::
88
89             compt.volume = vol
90
91         If everything
92         else is set up correctly, then this change propagates through to
93         all
94         reactions molecules.
95
96         For a deterministic reaction one would not see any change in
97         output
98         concentrations.
99         For a stochastic reaction illustrated here, one sees the level of
100        'noise'
101        changing, even though the concentrations are similar up to a
102        point.
103        This example creates a bistable model having two enzymes and a
104        reaction.
105        One of the enzymes is autocatalytic.
106        This model is set up within the script rather than using an
107        external
108        file.
109        The model is set up to run using the GSSA (Gillespie Stochastic
110        systems
111        algorithm) method in MOOSE.
112
113        To run the example, run the script
114
115            python scaleVolumes.py
116
117        and close the plots every cycle to see the outcome of stochastic
118        calculations at ever smaller volumes, keeping concentrations the
119        same.
120        """
121        makeModel()
122        moose.seed( 11111 )
123        gsolve = moose.Gsolve( '/model/compartment/gsolve' )
124        stoich = moose.Stoich( '/model/compartment/stoich' )
125        compt = moose.element( '/model/compartment' );
126        stoich.compartment = compt
127        stoich.ksolve = gsolve
128        stoich.path = "/model/compartment/##"
129        moose.setClock( 5, 1.0 ) # clock for the solver

```

(continues on next page)

(continued from previous page)

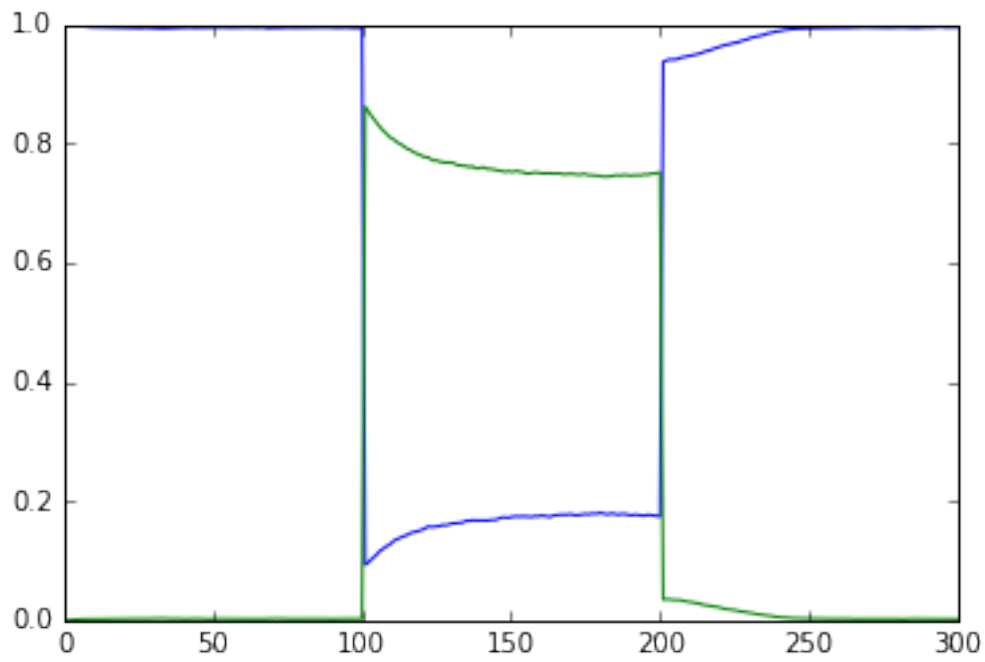
```

123 moose.useClock( 5, '/model/compartments/gsolve', 'process' )
124 a = moose.element( '/model/compartments/a' )
125
126 for vol in ( 1e-19, 1e-20, 1e-21, 3e-22, 1e-22, 3e-23, 1e-23 ):
127     # Set the volume
128     compt.volume = vol
129     print('vol = {}, a.concInit = {}, a.nInit = {}'.format( vol,
→a.concInit, a.nInit))
130     print('Close graph to go to next plot\n')
131
132     moose.reinit()
133     moose.start( 100.0 ) # Run the model for 100 seconds.
134
135     a = moose.element( '/model/compartments/a' )
136     b = moose.element( '/model/compartments/b' )
137
138     # move most molecules over to b
139     b.conc = b.conc + a.conc * 0.9
140     a.conc = a.conc * 0.1
141     moose.start( 100.0 ) # Run the model for 100 seconds.
142
143     # move most molecules back to a
144     a.conc = a.conc + b.conc * 0.99
145     b.conc = b.conc * 0.01
146     moose.start( 100.0 ) # Run the model for 100 seconds.
147
148     # Iterate through all plots, dump their contents to data.plot.
149     displayPlots()
150     pylab.show()
151
152     quit()
153
154 # Run the 'main' if this script is executed standalone.
155 if __name__ == '__main__':
156     main()

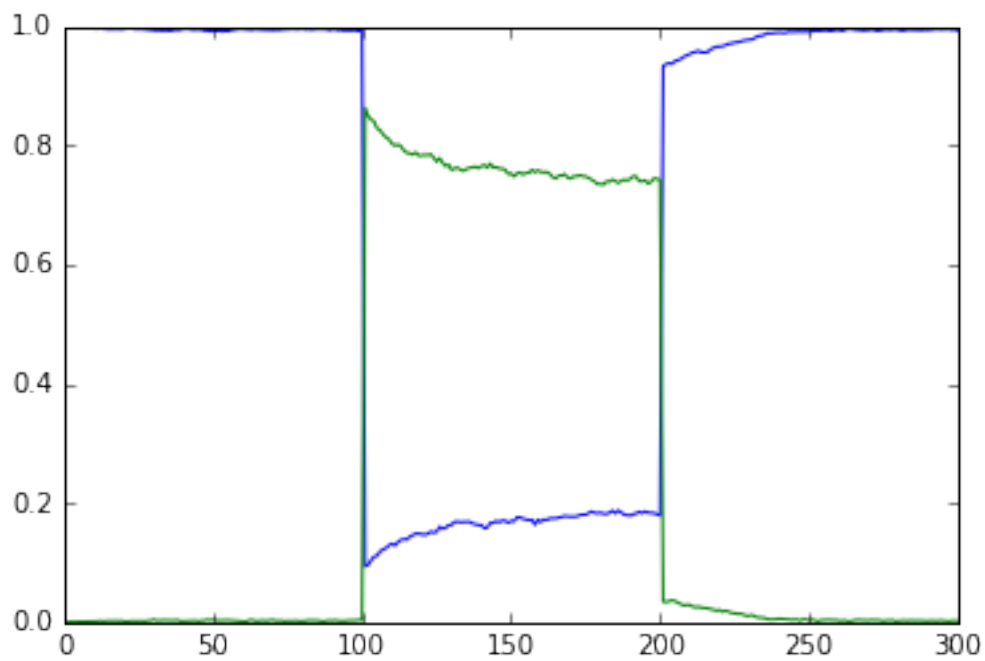
```

Output:

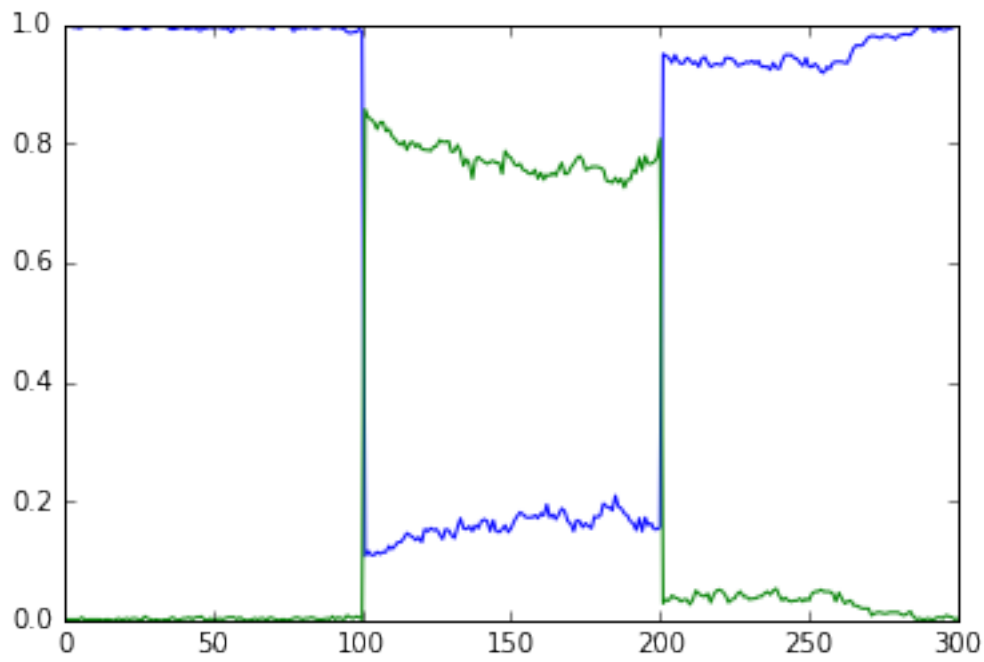
```
vol = 1e-19, a.concInit = 1.0, a.nInit = 60221.415
```



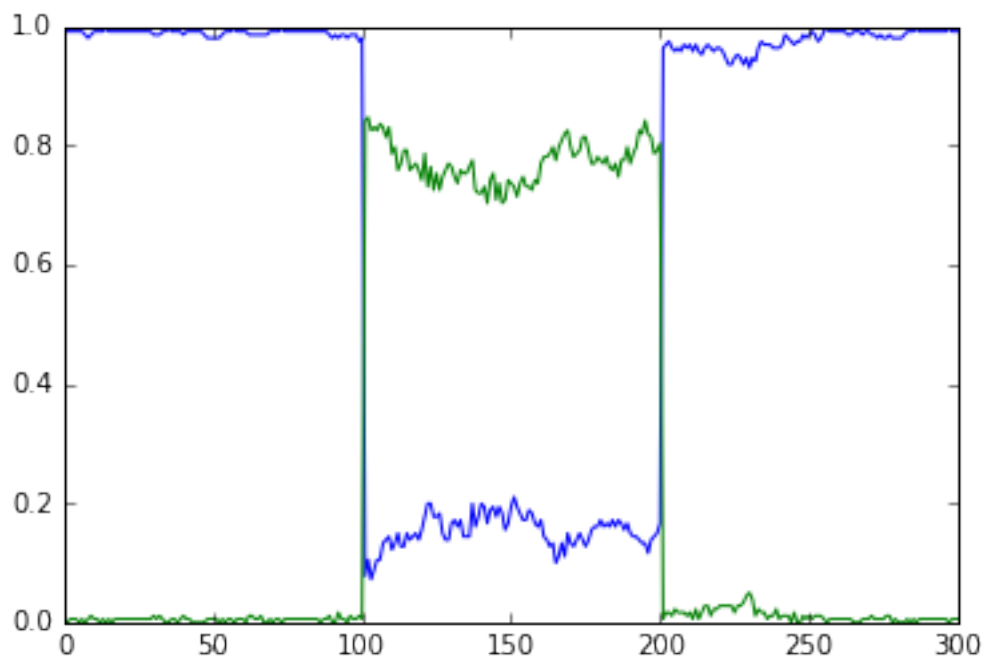
```
vol = 1e-20, a.concInit = 1.0, a.nInit = 6022.1415
```



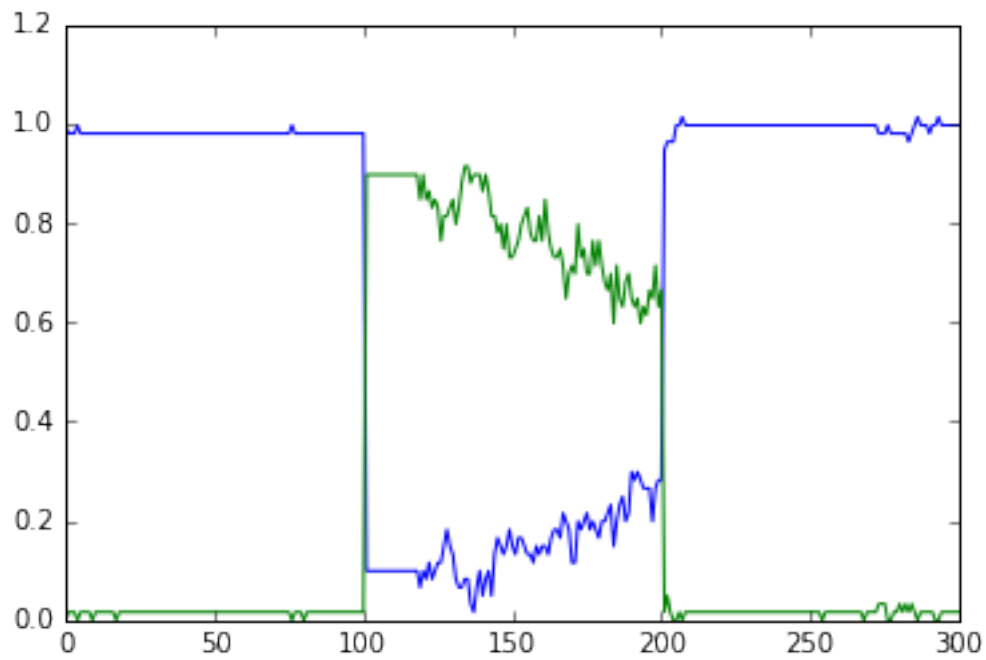
```
vol = 1e-21, a.concInit = 1.0, a.nInit = 602.21415
```



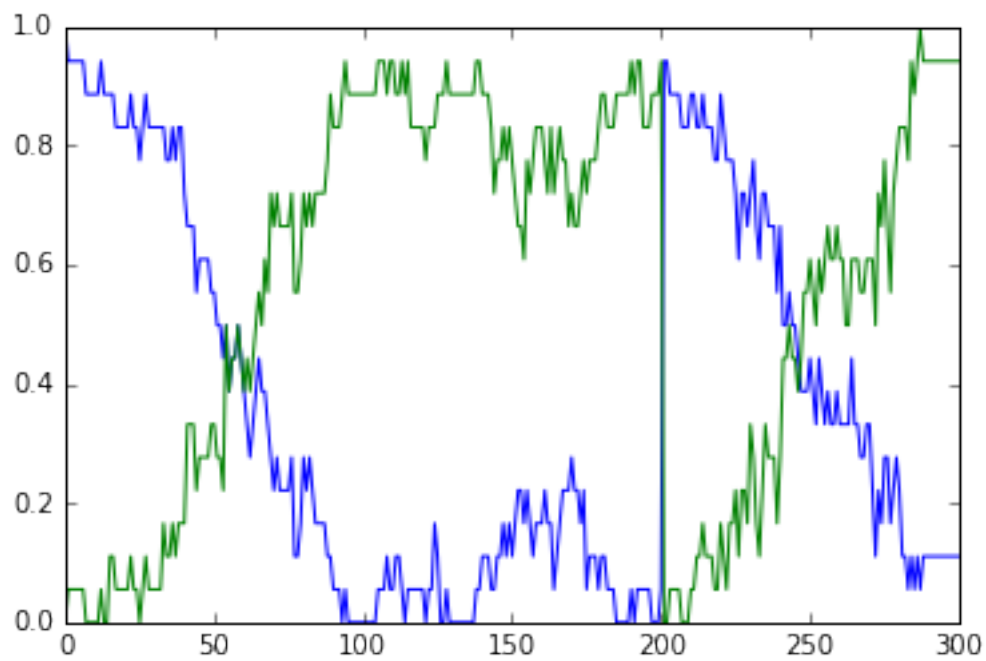
```
vol = 3e-22, a.concInit = 1.0, a.nInit = 180.664245
```



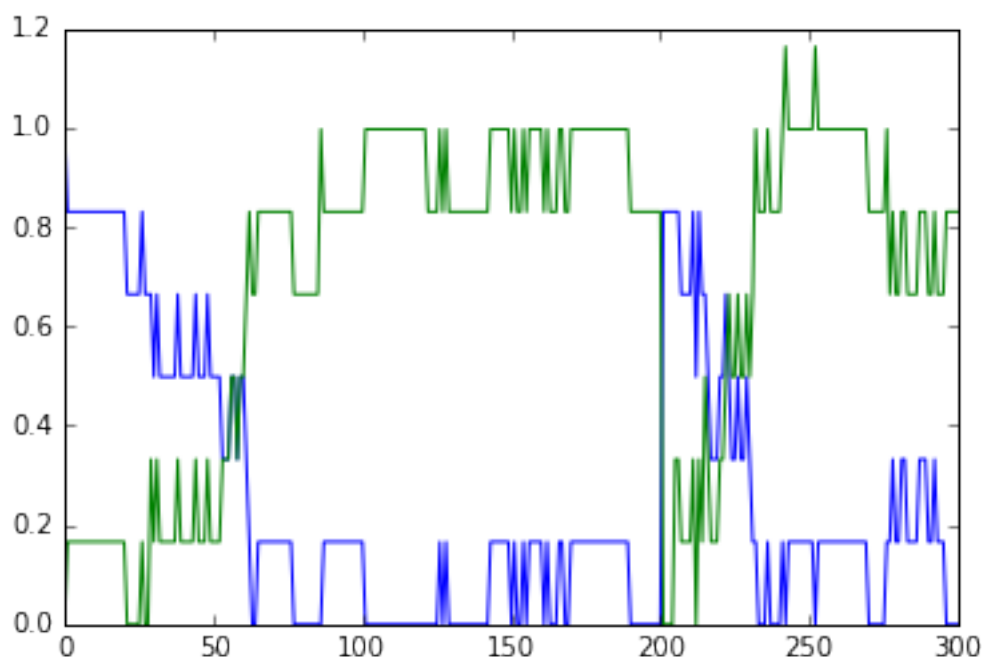
```
vol = 1e-22, a.concInit = 1.0, a.nInit = 60.221415
```



