NeuroManager 0.961 User and Programmer's Guide

David B. Stockton

September 18, 2015

1 Introduction

This Guide presents practical details of installing, configuring, and running the NeuroManager software, as well as code–level technical details that should help those working with the code itself. For an overview of NeuroManager, see the published paper (need reference). Familiarity with the Workflow Staging descriptions in the Supplemental Materials should also be helpful.

Note: this is not highly commercialized software produced by a software development team. If there are rough edges, find a way through or around them and let me know — I may be able to fix them in the next version. There are no warrantees, guarantees or anything else — use NeuroManager at your own risk.

2 Installation, configuration, and operation

We will go through a sequence of steps to install, configure, and test NeuroManager. The basic flow of this section can be seen in Figure 1.

We first install to run the simplest setup: NeuroManager installed on a UNIX host which is also acting as the only remote, so that the host and remote software are running on the same machine; called the 'SingleMachine' configuration. We use a simplest-possible simulator called 'SineSim' that is MATLAB only and creates a sine wave plot based on two input parameters. After that we proceed so that we run that same SineSim simulator on increasingly more complex machine setups: changing to a separate remote, adding dual key authentication, using mixed machine types, adding notifications.

Following that we set up and use NeuroManager to run a series of Neuron–based simulators, which are more complex than the SineSim simulator: SimpleSpike 01, 02A, 02B, and 02C. Finally, we use NeuroManager to run two simulators based on a model downloaded from ModelDB.

Table 1 lists the examples we will be covering in this User Guide.

There may also be an MCell example in the code; however, that will not be treated here.

Users with Windows hosts will not be able to run the SingleMachine configuration, since only UNIX machines can be remotes. Special instructions for Windows machines as hosts are embedded within the examples after the SingleMachine example so that Windows users can continue with the regular flow thereafter.

The host computer that actually runs NeuroManager must have a 2012a+ version of MATLAB running on it. For the host installation, no additional toolboxes are necessary, and the student edition works quite nicely. For the remote computers, there must be access to a MATLAB installation with the Compiler toolbox, which is not available in the student edition. For remotes that don't have MATLAB+Compiler, NeuroManager has a cross-compilation option that allows it to compile on a machine that does, then transfer the compiled files to the remote.

For UNIX non–cluster machines only, the host machine can be the same as one of the remote machines. Note that, for the SingleMachine configuration, the host is the remote, so in this case the host must have the Compiler Toolbox.

Example	Filename	Description
SS01	SineSimSet_SS01.m	Sine wave; no SimCore; same UNIX host and remote
SS03	SineSimSet_SS03.m	Sine wave; no SimCore; UNIX host with one remote server – SSH via dual key access
SS04	SineSimSet_SS04.m	Sine wave; no SimCore; UNIX host with remote cluster
SS05	SineSimSet_SS05.m	Sine wave; no Sim Core; UNIX host with heterogeneous Machine Set
SS07	SineSimSet_SS07.m	Sine wave; no Sim Core; Remote operation - UNIX host with Machine Set
SS08	SineSimSet_SS08.m	Sine wave; no Sim Core; Use of Notifications
SS09	SineSimSet_SS09.m	Sine wave; no Sim Core; Write SimSet automatically; multiple SimSets with same Machine Set; SimSet without text file; RNG used in creation of SimSet
NeuronTutorialA	NeuronTutorialASimSet.m	NEURON Tutorial A from University of Edinburgh; NEURON SimCore; hoc–file only
SimpleSpike01	SimpleSpike01SimSet.m	Soma only; NEURON (no Python); mod and hoc files as-is
SimpleSpike02A	SimpleSpike02ASimSet.m	Soma only; NEURON (no Python); mod and hoc files as-is except for parameters.hoc (host); focus on transfer of input parameters into simulation
SimpleSpike02B	SimpleSpike02BSimSet.m	Soma only; NEURON (no Python); mod and hoc files as-is except for parameters.hoc (remote); focus on transfer of input parameters into simulation
SimpleSpike02C	SimpleSpike02CSimSet.m	Soma only; NEURON (no Python); mod and hoc files as-is except for Khh.mod (host); focus on transfer of input parameters into simulation
NeuroML01 SimpleSpike03	SimpleSpike03SimSet.m	Soma only; NEURON (no Python); mod and hoc files as-is except for NaF.mod (host); focus on use and modification of NeuroML input format
MiyashoMOD	Miyasho2001SimSet.m	Full NEURON implementation via Miyasho2001 model from ModelDB; NEURON+Python is core simulator; mod and hoc files as is (except for slight reorganization for NeuroManager use) plus custom Python files; focus on full use of NEURON and to illustrate mixed NEURON-Python integration
KhStudy	KhStudySimSet.m	Full NEURON implementation via Miyasho2001 model from ModelDB; NEURON+Python is core simulator; hoc file modification to make ion channel distribution dependent on input parameter vectors; focus on subclassing of MiyashoMOD simulator
NMSessionML01	RunNMSessionML01.m	Same as SineSim03 but run from NMSessionML XML file
NMSessionML02	RunNMSessionML02.m	Same as SimpleSpike02A but run from NMSessionML XML file
NMSessionML03	RunNMSessionML03.m	Same as KhStudy but run from NMSessionML XML file
GPUSim	GPUSimSet.m	Runs an illustrative MATLAB GPU sine wave example

Table 1: NeuroManager Detailed Examples

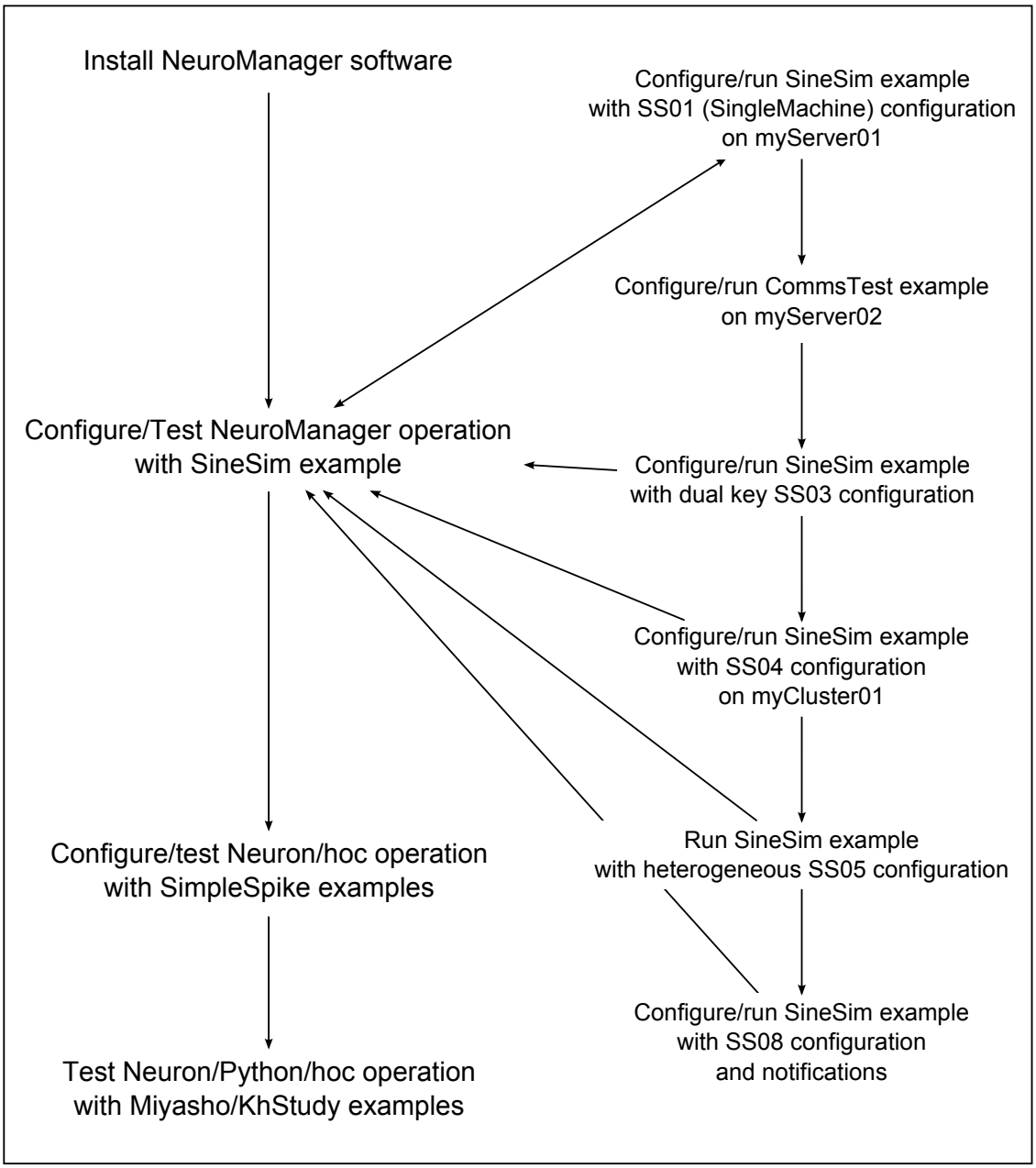

Figure 1: Installation, configuration, and test flow. Section 2 follows this sequence for getting NeuroManager to run on a new system.

2.1 Install NeuroManager software

Install the NeuroManager code by going to the repository site at https://github.com/SantamariaLab/NeuroManager, copying the clone URL (right side of the page), then opening a terminal window at the directory in which you want NeuroManager to be installed. Then run 'git clone https://github.com/SantamariaLab/NeuroManager'. Alternately you can download a zip file by clicking the button on the right side, or on Windows, use the 'Clone in Desktop' button which also requires installing GitHub Desktop. The NeuroManager directory will appear in the directory you gave the clone command. For the purposes of this document we will call this installed NeuroManager directory 'NMInstallDir', but you may wish to use a different name. This action will automatically create several subdirectories: Core (which holds the NeuroManager core files); LocalMachine (which holds machine resource files); SSHLib, which holds the SSH library files; and CommsTest, SineSim, and NeurSim, which hold example simulator files.

Add the installation directory to your MATLAB search path permanently by editing your **pathdef.m** or MATLAB **startup.m** file, as you prefer.

2.2 Quickstart with the SineSim simulator

The SineSim simulator is NeuroManager's 'Hello, World!' simulator. It simply creates a sine wave and plots it, all within MATLAB itself. The SineSim simulator doesn't access external simulators such as Neuron or MCell, nor does it use programs outside the compiled MATLAB executable. The success of each simulation, as well as the proper production of sine wave characteristics according to the input SimSpec, can be verified by examining the output plot's tiff format file, which is passively viewable through a directory viewer's preview pane. For details on the SineSim simulator, see Section 3.12 below.

In this and the following sections we use the SineSim simulator as a vehicle to configure and test your local machine resources. This first step is a single machine configuration which requires your machine to be both host and remote. To extend your Machine Set outside of the single machine, see the following Section 2.3. It's not necessary to do all the machine configuration sections; just enough to get you going with whatever machine resources you desire. Table 1 lists the m-files associated with these sections.

2.2.1 SineSim SS01: SingleMachine configuration

The SingleMachine configuration runs simulations on the same machine as the host. For those with a UNIX host, this example requires a MATLAB installation with the MATLAB Compiler Toolbox. This first step is not applicable to user with a Windows host.

Configuration: First we change a few entries in the file **LocalMachine/createMyServer01Data.m**, as follows:

1. Install the MATLAB MCR on the machine (see http://www.mathworks.com/products/compiler/mcr/). Then go into the **LocalMachine/createMyServer01Data.m** file and edit the line that starts with 'md.addSetting('mcrDir',' to indicate the full path location of your MCR installation, something like '/usr/local/MATLAB/MATLAB_Compiler_Runtime/v80'.

2. Within MATLAB type 'getenv('HOSTNAME')'. The result is the name MATLAB has for your machine. Then go into the file and edit the line that starts with 'md.addSetting('id',' to insert that name. This approach to obtaining the ID is only necessary for machines that may function as a host as well.

3. Edit the line that starts with 'md.addSetting('matlabCompilerDir', ' to indicate the location of your MATLAB compiler executable.

4. Edit the line that starts with 'md.addSetting('matlabCompiler',' to indicate the name of your MATLAB Compiler executable (probably 'mcc').

5. Edit the line that starts with 'md.addSetting('matlabExecutable',' to indicate the name of your MATLAB executable (probably 'matlab').

6. For this configuration none of the other settings are used so they can stay as they are.

Now go into the SineSim directory and edit one line in the file called **SineSimSet_SS01.m**: Change this line (Line 26):

config.addMachine(MachineType.MYSERVER01, 4, 'YourWorkDirectoryHere');

so that 'YourWorkDirectoryHere' is changed to the full path of a new directory on your machine that will become the 'remote' work directory. For example, my work directory is typically something like '/home/David.Stockton/NMWork', giving a Line 26 that looks like

config.addMachine(MachineType.MYSERVER01, 4, '/home/David.Stockton/NMWork');

CAUTION! The work directory will be written to and cleared without confirmation, so it must be totally independent of other directories.

Run: Change to the SineSim directory within MATLAB and run the script **SineSimSet_SS01.m**. Results will be seen in the 'SimResults_date/time' directory on the host machine as specified in the NeuroManager constructor (see Section 3.12 and Figure 15 below). A typical SineSim plot can be seen in Figure 19. Log entries are sent to file and also to the MATLAB command window. It's also instructive to monitor file and directory changes via a separate monitor (terminal window) on the remote.

Note: Sometimes a Java XPCMessageLoop exception appears which is outside of NeuroManager; ignore it and the application will continue in a few seconds.

Examining the results will show you a subdirectory called 'SineSimSet' which is the name of the simset; within that is a subdirectory for each simulation specified in the sim specification file **SineSimSpec.txt**. Examining the tiff file in one of those locations should show you a figure displaying a sine wave; this indicates proper execution of that simulation. Note that SineSimRun04 fails due to a deliberate error in the parameter specification and will have no tiff file; the others should succeed and have it.

By changing the number of simulators in Line 26 to more than one, you will run that number of simulations simultaneously (depending on machine scheduling, of course); watching the log will show you this. In the above example, four simulators were running simultaneously. Using '0' means that that machine will not be used.

2.2.2 What just happened in the SSO1 example?

So what just happened? We'll give a quick, highly abbreviated sketch of the SSO1 example. NeuroManager (NM) went to the work directory, cleaned it up, then constructed four directories, one for each simulator, so that their work space is independent. Each location is an aspect of a Simulator object, four of which make up the Simulator Pool. NM then loaded the remote m-files into that structure and compiled them to produce a MATLAB executable, the 'simulator', which it then copied into each of the simulator directories (not necessary here, but is suitable for other situations). NM pulls in the lines of SineSimSpec.txt to produce a SimSet composed of one simulation per SIMDEF line. Each line specifies a simulation ID and a set of up to ten input parameters for that simulation. Then, for each simulation in the SimSet, NM ran the executable on an available simulator with those input parameters. The results go into an output directory located in that simulator's directory structure, which NM subsequently zips and downloads. When a simulator is free, then NM places the next available simulation on it, until the entire SimSet has been processed.

The webpage that you saw updating in MATLAB shows each simulation ID, state, times, and what simulator/machine that it ran on, together with its result.