```
vol = 3e-23, a.concInit = 1.0, a.nInit = 18.0664245
```



```
vol = 1e-23, a.concInit = 1.0, a.nInit = 6.0221415
```

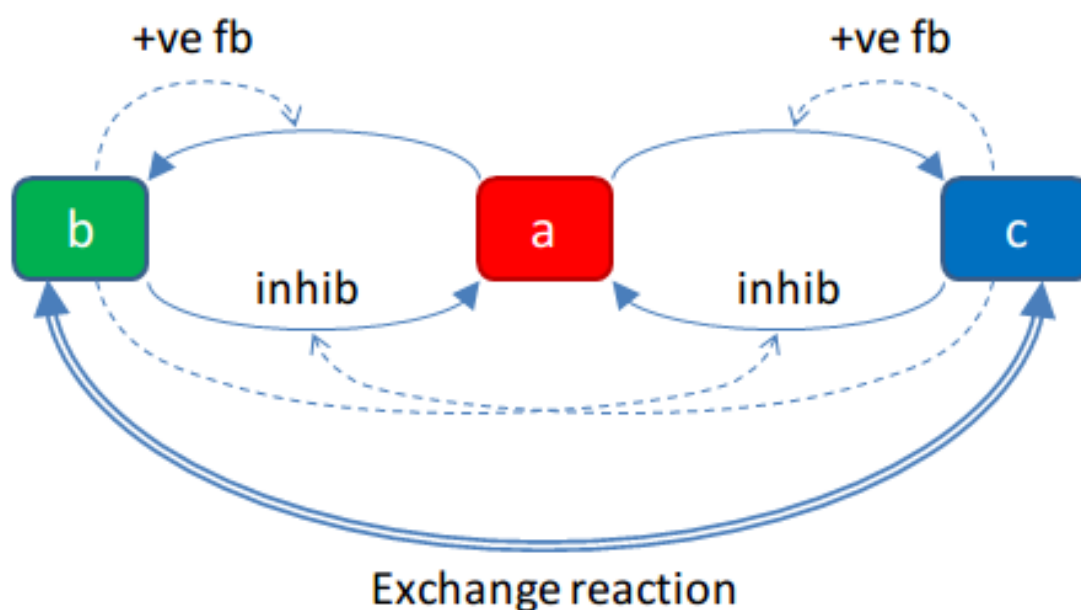
6.1.3 Strong Bistable System

File name: **strongBis.py**

This example illustrates a particularly strong, that is, parametrically robust bistable system. The model topology is symmetric between molecules **b** and **c**. We have both positive feedback of molecules **b** and **c** onto themselves, and also inhibition of **b** by **c** and vice versa.

Strong bistable tutorial

Ramakrishnan and Bhalla, PLoSCB 2008



Open the python file to see what is happening. The simulation starts at a symmetric point and

the model settles at precisely the balance point where **a**, **b**, and **c** are at the same concentration. At $t = 100$ we apply a small molecular 'tap' to push it over to a state where **c** is larger. This is stable. At $t = 210$ we apply a moderate push to show that it is now very stably in this state, and the system rebounds to its original levels. At $t = 320$ we apply a strong push to take it over to a state where **b** is larger. At $t = 430$ we give it a strong push to take it back to the **c** dominant state.

Code:

```

1 #####
2  →###
3  ## This program is part of 'MOOSE', the
4  ## Messaging Object Oriented Simulation Environment.
5  ## Copyright (C) 2014 Upinder S. Bhalla. and NCBS
6  ## It is made available under the terms of the
7  ## GNU Lesser General Public License version 2.1
8  ## See the file COPYING.LIB for the full notice.
9  #####
10  →###
11
12  import moose
13  import matplotlib.pyplot as plt
14  import matplotlib.image as mpimg
15  import pylab
16  import numpy
17  import sys
18
19  def main():
20
21      solver = "gsl" # Pick any of gsl, gssa, ee..
22      #solver = "gssa" # Pick any of gsl, gssa, ee..
23      #moose.seed( 1234 ) # Needed if stochastic.
24      mfile = '../genesis/M1719.g'
25      runtime = 100.0
26      if ( len( sys.argv ) >= 2 ):
27          solver = sys.argv[1]
28      modelId = moose.loadModel( mfile, 'model', solver )
29      # Increase volume so that the stochastic solver gssa
30      # gives an interesting output
31      compt = moose.element( '/model/kinetics' )
32      compt.volume = 0.2e-19
33      r = moose.element( '/model/kinetics/equil' )
34
35      moose.reinit()
36      moose.start( runtime )
37      r.Kf *= 1.1 # small tap to break symmetry
38      moose.start( runtime/10 )
39      r.Kf = r.Kb
40      moose.start( runtime )
41
42      r.Kb *= 2.0 # Moderate push does not tip it back.
43      moose.start( runtime/10 )
44      r.Kb = r.Kf
45      moose.start( runtime )
46
47      r.Kb *= 5.0 # Strong push does tip it over
48      moose.start( runtime/10 )

```

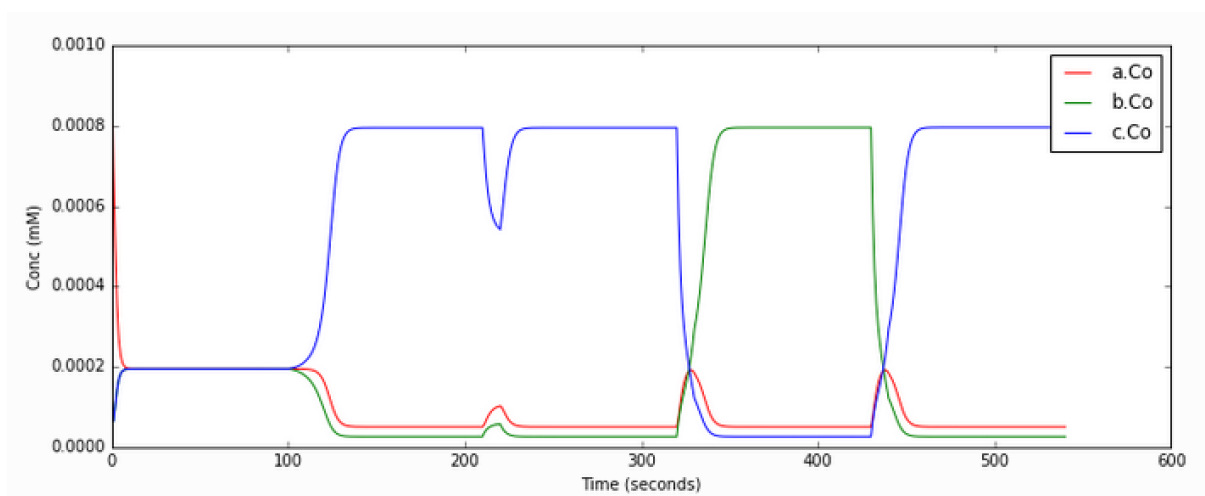
(continues on next page)

(continued from previous page)