The **userSimulation.m** seen in the SineSim directory is where the action is on the remote. This file is uploaded by NM and becomes part of the MATLAB executable. The input parameters are passed into that function as well as the directories to use for input and output; the sine waves are produced using those parameters, plotted, and the figures saved into the output directory for download. Paired with **userSimulation.m** is the host–side **SimSineSim.m** file which defines the ***SimSineSim*** class.

The following section shows you how togo beyond your single machine to include remote servers and clusters that you can SSH into. If you'd rather see Neuron simulations running directly on your single machine, skip ahead to Section 2.6.

2.3 Extending your Machine Set with the SineSim simulator

Since NeuroManager can handle multiple heterogeneous computational resources simultaneously, you will quickly want to move past the single machine option to extend your Machine Set to include other servers and clusters. Here we move the remote to a separate machine, then allow it to be a cluster, then compose a Machine Set out of more than one remote. We use the SineSim simulator for simplicity.

2.3.1 SineSim SS03: Configure/run using dual key authentication

In this step we separate the host and remote, and start using communications protected by SSH with dual key authentication. To do this, we must configure the data for the remote, and set up and use dual keys. Then we test communications before running the SineSim simulator once again.

Configure:

1. Choose a remote machine other than your host. It must be a machine that you can log into and SSH into, that has MATLAB with Compiler Toolbox, and that does not make use of a job submission manager (i.e., you run simulations directly from the command line). We will call this machine myServer02 and configure myServer02 files to match it.

2. Edit the **createMyServer02Data.m** file for proper username, id (obtained as above), ipAddress, and MATLAB locations (ignore the xCompDir entry for now). Ensure that Password is empty (''), so that dual key will be used. Placing a password in the string will override dual key for that machine and send the unencrypted password over the network (undesirable, but sometime necessary).

3. If you are on a Windows host, install Putty as detailed in Section 2.12.1. Ensure that Pageant has also installed.

4. Set up dual key communications with that machine as described below in Section 2.12.2. Ensure that you can SSH into Server02 using dual key but without using NeuroManager.

5. Place your private key(s) into the LocalMachine directory, then edit **myNMStaticData.m** with the key names as indicated by the comments in that file.

6. Test NeuroManager communications using the Testing Machine Communications section below (Section 2.12.3). This section tests all the types of communications that NeuroManager uses, for each machine in the MachineSetConfig (unless number of simulators == 0). Troubleshoot as necessary until communications pass for myServer02.

7. Go to the file **SineSimSet_SS03.m** and edit the work directory to match your work directory on myServer02.

Run: Change to the SineSim directory within MATLAB and run the file **SineSimSet_SS03.m**. The results will be placed in that same directory, and should be the same as obtained in the SingleMachine step. In this case, however, the simulations have been run on a separate machine, and all communications have been done with SSH using dual key authentication.

If you will be using only the one myServer02, then you may skip to the SS08 notifications section, or simply go on to the Neuron examples (Section 2.7.2). If you will only be running one or more single servers, not clusters, then skip to SS05.

2.3.2 SineSim SS04: Using a remote cluster

Configuring and running a remote cluster is more involved than a single server: ▷ A cluster may involve separate machines for job submission and file system, ▷ a cluster will generally use a job submission utility and require a job file to run a simulation, ▷ a cluster may have restrictions on where MATLAB or other

compilation can occur, ▷ a cluster may involve cluster subsets called queues, and ▷ a cluster may have load–balancing login servers. Configuring NeuroManager to work with a new cluster type can be a little involved, but is well worth the effort (as opposed to using cluster-native multiple job submission) because of NeuroManager's automation of the overall workflow. We address each in turn just below. Typically you will eliminate lots of file transfers and the like, hopefully for good! Moreover, clusters tend to handle more simulators than do normal servers, so your throughput can be higher just by changing the number of simulators per machine.

If you have a Sun Grid Engine cluster, then make use of the mySGECluster01 files; if you have a SLURM cluster, then make use of the mySLURMCluster01 files. If you have a different cluster type, you will need to study one of those two to learn how they work, then create your own cluster type. This is a profitable thing to do, however it would be helpful to get experience first with an SGE or SLURM cluster if possible. See Section 2.3.5 below.

We proceed assuming you are working with an SGE cluster. See also Section 2.3.6.

Separate machines: Our local Sun Grid Engine cluster is organized so that users move files to and from one specific machine with a specific IP address (the 'filesystem machine'), where executables cannot be run. The user must submit jobs from another machine at a different IP address (the 'job submission machine'); the filesystems are shared. NeuroManager handles this by thinking of a remote machine as a combination of the two. See **createMySGECluster01Data.m** to see that each has its own username, ip address, and (optional) password. NeuroManager handles commands in a way that directs them to the proper machine. See also Figure 2.

Job submission utility: Job submission systems like Sun Grid Engine involve creation of a job file and its submission using a command such as `qsub`. NeuroManager writes the job files automatically and submits them for you; however our implementation of that is localized and you may need to alter it. See, for example, `preRunCreateJobFile()` in **QSUBMachine.m**.

Compilation restrictions: NeuroManager does MATLAB compilation remotely; on our SGE cluster this is done on the head node (our job submission machine). Cluster administrators gave permission to run mcc on the head node since the execution time is relatively short. If our SGE cluster had a short queue, we could use that instead. You will have to adapt to your own cluster administration policies. Neuron model compilation is similar and will be dealt with below.

Queues: SGE and SLURM use queues. NeuroManager treats a queue as a separate machine and handles a queue using a subclass; for example, the combination of **mySGEClusterQueueData.m** and **MachineMySGEClus01Queue01.m** allow us to specify the QUEUE01 queue on the SGE cluster. You will need to create a similar subclass and edit **mySGEClusterQueueData.m** to choose a queue for your system. In addition, **MachineType.m** will need to be modified with your new machine. See also Section 2.11.4.

Login servers: Larger clusters such as Stampede have several login servers which work in a load–balancing fashion. Although the user accesses 'stampede.tacc.utexas.edu', the actual IP address may vary. NeuroManager does not handle this situation; it deals only with numerical IP addresses. Our experience so far is that this almost never causes issues. You will need to determine a numerical IP address for your cluster's filesystem and jobsubmission machines.

The **SineSimSet_SS04.m** script looks much like the SS03 script for good reason: NeuroManager was designed to abstract machines. There are two things to note about the SS04 script. First, the machine type is MYSGECLUSTER01QUEUE01, which is a combination of the machine name and the queue name (see Section 3.9). Second, the working directory is a subdirectory named after the queue. Although this is not required, it is highly desired in the case that the user wishes to use two separate queues at the same time (as, of course, separate machines), since it keeps the different queues in labeled, nonoverlapping directories.

Given that the cluster is running the **SineSimSet_SS04.m** script should give the same results as the previous two examples.

2.3.3 SineSim SS05: Using a heterogeneous Machine Set

The user is free to add more than one remote to create a multiple machine 'Machine Set'. As long as they work correctly as individual remotes, they should work together as part of the Machine Set — although

the user should take care to ensure that machine names are unique (NeuroManager does not check). Each simulator will be placed into a machine–independent Simulator Pool and simulations will placed on them in turn, in the order in which the machine was added to the MachineSetConfig. Using zero for the number of simulators on a machine means that machine will not be used, which facilitates the approach seen in **SineSimSet_SS05.m**. One gathers experience with the machines in question and configures the machines and number of simulators on each to achieve substantial gains in throughput without huge code.

Use the two machines configured above to form a Machine Set and run **SineSimSet_SS05.m** with four simulators on each; the results should be the same, but the user can see in the log and in the titles of the figures where they were produced.

2.3.4 SineSim SS08: Configuring notifications

User data for sending text and email notifications are entered in **myNMStaticData.m**. Overall notifications are controlled in the ***NeuroManager*** constructor options (NotificationsType $\in$ {NONE, OFF, BOTH, EMAIL, TEXT}). More detailed notifications for individual simulations are specified by using 'SIMDEFN' instead of 'SIMDEF'.

Script **SineSimSet_SS08** shows the use of notifications after setting the values in **myNMStaticData.m**. Supported carriers can be seen in **Core/@NotificationSet/sendTextMsg.m**. Additional carriers and smtp data are easily found with an internet search. Note: this carrier data is subject to change and other complications. If you have problems, try doing the email–to–text–message thing outside of NeuroManager; once that works, use the m–file alone with a little hacking; when that works use NeuroManager.

See also Section 3.12 for automatically attaching a plot or other file to the email and/or text message upon simulation completion.

2.3.5 Configuring additional machine resources

Adding additional machines will be simple or complex, depending on the machine's differences from those supplied in the code. Adding an SGE or SLURM cluster can be simple, like that of the first server above — just inserting a username, a couple of IP addresses, and setting a few directory locations in the appropriate **create....m** file. Adding additional machines or a new cluster type will be a little more complex, but we hope that most of the ground in this area has been broken by the current software and that extensions can be mostly done by analogy. By determining the simulation submission workflow for a new machine and comparing that stage–by–stage with the existing implementation, it should become apparent where and how the new machine fits into NeuroManager operation.

Creating a new, additional SGE machine, say 'mySGECluster02', requires creation of the ***MachineMySGECluster02*** class as a subclass of ***MachineSGECluster***, and the ***MachineMySGECluster02Test*** class as a subclass of ***MachineCommsTest***. At least one queue subclass ***MachineMySGECluster02Queue*** class is necessary. The machine will have to be included in the ***MachineType*** class, and the user will need to form the files **createMySGECluster02Data.m** and **mySGECluster02QueueData.m** for use by the class constructor. The user will need to publish the public key to the new machine in the same way as above. If administrative policies are different than those presented in mySGECluster01, the new classes may have to override methods in the superclasses — this will be most likely be true for the job file formation at least (see ***MachineMySGECluster01*** for where to override ***QSUBMachine***'s job file creation method `preRunCreateJobFile()`). For the most part, however, we anticipate that these operations will be mostly copy/paste/tweak — it should be relatively straightforward to add a new machine.

The same comments apply to additional SLURM machines. We expect that other cluster batch systems such as Moab, Argent Job Scheduler, Univa Grid Engine, Condor, and Platform LSF (and others listed at http://en.wikipedia.org/wiki/Job_scheduler) would be suitable for NeuroManager, but we have not investigated these other systems.

Side Note: Creating full–fledged general SGE or SLURM solutions is beyond the scope of this project, or at least this version. The user should feel free to modify Core files as needed to achieve desired operation; or better yet, override the Core file with a new file and careful use of the MATLAB path.

2.3.6 Our experience with our local Sun Grid Engine cluster

Interacting with our local Sun Grid Engine (SGE) cluster required filesystem operations to be done on one physical machine and job submission to be done on another with a different IP address. In order to deal with this distribution of work we designed every NeuroManager machine to work with two IP addresses: a 'filesystem' machine and a 'jobsubmission' machine. Figure 2 shows an example of this split. Machines that don't have this split just duplicate the IP address to both (for example, see the supplied **createMyServer01Data.m** file). We split all machine commands into those two types so that host software would know which IP address to use (see the ***CommandType*** class). Second, the SGE Cluster uses the Sun Grid Engine, so we created the ***QSUBMachine*** subclass and job file handling. Also the SGE uses queues, each with its own characteristics; the subclassing approach worked very well for this. Third, users are required to run jobs using `qsub` on the jobsubmission machine; however for us this resulted in huge delays with the need to compile both MATLAB and model files before actually running the simulation itself; so we were in effect waiting for job submission three times to do one simulation. We solved this by working with the administrators to find an alternate way to do the MATLAB compilations. Some machines may have a short–job queue available that would work well for MATLAB and model file compilation. We also bundled Neuron model file compilation in with the simulation so that it was run as part of a single job submission. Since TACC compilation rules are basically the opposite than the ones described for our local SGE, we developed the H/P/D split of the *PreRun Model Processing Stage*.

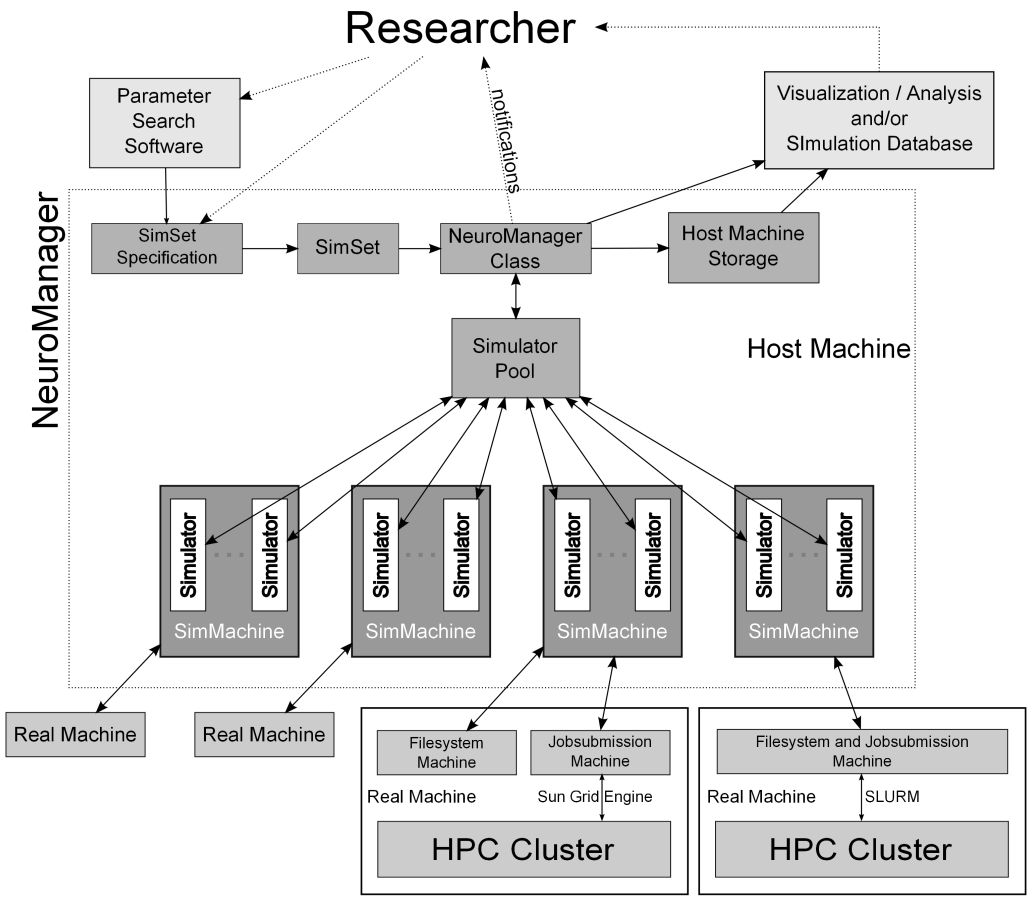

Figure 2: NeuroManager Block Diagram. Each machine has two components: a filesystem machine and a jobsubmission machine, which can be the same

2.3.7 Our experience with TACC's Stampede (a SLURM cluster)

Our MATLAB licenses didn't involve TACC, so we developed NeuroManager's cross–compilation facility that makes use of a separate external compilation machine where a license was available and transfers the MATLAB executable files to the Stampede cluster. Section 2.11.6 has more details on this process.

We installed the appropriate MCR in our work directories on the SLURM cluster. TACC's policy for the NORMAL queue doesn't allow compilation on work nodes, so we use the PreSubmission Phase of the *PreRun Model Processing Stage* to do Neuron model file compilation.

Installing Neuron on Stampede in user directories was a little tricky and the details are beyond the scope of this document. The Stampede and CBI staff were very helpful. It's possible that by now Neuron is available on Stampede as a module which would likely make things easier; then the code would load the Neuron module in the same way it loads the matlab (see **SLURMMachine.m**) and python (see **createStampedeClusterData.m**) modules now.

2.4 Automatically generating SimSets

The SineSim SS09 example demonstrates two different ways to use of a script to produce a SimSet automatically, the use of multiple SimSets for a single Machine Set, and a simple use of a random number generator in a SimSet.

In this example, we use the MATLAB RNG to produce frequency and duration. The first way to generate the SimSet is by producing the same kind of text file as used in the previous examples, then referencing it in the `runFromFile()` method. The second way is to create a SimSpec directly using the `addTokenSet()` method.

Note that, since we create multiple SimSets that use the same MachineSet, the overhead of constructing the Simulators is spread over all the SimSets. At the same time, the risk of having a remote failure due to resource problems is limited to one SimSet rather than all of the Simulations.

2.5 Running your own MATLAB–only code

The SineSim example is a good way to start running your own MATLAB code within NeuroManager, using a copy/paste/modify approach. Your code needs to be wrapped in a new simulator we'll call YourSim. We have to create a new subclass defining the simulator, together with its companion userSimulation function; new simulator type; new script; and new simspec file — but it's easy! Here's what to do:

1. Set up your own code as a self-contained function, perhaps using a master function calling your own code; the input parameters of which will be what's listed in the simspec file. You will call this function from userSimulation(). A good example is the `createMySineWave()` found in the SineSim **userSimulation.m** file. Test this new function to ensure it works before you try to use it in NeuroManager.

2. In the installation directory, create a new directory 'YourSim' where your new simulator will go.

3. Copy SineSim's **userSimulation.m** into YourSim, and modify it to match your function's needs. Any parameters you wish to pass in will come in as strings via varargin in the same order you list them in the SimSpec file. Any files you want to keep need to end up in 'outdir' before `userSimulation()` exits.

4. Copy **SimSineSim.m** into YourSim, rename it 'SimYourSim.m', and edit it to replace all 'SineSim' with 'YourSim'. In addition, comment out the two lines in the `defineSimulationInputDataFiles()` function. If your function calls other functions in other files, you will need to add those files to the YourSim directory, and also put their names in the 'addlcustfilelist' variable seen in **SimSineSim.m**, like this:

   addlcustfilelist = {'file1', 'file2'};

   NM will upload those files for you and compile them into the executable.

5. Open **SimType.m** and make a new line within it, emulating the one that has 'SIM_SINESIM in it. This makes a new simulator type called SIM_YOURSIM. Alternately, you can make a copy of **SimType.m** and put it in your YourSim directory, and put your new line

   SIM_YOURSIM (@Sim_YourSim)

   as the only line in the enumeration section (except the UNASSIGNED line, which must be present); this file will override the Core file since this custom directory will be above Core in the MATLAB Path when your simulation runs.

6. Copy **SineSimSet_SS01.m** (or whichever SS0X you prefer) into YourSim, rename it 'YourSimSet.m', then edit Part II with 'YourSim', and in Part VI replace 'SIM_SINESIM' with 'SIM_YOURSIM' — the type you developed when you edited **SimType.m**. This means the simulators constructed on the remotes will be of your new type. Finally, replace 'SineSimSpec.txt' in Part VII with 'YourSimSpec.txt'.

7. Copy **SineSimSpec.txt** into YourSim and rename it 'YourSimSpec.txt'. Then edit its contents to utilize YOURSIM instead of SINESIM and YourSim instead of SineSim, then edit the parameters to match your needs.

2.6 Quickstart with Neuron

As a companion to the SineSim Quickstart example, we have constructed two simple Neuron examples. The first is called NeuronTutorialA, which runs and plots The University of Edinburgh's hoc–file–only Neuron tutorial part A seen at http://www.anc.ed.ac.uk/school/neuron/. The second Quickstart simple Neuron example is called SimpleSpike01, which uses both mod and hoc files for a soma–only neuron. Here we describe how to run the SimpleSpike01 example quickly using the SingleMachine configuration; the NeuronTutorialA example is trivially similar and will not be explained here. Later, in Section 2.7.2 we will extend the SimpleSpike example in three powerful ways, then move to two versions of a hoc/Python Neuron simulator based on a model from ModelDB. Here, though, we emulate the 'Quickstart with the SineSim Simulator' section (Section 2.2) above to run the SimpleSpike01 example. This example does not have input parameters, but in Section 2.7.2 we will go into that topic in excruciating detail and variety in order to highlight the power and flexibility of NeuroManager. We recommend using Neuron 7.3 which is what we used.

Configuration:

1. First ensure that the MATLAB and ID settings are correctly done as in Section 2.2.

2. Still within **createMyServer01Data.m**, edit the Neuron settings as indicated to match your system. The 'NeuronDir' acts as a prefix for the other settings, and may be the only setting that needs to change. We recommend also editing the 'PythonPath' line. The two lines 'PythonEnvAddlLines' and 'EnvAddlLines' should not need to be edited unless your installation requires.

Now go into the NeurSim/SimpleSpike01 directory and edit the file called **SimpleSpike01SimSet.m**, which assumes you are using SingleMachine configuration:

1. Change this line :

   config.AddMachine(MachineType.MYSERVER01, 4, 'YourWorkDirectoryHere');

   so that 'YourWorkDirectoryHere' is changed to the same work directory that you used for the SineSim example. **CAUTION! The work directory will be written to and cleared without confirmation, so ensure this edit is done correctly.**

Run: Change to the NeurSim/SimpleSpike01 directory within MATLAB and run the script **SimpleSpike01SimSet.m**. Results will be seen in the 'SimResults_date/time' directory on the host machine within the Results directory specified in the NeuroManager constructor, including a plot that should look like Figure 3. The NeuronTutorialA example and subsequent SimpleSpike examples also plot the data and download to the host automatically. See Section 2.7.1 for the subsequent SimpleSpike examples.

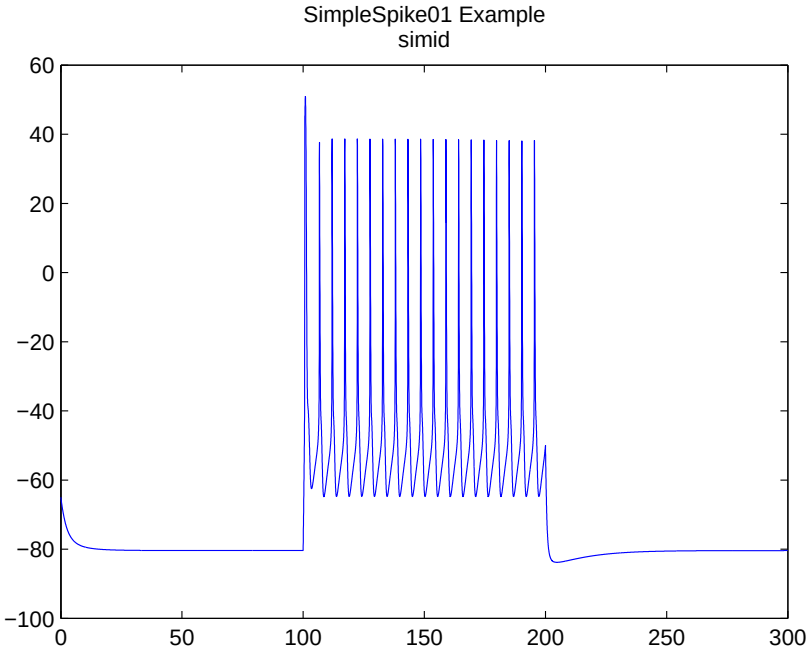

Figure 3: SingleSpike01 Plot of the dataS0R11.dat file

2.7 Extending your simulations beyond simple submission

2.7.1 Six advanced example simulators

We have developed six examples for demonstrating advanced manipulation of Neuron simulations. The concepts also hold true with other simulator types, such as MCell. In the next section (Section 2.7.2, Discussion of example simulators) we discuss the SimpleSpike, Miyasho2001, and KhStudy simulators from a design point of view. To run the example simulators requires a Neuron installation on the remote machine, together with proper entry of machine data in the **createMyXXXData.m** file (or its renamed equivalent), as already accomplished above in Section 2.6. The SimpleSpike simulators do not use Python but there may be some dependencies based on the Neuron installation. The Miyasho2001 and KhStudy simulators do use Python. For each simulator, there is a script that runs simulations on that simulator, with a name similar to **SimpleSpike02ASimSet.m**. Like the CommsTest and SimSpec examples, you will have to adapt each script slightly to your file and machine configuration.

The six example simulators are as follows (see also Table 1):

1. SimpleSpike01: Runs a hoc–file–only simple simulation with no input parameters — this is the example used in the Quickstart

2. SimpleSpike02A: Runs the same simple simulation but handles input parameters by creating a hoc file on the fly on the host within the subclass

3. SimpleSpike02B: Runs the same simple simulation but handles input parameters by creating a hoc file on the fly on the remote within **userSimulation.m**

4. SimpleSpike02C: Runs the same simple simulation but handles input parameters by creating a mod file on the fly on the host within the subclass

5. Miyasho2001: Runs the Miyasho2001 ModelDB model using a hoc/Python combination and manipulates a few input parameters that deal with model external parameters

6. KhStudy: Runs the Miyasho2001 ModelDB model using a hoc/Python combination and manipulates the distribution of the Kh ion channel by creating a hoc file on the fly on the host and merging with others on the remote

2.7.2 Discussion of example simulators

In this section we get off the main highway and onto dirt track as we talk about nitty gritty programming to extend the Neuron-based simulator in powerful ways. We deal with modifying hoc and/or mod files on a simulation–by–simulation basis in response to text entries in the simspec file; we also deal with on–the–fly writing of Python files. Primarily we point out salient points of each of the example simulators listed in Section 2.7 and rely on the reader to peruse the files in question. The key ideas to keep in mind are: • the submission workflow is the basis for organization (see the paper and Supplemental Materials), • the simulator class file forms a pair with the **userSimulation.m** file, and • simulation–specific activities that need to be done before a simulation runs can be done on the host or on the remote. These examples show interplay between these factors, and the advantages of one approach over another.

Because Neuron is a simulator that uses model files, we subclass the ***ModelFileSim*** abstract class, which itself takes all the features of ***Simulator*** class and adds facilities for uploading of model files. Our new subclass is then called ***SimNeuron*** which implements the host–side aspects of the Neuron–specific actions demanded by the Workflow Stages — primarily model file copying and modification, and the creation of a shell file for mod file compilation and its execution or addition to the job file (please refer to Figure 13). The `preRunModelProcPhaseH()` method is where the model file compilation script file is constructed, uploaded, and made executable. Note that ***SimNeuron*** is an abstract class because it declares two abstract methods that can be used for modifying hoc and mod files on the fly; these methods must be made concrete in any subclass of ***SimNeuron***. In the ***SimNeurPurkinjeMiyasho2001*** class, for example, we use one of them to create the biomechanism hoc file which is then combined with the morphology hoc file on the remote by the corresponding `userSimulation()`.

For each of our six example Neuron–based simulators, we create a new subclass of the ***SimNeuron*** class, which will be in play on the host, and a new companion **userSimulation()** for the remote, which in addition to running the simulation by calling Neuron itself, also allows the user to preprocess and postprocess anything on the remote based on the simulation's input parameters (the subclass also has access to the simulation's input parameters, as the reader will see). The first four simulators are simple spiking simulators with just a few hoc and mod files; all are called 'SimpleSpike'. The fifth simulator is called 'Miyasho2001' because it essentially uses the data of the the ModelDB Miyasho2001 model (Miyasho et al, 2001). The final simulator is called 'KhStudy' because it modifies the distribution of the Kh channel in that same Miyasho2001 model.

The *SimpleSpike01* simulator doesn't modify hoc or mod files, so the ***SimNeuronSimpleSpike01*** just provides the lists of each to its superclass ***SimNeuron***. Its companion **userSimulation.m** simply calls `runHocOnlySimulation()`, which runs the simulation using **nrniv** and transfers the resulting datafile to the output directory. This simplicity, however, doesn't allow any of the simulation parameters in **parameters.hoc** to change.

The remaining SimpleSpike simulators allow simulation–by–simulation control of three input parameters: input current level, leak conductance, and duration. The actual simulation values are taken from the SimSpec. The three examples accomplish the same thing in three different ways.

The *SimpleSpike02A* simulator modifies **parameters.hoc** by constructing it on the host for each simulation during the *Pre Run Model Processing Host (H) Stage* discussed above and then uploading it to the

simulation's input directory (see **SimNeuronSimpleSpike02A.m**). A major advantage of this approach is that the operation is part of the simulator class. There is no effective difference in the `userSimulation()` functions of this example and that of SimpleSpike01 (though we have added a visualization component to *SimpleSpike02A*).

The *SimpleSpike02B* simulator modifies **parameters.hoc** by constructing it on the remote for each simulation. The construction is done in **userSimulation.m** using the parameter values passed in through *varargin*. A minor advantage of this approach is that the file construction is done in parallel on the remote. A major disadvantage is that the construction is disconnected from the formal simulator definition and is not part of the simulator class hierarchy. For that reason, this is not the preferred approach.

The *SimpleSpike02C* simulator shows the power of the NeuroManager approach by actually creating a new *mod file* for each simulation, based on input parameter values. To keep our example simple and consistent with the 02A and 02B examples, we modify only the default Leak_gl value, although mod files and their manipulations can be more complex. So we are modifying the default value to match the desired value as seen in the input parameter vector. We modify the **channelproperties.hoc** file by hand so that the hoc file relies on the default Leak_gl setting. Then we construct the new mod file during the *Pre Run Model Processing Host (H) Stage* discussed above and then upload it to the simulation's input directory (see **SimNeuronSimpleSpike02C.m**). The new mod file will be compiled into the Neuron dynamic library **libnrnmech.so** along with the other mod files. Figure 4 shows an example SimpleSpike02C plot, in which the simid and some of the input parameters have been used in the plot's title.