```

47     r.Kb = r.Kf
48     moose.start( runtime )
49     r.Kf *= 5.0 # Strong push tips it back.
50     moose.start( runtime/10 )
51     r.Kf = r.Kb
52     moose.start( runtime )
53
54
55     # Display all plots.
56     img = mpimg.imread( 'strongBis.png' )
57     fig = plt.figure( figsize=(12, 10) )
58     png = fig.add_subplot( 211 )
59     imgplot = plt.imshow( img )
60     ax = fig.add_subplot( 212 )
61     x = moose.wildcardFind( '/model/#graphs/conc#/' )
62     dt = moose.element( '/clock' ).tickDt[18]
63     t = numpy.arange( 0, x[0].vector.size, 1 ) * dt
64     ax.plot( t, x[0].vector, 'r-', label=x[0].name )
65     ax.plot( t, x[1].vector, 'g-', label=x[1].name )
66     ax.plot( t, x[2].vector, 'b-', label=x[2].name )
67     plt.ylabel( 'Conc (mM)' )
68     plt.xlabel( 'Time (seconds)' )
69     pylab.legend()
70     pylab.show()
71
72     # Run the 'main' if this script is executed standalone.
73     if __name__ == '__main__':
74         main()

```

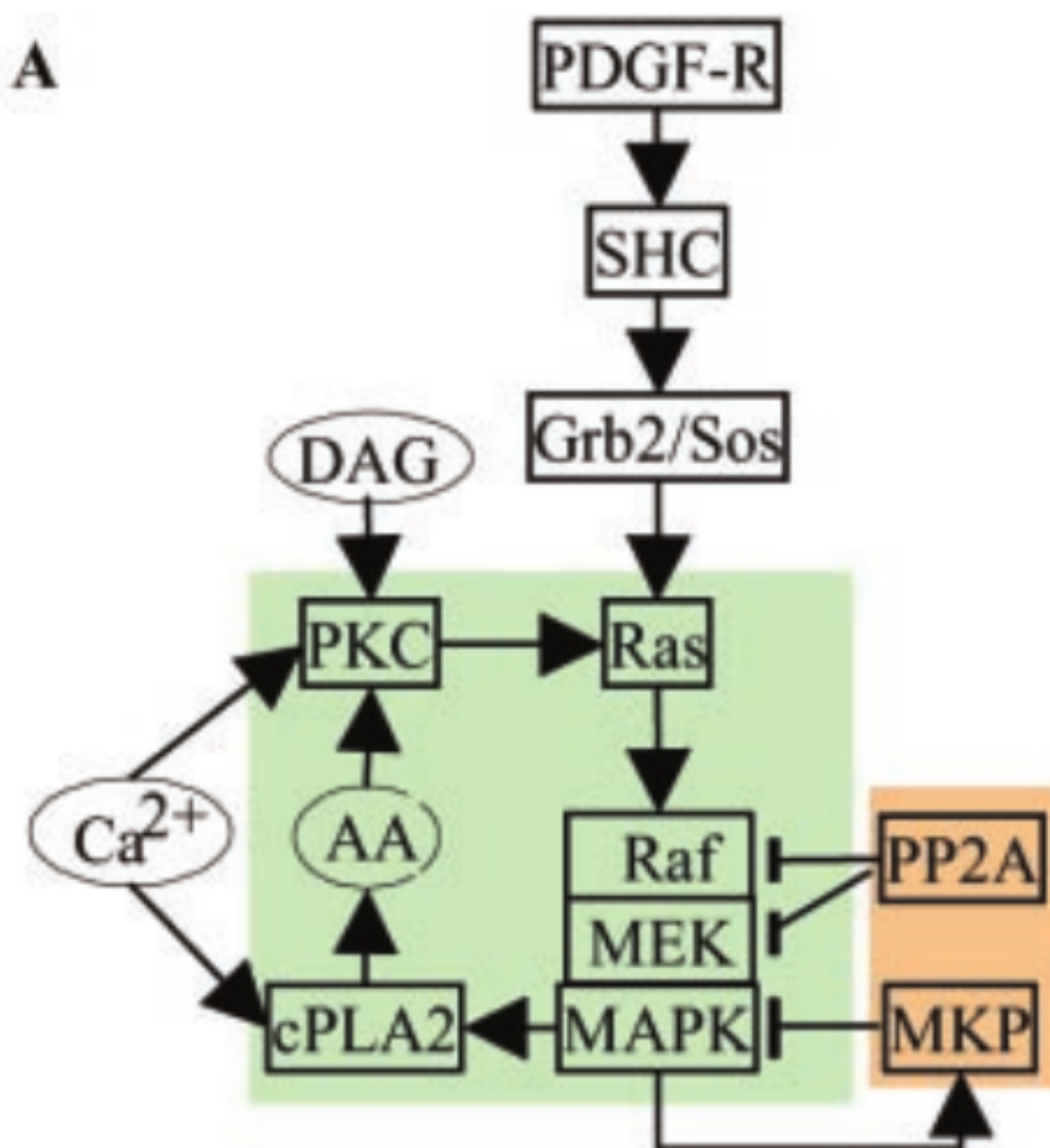
Output:

6.1.4 MAPK Feedback Model

File name: `mapkFB.py`

This example illustrates loading, and running a kinetic model for a much more complex 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^{2+} as inputs.



This model is quite a large one and due to some stiffness in its equations, it takes about 30 seconds to execute. Note that this is still 200 times faster than the events it models.

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, starting at $t = 500$ s. After it has settled to the new 'on' state, the model is made to 'turn off' by setting the system calcium levels to zero. This inhibition starts at $t = 2900$ and goes on for 500 s.

Note that this is a somewhat unphysiological manipulation! Following this the model settles back to the same 'off' state it was in originally.

Code:

```

1 #####
  ↳###
2 ## This program is part of 'MOOSE', the
3 ## Messaging Object Oriented Simulation Environment.
4 ##      Copyright (C) 2014 Upinder S. Bhalla. and NCBS
5 ## It is made available under the terms of the
6 ## GNU Lesser General Public License version 2.1
7 ## See the file COPYING.LIB for the full notice.
8 #####
  ↳###
9
10 import moose
11 import matplotlib.pyplot as plt
12 import matplotlib.image as mpimg
13 import pylab
14 import numpy
15 import sys
16 import os
17
18 scriptDir = os.path.dirname( os.path.realpath( __file__ ) )
19
20 def main():
21     """
22     This example illustrates loading, and running a kinetic model
23     for a bistable positive feedback system, defined in kkit format.
24     This is based on Bhalla, Ram and Iyengar, Science 2002.
25
26     The core of this model is a positive feedback loop comprising of
27     the MAPK cascade, PLA2, and PKC. It receives PDGF and Ca2+ as
28     inputs.
29
30     This model is quite a large one and due to some stiffness in its
31     equations, it runs somewhat slowly.
32
33     The simulation illustrated here shows how the model starts out in
34     a state of low activity. It is induced to 'turn on' when a
35     a PDGF stimulus is given for 400 seconds.
36     After it has settled to the new 'on' state, model is made to
37     'turn off'
38     by setting the system calcium levels to zero for a while. This
39     is a somewhat unphysiological manipulation!
40
41     """
42
43     solver = "gsl" # Pick any of gsl, gssa, ee..
44     #solver = "gssa" # Pick any of gsl, gssa, ee..
45     mfile = os.path.join( scriptDir, '..', '..', 'genesis' , 'acc35.g
  ↳' )
46     runtime = 2000.0
47     if ( len( sys.argv ) == 2 ):
48         solver = sys.argv[1]
49     modelId = moose.loadModel( mfile, 'model', solver )
50     # Increase volume so that the stochastic solver gssa
51     # gives an interesting output
52     compt = moose.element( '/model/kinetics' )

```

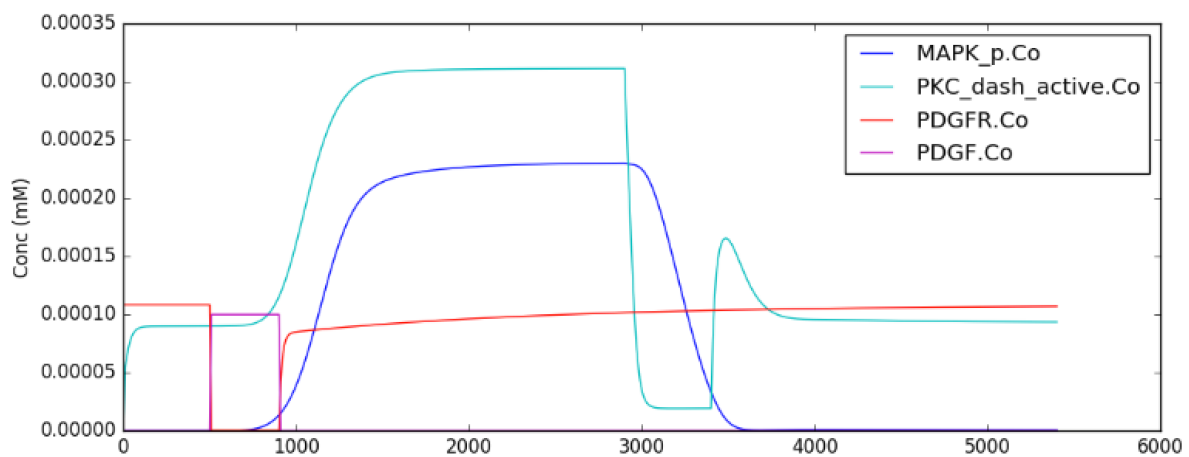
(continues on next page)

(continued from previous page)

```

53     compt.volume = 5e-19
54
55     moose.reinit()
56     moose.start( 500 )
57     moose.element( '/model/kinetics/PDGFR/PDGF' ).concInit = 0.0001
58     moose.start( 400 )
59     moose.element( '/model/kinetics/PDGFR/PDGF' ).concInit = 0.0
60     moose.start( 2000 )
61     moose.element( '/model/kinetics/Ca' ).concInit = 0.0
62     moose.start( 500 )
63     moose.element( '/model/kinetics/Ca' ).concInit = 0.00008
64     moose.start( 2000 )
65
66     # Display all plots.
67     img = mpimg.imread( 'mapkFB.png' )
68     fig = plt.figure( figsize=(12, 10) )
69     png = fig.add_subplot( 211 )
70     imgplot = plt.imshow( img )
71     ax = fig.add_subplot( 212 )
72     x = moose.wildcardFind( '/model/#graphs/conc#/' )
73     t = numpy.arange( 0, x[0].vector.size, 1 ) * x[0].dt
74     ax.plot( t, x[0].vector, 'b-', label=x[0].name )
75     ax.plot( t, x[1].vector, 'c-', label=x[1].name )
76     ax.plot( t, x[2].vector, 'r-', label=x[2].name )
77     ax.plot( t, x[3].vector, 'm-', label=x[3].name )
78     plt.ylabel( 'Conc (mM)' )
79     plt.xlabel( 'Time (seconds)' )
80     pylab.legend()
81     pylab.show()
82
83     # Run the 'main' if this script is executed standalone.
84     if __name__ == '__main__':
85         main()

```

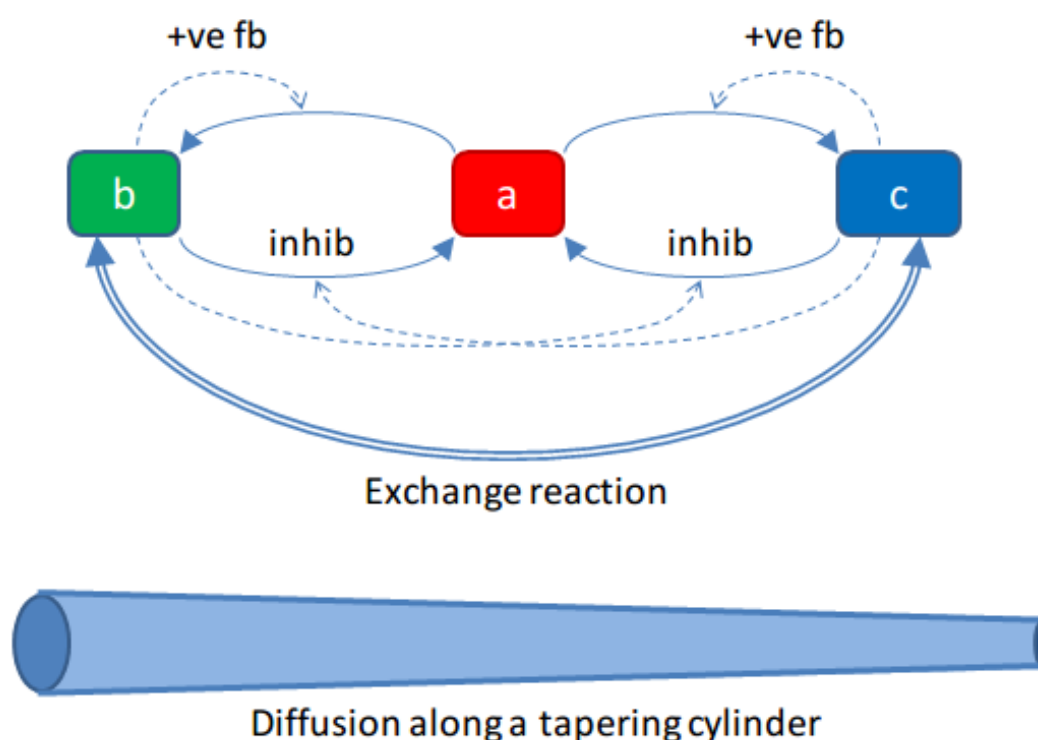
Output:

6.1.5 Propagation of a Bistable System

File name: **propagationBis.py**

All the above models have been well-mixed, that is point or non-spatial models. Bistables do interesting things when they are dispersed in space. This is illustrated in this example. Here we have a tapering cylinder, that is a pseudo 1-dimensional reaction-diffusion system. Every point in this cylinder has the bistable system from the strongBis example.

Propagating bistable tutorial



The example has two stages. First it starts out with the model in the unstable transition point, and introduces a small symmetry-breaking perturbation at one end. This rapidly propagates through the entire length model, leaving molecule **b** at a higher value than **c**.

At $t = 100$ we carry out a different manipulation. We flip the concentrations of molecules **b** and **c** for the left half of the model, and then just let it run. Now we have opposing bistable states on either half. In the middle, the two systems battle it out. Molecule **c** from the left side diffuses over to the right, and tries to inhibit **b**, and vice versa. However we have a small asymmetry due to the tapering of the cylinder. As there is a slightly larger volume on the left, the transition point gradually advances to the right, as molecule **b** yields to the slightly larger amounts of molecule **c**.

Code:

```
1 #####
2  ->###
3  ## This program is part of 'MOOSE', the
4  ## Messaging Object Oriented Simulation Environment.
   ## Copyright (C) 2014 Upinder S. Bhalla. and NCBS
```

(continues on next page)

(continued from previous page)

```

5  ## It is made available under the terms of the
6  ## GNU Lesser General Public License version 2.1
7  ## See the file COPYING.LIB for the full notice.
8  #####
   ↪###
9
10 """
11 This example illustrates propagation of state flips in a
12 linear 1-dimensional reaction-diffusion system. It uses a
13 bistable system loaded in from a kkit definition file, and
14 places this in a tapering cylinder for pseudo 1-dimentional
15 diffusion.
16
17 This example illustrates a number of features of reaction-diffusion
18 calculations.
19
20 First, it shows how to set up such systems. Key steps are to create
21 the compartment and define its voxelization, then create the Ksolve,
22 Dsolve, and Stoich. Then we assign stoich.compartment, ksolve and
23 dsolve in that order. Finally we assign the path of the Stoich.
24
25 For running the model, we start by introducing
26 a small symmetry-breaking increment of concInit
27 of the molecule **b** in the last compartment on the cylinder. The_
   ↪model
28 starts out with molecules at equal concentrations, so that the system_
   ↪would
29 settle to the unstable fixed point. This symmetry breaking leads
30 to the last compartment moving towards the state with an
31 increased concentration of **b**,
32 and this effect propagates to all other compartments.
33
34 Once the model has settled to the state where **b** is high_
   ↪throughout,
35 we simply exchange the concentrations of **b** with **c** in the left
36 half of the cylinder. This introduces a brief transient at the_
   ↪junction,
37 which soon settles to a smooth crossover.
38
39 Finally, as we run the simulation, the tapering geometry comes into_
   ↪play.
40 Since the left hand side has a larger diameter than the right, the
41 state on the left gradually wins over and the transition point slowly
42 moves to the right.
43
44 """
45
46 import math
47 import numpy
48 import matplotlib.pyplot as plt
49 import matplotlib.image as mpimg
50 import moose
51 import sys
52
53 def makeModel():
54     # create container for model

```

(continues on next page)

(continued from previous page)

```

55     r0 = 1e-6          # m
56     r1 = 0.5e-6       # m. Note taper.
57     num = 200
58     diffLength = 1e-6 # m
59     comptLength = num * diffLength          # m
60     diffConst = 20e-12 # m^2/sec
61     concA = 1 # millimolar
62     diffDt = 0.02 # for the diffusion
63     chemDt = 0.2 # for the reaction
64     mfile = '../..genesis/M1719.g'
65
66     model = moose.Neutral( 'model' )
67     compartment = moose.CylMesh( '/model/kinetics' )
68
69     # load in model
70     modelId = moose.loadModel( mfile, '/model', 'ee' )
71     a = moose.element( '/model/kinetics/a' )
72     b = moose.element( '/model/kinetics/b' )
73     c = moose.element( '/model/kinetics/c' )
74
75     ac = a.concInit
76     bc = b.concInit
77     cc = c.concInit
78
79     compartment.r0 = r0
80     compartment.r1 = r1
81     compartment.x0 = 0
82     compartment.x1 = comptLength
83     compartment.diffLength = diffLength
84     assert( compartment.numDiffCompts == num )
85
86     # Assign parameters
87     for x in moose.wildcardFind( '/model/kinetics/#
↪#[ISA=PoolBase]' ):
88         #print 'pools: ', x, x.name
89         x.diffConst = diffConst
90
91     # Make solvers
92     ksolve = moose.Ksolve( '/model/kinetics/ksolve' )
93     dsolve = moose.Dsolve( '/model/dsolve' )
94     # Set up clocks.
95     moose.setClock( 10, diffDt )
96     for i in range( 11, 17 ):
97         moose.setClock( i, chemDt )
98
99     stoich = moose.Stoich( '/model/kinetics/stoich' )
100    stoich.compartment = compartment
101    stoich.ksolve = ksolve
102    stoich.dsolve = dsolve
103    stoich.path = "/model/kinetics/##"
104    print(( 'dsolve.numPools, num = ', dsolve.numPools, _
↪num))
105    b.vec[num-1].concInit *= 1.01 # Break symmetry.
106
107    def main():
108        runtime = 100

```

(continues on next page)

(continued from previous page)

```

109         displayInterval = 2
110         makeModel()
111         dsolve = moose.element( '/model/dsolve' )
112         moose.reinit()
113         #moose.start( runtime ) # Run the model for 10_
↪seconds.
114
115         a = moose.element( '/model/kinetics/a' )
116         b = moose.element( '/model/kinetics/b' )
117         c = moose.element( '/model/kinetics/c' )
118
119         img = mpimg.imread( 'propBis.png' )
120         #imgplot = plt.imshow( img )
121         #plt.show()
122
123         plt.ion()
124         fig = plt.figure( figsize=(12,10) )
125         png = fig.add_subplot(211)
126         imgplot = plt.imshow( img )
127         ax = fig.add_subplot(212)
128         ax.set_ylim( 0, 0.001 )
129         plt.ylabel( 'Conc (mM)' )
130         plt.xlabel( 'Position along cylinder (microns)' )
131         pos = numpy.arange( 0, a.vec.conc.size, 1 )
132         line1, = ax.plot( pos, a.vec.conc, 'r-', label='a' )
133         line2, = ax.plot( pos, b.vec.conc, 'g-', label='b' )
134         line3, = ax.plot( pos, c.vec.conc, 'b-', label='c' )
135         timeLabel = plt.text(60, 0.0009, 'time = 0')
136         plt.legend()
137         fig.canvas.draw()
138
139         for t in range( displayInterval, runtime,
↪displayInterval ):
140             moose.start( displayInterval )
141             line1.set_ydata( a.vec.conc )
142             line2.set_ydata( b.vec.conc )
143             line3.set_ydata( c.vec.conc )
144             timeLabel.set_text( "time = %d" % t )
145             fig.canvas.draw()
146
147         print('Swapping concs of b and c in half the cylinder
↪')
148
149         for i in range( b.numData/2 ):
150             temp = b.vec[i].conc
151             b.vec[i].conc = c.vec[i].conc
152             c.vec[i].conc = temp
153
154         newruntime = 200
155         for t in range( displayInterval, newruntime,
↪displayInterval ):
156             moose.start( displayInterval )
157             line1.set_ydata( a.vec.conc )
158             line2.set_ydata( b.vec.conc )
159             line3.set_ydata( c.vec.conc )
160             timeLabel.set_text( "time = %d" % (t + runtime) )
161             fig.canvas.draw()

```


(continues on next page)

(continued from previous page)

```

161         print( "Hit 'enter' to exit" )
162         sys.stdin.read(1)
163
164
165
166
167 # Run the 'main' if this script is executed standalone.
168 if __name__ == '__main__':
169     main()

```

Output:


propBis-gif-converted-to.png

6.1.6 Steady-state FinderFile name: **findSteadyState**

This is an example of how to use an internal MOOSE solver to find steady states of a system very rapidly. The method starts from a random position in state space that obeys mass conservation. It then finds the nearest steady state and reports it. If it does this enough times it should find all the steady states.

We illustrate this process for 50 attempts to find the steady states. It does find all of them. Each time it plots and prints the values, though the plotting is not necessary.

The printout shows the concentrations of all molecules in the first 5 columns. Then it prints the type of solution, and the numbers of negative and positive eigenvalues. In all cases the calculations are successful, though it takes different numbers of iterations to arrive at the steady state. In some models it would be necessary to put a cap on the number of iterations, if the system is not able to find a steady state.

In this example we run the bistable model using the ODE solver right at the end, and manually enforce transitions to show where the target steady states are.

For more information on the algorithm used, look in the comments within the main method of the code below.

Code:

```

1 #####
  ↳###
2 ## This program is part of 'MOOSE', the
3 ## Messaging Object Oriented Simulation Environment.
4 ##           Copyright (C) 2013 Upinder S. Bhalla. and NCBS

```

(continues on next page)

(continued from previous page)

```

5  ## It is made available under the terms of the
6  ## GNU Lesser General Public License version 2.1
7  ## See the file COPYING.LIB for the full notice.
8  #####
   →###
9
10 from __future__ import print_function
11
12 import math
13 import pylab
14 import numpy
15 import moose
16
17 def main():
18     """
19     This example sets up the kinetic solver and steady-state finder,
   →on
20     a bistable model of a chemical system. The model is set up within
   →the
21     script.
22     The algorithm calls the steady-state finder 50 times with
   →different
23     (randomized) initial conditions, as follows:
24
25     * Set up the random initial condition that fits the conservation
   →laws
26     * Run for 2 seconds. This should not be mathematically necessary,
   →but
27     for obscure numerical reasons it makes it much more likely that
   →the
28     steady state solver will succeed in finding a state.
29     * Find the fixed point
30     * Print out the fixed point vector and various diagnostics.
31     * Run for 10 seconds. This is completely unnecessary, and is done
   →here
32     just so that the resultant graph will show what kind of state
   →has
33     been found.
34
35     After it does all this, the program runs for 100 more seconds on
   →the
36     last found fixed point (which turns out to be a saddle node), then
37     is hard-switched in the script to the first attractor basin from
   →which
38     it runs for another 100 seconds till it settles there, and then
39     is hard-switched yet again to the second attractor and runs for
   →400
40     seconds.
41
42     Looking at the output you will see many features of note:
43
44     * the first attractor (stable point) and the saddle point
   →(unstable
45     fixed point) are both found quite often. But the second
46     attractor is found just once.
47     It has a very small basin of attraction.

```

(continues on next page)

(continued from previous page)

```

48  * The values found for each of the fixed points match well with
→the
49  values found by running the system to steady-state at the end.
50  * There are a large number of failures to find a fixed point.
→These are
51  found and reported in the diagnostics. They show up on the plot
52  as cases where the 10-second runs are not flat.
53
54  If you wanted to find fixed points in a production model, you
→would
55  not need to do the 10-second runs, and you would need to
→eliminate the
56  cases where the state-finder failed. Then you could identify the
→good
57  points and keep track of how many of each were found.
58
59  There is no way to guarantee that all fixed points have been found
60  using this algorithm! If there are points in an obscure corner of
→state
61  space (as for the singleton second attractor convergence in this
62  example) you may have to iterate very many times to find them.
63
64  You may wish to sample concentration space logarithmically rather
→than
65  linearly.
66  """
67  compartment = makeModel()
68  ksolve = moose.Ksolve( '/model/compartment/ksolve' )
69  stoich = moose.Stoich( '/model/compartment/stoich' )
70  stoich.compartment = compartment
71  stoich.ksolve = ksolve
72  stoich.path = "/model/compartment/##"
73  state = moose.SteadyState( '/model/compartment/state' )
74
75  moose.reinit()
76  state.stoich = stoich
77  state.showMatrices()
78  state.convergenceCriterion = 1e-6
79  moose.seed( 111 ) # Used when generating the samples in state
→space
80
81  for i in range( 0, 50 ):
82      getState( ksolve, state )
83
84  # Now display the states of the system at more length to compare.
85  moose.start( 100.0 ) # Run the model for 100 seconds.
86
87  a = moose.element( '/model/compartment/a' )
88  b = moose.element( '/model/compartment/b' )
89
90  # move most molecules over to b
91  b.conc = b.conc + a.conc * 0.9
92  a.conc = a.conc * 0.1
93  moose.start( 100.0 ) # Run the model for 100 seconds.
94
95  # move most molecules back to a

```

(continues on next page)

(continued from previous page)

```

96     a.conc = a.conc + b.conc * 0.99
97     b.conc = b.conc * 0.01
98     moose.start( 400.0 ) # Run the model for 200 seconds.
99
100    # Iterate through all plots, dump their contents to data.plot.
101    displayPlots()
102
103    quit()
104
105    def makeModel():
106        """ This function creates a bistable reaction system using_
107        ↪explicit
108            MOOSE calls rather than load from a file
109            """
110        # create container for model
111        model = moose.Neutral( 'model' )
112        compartment = moose.CubeMesh( '/model/compartment' )
113        compartment.volume = 1e-15
114        # the mesh is created automatically by the compartment
115        mesh = moose.element( '/model/compartment/mesh' )
116
117        # create molecules and reactions
118        a = moose.Pool( '/model/compartment/a' )
119        b = moose.Pool( '/model/compartment/b' )
120        c = moose.Pool( '/model/compartment/c' )
121        enz1 = moose.Enz( '/model/compartment/b/enz1' )
122        enz2 = moose.Enz( '/model/compartment/c/enz2' )
123        cplx1 = moose.Pool( '/model/compartment/b/enz1/cplx' )
124        cplx2 = moose.Pool( '/model/compartment/c/enz2/cplx' )
125        reac = moose.Reac( '/model/compartment/reac' )
126
127        # connect them up for reactions
128        moose.connect( enz1, 'sub', a, 'reac' )
129        moose.connect( enz1, 'prd', b, 'reac' )
130        moose.connect( enz1, 'enz', b, 'reac' )
131        moose.connect( enz1, 'cplx', cplx1, 'reac' )
132
133        moose.connect( enz2, 'sub', b, 'reac' )
134        moose.connect( enz2, 'prd', a, 'reac' )
135        moose.connect( enz2, 'enz', c, 'reac' )
136        moose.connect( enz2, 'cplx', cplx2, 'reac' )
137
138        moose.connect( reac, 'sub', a, 'reac' )
139        moose.connect( reac, 'prd', b, 'reac' )
140
141        # Assign parameters
142        a.concInit = 1
143        b.concInit = 0
144        c.concInit = 0.01
145        enz1.kcat = 0.4
146        enz1.Km = 4
147        enz2.kcat = 0.6
148        enz2.Km = 0.01
149        reac.Kf = 0.001
150        reac.Kb = 0.01

```

(continues on next page)

(continued from previous page)