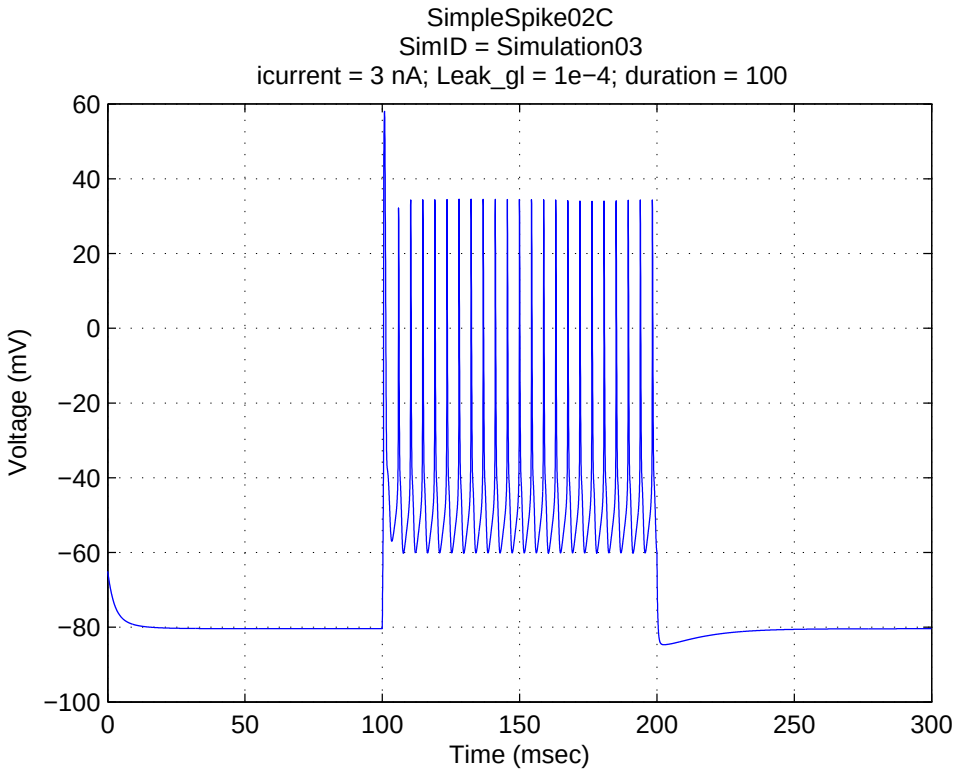

Figure 4: Example SimpleSpike02C Output

The *Miyasho2001* simulator is a NeuroManager-ization of the published model and serves to illustrate one variation of Hoc + Python handling within NeuroManager. It uses the hoc morphology and biomechanism definitions and a Python implementation of the actual simulation mechanism. The input parameters that are available to change from simulation to simulation are stimulation current, initial membrane

voltage, stimulation delay, stimulation duration, timestep, stoptime, and record interval. The class is the ***SimNeurPurkinjeMiyasho2001*** class, a subclass of the ***SimNeuron*** class; this new class specifies the specific Miyasho ***.mod** files and a modified version of their **purkinje.hoc** file (modified to remove all GUI code). We add **PySim.py**, which is a Python file that imports the neuron module, pulls in the hoc file (the morphology and biomech hoc files are combined on the remote), and runs the simulation with the given input parameters. In the companion **userSimulation.m**, we use the remote utility function called `runPythonSimulation()` that itself constructs a Python harness, or wrapper, that allows the passing in of all the simulation's input parameters. `runPythonSimulation()` also creates the **nrnivsh.sh** shell file alluded to in Section 2.7.5 which actually calls the Neuron simulator itself. So the reader can see that the class file/userSimulation pair form an extended simulator with Neuron at its core.

Figure 5 was developed as part of a parameter search run of 20 simulations using the ***SimNeurPurkinjeMiyasho2001*** simulator within NeuroManager (see **Miyasho2001StraightSimSpec.txt**) and has the approximate appearance of Figure 2B bottom of the Miyasho paper Miyasho et al (2001), as claimed in their **readme** file at `http://senselab.med.yale.edu/modeldb/showmodel.asp?model=17664`.

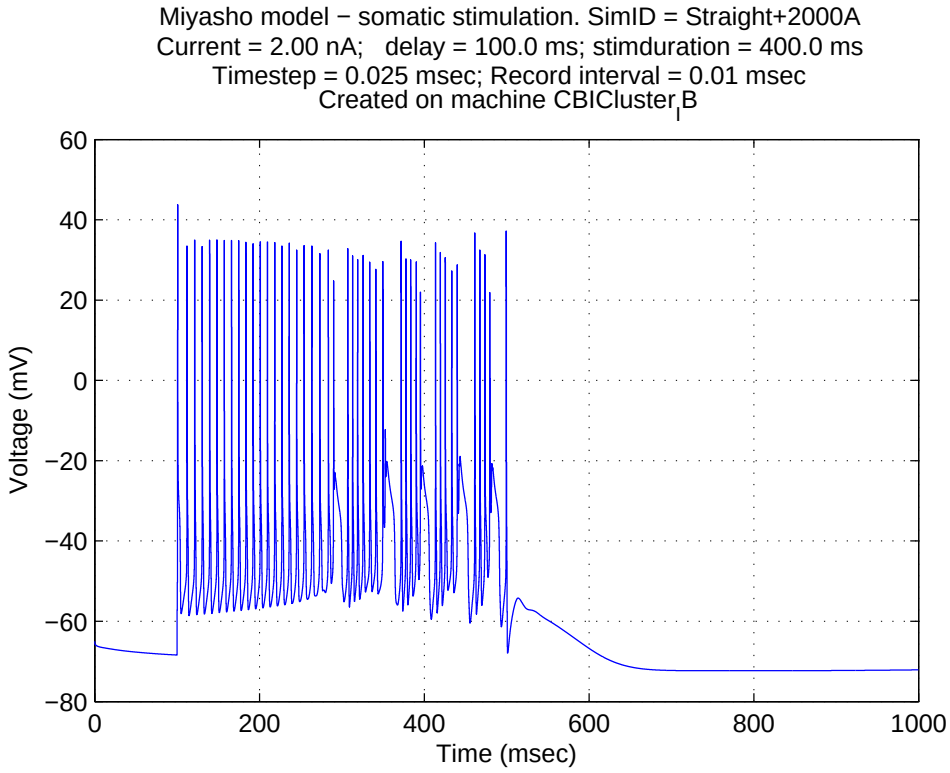

Figure 5: Example Miyasho2001 Plot

The *KhStudy* simulator adds one more more complexity to the *Miyasho2001* simulator. We have subclassed ***SimNeurPurkinjeMiyasho2001*** to form ***SimNeurPurkinjeMiyasho2001Kh***. The latter overrides the methods that insert the Kh ion channel so that they are SimSpec–dependent and specifies a further–modified version of biomech file **purkinje.hoc** which also removes all Kh channel insertions. Together with the **userSimulation.m** file, which acts on the remote machine, these files work together to form a simulator that uses the Miyasho cell configuration but allows Kh concentration on soma, smooth dendrites and spiny dendrites to be free parameters.

It's not always necessary to cut out gui code from one's hoc files. Work we have done in our lab on a Purkinje cell modeling project used NeuroManager to run Neuron hoc-only simulations by cre-

ating a file called 'parameters.hoc' within `userSimulation()`. A **runme.hoc** file was the input to `runHocOnlySimulation()`; it loaded first **nrngui.hoc**, then **parameters.hoc**, then the other hoc files. The advantage to this approach is that the same hoc files were runnable under the Neuron gui for debugging of the hoc files; then NeuroManager was able to run multiple longer simulations on multiple machines for maximum simulation throughput and greatly simplified researcher activity.

2.7.3 Neuron installation locations

The locations of all Neuron–specific directories and executables are different on different machines; since this is a machine–specific characteristic this is done through properties in the ***RMD*** class, which are set by the various `createMyXXXData()` functions. The machine constructors (`MachineStampede()`, for example) call those functions via the ***NeuronMachine*** superclass to get the data they contain.

2.7.4 Neuron mod file compilation

NeuroManager's approach is to compile mod files for every simulation. This allows mod file structure and settings to be part of the input parameters to each simulation and adds another dimension to Neuron's simulation power. This also allows simultaneous use of Neuron installations that are not identical, since the compilation takes place on each remote machine within its own installation of Neuron. Accordingly, handling and compilation of mod files are part of NeuroManager's implementation of Neuron simulation. The user's interaction with the mod files, however, is simplified, since all uploads, modifications, compilation, and mod file movements are automated.

The administration policies of the remote machine play a major role in the compilation of mod files, since compilation software is not always available on cluster nodes. In the case of our local SGE Cluster, compilation needed to be performed on a work node as a preliminary part of the simulation job itself. In the case of the Stampede Cluster, compilation needed to be performed from the user's working directory, and was not available on cluster nodes. We used a combination of three different stages of the workflow of the Pre Run Model Processing: PhaseH (**H**ost), PhaseP (**P**re Job submission), and PhaseD (**D**uring Job Submission). Note that mod file modifications cannot be made on the remote using `userSimulation()` because that function is not run until after mod file compilation is complete; so we do any model file modifications on the host and upload the resulting files for remote compilation.

2.7.5 The packaging of nrniv

We are using the double abstractions of object-oriented programming and workflow analysis, which gains us flexibility and extensibility, but tracing actual procedural steps through the code can be a little difficult. To help the reader understand a little better how a simulation is actually run, we discuss in this section how a Neuron-based Hoc/Python simulation is actually run on the remote. This is the most complex path currently employed by NeuroManager; other simulators can have a simpler process. To help the discussion, we refer to Figure 6, which shows a skeletal view of the major players.

The reader may also find it useful to refer to the Miyasho2001 or KhStudy examples for this discussion.

`nrniv`, the command to run Neuron, or the GUI gateway `nrngui`, are the commands most Neuron users are familiar with. NeuroManager doesn't do user interaction and thus doesn't use `nrngui`. In order to run `nrniv` with the proper environment, with the proper mod file library, the proper hoc/Python inputs, on a variety of platforms, based on just a couple of configuration files, we package the call to `nrniv` in layers. Additional layers are added as one proceeds down the figure, following the arrows.

The call to `nrniv` is performed within **nrnivsh.sh**, created at runtime within **createPythonNrnivsh.m**, which is run by `runPythonSimulation()`, which is called by the user in `userSimulation()`. **nrnivsh.sh** sets up the remote environment, imports the *.mod library **libnrnmech.so** from its simulation–dependent (and hidden) location, and sets up output redirection files so that runtime Neuron errors can be identified if necessary.

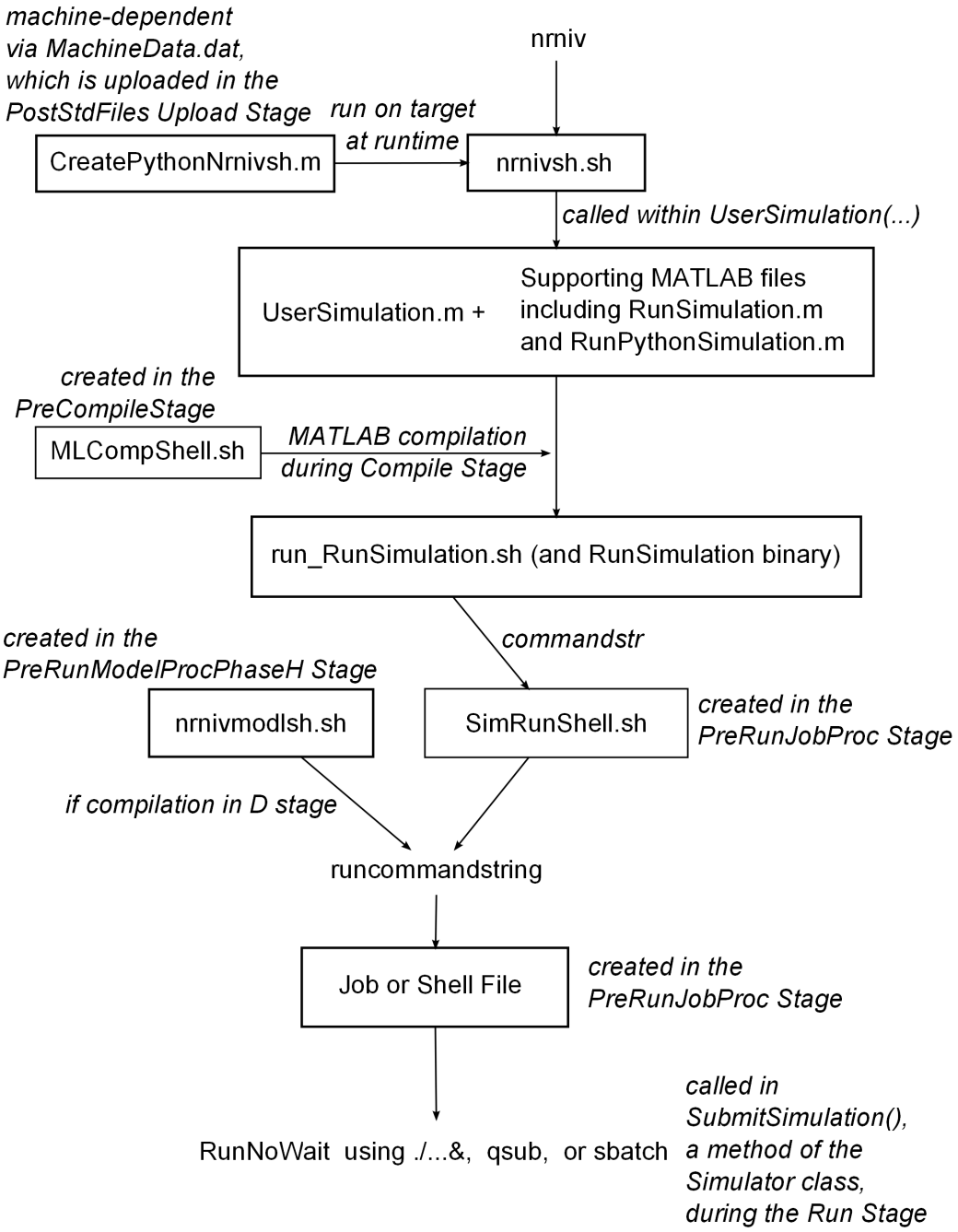

Figure 6: Layers involved in running a simulation on a simulator with Neuron at its core. Multiple shell layers provide for model compilation as well as the ability for simulations to fail gracefully and verbosely on a variety of platforms.

`userSimulation()` is itself called by Core file `runSimulation()`, which is basically the lead MATLAB function on the remote. **runSimulation.m** and all the other remote-side m-files are compiled into a MATLAB executable called **runSimulation** through a call to the MATLAB compiler. The MATLAB compiler also produces a shell file called **run_RunSimulation.sh**, which sets up the proper environment for the executable then runs it. The actual MATLAB compilation on the remote is run by a NeuroManager shell called **MLCompShell.sh**, which allows graceful failure and reporting to the host in the case of MATLAB compilation errors. **MLCompShell.sh** is created on the host in the *PreCompile Stage* within the ***MATLABCompileMachine*** class, and invoked during the *Compile Stage*.

In order to have graceful error detection and reporting during running of **RunSimulation**, we package the call to **run_RunSimulation.sh** in another shell script file called **SimRunShell.sh**, which is created during the *PreRunJobProc Stage* by the ***Simulator*** class. This shell also checks for model file compilation success as discussed immediately below.

The call to **run_RunSimulation.sh** is the **second** part of a command string called *commandstr* that is run in the **SimRunShell.sh** script just mentioned. The **first** part of *commandstr* is the optional model compilation discussed elsewhere. Briefly, Neuron ***.mod** files have to be compiled into a library before Neuron is run; in NeuroManager we do that every simulation to provide a more powerful tool with very little time cost. On some clusters, the compiler is not provided on work nodes (for example, Stampede at TACC). On other clusters (for example, our local SGE cluster), compilation is required to be on work nodes. In that case, the model compilation must be done during job submission, during *PreRun Model Processing Stage D*. The first part of *commandstr* is where this is done, through the running of a model file compilation shell file called **nrnivmodsh.sh**, which among other things contains the call to **nrnivmodl**, which Neuron users will recognize as the command–line mod file compiler. The **nrnivmodsh.sh** shell file not only sets up environment, but also allows for graceful error detection and reporting during mod file compilation, and is created on the host during *PreRun Model Processing Stage H*.