```

151 # Create the output tables
152 graphs = moose.Neutral( '/model/graphs' )
153 outputA = moose.Table2 ( '/model/graphs/concA' )
154 outputB = moose.Table2 ( '/model/graphs/concB' )
155 outputC = moose.Table2 ( '/model/graphs/concC' )
156 outputCplx1 = moose.Table2 ( '/model/graphs/concCplx1' )
157 outputCplx2 = moose.Table2 ( '/model/graphs/concCplx2' )
158
159 # connect up the tables
160 moose.connect( outputA, 'requestOut', a, 'getConc' );
161 moose.connect( outputB, 'requestOut', b, 'getConc' );
162 moose.connect( outputC, 'requestOut', c, 'getConc' );
163 moose.connect( outputCplx1, 'requestOut', cplx1, 'getConc' );
164 moose.connect( outputCplx2, 'requestOut', cplx2, 'getConc' );
165
166 return compartment
167
168 def displayPlots():
169     for x in moose.wildcardFind( '/model/graphs/conc#' ):
170         t = numpy.arange( 0, x.vector.size, 1 ) #sec
171         pylab.plot( t, x.vector, label=x.name )
172     pylab.legend()
173     pylab.show()
174
175 def getState( ksolve, state ):
176     """ This function finds a steady state starting from a random
177     initial condition that is consistent with the stoichiometry rules
178     and the original model concentrations.
179     """
180     scale = 1.0 / ( 1e-15 * 6.022e23 )
181     state.randomInit() # Randomize init conditions, subject to_
182     ↪stoichiometry
183     moose.start( 2.0 ) # Run the model for 2 seconds.
184     state.settle() # This function finds the steady states.
185     for x in ksolve.nVec[0]:
186         print( "{:.2f}".format( x * scale ), end=' ' )
187
188     print( "Type={} NegEig={} PosEig={} status={} {} Iter={:2d}".
189     ↪format( state.stateType, state.nNegEigenvalues, state.
190     ↪nPosEigenvalues, state.solutionStatus, state.status, state.nIter))
191     moose.start( 10.0 ) # Run model for 10 seconds, just for display
192
193 # Run the 'main' if this script is executed standalone.
194 if __name__ == '__main__':
195     main()

```

Output:

```

0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
↪Iter=16

```

(continues on next page)

(continued from previous page)

```

0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=29
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=10
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=26
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=27
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=30
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=12
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=29
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=12
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=41
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=29
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=18
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=27
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=14
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=12
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=19
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter= 6
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=14
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=23
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=25
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=16
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter= 5
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=43
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter= 9
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=43
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=29
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=27
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter= 9
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=12

```

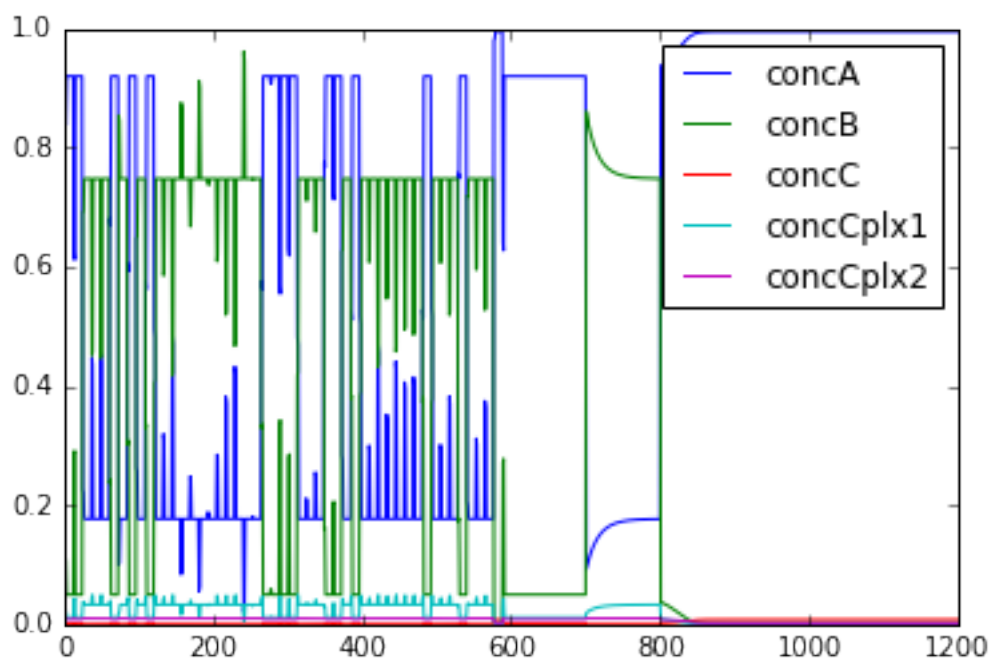
(continues on next page)

(continued from previous page)

```

0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=24
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=26
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=14
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=14
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=10
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=13
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=26
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=21
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=26
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=24
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=24
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=18
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=26
0.18 0.75 0.00 0.03 0.01 Type=5 NegEig=4 PosEig=0 status=0 success_
  ↳Iter=13
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=23
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=24
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter= 8
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=18
0.18 0.75 0.00 0.03 0.01 Type=0 NegEig=3 PosEig=1 status=0 success_
  ↳Iter=21
0.99 0.00 0.01 0.00 0.00 Type=0 NegEig=3 PosEig=0 status=0 success_
  ↳Iter=15
0.92 0.05 0.00 0.01 0.01 Type=2 NegEig=2 PosEig=1 status=0 success_
  ↳Iter=29

```



6.1.7 Dose Response (Under construction)

File name: **doseResponse.py**

This example generates a doseResponse plot for a bistable system, against a control parameter (dose) that takes the system in and out again from the bistable regime. Like the previous example, it uses the steady-state solver to find the stable points for each value of the control parameter. Unfortunately it doesn't work right now. Seems like the kcat scaling isn't being registered.

Code:

```

1  ## Makes and plots the dose response curve for bistable models
2  ## Author: Sahil Moza
3  ## June 26, 2014
4
5  import moose
6  import pylab
7  import numpy as np
8  from matplotlib import pyplot as plt
9
10 def setupSteadyState(simdt,plotDt):
11
12     ksolve = moose.Ksolve( '/model/kinetics/ksolve' )
13     stoich = moose.Stoich( '/model/kinetics/stoich' )
14     stoich.compartment = moose.element('/model/kinetics')
15
16     stoich.ksolve = ksolve
17     #ksolve.stoich = stoich
18     stoich.path = "/model/kinetics/##"
19     state = moose.SteadyState( '/model/kinetics/state' )
20
21     #### Set clocks here

```

(continues on next page)

(continued from previous page)

```

22     #moose.useClock(4, "/model/kinetics/##[]", "process")
23     #moose.setClock(4, float(simdt))
24     #moose.setClock(5, float(simdt))
25     #moose.useClock(5, '/model/kinetics/ksolve', 'process' )
26     #moose.useClock(8, '/model/graphs/#', 'process' )
27     #moose.setClock(8, float(plotDt))
28
29     moose.reinit()
30
31     state.stoich = stoich
32     state.showMatrices()
33     state.convergenceCriterion = 1e-8
34
35     return ksolve, state
36
37 def parseModelName(fileName):
38     pos1=fileName.rfind('/')
39     pos2=fileName.rfind('.')
40     directory=fileName[:pos1]
41     prefix=fileName[pos1+1:pos2]
42     suffix=fileName[pos2+1:len(fileName)]
43     return directory, prefix, suffix
44
45 # Solve for the steady state
46 def getState( ksolve, state, vol):
47     scale = 1.0 / ( vol * 6.022e23 )
48     moose.reinit
49     state.randomInit() # Removing random initial condition to
50     →systematically make Dose reponse curves.
51     moose.start( 2.0 ) # Run the model for 2 seconds.
52     state.settle()
53
54     vector = []
55     a = moose.element( '/model/kinetics/a' ).conc
56     for x in ksolve.nVec[0]:
57         vector.append( x * scale)
58     moose.start( 10.0 ) # Run model for 10 seconds, just for display
59     failedSteadyState = any([np.isnan(x) for x in vector])
60
61     if not (failedSteadyState):
62         return state.stateType, state.solutionStatus, a, vector
63
64 def main():
65     # Setup parameters for simulation and plotting
66     simdt= 1e-2
67     plotDt= 1
68
69     # Factors to change in the dose concentration in log scale
70     factorExponent = 10 ## Base: ten raised to some power.
71     factorBegin = -20
72     factorEnd = 21
73     factorStepsize = 1
74     factorScale = 10.0 ## To scale up or down the factors
75
76     # Load Model and set up the steady state solver.

```

(continues on next page)

(continued from previous page)

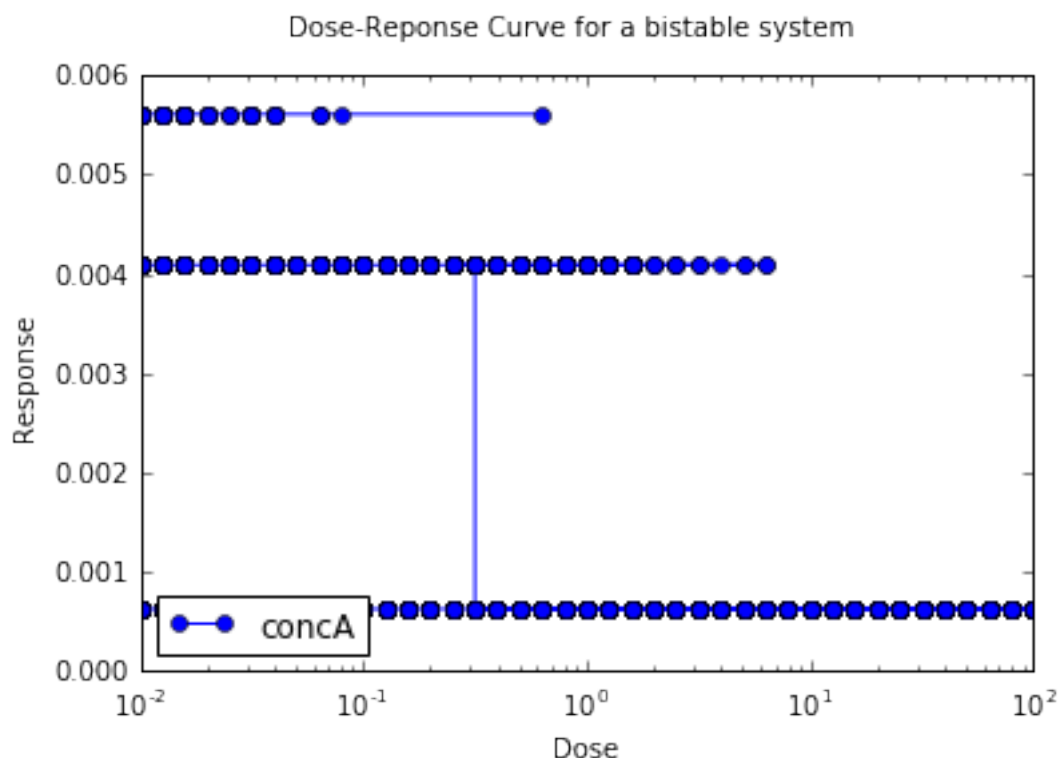
```

77  # model = sys.argv[1] # To load model from a file.
78  model = './19085.cspace'
79  modelPath, modelName, modelType = parseModelName(model)
80  outputDir = modelPath
81
82  modelId = moose.loadModel(model, 'model', 'ee')
83  dosePath = '/model/kinetics/b/DabX' # The dose entity
84
85  ksolve, state = setupSteadyState( simdt, plotDt)
86  vol = moose.element( '/model/kinetics' ).volume
87  iterInit = 100
88  solutionVector = []
89  factorArr = []
90
91  enz = moose.element(dosePath)
92  init = enz.kcat # Dose parameter
93
94  # Change Dose here to .
95  for factor in range(factorBegin, factorEnd, factorStepsize ):
96      scale = factorExponent ** (factor/factorScale)
97      enz.kcat = init * scale
98      print( "scale={:.3f}\tkcat={:.3f}".format( scale, enz.kcat ) )
99      for num in range(iterInit):
100          stateType, solStatus, a, vector = getState( ksolve, state,
↪ vol)
101          if solStatus == 0:
102              #solutionVector.append(vector[0]/sum(vector))
103              solutionVector.append(a)
104              factorArr.append(scale)
105
106  joint = np.array([factorArr, solutionVector])
107  joint = joint[:,joint[1,:].argsort()]
108
109  # Plot dose response.
110  fig0 = plt.figure()
111  pylab.semilogx(joint[0,:],joint[1,:],marker="o",label = 'concA')
112  pylab.xlabel('Dose')
113  pylab.ylabel('Response')
114  pylab.suptitle('Dose-Reponse Curve for a bistable system')
115
116  pylab.legend(loc=3)
117  #plt.savefig(outputDir + "/" + modelName + "_doseResponse" + ".png
↪ ")
118  plt.show()
119  #plt.close(fig0)
120  quit()
121
122
123
124  if __name__ == '__main__':
125      main()

```

Output:

```
scale=0.010 kcat=0.004
scale=0.013 kcat=0.005
scale=0.016 kcat=0.006
scale=0.020 kcat=0.007
scale=0.025 kcat=0.009
scale=0.032 kcat=0.011
scale=0.040 kcat=0.014
scale=0.050 kcat=0.018
scale=0.063 kcat=0.023
scale=0.079 kcat=0.029
scale=0.100 kcat=0.036
scale=0.126 kcat=0.045
scale=0.158 kcat=0.057
scale=0.200 kcat=0.072
scale=0.251 kcat=0.091
scale=0.316 kcat=0.114
scale=0.398 kcat=0.144
scale=0.501 kcat=0.181
scale=0.631 kcat=0.228
scale=0.794 kcat=0.287
scale=1.000 kcat=0.361
scale=1.259 kcat=0.454
scale=1.585 kcat=0.572
scale=1.995 kcat=0.720
scale=2.512 kcat=0.907
scale=3.162 kcat=1.142
scale=3.981 kcat=1.437
scale=5.012 kcat=1.809
scale=6.310 kcat=2.278
scale=7.943 kcat=2.868
scale=10.000      kcat=3.610
scale=12.589      kcat=4.545
scale=15.849      kcat=5.722
scale=19.953      kcat=7.203
scale=25.119      kcat=9.068
scale=31.623      kcat=11.416
scale=39.811      kcat=14.372
scale=50.119      kcat=18.093
scale=63.096      kcat=22.778
scale=79.433      kcat=28.676
scale=100.000     kcat=36.101
```



6.2 Chemical Oscillators

Chemical Oscillators (https://en.wikipedia.org/wiki/Chemical_clock), also known as chemical clocks, are chemical systems in which the concentrations of one or more reactants undergoes periodic changes.

These Oscillatory reactions can be modelled using moose. The examples below demonstrate different types of chemical oscillators, as well as how they can be simulated using moose. Each example has a short description, the code used in the simulation, and the default (gsl solver) output of the code.

Each example can be found as a python file within the main moose folder under

```
(...)/moose/moose-examples/tutorials/ChemicalOscillators
```

In order to run the example, run the script

```
python filename.py
```

in command line, where `filename.py` is the name of the python file you would like to run. The filenames of each example are written in **bold** at the beginning of their respective sections, and the files themselves can be found in the aforementioned directory.

In chemical models that use solvers, there are optional arguments that allow you to specify which solver you would like to use.

```
python filename.py [gsl | gssa | ee]
```

Where:

- **gsl**: 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.
- **gssl**: Optimized Gillespie stochastic systems algorithm, custom implementation. This uses variable timesteps internally. 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 method computes the solution of partial and ordinary differential equations.

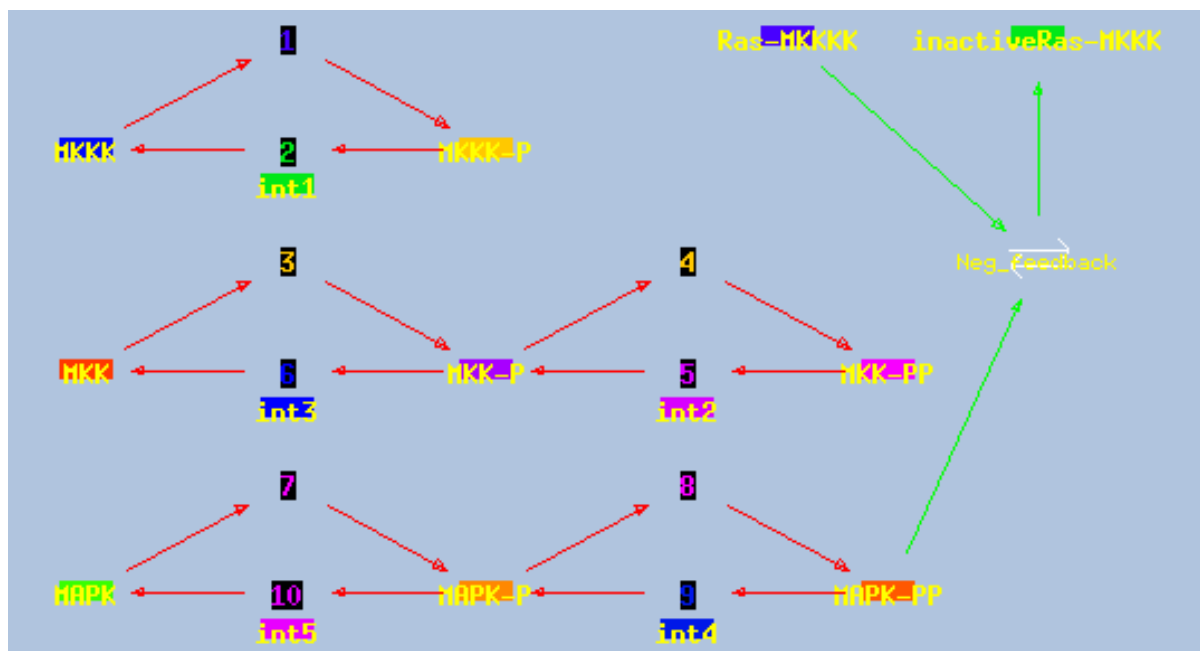
All the following examples can be run with either of the three solvers, each of which has different advantages and disadvantages and each of which might produce a slightly different outcome.

Simply running the file without the optional argument will by default use the **gsl** solver. These **gsl** outputs are the ones shown below.

6.2.1 Slow Feedback Oscillator

File name: **slowFbOsc.py**

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
```

Code:

```
1 #####
2  ↳###
3  ## This program is part of 'MOOSE', the
4  ## Messaging Object Oriented Simulation Environment.
5  ## Copyright (C) 2014 Upinder S. Bhalla. and NCBS
6  ## It is made available under the terms of the
7  ## GNU Lesser General Public License version 2.1
8  ## See the file COPYING.LIB for the full notice.
9  #####
10  ↳###
11
12 import moose
13 import matplotlib.pyplot as plt
14 import matplotlib.image as mpimg
15 import pylab
16 import numpy
17 import sys
18
19 def main():
20     solver = "gsl"
21     mfile = '../genesis/Kholodenko.g'
22     runtime = 5000.0
23     if ( len( sys.argv ) >= 2 ):
24         solver = sys.argv[1]
```

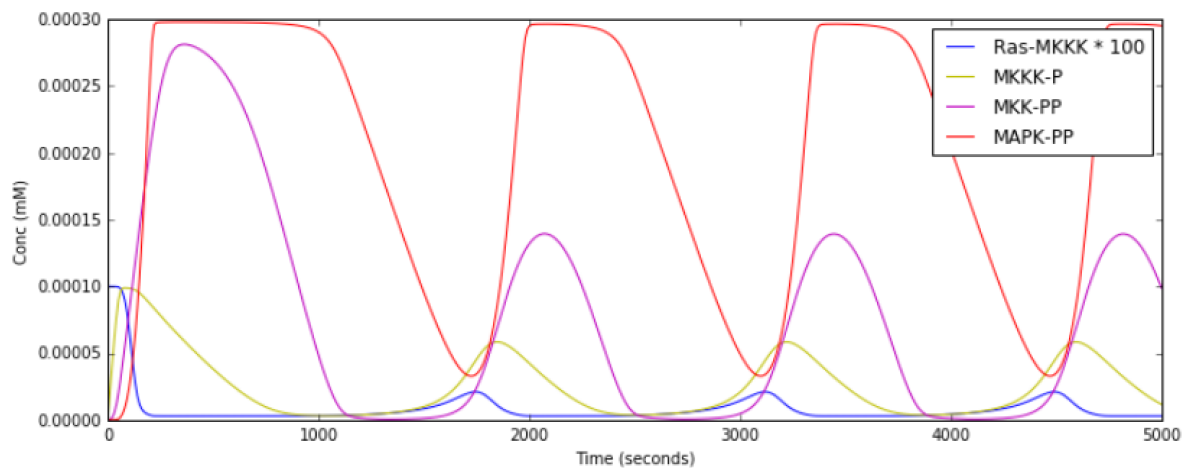
(continues on next page)

(continued from previous page)

```

24 modelId = moose.loadModel( mfile, 'model', solver )
25 dt = moose.element( '/clock' ).tickDt[18]
26 moose.reinit()
27 moose.start( runtime )
28
29 # Display all plots.
30 img = mpimg.imread( 'Kholodenko_tut.png' )
31 fig = plt.figure( figsize=( 12, 10 ) )
32 png = fig.add_subplot( 211 )
33 imgplot = plt.imshow( img )
34 ax = fig.add_subplot( 212 )
35 x = moose.wildcardFind( '/model/#graphs/conc#/' )
36 t = numpy.arange( 0, x[0].vector.size, 1 ) * dt
37 ax.plot( t, x[0].vector * 100, 'b-', label='Ras-MKKK * 100' )
38 ax.plot( t, x[1].vector, 'y-', label='MKKK-P' )
39 ax.plot( t, x[2].vector, 'm-', label='MKK-PP' )
40 ax.plot( t, x[3].vector, 'r-', label='MAPK-PP' )
41 plt.ylabel( 'Conc (mM)' )
42 plt.xlabel( 'Time (seconds)' )
43 pylab.legend()
44 pylab.show()
45
46 # Run the 'main' if this script is executed standalone.
47 if __name__ == '__main__':
48     main()

```

Output:

6.2.2 Turing Pattern Oscillator in One Dimension

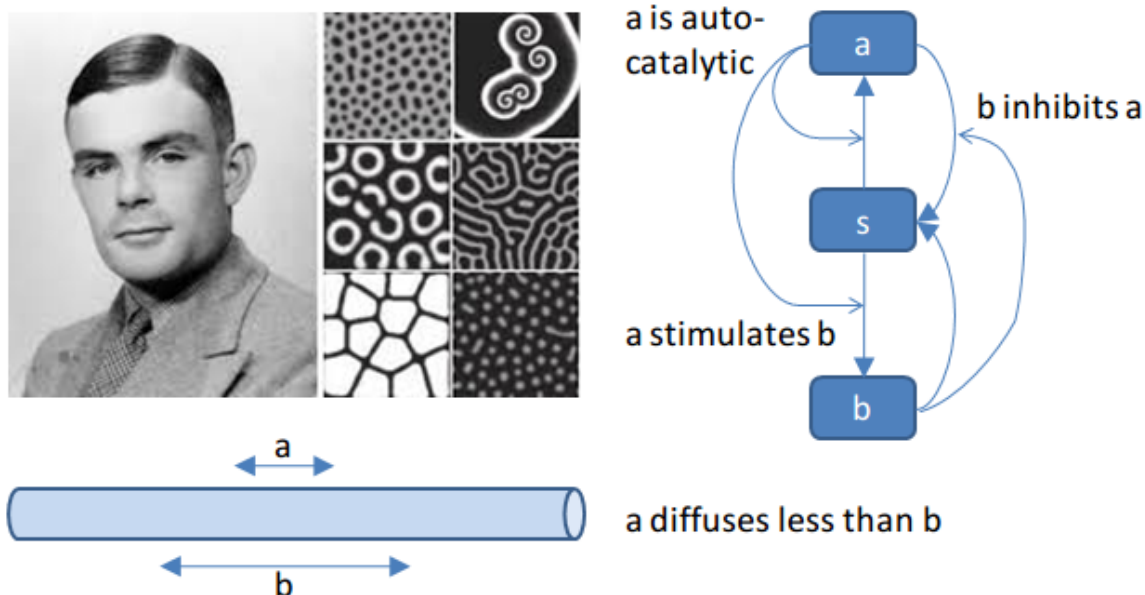
File name: **TuringOneDim.py**

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**.