The complete *commandstr* is then packaged within a job file, if it is running on a cluster that requires a job file, or within a shell file, if it is running on a machine that doesn't require a job file. The job file is created during the *PreRunJobProc* stage by the ***QSubMachine*** or ***SLURMMachine*** class, or the shell file is created during the *PreRunJobProc* stage by the ***NoSubMachine*** class, whichever is appropriate. Users can override this operation to make their own job files if desired (which is likely, since job files tend to be installation–dependent) — see ***MachineMySGECluster01*** for a tip on where and how to do this.

Finally, the job file or shell file is run during the *Run Stage* via the `runNoWait()` method of the ***NoSubMachine***, ***QSubMachine***, or ***SLURMMachine*** class, whichever is appropriate. `runNoWait()` will make use of `./...&`, `qsub`, or `sbatch` to actually run the simulation job. `runNoWait()` also does output redirection to files that can be used for debugging. `runNoWait()` is called by the `submitSimulation()` method of the ***Simulator*** class.

2.8 The NMSessionML examples

The three NMSessionML examples listed in Table 1 can be run from the emulated directories (SineSim03, SimpleSpike02A, and KhStudy). The scripts listed in the table are run under the MATLAB environment and automatically produce the SimSpec and script files, then run the script. The reader will want to modify the XML file in each case to local circumstances as already done with the emulated examples.

2.9 The NeuroML example

The SimpleSpike03 example shows use of the NeuroML input format. The specification of the NaF ion channel is done using the NeuroML version from `neuroml.org` called **NaF_Chan.xml**. We load the file into memory using the MATLAB `xmlread()` function, then use XPath expressions to locate and modify each specified parameter. We also comment in metadata that the file was modified, and how. In this example we have pulled eight parameters out to use as free parameters that are specified in the nine Input Parameter Vectors seen in the SimSpec file. Once the file has been modified, we write it back out to the scratch directory, transform it to mod file format using the stylesheet found at `https://neuroml.org/neuron_tools`,

then upload it to the remote. In addition, the modified files are stored in the Simulation Results directory for provenance. All of the work was done in the ***SimNeuronSimpleSpike03*** subclass with no alterations to other code. We use the Saxon Home Edition XML software to do the transformation, but there are a number of other possibilities.

2.10 Troubleshooting facilities within NeuroManager

Debugging problems on any remote system can be very challenging and time–consuming. For minimizing debugging on NeuroManager, the first rule of thumb is to debug as much code as possible in a way that doesn't involve remote operation. In addition, a thorough experience with doing simulations on the remote system without the use of NeuroManager is invaluable. Running a simulation set with extremely short simulation time (say, 1–5 min per simulation) and a minimal machine set can help minimize time lost to misconfiguration issues. Getting to know the dynamics of NeuroManager's remote footprint is also quite valuable (see Section 16).

For the most part, faults and errors that occur before start of simulation submission will result in immediate halt of the program with (hopefully) informative messages.

Most faults and errors that occur remotely after start of simulation submission will result in a graceful simulation failure, often with hints about where to look for information, and NeuroManager will continue running other simulations. NeuroManager collects and downloads standard out and error streams from the MATLAB compiler, from model compilation, and simulation job submission and run commands, and those files are placed in the simulation's directory on the host (list just below). In addition, within `userSimulation()` the user can copy or move any runtime files to the simulation's remote output directory for subsequent download and analysis.

The **SimulationResults.txt** file which is created automatically on the remote and downloaded to the simulation directory on the host often will have a useful error message. Also useful are redirection files that appear in the simulation directory on the host after download, including:

- **stdout.txt** and **error.txt**: the stdout and stderr of the running of the **run_runSimulation.sh** script from **SimRunShell.sh**.
- **nrniverror.txt** and **nrnivoutput.txt** from the execution of **nrniv**, the Neuron simulator itself.
- **nrnivmodlcompile.txt** and **stderrnrnivmodlcompile.txt** from the running of **nrnivmodl** that compiles the Neuron model files.
- **[simid]_output.log%[job#]**: the output of the job submission command as produced by the job submission process and specified in the job file
- **[simid]_error.log%[job#]**: the error stream of the job submission command as produced by the job submission process and specified in the job file
- **stdout[simid].txt**: the UNIX output stream from the job submission command. The job ID can be found in this file, except for NoSubMachines.
- **stderr[simid].txt**: the UNIX error stream from the job submission command
- **dwnldstdout.txt**: the stdout stream from the scp or pscp command used for downloading simulation output files to the host.
- **dwnldstderr.txt**: the stderr stream from the scp or pscp command used for downloading simulation output files to the host. Note: may contain the user's password if the 'exp_internal 1' line of the expect file is uncommented.

2.11 Advanced NeuroManager operation

2.11.1 Running multiple simsets and multiple machine configurations in one NeuroManager session

It's possible to run multiple SimSets serially with a single machine configuration (see Figure 7). This saves machine setup/compile time and may help the user organize results. There are two caveats. The first is that a given SimSet will finish before the next begins. Highest throughput is achieved instead by consolidating the SimSets together in one. The second caveat is that all SimSets run on a given machine configuration must have the same SimType because the MachineSet is created with a specific SimType (see the `constructMachineSet()` function).

It's also possible to change machine configurations on the fly if no SimSets are running. Invoking the NeuroManager method `removeMachineSet()` allows the construction of a new one (with a possibly different SimType) with the `constructMachineSet()` method, without the need to run the NeuroManager constructor again (Figure 8).

```
...
nm = NeuroManager(SMDirectorySet, SMAuthData, UserData);

config = MachineSetConfig();
config.AddMachine(MachineType.MYUNIXMACHINE01,  2,...
                              '/home/username/MyWorkDirOn01');
config.AddMachine(MachineType.MYUNIXMACHINE02,  3,...
                              '/home/username/MyWorkDirOn02');

nm.ConstructMachineSet(SimType.SIM_SINESIM, config);

result = nm.RunFromFile('SineSimSpec.txt');
result = nm.RunFromFile('SineSimSpec2.txt');

nm.Shutdown();
```

Figure 7: Running multiple SimSets serially with a single machine configuration

2.11.2 Running multiple parallel NeuroManager sessions

It's also possible to run multiple SimSets of different types in parallel by starting separate MATLAB sessions at least a couple of seconds apart. This works on both Windows and UNIX hosts. On the host side, the SimResults files are tagged with time to the second; so host side directories are not in conflict. On the remote side, one must use non-overlapping work directories and non-overlapping cross-compiling directories. Of course, this may increase or decrease aggregate throughput depending on many factors.

One of our researchers has used this technique to great advantage and often has three independent NeuroManager sessions operating simultaneously, each of which is running multiple simulators on multiple machines.

```
...
nm = NeuroManager(SMDirectorySet, SMAuthData, UserData);
...
config1 = MachineSetConfig();
config1.AddMachine(MachineType.MYUNIXMACHINE01, 1,...
                               '/home/username/MyWorkDirOn01');
config1.AddMachine(MachineType.MYUNIXMACHINE02, 1,...
                               '/home/username/MyWorkDirOn02');
nm.ConstructMachineSet(SimType.SIM_SINESIM, config1);

result = nm.RunFromFile('SineSimSpec.txt');

nm.RemoveMachineSet();

% --
config2 = MachineSetConfig();
config2.AddMachine(MachineType.MYUNIXMACHINE02, 3,...
                               '/home/username/MyWorkDirOn02');
config2.AddMachine(MachineType.MYUNIXMACHINE03, 3,...
                               '/home/username/MyWorkDirOn03');
config2.AddMachine(MachineType.MYUNIXMACHINE04, 3,...
                               '/home/username/MyWorkDirOn04');
nm.ConstructMachineSet(SimType.SIM_SINESIM, config2);

result = nm.RunFromFile('SineSimSpec2.txt');

nm.Shutdown();
```

Figure 8: Changing machine configurations within a single NeuroManager session

2.11.3 Remote operation

For UNIX hosts, NeuroManager can be operated remotely; that is, without a MATLAB installation at the site of the user's activity. This powerful feature allows the user to make use of NeuroManager on many devices, including phones and tablets. All of the facilities of NeuroManager are available through remote operation, including user notifications and use of dual key. Moreover, by using the UNIX **screen** utility, the user can disconnect and reconnect to the same session without interrupting NeuroManager operation. **SineSimSet_SS07.m** shows how to set up a script so that it operates properly both remotely and using the MATLAB GUI.

As a demonstration of this concept, this author used his iPhone to edit and initiate a set of Neuron simulations while taxiing the runway on the way to SFN 2014, then used that same phone to examine the results after the plane had landed.

Here's a detailed scenario. The user sets up and starts a NeuroManager session in the lab on HostMachine, ensures that things are operating reasonably, then drives to a downtown meeting. Before the meeting starts, the user uses a VPN app and a Server Maintenance app to log into HostMachine via cellphone, and checks to see that Simulations A-F have been submitted, but have not yet started running (they are in the cluster's waiting queue). An hour into the meeting, the user's phone buzzes with a text message saying that Simulation A is running. Finally, back at the lab, the user logs into HostMachine and sees that A-E have finished and been downloaded and that F has been running for an hour. The user checks some of the results from A-E. At home for the night, the user logs into HostMachine from a laptop, examines the results from F as well,

creates a new simulation specification, starts a new session, then shuts down the laptop for the night.

We will assume that the user has not requested notifications and just wants to do remote operation. The actions to accomplish this are fairly simple...

At the lab, the user:

- Logs into the lab's UNIX host ('HostMachine' here) and edits a simulation specification for, say, 20 simulations.
- Edits a script similar to **SineSimSet.m** that configures a Machine Set with four 4 SimMachines.
- Edits the NeuroManager constructor to set the LogEchoFlag to true.
- Types 'screen' to start a screen session.
- Navigates to a directory on the MATLAB search path.
- Executes a command similar to:

```
matlab -nodesktop -r "cd('/home/...../SineSim'); SineSimSet"
```

- Enters the private key passphrases as requested.
- Watches the log echo to determine that things are running properly.
- Detaches the screen using 'ctrl-a d'.
- Heads off to the meeting.

Before the meeting, the user:

- Connects to the lab institution via VPN app.
- Logs into HostMachine via Server Maintenance app.
- Attaches to the screen using `screen -r`.
- Sees the log echo.
- Forgets to detach the screen.

At home on the Windows laptop:

- Connects to the lab institution via VPN web page.
- Logs into HostMachine via Putty (say).
- Detaches the screen from the phone remotely with `screen -d`
- Attaches to the screen using `screen -r`.
- Sees the log echo and realizes that NeuroManager has finished.
- Peruses the results, etc.
- Starts a new NeuroManager session and verifies that it is proceeding properly.
- Detaches from the screen and shuts down the laptop.

At the lab the next day, the user logs into HostMachine, attaches the screen, and sees the elapsed log for the second session.

By setting up a couple of simple scripts, the user can make it very easy to start up a new session, even using a smart phone. This author uses vi on an iPhone to edit specification files, then starts and monitors NeuroManager remotely with that same phone. With use of DropBox or other synchronization utility, together with saving output files in a suitable format (say, as a jpg as well as a fig), it's possible to browse output results in the same way. By printing a figure in pdf format with code like `print(gcf, fullfile(outdir, 'filename'), '-dpdf')` it's also possible to create and email pdfs that contain output results, which are ideal for remote inspection. It's also possible to access the latest snapshot of the real-time status webpage in the SimSet directory on the host: ***.html**, but there is no currently implemented way to access that page as a normal webpage.

2.11.4 Adding a new simulator

NeuroManager's notion of simulator is discussed elsewhere, however there are a few practical points to make. A NeuroManager Simulator is a software entity distributed between host and remote. The *host–side component* of a NeuroManager Simulator is represented by a terminal subclass in the Simulator class hierarchy, such as ***SimNeuron_SimpleSpike02A***. For clarity, many methods in the Simulator class hierarchy that form the host-side aspects of the Simulator are named after stages in the workflow analysis (see Supplemental Materials), such as `postModelFilesUpload()` in the ***ModelFileSim*** class.

The *remote-side component* of the simulator is a compiled MATLAB program residing within a remote-side file structure. This remote-side component may or may not have an embedded call to an external simulator program such as Neuron (for example, the SineSim simulator does not make a call to an external simulator program); but if so, one can consider the external program to be the core of the simulator, with user–defined files forming an extending wrapper around the core via `userSimulation()`. These member files are shipped up to the remote during execution of the Simulator's constructor. MATLAB compilation (which will include any uploaded simulator m-files), and model compilation if appropriate, take place upon the remote.

The host-side and remote-side components of the Simulator work together to run Simulations.

In practice, adding a new simulator is quite similar to adding a new machine. Adding a new simulator typically involves

1. analyzing simulator operation from the workflow point of view, especially with respect to the Workflow Stages as described in Supplemental Materials;
2. developing a new subclass of an existing abstract simulator class such as ***NoModelFileSim*** for simulators with no model files and ***ModelFileSim*** for simulators that use model files;
3. adding the new simulator to **SimType.m**[1]; and
4. creating a compatible **userSimulation.m**. Note that the user can have as extensive a MATLAB application using as many m-files as desired on the remote, although there are no GUI or user interaction facilities. `userSimulation()` is only the entry/exit point for user code on the remote. Any data files or other simulation results files should be placed in the simulation's output directory; from there they will be automatically downloaded using class methods.

2.11.5 Extending the Input Parameter Vector

To handle a larger Input Parameter Vector, which is currently set at ten parameters, there are at least two options. First, simply edit **SimSet.m** to make the number larger. Although this file is part of the core code, having a larger number will not affect SimSets with ten or fewer parameters and will allow larger numbers.

[1]Note: a **SimType.m** in the Custom dir will override the **SimType.m** in the Core dir because of MATLAB search path precedence. The user must keep the **UNASSIGNED** enumeration member in both files, though.

Second, create input files, either ahead of time or on the fly, with as many parameters as you want to change, put the filename(s) into the Simulator leaf class or into the IPV as one parameter, upload them within the workflow, and process them in `userSimulation()`.

2.11.6 Using the NeuroManager cross–compilation feature

NeuroManager provides the ability to do the MATLAB compilation on a machine other than the remote on which simulation will take place. This is useful in situations where the user doesn't have access to a MATLAB installation with Compiler Toolbox on the intended remote, but does have access on another machine.

The cross–compilation machine is actually an subclass of ***NoSubMachine***, which assumes that the cross–compilation machine can do the compilation without use of a cluster manager. The cross–compilation machine is formed during the constructor of the SimMachine that will be using the cross–compilation feature. The ***MachineStampede*** class in the LocalMachine directory shows how to configure the cross–compilation machine. Everything in the constructor of the using machine is the same as any other, except that the xCompilationMachine and xCompilationScratch dir properties supplied by the ***MATLABCompileMachine*** class, normally null, are set by constructing an ***xCompilationMachine*** in the former case, and referencing the machine's *xCompDir* setting in the latter case. The *xCompDir* setting requires need a valid directory path on the cross–compilation machine that is empty (it will be automatically cleaned out without verification or notice). Also the user will need SSH access to the cross–compilation machine, just like the other SimMachines. Note that a) the Communications test procedure doesn't currently test communications with the cross–compilation machine, and b) the MATLAB Compiler version on the cross–compilation machine must match the MCR version on the simulation machine.