Turing pattern tutorial



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

Code:

```
1 #####
2  ↳###
3  ## This program is part of 'MOOSE', the
4  ## Messaging Object Oriented Simulation Environment.
5  ## Copyright (C) 2014 Upinder S. Bhalla. and NCBS
6  ## It is made available under the terms of the
7  ## GNU Lesser General Public License version 2.1
8  ## See the file COPYING.LIB for the full notice.
9  #####
10 ↳###
11
12 import math
import numpy
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```

13 import matplotlib.image as mpimg
14 import moose
15
16 def makeModel():
17
18     # create container for model
19     r0 = 1e-6          # m
20     r1 = 1e-6          # m
21     num = 100
22     diffLength = 1e-6 # m
23     len = num * diffLength # m
24     diffConst = 5e-12 # m^2/sec
25     motorRate = 1e-6 # m/sec
26     concA = 1 # millimolar
27     dt4 = 0.02 # for the diffusion
28     dt5 = 0.2 # for the reaction
29
30     model = moose.Neutral( 'model' )
31     compartment = moose.CylMesh( '/model/compartment' )
32     compartment.r0 = r0
33     compartment.r1 = r1
34     compartment.x0 = 0
35     compartment.x1 = len
36     compartment.diffLength = diffLength
37
38     assert( compartment.numDiffCompts == num )
39
40     # create molecules and reactions
41     a = moose.Pool( '/model/compartment/a' )
42     b = moose.Pool( '/model/compartment/b' )
43     s = moose.Pool( '/model/compartment/s' )
44     e1 = moose.MMenz( '/model/compartment/e1' )
45     e2 = moose.MMenz( '/model/compartment/e2' )
46     e3 = moose.MMenz( '/model/compartment/e3' )
47     r1 = moose.Reac( '/model/compartment/r1' )
48     moose.connect( e1, 'sub', s, 'reac' )
49     moose.connect( e1, 'prd', a, 'reac' )
50     moose.connect( a, 'nOut', e1, 'enzDest' )
51     e1.Km = 1
52     e1.kcat = 1
53
54     moose.connect( e2, 'sub', s, 'reac' )
55     moose.connect( e2, 'prd', b, 'reac' )
56     moose.connect( a, 'nOut', e2, 'enzDest' )
57     e2.Km = 1
58     e2.kcat = 0.5
59
60     moose.connect( e3, 'sub', a, 'reac' )
61     moose.connect( e3, 'prd', s, 'reac' )
62     moose.connect( b, 'nOut', e3, 'enzDest' )
63     e3.Km = 0.1
64     e3.kcat = 1
65
66     moose.connect( r1, 'sub', b, 'reac' )
67     moose.connect( r1, 'prd', s, 'reac' )
68     r1.Kf = 0.3 # 1/sec

```

(continues on next page)

(continued from previous page)

```

69     r1.Kb = 0 # 1/sec
70
71     # Assign parameters
72     a.diffConst = diffConst/10
73     b.diffConst = diffConst
74     s.diffConst = 0
75
76     # Make solvers
77     ksolve = moose.Ksolve( '/model/compartment/ksolve' )
78     dsolve = moose.Dsolve( '/model/dsolve' )
79     # Set up clocks. The dsolver to know before assigning stoich
80     moose.setClock( 4, dt4 )
81     moose.setClock( 5, dt5 )
82     moose.useClock( 4, '/model/dsolve', 'process' )
83     # Ksolve must be scheduled after dsolve.
84     moose.useClock( 5, '/model/compartment/ksolve', 'process' )
85
86     stoich = moose.Stoich( '/model/compartment/stoich' )
87     stoich.compartment = compartment
88     stoich.ksolve = ksolve
89     stoich.dsolve = dsolve
90     stoich.path = "/model/compartment/##"
91     assert( dsolve.numPools == 3 )
92     a.vec.concInit = [0.1]*num
93     a.vec[0].concInit *= 1.2 # slight perturbation at one end.
94     b.vec.concInit = [0.1]*num
95     s.vec.concInit = [1]*num
96
97     def displayPlots():
98         a = moose.element( '/model/compartment/a' )
99         b = moose.element( '/model/compartment/b' )
100         pos = numpy.arange( 0, a.vec.conc.size, 1 )
101         pylab.plot( pos, a.vec.conc, label='a' )
102         pylab.plot( pos, b.vec.conc, label='b' )
103         pylab.legend()
104         pylab.show()
105
106     def main():
107         runtime = 400
108         displayInterval = 2
109         makeModel()
110         dsolve = moose.element( '/model/dsolve' )
111         moose.reinit()
112         #moose.start( runtime ) # Run the model for 10 seconds.
113
114         a = moose.element( '/model/compartment/a' )
115         b = moose.element( '/model/compartment/b' )
116         s = moose.element( '/model/compartment/s' )
117
118         img = mpimg.imread( 'turingPatternTut.png' )
119         #imgplot = plt.imshow( img )
120         #plt.show()
121
122         plt.ion()
123         fig = plt.figure( figsize=(12,10) )
124         png = fig.add_subplot(211)

```

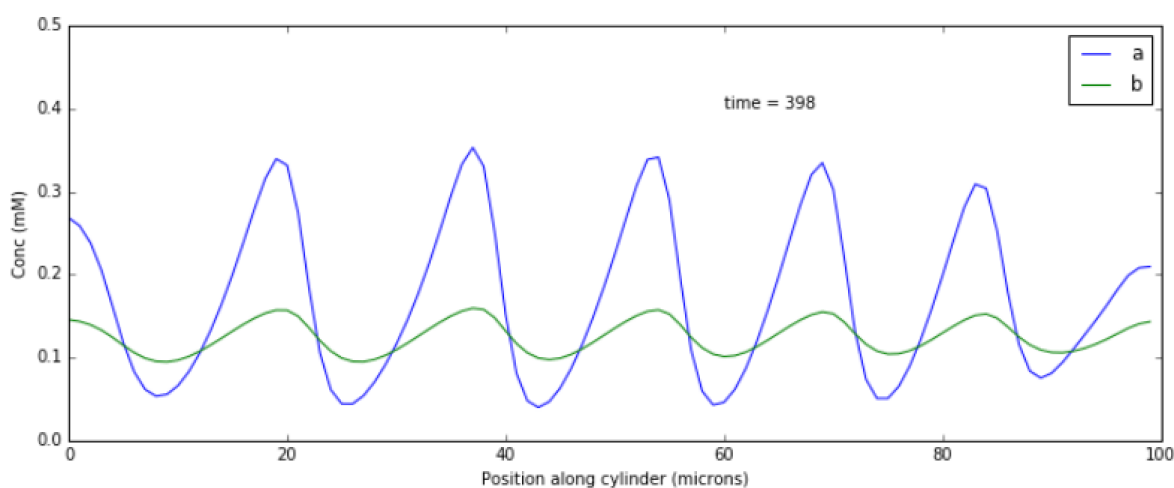
(continues on next page)

(continued from previous page)

```

125 imgplot = plt.imshow( img )
126 ax = fig.add_subplot(212)
127 ax.set_ylim( 0, 0.5 )
128 plt.ylabel( 'Conc (mM)' )
129 plt.xlabel( 'Position along cylinder (microns)' )
130 pos = numpy.arange( 0, a.vec.conc.size, 1 )
131 line1, = ax.plot( pos, a.vec.conc, label='a' )
132 line2, = ax.plot( pos, b.vec.conc, label='b' )
133 timeLabel = plt.text(60, 0.4, 'time = 0')
134 plt.legend()
135 fig.canvas.draw()
136
137 for t in range( displayInterval, runtime, displayInterval ):
138     moose.start( displayInterval )
139     line1.set_ydata( a.vec.conc )
140     line2.set_ydata( b.vec.conc )
141     timeLabel.set_text( "time = %d" % t )
142     fig.canvas.draw()
143
144 print( "Hit 'enter' to exit" )
145 raw_input( )
146
147
148
149 # Run the 'main' if this script is executed standalone.
150 if __name__ == '__main__':
151     main()

```

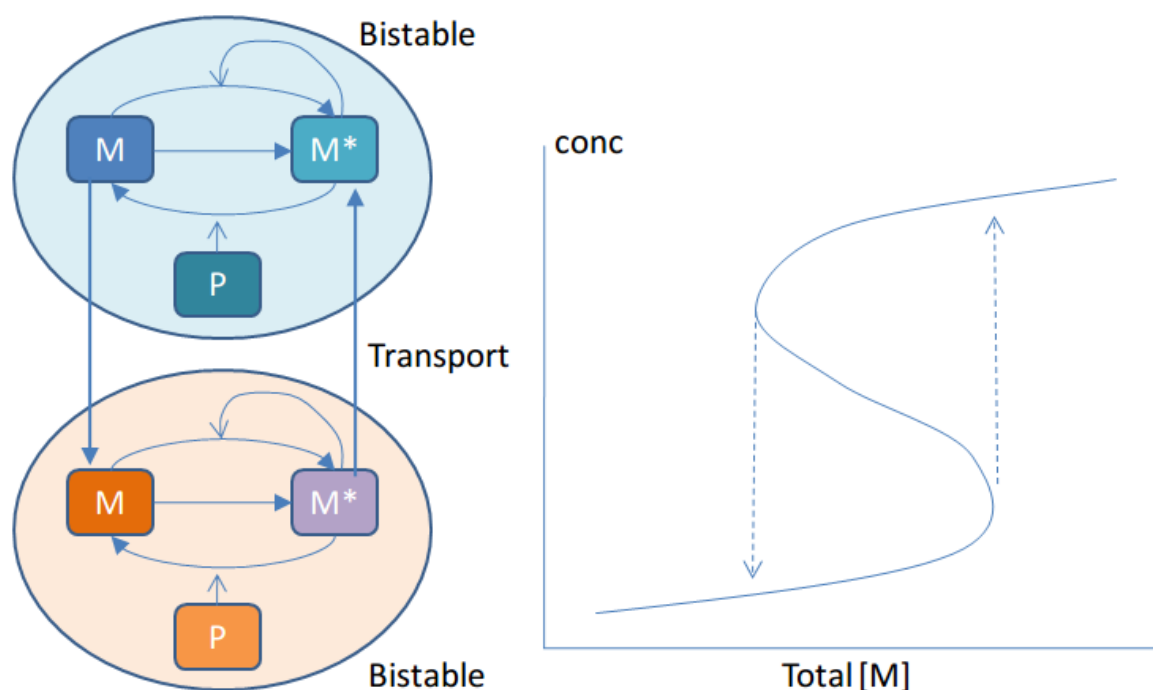
Output:

6.2.3 Relaxation Oscillator

File name: `relaxationOsc.py`

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.

Relaxation oscillator tutorial



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
```

(continues on next page)

(continued from previous page)

* 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
```

Code:

```
1 #####
2 ↪###
3 ## This program is part of 'MOOSE', the
4 ## Messaging Object Oriented Simulation Environment.
5 ## Copyright (C) 2014 Upinder S. Bhalla. and NCBS
6 ## It is made available under the terms of the
7 ## GNU Lesser General Public License version 2.1
8 ## See the file COPYING.LIB for the full notice.
9 #####
10 ↪###
11
12 import moose
13 import matplotlib.pyplot as plt
14 import matplotlib.image as mpimg
15 import pylab
16 import numpy
17 import sys
18
19 def main():
20
21     solver = "gsl" # Pick any of gsl, gssa, ee..
22     #solver = "gssa" # Pick any of gsl, gssa, ee..
23     mfile = '../..genesis/OSC_Cspace.g'
24     runtime = 4000.0
25     if ( len( sys.argv ) >= 2 ):
26         solver = sys.argv[1]
27     modelId = moose.loadModel( mfile, 'model', solver )
28     # Increase volume so that the stochastic solver gssa
29     # gives an interesting output
30     compt = moose.element( '/model/kinetics' )
31     compt.volume = 1e-19
32     dt = moose.element( '/clock' ).tickDt[18] # 18 is the plot clock.
33
34     moose.reinit()
35     moose.start( runtime )
36
37     # Display all plots.
38     img = mpimg.imread( 'relaxOsc_tut.png' )
39     fig = plt.figure( figsize=(12, 10) )
40     png = fig.add_subplot( 211 )
41     imgplot = plt.imshow( img )
42     ax = fig.add_subplot( 212 )
43     x = moose.wildcardFind( '/model/#graphs/conc#/' )
44     t = numpy.arange( 0, x[0].vector.size, 1 ) * dt
45     ax.plot( t, x[0].vector, 'b-', label=x[0].name )
46     ax.plot( t, x[1].vector, 'c-', label=x[1].name )
47     ax.plot( t, x[2].vector, 'r-', label=x[2].name )
```

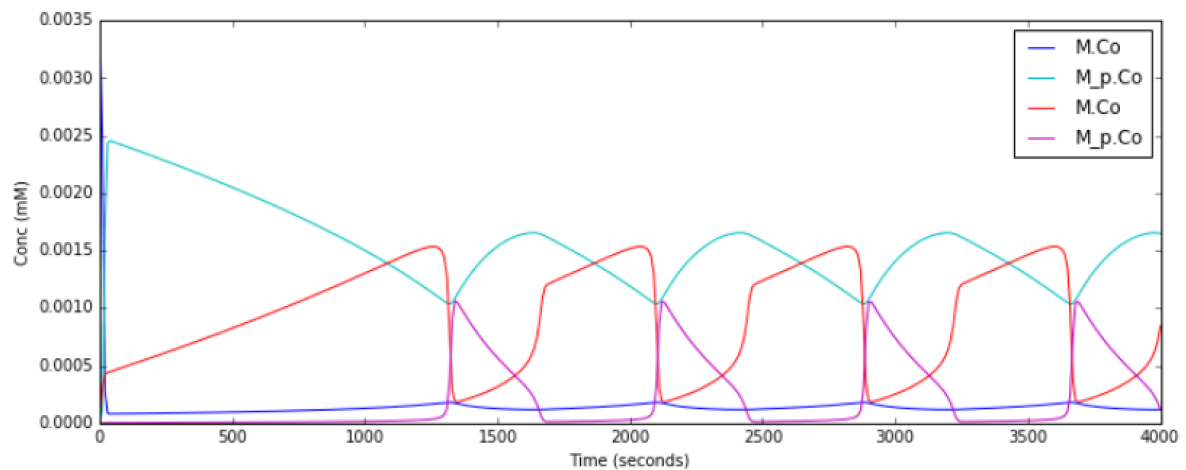
(continues on next page)