2.12 Miscellaneous specific installation procedures

These sections are referenced earlier in the text and moved here for simplification.

2.12.1 Installation of PuTTY

If you are using a Windows host, the PuTTY SSH package can be downloaded from `http://www.chiark.greenend.org.uk/~sgtatham/putty` and installed. NeuroManager makes use of most elements of that package, including the Pageant agent. Ensure that the Windows environment knows where the PuTTY elements are located because NeuroManager doesn't store that information. Basically if you can pop up a command window and run PuTTY from the command line with no additional location qualifiers then things are probably ok. Connect to your remote machine(s) using PuTTY for verification of its operation.

2.12.2 Setting up SSH2 communications

For either type of host, it's advantageous to gain familiarity with SSH2 communications between the host and remote without the use of NeuroManager before bringing NeuroManager into the blend.

Overall, SSH2 communications with a remote server authenticate in one of two ways: 1) with a password that is sent to the remote (very nonsecure), or 2) using a dual key authentication process that still uses a passphrase, but the passphrase is not sent through the network — instead, an exchange of crytographic keys is used to verify the identities of the client and host.

The user chooses *true* or *false* for the 'UseDualKey' option in the ***NeuroManager*** class constructor. A *false* indicates that the user will supply a password for every machine in the machine configuration that has nonzero number of simulators. A *true* indicates that NeuroManager will use dual key authentication for communications with each machine that has an empty password.

In this section, to avoid sending a password, we will describe setup and test of communications with the dual key system. In the case this this proves intractable, the user can use password authentication by simply using the 'Password' entry in the `create...Data()` configuration function as listed above.

Though all host–remote interactions will take place over SSH2[2], each host–remote connection's use of password versus dual-key authentication is determined by the password in the `create...Data()` configuration function. An empty string for a password is a direction to NeuroManager to use dual-key authentication for the interaction between the host and that particular remote. In addition, as mentioned, the user must set 'UseDualKey' to *true* in the ***NeuroManager*** class constructor. If 'UseDualKey' is set to *false*, then an empty password will result in an error. The ability to use dual-key, of course, also depends on server configuration and the user's key setup. We recommend that the user use a key pair that is for NeuroManager use only. We like to store the keys in the **LocalMachine** directory with the other local–machine–specific files, but that is optional.

For UNIX hosts: SSH2 communications take place using both the MATLAB/Ganymed library and the `scp` UNIX command via `ssh-agent`.

1. Change directory to the LocalMachine directory on the host.
2. Create a new key set for NeuroManager use only by issuing the following commands, where *keyname* is a name of your choice that has no extension, and *passphrase* is a password for the key:

   ```
   ssh-keygen -t 'rsa' -C 'NeuroManager Use' -N passphrase -f keyname
   chmod 600 keyname
   ```

3. Install the public key on the remote machine by issuing the following command from the LocalMachine directory, where *keyname* is the same as in the previous step, *username* is your username on the remote, and *IPAddress* is the numerical address of the remote. This command will ask for your password on the remote machine; this is your login password, NOT the passphrase entered above.

   ```
   ssh-copy-id -i keyname.pub username@IPAddress
   ```

4. Test your SSH key setup by running (still from the LocalMachine directory):

   ```
   ssh-agent
   ssh-add keyname
   ssh -l username IPAddress
   ```

   The `ssh-add keyname` will ask for your passphrase, but the `ssh` itself should not; it should be using the agent. If this is the case, you are set. If not, you will need to troubleshoot this. Finally, kill the agent just created with

   ```
   ssh-agent -k
   ```

Side Note: NeuroManager makes use of the MATLAB/Ganymede library for most communications and also `ssh-agent` as well, for asynchronous downloads, and in fact creates its own instance of `ssh-agent` for each NeuroManager session, then kills that instance at the end of the session, helping to ensure that use of those keys is deliberate. The constructor call to NeuroManager asks the user to provide the location and name of the previously–generated host–based private key (see any of the code examples). When NeuroManager starts, it will ask the user to provide the passphrase for the private key (unless a login password has already been provided in the `create...()` function, in which case interchange with that specific machine will be password–based and not key–based). As a result of this overall construction, NeuroManager may ask twice for passwords. To remedy this is outside the current version of the software but hopefully we can rectify this in the future.

For Windows hosts: SSH2 communications take place using both the MATLAB/Ganymed library and the resources of PuTTY.

[2]Except for same-machine interactions, which use normal UNIX system commands.

1. Run PuttyGen and create a public/private key pair of type 'SSH2–RSA'.
2. Change the comment to 'NeuroManager Use'.
3. Insert a passphrase of your choice.
4. Save the private key in the LocalMachine directory with the standard '.ppk' extension. This saves the private key in the Putty format.
5. Save the public key in the LocalMachine directory with a '.pub' extension.
6. Under 'Conversions', export the key in OpenSSH format, saving it in the LocalMachine directory with no extension.
7. Copy the contents of ***keyname*.pub**, not including the first and last lines, and paste it into the user's **/.ssh/authorizedkeys** file on the remote machine (or create such if there is none)[3]. The public key will need to be copy-paste-added to each remote machine the user intends to access with NeuroManager.
8. Test your setup by running Pageant. Add to it the new ***keyname*.ppk** private key you just created; Pageant will ask for your passphrase. If you do 'View keys' you will see your comment as part of the listing. Then create a 'New Session'. Under 'Session' enter the *IPAddress* of the remote and check 'SSH' as Connection Type. Under 'SSH—Auth' ensure that 'Attempt authentication using Pageant' is checked. Click Open. The new terminal window will open and ask for *username*. Given that, it should not ask for your password; instead it should connect with the dual key authentication. If this is not the case you will have to troubleshoot or not use dual key for this machine.

Side Note: NeuroManager will make use of both private keys (actually one key in two formats) and the public key; its constructor asks for both private keys. NeuroManager will start Pageant with the PuTTY format private key; Pageant will ask the user for the passphrase. In general the two passphrases will be the same, but do not need to be. If Pageant is already running with that key, then NeuroManager will not ask for the passphrase. If Pageant is already running without that key, then NeuroManager will not ask for the passphrase and will continue[4] but communications will fail with an error message similar to 'Communications Test with MyMachine failed.' See Testing machine communications below for more on testing communications.

Side Note: Please note that the MATLAB SSH2 library (which NeuroManager uses) doesn't support use of an agent. As a result, the passphrase for SSH communications must be entered manually within MATLAB and stored there during the session. There is no current beautifully secure way to do this. In particular, the user's entry of the passphrase will go into the MATLAB command history. There is no MATLAB function that can securely delete this entry, but it can be done manually by the user by right-clicking on the command history frame and selecting 'Clear command history'. Although this is not desirable, it is far better than storing passwords and/or transmitting them between computers, which is the case without the use of dual-key authentication.

More information about SSH public/private key communications can be found easily, including Barrett et al (2005).

2.12.3 Testing machine communications

With a remote machine set up and SSH2 communications in place and verified, the next step is to ensure that machine communications are operating properly from within NeuroManager. The CommsTest example tests communications with the myServer02 remote machine. Use this example to test your setup within

[3]Note: the administrator of the remote machine may have to configure SSH communications for dual key operation; that is beyond the scope of this document.

[4]PuTTY provides no way to automatically determine whether the proper key is loaded or not.

MATLAB on the host as follows. The SineSim simulator described below (Section 3.12) goes into more detail as to what each part of the script does. *Note:* Unless this test passes completely, NeuroManager itself will not run simulations properly on myServer01.

1. Change to the CommsTest directory on the host within MATLAB.
2. Open the **CommsTest.m** file in the MATLAB editor.
3. Part I: Edit **myNMStaticData.m** to fit your host UNIX/Windows setup (this step should already have been done).
4. Part II: Edit the run-specific directory paths to fit your setup (there is no need since we have automated this part for you).
5. Part III: Ensure that 'UseDualKey' is set to true.
6. Part IV: Alter the machine work directory in the 'AddMachine' line to match the NeuroManager work directory on your remote machine. *Note:* this directory must already exist when this script is run. *Warning!* the contents of this directory and its subdirectories, if any, will be destroyed without confirmation.
7. Run the script. You will see log entries in the Command Window as well as in the log file, which can be found in the 'SimResults...' directory created within the CommsTest directory.
8. If there are problems with the test, the error messages should be helpful. In addition the 'Machine-Scratch' directory under 'SimResults...' will have a code such as 'A5', which indicates at which point the test failed. Examine **CommunicationsOK.m** and **FileTransferOK.m** to see where these codes are produced within the tests .

Failure here can be caused by many things, including: 1) The remote machine is down. 2) The remote machine is not configured for remote connection with dual key SSH2. 3) The remote machine uses SSH1 instead of SSH2. 4) The public key is not installed properly on the remote machine. 5) The **createMyServer01Data.m** file has incorrect information. 6) The Java library file is not in the Core directory. 7) The MATLAB search path does not point properly to the MATLAB SSH2 library. 8) The remote work directory doesn't exist. 9) The remote work directory permissions are not set correctly.

Side Note: As demonstrated in **CommsTest.m**, the ***NeuroManager*** class has a method specifically for communications testing called `testCommunications()`. One creates an instance of the ***MachineSetConfig*** and adds the new machine (Figure 9). `testCommunications()` exercises basic communications and file–passing between the host and remote machines using SSH2, using methods in the ***MachineCommsTest*** class. Any failure will cause a MATLAB error that will provide debugging clues as to the stage of the failure. This testing is fairly quick, and since network conditions and remote machine availability are dynamic, we recommend testing as a routine preliminary to each set of simulations.

```
...
nm = NeuroManager(SMDirectorySet, SMAuthData, UserData);

config = MachineSetConfig();
% MyWorkDir must preexist and will be overwritten
config.AddMachine(MachineType.MYUNIXMACHINE, 1, '/home/username/MyWorkDir');
config.Print();

nm.TestCommunications(config);

nm.Shutdown();
```

Figure 9: Using the `testCommunications()` method of the NeuroManager class to test communications with a single remote

3 Theory and design

3.1 NeuroManager detailed workflow

See Supplemental Material for descriptions of the detailed workflow and descriptions of their connection to the NeuroManager code.

3.2 NeuroManager ontology

This section was cut from the main paper for space reasons, but should be of use for users. In this subsection we present definitions and discussion of some of the concepts in NeuroManager. Figure 10 illustrates the fundamental principle that each Simulation is run on a Simulator, producing Results, which can be processed to produce Products.

Free Parameters. The simulator/simulation process in computational neuroscience consists entirely of programmable software elements that have a large number of configurable entities, such as sections, algorithms, biomechanism population, concentrations, time steps, durations, models, links, and many others, each of which has parameters that determine its character. In any given set of simulations, the user selects a small subset of these parameters to vary, keeping the remainder constant. We call these varying parameters 'Free Parameters'. In NeuroManager a Free Parameter is represented by a string (e.g., a number, a node identity, a filename, or an algorithm id). The remainder of the parameters, being static, become part of the sense of 'Simulator' (see below).

An **Input Parameter Vector**, or **IPV**, is a set of string values to be assigned to each of the free parameters for a specific Simulation.

A **Simulator** is a software device that acts upon an Input Parameter Vector to produce results, data, and other output. This includes a specific configuration of the software device's attributes (the static parameters mentioned above). The Simulator may be partially composed of a configurable core software product such as NEURON, in which case NEURON forms the core of the Simulator and we say that NEURON is the 'SimCore' of the Simulator. A NeuroManager Simulator's precise SimType is defined by subclassing the Simulator class, as discussed below, and all Simulators in use at one time are constrained to be of the same SimType. The SimType of a Simulator class has the same name as the leaf class, but in all caps with underscores separating words (e.g., class '***SimNeurPurkinjeMiyasho2001***' gives rise to SimType '***SIM_NEUR_PURKINJE_MIYASHO2001***').

In NeuroManager, the Simulator object is shared between host–side subclasses of a Simulator base class and a set of functions uploaded to the remote. The user's primary access to the core simulator ('SimCore'; see below) is one of these functions called `userSimulation()`, which calls the SimCore through the use of

utilities also in that set. These functions are part of the compiled MATLAB program and use the MATLAB **system(***'commandstring'***)** function to run the core simulator, if there is one. The command string includes command line entities that the user is likely familiar with (such as NEURON's `nrniv` executable). The `userSimulation()` function also allows the Simulator to do pre– and post–processing and is where the Input Parameter Vector strings are turned into inputs to the core simulator (sometimes through creating or modifying files).

A **SimCore** is the heart of a Simulator and can be an established simulator such as NEURON or MCell, or simply a custom executable that is constructed to have a set of inputs and outputs. Generally a SimCore will be configured with default or explicit configuration parameters as well as one or more Models. NeuroManager incorporates the SimCore into the Simulator object, including configuration, Models, and pre/post–processing, thus allowing a highly generalized notion of what an input parameter is. Although unproven, the SimCore could also be constructed of a chain of established simulators and custom code.

A **Simulation** is the act of using a Simulator on an Input Parameter Vector, together with that input vector and any resulting data or other output produced. One runs Simulations on a Simulator, which is hosted on a SimMachine.

A **SimSet** is a set of Simulations to be run using a specific choice of Free Parameters and Simulator. Each Simulation in a SimSet will have its own ID and its own values for the Input Parameter Vector, but the Simulator/Free Parameters choice is fixed for the entire SimSet. All the output of the Simulations in the SimSet will be stored in its own isolated subdirectory located within a directory named after the SimSet, which also has its own ID.

A **Model** is those aspects of the configuration of a Simulator which make it specific to the biological entity being studied. A Model can include other models, such as those describing ion channels. As alluded to above, various entities within a Model, or even the Model itself, can be selected by the user to be part of the Free Parameters. The examples demonstrate this in detail (see the User Guide).

Results are the output of the Simulator acting on a Simulation's Input Parameter Vector, such as a voltage time series. They also become part of the Simulation.

Products are the output of the Simulator acting on a Simulation's Results. They also become part of the Simulation. Example: a graph of the voltage vector.

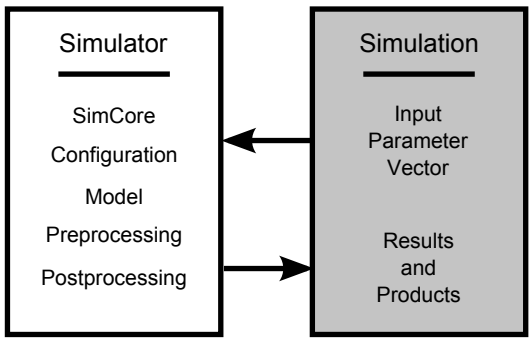

Figure 10: Each Simulation is run on a Simulator. The Simulator acts upon an Input Parameter Vector, data that instantiates the Free Parameters chosen by the user, to produce Results and Products. The Simulation is the act of simulation itself, together with the Input Parameter Vector, Results, and Products formed from those results. The Simulator is made up of the SimCore set up with the Configuration (including step size, choice of algorithms), the Model (which adds the biological content), and user routines that do Preprocessing and Postprocessing.