(continued from previous page)

```

46     ax.plot( t, x[3].vector, 'm-', label=x[3].name )
47     plt.ylabel( 'Conc (mM)' )
48     plt.xlabel( 'Time (seconds)' )
49     pylab.legend()
50     pylab.show()
51
52     # Run the 'main' if this script is executed standalone.
53     if __name__ == '__main__':
54         main()

```

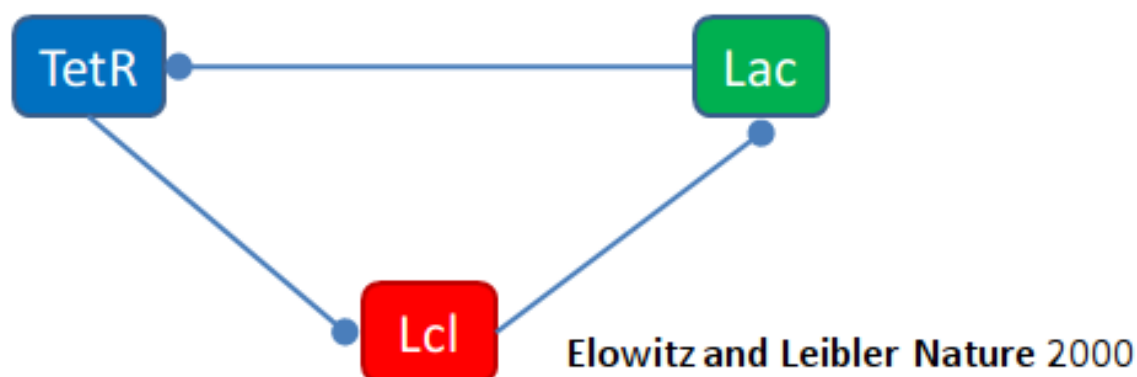
Output:

6.2.4 Repressilator

File name: **repressilator.py**

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

Repressillator tutorial



where **TetR**, **Lac**, and **Lcl** are genes whose products repress each other. The circle 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.
  
```

Code:

```

1 #####
2 ## This program is part of 'MOOSE', the
3 ## Messaging Object Oriented Simulation Environment.
4 ## Copyright (C) 2014 Upinder S. Bhalla. and NCBS
5 ## It is made available under the terms of the
6 ## GNU Lesser General Public License version 2.1
7 ## See the file COPYING.LIB for the full notice.
  
```

(continues on next page)

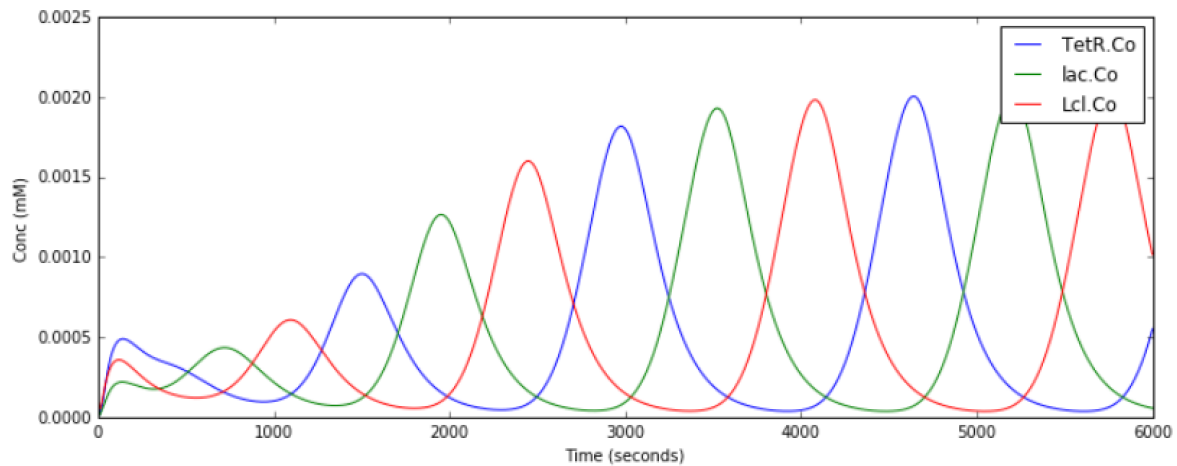
(continued from previous page)

```

8 #####
9 ↪###
10
11 import moose
12 import matplotlib.pyplot as plt
13 import matplotlib.image as mpimg
14 import pylab
15 import numpy
16 import sys
17
18 def main():
19
20     #solver = "gsl" # Pick any of gsl, gssa, ee..
21     solver = "gssa" # Pick any of gsl, gssa, ee..
22     mfile = '../genesis/Repressillator.g'
23     runtime = 6000.0
24     if ( len( sys.argv ) >= 2 ):
25         solver = sys.argv[1]
26     modelId = moose.loadModel( mfile, 'model', solver )
27     # Increase volume so that the stochastic solver gssa
28     # gives an interesting output
29     compt = moose.element( '/model/kinetics' )
30     compt.volume = 1e-19
31     dt = moose.element( '/clock' ).tickDt[18]
32
33     moose.reinit()
34     moose.start( runtime )
35
36     # Display all plots.
37     img = mpimg.imread( 'repressillatorOsc.png' )
38     fig = plt.figure( figsize=(12, 10) )
39     png = fig.add_subplot( 211 )
40     imgplot = plt.imshow( img )
41     ax = fig.add_subplot( 212 )
42     x = moose.wildcardFind( '/model/#graphs/conc#/' )
43     plt.ylabel( 'Conc (mM)' )
44     plt.xlabel( 'Time (seconds)' )
45     for x in moose.wildcardFind( '/model/#graphs/conc#/' ):
46         t = numpy.arange( 0, x.vector.size, 1 ) * dt
47         pylab.plot( t, x.vector, label=x.name )
48     pylab.legend()
49     pylab.show()
50
51 # Run the 'main' if this script is executed standalone.
52 if __name__ == '__main__':
53     main()

```

Output:



6.3 Squid giant axon

This tutorial is an interactive graphical simulation of a squid giant axon, closely based on the ‘Squid’ demo by Mark Nelson which ran in GENESIS.

The [squid giant axon](https://en.wikipedia.org/wiki/Squid_giant_axon) (https://en.wikipedia.org/wiki/Squid_giant_axon) is a very large axon that plays a role in the water jet propulsion systems of squid.

Alan Hodgkin, Andrew Huxley, and John Eccles won the nobel prize in physiology or medicine for their pioneering work on the squid axon. Hodgkin and Huxley were the first to qualitatively describe action potentials within neurons. The large diameter of the squid giant axon (0.5 mm to 1 mm) allowed them to affix electrodes and voltage clamps to precisely measure the action potential as it travelled through the axon. They later went on to mathematically describe this in an equation that paved the road for mathematical and computational biology’s development.

This tutorial models the Hodgkin-Huxley equation within a neat graphical interface that allows one to change different parameters and see how it affects the resulting action potential.

The tutorial can be run from within the `.../moose/moose-examples/squid` directory by running

```
python squid_demo.py
```

in command line from within the directory.

Once the model loads, you can access the inbuilt documentation by clicking on `Help running` in the top right of the window as shown below

The page that pops up will take you through using the GUI, changing the parameters and understanding the model.

For more details on the biophysics behind the model, you can click on `Help biophysics` tab to the immediate right of the `Help running` tab (note that for smaller default window sizes, the tab might not be visible and can be accessed by clicking the `>>` on the top right corner of the GUI).

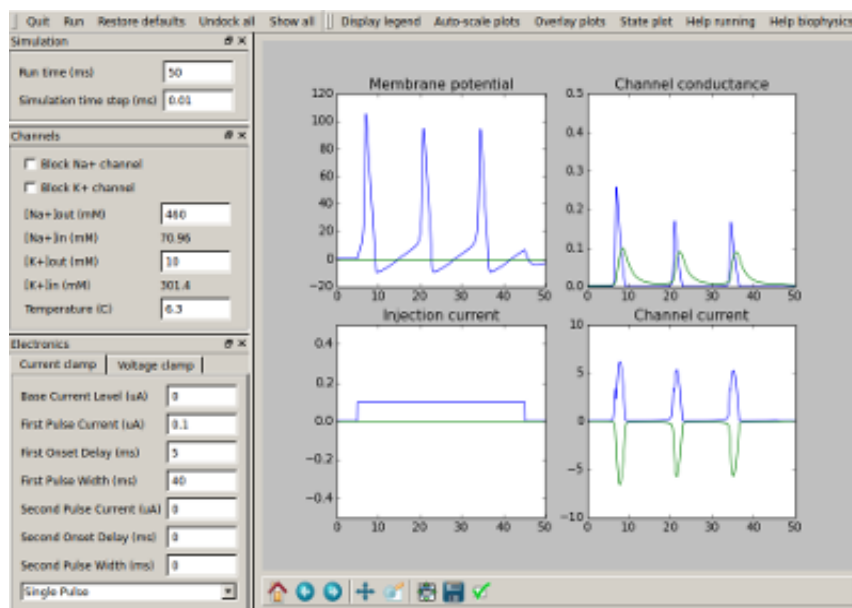


Fig. 1: The GUI of the simulation.

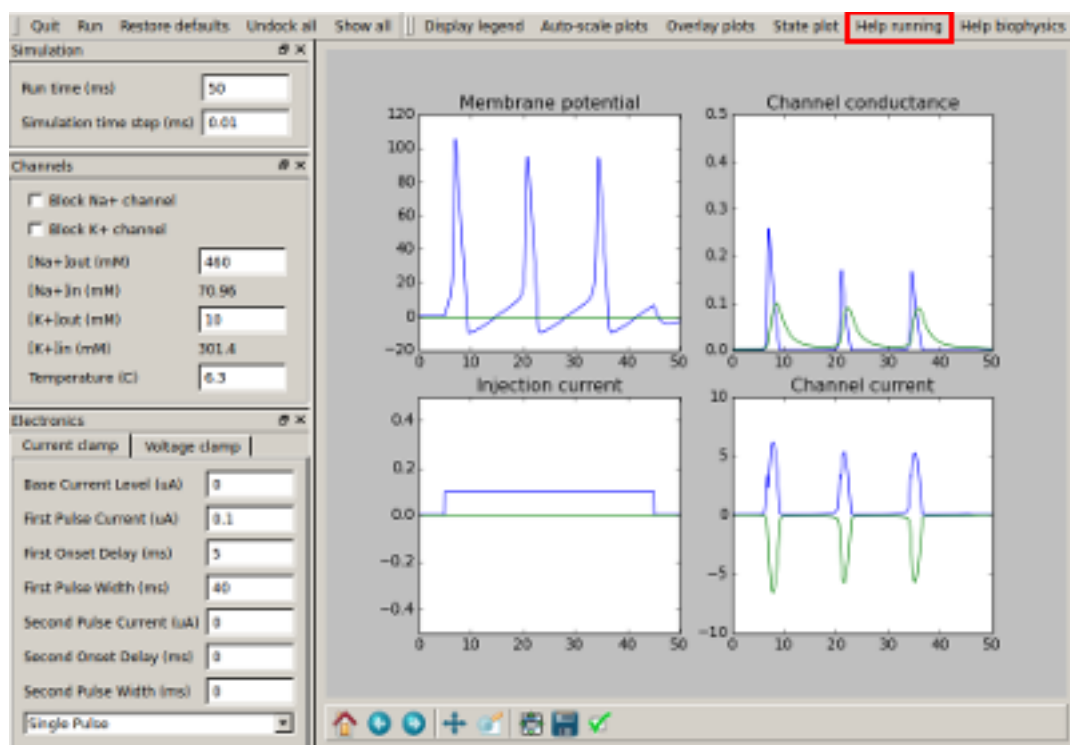


Fig. 2: The “Help running” tab is highlighted in red

7.1 MOOGLI

7.1.1 Use Moogli for plotting

7.2 Matplotlib

7.2.1 Displaying time-series plots

7.2.2 Animation of values along axis

8.1 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')
```

8.1.1 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 simultion 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 *valueFinfo*'s are returned when '*finfoType*' 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
```

(continues on next page)

(continued from previous page)

{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.

8.1.2 Class Hierarchy

- `__builtin__.object` - Melement
 - **Neutral**
 - * Adaptor
 - * Annotator
 - * Arith
 - * **CaConcBase**
 - CaConc
 - ZombieCaConc
 - * **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

CHAPTER 9

Doxygen

Here you can find all the references necessary for MOOSE.

[Click here](#)

CHAPTER 10

Release Notes

Todo: Collect release notes from github.

11.1 Version 3.2.0

- Improved SBML support.
- NeuroML2 support (Thanks to Padraig Glesson)
- Various bugfixes.

CHAPTER 12

Series *chamcham*

12.1 Version 3.1.3

CHAPTER 13

Known issues

Full report can be found at the following places

- Related to [build, packages and documentation](https://github.com/BhallaLab/moose/issues) (<https://github.com/BhallaLab/moose/issues>)
- Related to [moose-core](https://github.com/BhallaLab/moose-core/issues) (<https://github.com/BhallaLab/moose-core/issues>)
- Related to [MOOSE-GUI](https://github.com/BhallaLab/moose-gui/issues) (<https://github.com/BhallaLab/moose-gui/issues>)
- Related to [moogli](https://github.com/BhallaLab/moogli/issues) (<https://github.com/BhallaLab/moogli/issues>)

CHAPTER 14

Indices and tables

- `genindex`
- `modindex`
- `search`