Simulator–Free Parameter boundary. As alluded to under 'Free Parameters' above, the boundary between Simulator and Free Parameters is a reconfigurable one, depending on the user's research goals for the Simulations in the SimSet (Figure 11). The Simulator subclass chosen for the SimSet determines which parameters are fixed and which are Free Parameters for that SimSet. A given period in research activity

will have a focus on specific aspects of a model, simulator configuration, or other aspect of the simulation work. The other aspects are, for that period, of less interest; in fact the art of research is to know which parameters are of interest at any given time. All other parameters need specific values, but they are generally static during that period. Those variables can be handled by the Simulator class. Simple examples might be temperature, or cell morphology, or stimulation waveform.

With that stable background, the Free Parameter space will consist of those parameters that are of interest at the moment. If the user's goal is to investigate the effects of various ion channel concentrations on excitability, then the Free Parameters will be ion channel concentrations and each Input Parameter Vector will consist of a set of specific numerical values to be assigned to each concentration for that simulation. If the goal is to investigate the effects of various state transition probabilities of an ion channel model, then the Free Parameters will be the set of transition probabilities, and each Input Parameter Vector will consist of a set of specific numerical values to be assigned to each transition probability for that simulation. If the goal is to investigate the effect of various state transition diagrams for a specific ion channel, then the Free Parameters will be the various definitions of that ion channel's state transition configuration, and the Input Parameter Vector will consist of the names of the files or functions to be used for the state transition definitions for that simulation.

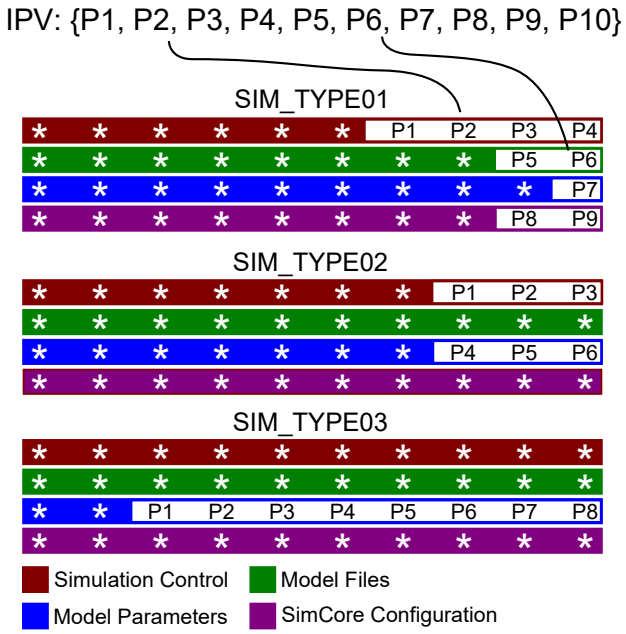

Figure 11: The SimType captures the relationship between fixed and free parameters in a Simulation. The Free Parameters of a Simulation (P1 - P10) form the Input Parameter Vector (IPV), and can come from the Simulation Control parameters (such as time step and duration), Model files (different morphologies or ion channel files), Model parameters (concentrations and location of synapses) and the SimCore configuration parameters (such as integration method). All parameters that are not free become part of the Simulator, which is defined by its class/SimType. The illustration shows three different choices of free parameters as captured by SimTypes. The black P1 ... P10 in each white section represent the free parameters that will be represented in the IPV; the white '*'s in each colored section represent the large number of parameters that are fixed or dependent on the free parameters. In SIM_TYPE01, the IPV has nine parameters from each of the four areas. In SIM_TYPE03, the IPV is composed of eight parameters from the model alone.

A **Session** is a single use of NeuroManager via a script or NMSessionML file (see Section 2.8), which may involve multiple SimSets and multiple Simulations. The term 'Session' does not apply to embedded use

of NeuroManager.

A **SimMachine** is a hardware/software entity which is capable of hosting one or more Simulators of the same SimType. A SimMachine is a UNIX computer, server, or cluster that is SSH–accessible by the user and hosts the SimCore to be used. Since NeuroManager acts as a Virtual User, if the user can run a simulation manually on the remote, then NeuroManager is capable of that as well. It is possible to make more than one SimMachine out of a single computational resource (using non–overlapping work directories), and different queues on a cluster can be different machines. See Section 2.11.2 for practical details.

A **Machine Set** is a collection of different SimMachines that are capable of hosting the same Simulator type. NeuroManager creates the Machine Set using the MachineSetConfig class with user–supplied data, including the number of Simulators each machine should host.

3.3 The machine and simulator class hierarchies

The core of NeuroManager is the the two class hierarchies: machine and simulator, which maximize extensibility while minimizing code duplication. Although we do not discuss this structure here, we present diagrams for the user's convenience.

The machine class hierarchy is shown in Figure 12. The simulator class hierarchy is shown in Figure 13. Not all classes mentioned in the Figures have been supplied with the code.

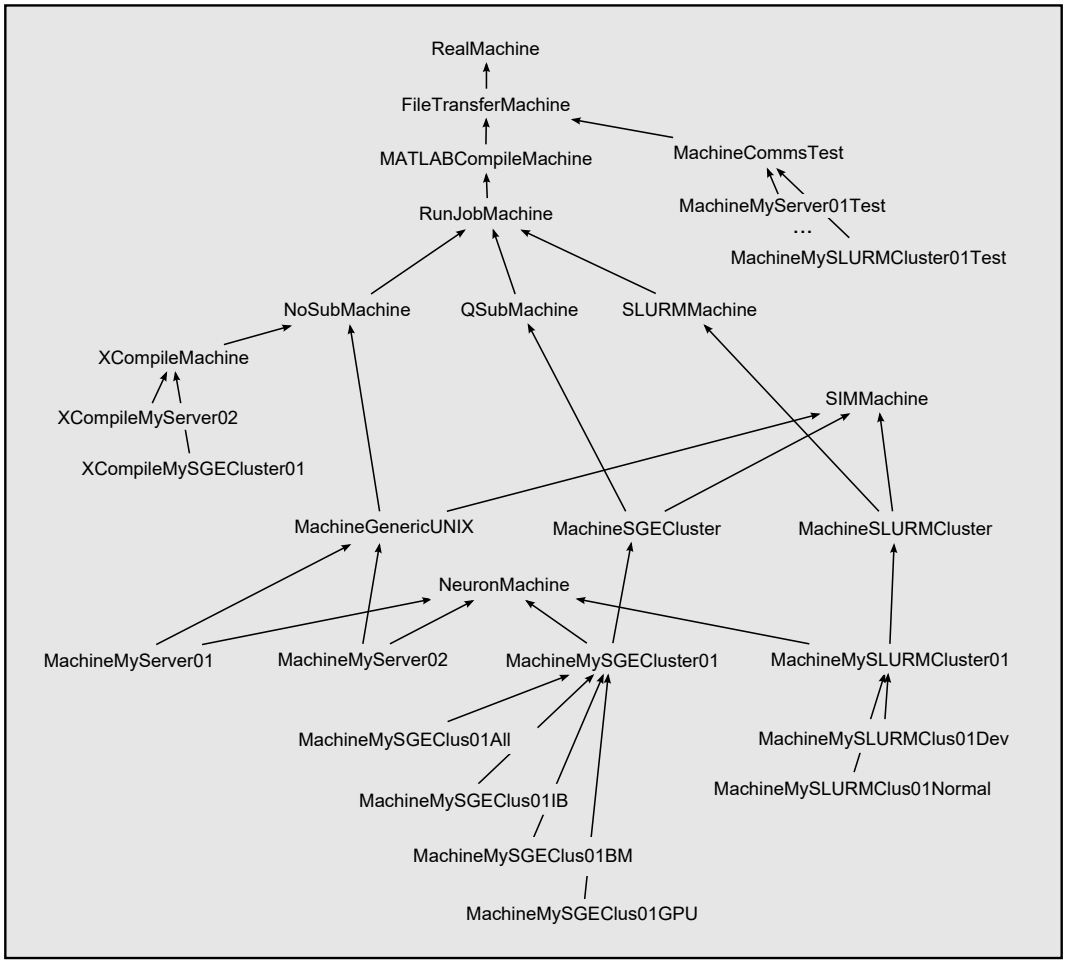

Figure 12: Machine Class Hierarchy. Class inheritance creates a single source for most code and maximizes extensibility.

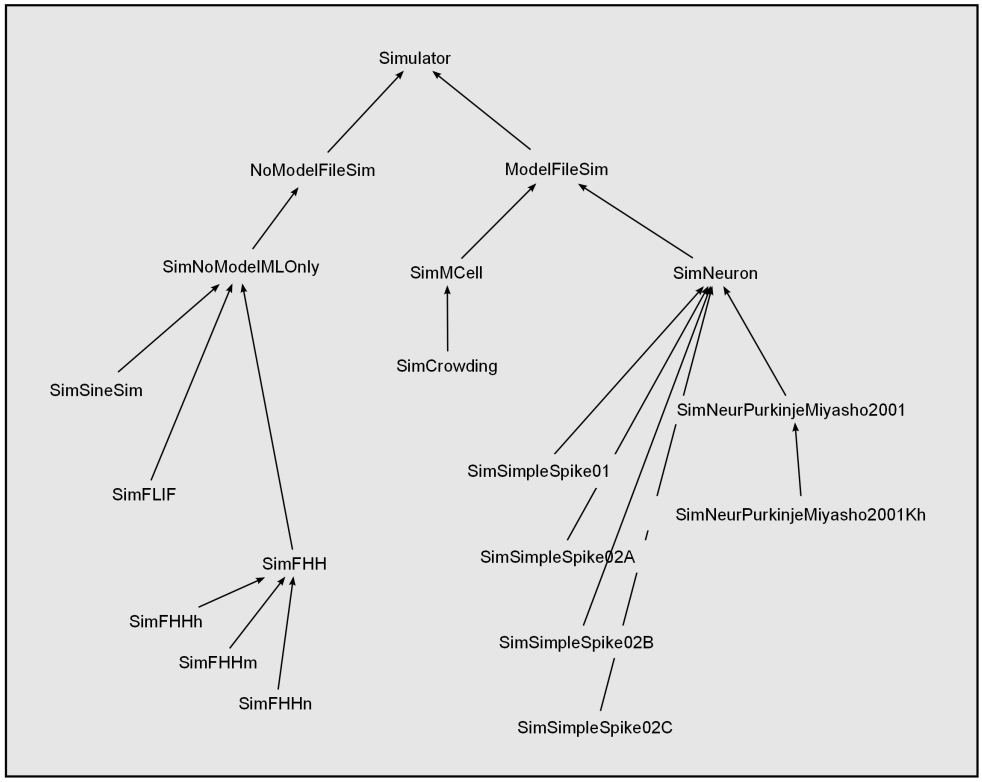

Figure 13: Simulator Class Hierarchy. Class inheritance creates documented evolution of simulator design.

3.4 The SimSet specification

The SimSet Specification allows the user to define a single or multiple simulations to be run on the simulators provided by the configured machine set. The user assigns a unique id to each simulation and specifies a vector of input free parameters. The SimSet Spec can be constructed on the fly or loaded from a text file.

The SimSet Specification File format is described in Figure 14. The first non–comment line in this text file must be the SIMSETDEF line with an id string SimSetID and the specification of which simulator type that NeuroManager should use, indicated here as 'SimType'. **SimType.m** defines simulator types; and the user will make use of this file to extend NeuroManager's set of simulator types. Comments have a percent sign in the first column and can be interspersed anywhere. Comments will not be processed, but will appear in the copy of this file that is saved with the results, so that comments can be used for documentation purposes. Note that individual simulations can easily be commented out/in with the MATLAB editor's 'ctrl-r' / 'ctrl-t' keyboard shortcuts. The third type of line is the specification of a single simulation input parameter vector, which begins with SIMDEF or SIMDEFN, where the 'N' indicates that the user wants a notifications upon start, running, and end of this specific simulation. The second token in a SIMDEF line is the id of this simulation. Simulation IDs within a file must be unique and duplicates will result in an error. The remaining tokens in the line are no-whitespace strings representing the input parameters to that specific simulation. These tokens can represent numbers, ion type, colors, filenames, switch settings, or whatever else the user requires, and will a) be presented verbatim to the `userSimulation()` function on the remote, and b) be accessible on the host through the Simulation object. A copy of the file is automatically saved under the SimResults directory.

```
SIMSETDEF SimSetID          SimType
% Comments begin with a percent sign
% Simids must be unique
%       simid  P-01 P-02 P-03 P-04 P-05 P-06 P-07 P-08 P-09 P-10
SIMDEF  ID01   str  str  str  str  str  str  str  str  str  str
SIMDEF  ID02   str  str  str  str  str  str  str  str  str  str
% "N" adds automatic notification at begin and end of this simulation
%    if Notifications are turned on.
SIMDEFN ID03   str  str  str  str  str  str  str  str  str  str
SIMDEF  ID04   str  str  str  str  str  str  str  str  str  str
...
```

Figure 14: The SimSet Specification File allows the user to assign a unique id to each simulation as well as specify input free parameters

A non–file version can easily be constructed on–the-fly; see, for example, the SimpleSpike02A example.

3.5 Host side directories

The **host–side** directory structure (Figure 15) is designed to make it possible for the researcher to do simulation sets in quick succession without worrying about them overwriting one another. The top directory is called 'SimResults' appended with the date and time. Subdirectories are named with a unique SimSet ID and then the IDs of individual simulations, which were specified by the user in the Simulation Specification file. Those IDs ('simids') are also available programmatically to remote–side code for the purpose of naming data files, providing titles for plots, custom captions for figures, and the like.

Note: Running additional SimSets in the same session (see Running multiple SimSets serially with a single machine configuration) will produce additional SimSet directories below the SimResults directory; named by the SimSet ID.

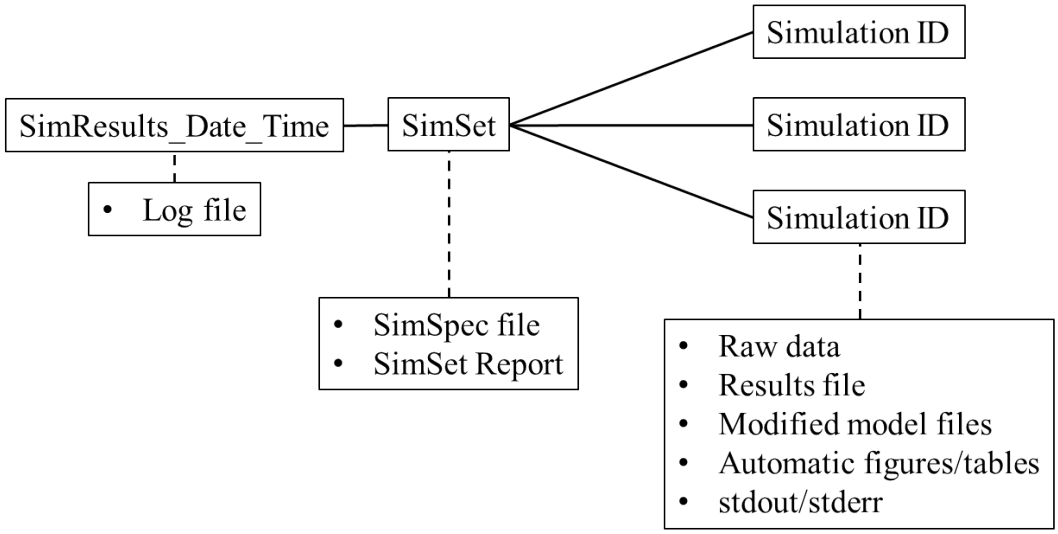

Figure 15: Host Directory Structure. Every NeuroManager session is Date/Time stamped

3.6 Remote side directories

The **remote–side** directory structure (Figure 16) starts with a user–supplied work/base directory for the machine ('Machine Base'), below which are built individual directories ('Simulator Base') for each simulator the machine supports. The Machine Base directory will be cleared and should be provided empty — no subdirectories or files. Below the simulator directory is the simulation directory ('Simulation Base'), which itself has input and output directories. The temp directory off the input directory is used for model files and other input files that are specific to that simulation. This structured, compartmented approach provides debugging observability and resource exclusivity for operating simulators in parallel without the need for locking files. Also off the machine base is the 'SimulatorCommon' directory, which is used as a source for files that are common to all simulators. Off the simulator base is the 'SimulationCommon' directory, which acts as a model file repository for simulations, since previous simulations may have modified model files and the current simulation needs to have an untouched source. Once the remote–side directory structure is no longer needed, it is removed, typically but not exclusively by class destructors. Here is a description of the remote–side directory structure.

- MachineBasedir: Holds all files relating to NeuroManager operation on that particular machine. Must be already in existence, created by the user. This directory and all subdirectories will be purged by NeuroManager without confirmation. Any user files will be destroyed. Do not use this for anything except normal NeuroManager operation.

- SimulatorCommons: a subdirectory of the Machine Basedir; holds files common to all simulators on that machine. Acts as a repository.

- SimulatorBasedir: created automatically as a subdirectory of the Machine Basedir by the simulator constructor. Holds standard and custom simulator files for a specific simulator. Has subdirectory **temp** where all MATLAB compilation takes place to keep all auxiliary files from clogging the base directory (basedir). The MATLAB shell and executable are copied from **temp** to the simulator basedir. If the MATLAB compilation has already been done, the shell and executable are copied from SimulatorCommons. SimulatorBasedir is the working directory for the actual simulation run. Simulators have only one simulation at a time. Simulations can use the Simulator Basedir to hold modified model files (with different names than the uploads) but must remove them before simulation is done so that they don't

build up. In general it's better to use the simulation input directory instead to hold simulation-specific files of all kinds.

- SimulationBasedir: The directory for a given simulation. Not used itself; only its subdirectories **input** and **output** are used.
- SimulationCommons: a subdirectory of the Simulator Basedir; holds simulation–centric files common to all simulations (i.e., independent of input parameters) on a given simulator. Used primarily as a repository for simulations to pull model files from and helps ensure simulators are independent of one another.
- SimulationInputdir: is where all simulation-specific input files are kept, whether uploaded or constructed on the fly. For model file processing such as in Neuron, model files are copied into this directory and compiled here to keep directory structure clean; this directory is removed with the simulation so that there is no cross-simulation contamination due to leftover compilation products. Simulation-specific model files that are constructed on–the–fly are constructed here or uploaded to here.
- SimulationOutputdir: is where all simulation-specific output files are kept. All files in this directory will be zipped and downloaded to the host. All debugging files will be copied into here. The user should put all results into this directory. Special procedures for downloading for specific machines or simulators (such as alternate methods for large file downloading) can be performed by overriding class methods (unproven as of 7-20-14).

During the construction of each machine's first simulator, **SimulatorCommon** is populated with files that are common to all simulators, including uploaded std, custom, and model files and the results of the MATLAB compilation. Then, when each subsequent simulator is constructed, instead of doing uploads and compilations, they draw from the SimulatorCommon directory.

Similarly, since each simulation may use different model files, and possibly modify them or build new ones, we have employed the **SimulationCommon** directory which holds untouched model files (and possibly other simulation common files). To use files in **SimulationCommon**, a simulation copies them into the simulation input directory where it is free to use and modify them at will; the input directory and its contents will be destroyed when the simulation is cleaned off the remote.

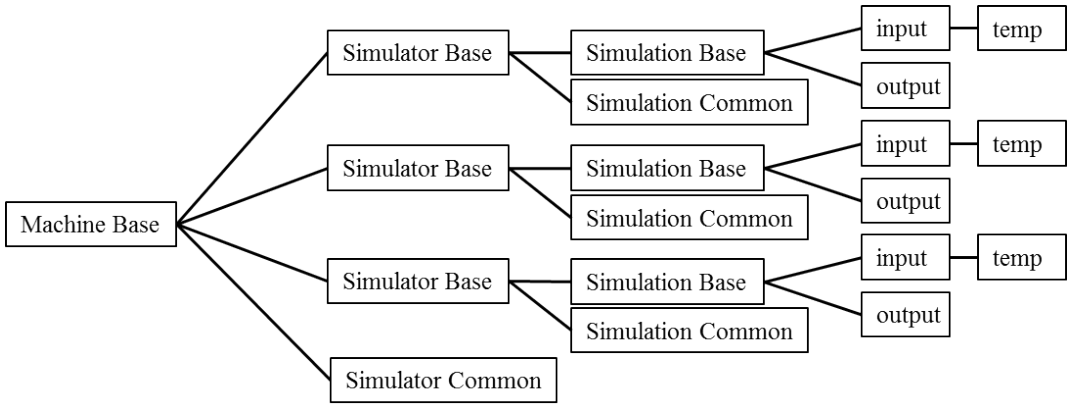

Figure 16: Remote Directory Structure. Careful isolation prevents cross–contamination between simulations operating in parallel

3.7 The userSimulation function

On the remote side, the `runSimulation()` core function is designed to call the user's `userSimulation()` custom MATLAB function and pass the simulation vector of input parameters in a standard way, together with run, input, and output directory paths. The user has to write the code that runs the simulation and processes its output data using those parameters and those directories. See the sections below for our experiences in adapting NeuroManager to different machines, simulators, and simulations.

In more detail: on the remote side, the user's interface to NeuroManager is through a MATLAB function call to the `userSimulation()` function (see Figure 17). The user supplies this function and whatever supporting files, functions, data, and other files that the running simulation requires. User–written `userSimulation()` is called by `runSimulation()` and supported by files **SimulationState.m**, **RMD.m**, and **OSType.m**, all of which are automatically shipped to the remote machine and none of which is intended to be modified by the user. For Neuron simulations we have also provided supporting m-files **runHocOnlySimulation.m**, **runPythonSimulation.m**, **createHocOnlyNrnivsh.m**, **createPython-Nrnivsh.m**, and **NeuronPythonPrep.m**, which are automatically uploaded as part of the ***SimNeuron*** class. In addition, as seen in the examples, **MachineData.dat** is uploaded automatically for all simulators and contains information about the machine in question, in the form of an ***RMD*** class instance, which the user can access using MATLAB's load utility.

```
[result, errmsg] = userSimulation(machineid, simid, rundir, indir, outdir, varargin)
```

Figure 17: The userSimulation function. The interface between NeuroManager and user code on the remote is the `userSimulation()` function, which supplies the user code with machine-specific details such as pathnames necessary for locating and placing input and output data, and the input parameter strings from the simspec

The parameters to the userSimulation function are: *machineid* is the identifier of the SimMachine on which the simulation is running, in case the user has need for that information during simulation processing or plot labeling. In addition, **MachineData.dat** contains all relevant information on the specific machine (it contains the appropriate instance of the ***RMD*** class) and is shipped up to the remote automatically. The simulation has the identifier *simid* which comes from the SimSet SIMDEF line in the SimSpec file (see below). The simid can be utilized by the simulation in labels and titles of plots that are produced as part of the simulation. The simulation will be run in the Simulator Base Directory (supplied to the user as *rundir*). The simulation model and other input files will be uploaded from the host into the *indir* directory and are available for the simulation to access. The simulation should place all output files (such as raw data, plots, or analyses) into the *outdir* directory. Each of these three directories (rundir, indir, and outdir) are supplied in remote format as full pathnames for safety and need no further processing by the user. The variable *varargin* is a cell array of strings that are input arguments to the simulation. The values placed in these strings are supplied by the user in the SimSpec file. Typically the user will convert these strings on a argument–by–argument basis at the top of the function definition using such converters as seen in Figure 18, which converts the third parameter string in the varargin array to a double named **mythirdvar**. In this way the remainder of the code can use the user's normal variable names. The variable *result* is 0 for success and 1 for failure; the user needs to assign this value appropriately. Finally, the variable *errmsg* is a string that the user can use to indicate the nature of a failure.

3.8 Machine data description

In this section we describe the contents of the **create...Data.m** files in more detail.

```
mythirdvar = str2double(varargin{3})
```

Figure 18: Converting the input parameter vector's third parameter string value to a double named mythirdvar

1. id — the computer name of the server as seen by MATLAB; a string such as 'MyUNIXMachine' or 'MyCluster'. This string must match the string received when `getenv('HOSTNAME')` is run within MATLAB on the server.
2. userName — the user login name for the server; a string such as 'username'.
3. password — the password for that user login; a string. It's best to use the SSH dual key approach, though, in which case this must be the empty string.
4. ipAddress — the numerical (decimal) IP address of the server; a string something like '000.000.000.000'. *Side Note:* this version of NeuroManager does not do DNS lookup.
5. matlabCompilerDir — the location of the MATLAB compiler binary on the server; a string something like '/usr/local/bin'.
6. matlabCompiler — the name of the MATLAB compiler binary on the server; a string something like 'mcc'.
7. xCompDir — the location where MATLAB cross-compilation should be done on this machine; a string something like '/home/username/MySMXComp'. This directory must be created ahead of time by the user and should be empty (its contents will not be preserved). If this machine is not to be used for cross-compilation (such as in our simple example), then this can be the empty string.
8. mcrDir — the location of the MATLAB Compiler Runtime Environment as installed by the user; a string something like '/usr/local/MATLAB/MATLAB_Compiler_Runtime/v81'.

The following entries are necessary for Neuron simulations (such as the SimpleSpikeXX examples), but not for the CommsTest or SineSim examples, and do not need modification at this point:

9. neuronDir — the location of the Neuron installation on this machine; a string such as '/opt/neuron'.
10. neuronHomeExt — relative to the NeuronDir, the location of the Neuron home directory; a string something like '/share/nrn'.
11. neuronBinExt — relative to the NeuronDir, the location of the Neuron bin files; a string something like '/x86_64/bin'.
12. neuronLibExt — relative to the NeuronDir, the location of the Neuron lib files; a string something like '/x86_64/lib'.
13. neuronPythonLibExt — relative to the NeuronDir, the location of the Neuron Python utilities library; a string something like '/lib/python2.7'.
14. envAddlLibLines — added for TACC Stampede, which needed a special line in job submission for NEURON, to tell NeuroManager where to look for Intel libraries that NEURON depends on.
15. pythonPath — the location of the Python binary on this machine; a string something like '/usr/bin/python'.
16. pythonEnvAddlLines — optional additional lines to add to the environment setup for Neuron–based simulators that use Python.

17. envAddlLines — optional additional lines to add to the environment setup for Neuron–based simulators that don't use Python.

Side Note: For clusters that have separate filesystem and jobsubmission machines, the Username, IP Address, and Password settings will be split between filesystem and jobsubmission machines; however for our simple UNIX server example, there is only one set.

3.9 Machine IDs

This section discusses Machine IDs in NeuroManager because they can be a little confusing. First, we remind the reader that a SimMachine in NeuroManager could be a single computer, in which case the filesystem and jobsubmission IP addresses are the same and there are no queues, or it could be a cluster of computers with multiple queues and separate filesystem and jobsubmission machines. In NeuroManager design, a queue becomes a separate SimMachine with its own set of Simulators, so that the user can have, say, 3 Simulators on Queue A and 4 Simulators on Queue B, where both queues are on the same cluster. Both queues are accessed with the same IP addresses and have the same login specifics. To deal with these concepts in NeuroManager, each SimMachine has an ID and a CommsID. The CommsID is the ID supplied by the `create...()` function; for example, 'StampedeCluster'. All queues on the cluster have the same CommsID. The CommsID is used for communications testing to avoid testing communications for every queue on that machine.

The ID for the machine itself is formed by appending the Queue extension to the CommsID (in the ***RealMachine*** class constructor). For example, the NORMAL queue on Stampede becomes 'StampedeCluster_NORMAL', and this ID is used for everything else in NeuroManager a machine ID is used for, such as log entries, directory names, plot labels, etc. The Queue extension is defined and associated with the specific queue and with the queue's jobfile entry string in a *queue enumeration class file* such as **mySGECluster01QueueData.m**, which handles all the queues on SGECluster01 that the user is interested in. The queue class (say ***MachineMySGEClus01ALL***) refers to the enumeration class ***MySGECluster01QueueData***'s ALL member, thus initializing the ID, Extension, and JobString properties of that enumeration class. For non–cluster machines without queues, queue data is not used and the extension is simply the empty string.

Note that machine ID is not used for software machine referencing; that is done by the machine object's file handle. Note also that the NeuroManager IDs for remote machines that might also be a host must be the same as that obtained by MATLAB using `getenv('HOSTNAME')` in order for NeuroManager to be able to detect that the host and remote are the same machine. This is not an issue for cluster queues since a cluster queue cannot be a NeuroManager host.

Observe a fundamental design feature of NeuroManager related to Machine ID: NeuroManager machine types act a bit like instantiations. That is, the `addMachine()` method of ***MachineSetConfig*** eventually adds a machine of that type to NeuroManager. Nothing in the software prevents the user from adding another machine of that type and using a different work directory. However, since the ID is automatically constructed from the type, there will be confusion when it comes to log notifications, since both machines will have the same ID. The reason this is done is because there is currently no good reason to add another machine of the same type, since one can just add more simulators to the original. Adding another machine of the same type just adds overhead with no other apparent benefit. So overall, in NeuroManager a machine type is also the instantation of that type.

In contrast, simulations are checked to ensure that their IDs, as assigned by the user, are unique. In similar contrast, simulators, since they must occupy the same remote directory structure, are assigned machine-unique IDs by the machine constructor.

3.10 Provisions for simulator version control

One of the principle goals of NeuroManager is to facilitate rapid yet tracked simulator evolution. In light of this, NeuroManager has several facilities for labeling versions of things. First, NeuroManager itself is

versioned within the ***NeuroManager*** class. Second, the Simulator class hierarchy approach makes it easy to spawn a new version of a simulator without affecting other members of the current tree. The SimpleSpike examples described below show this feature well. Third, the ***Simulator*** class declares an abstract property 'Version', a string. As seen in the examples, the user must declare 'Version' as a property in a new simulator subclass, and can update this instead of creating a new simulator class (or as a sort of subversion of the version created by the simulator subclass itself).

3.11 Some comments on MATLAB licenses

The host computer needs a MATLAB license/installation because the software runs within the MATLAB environment. The host does not require the MATLAB compiler. At least one remote machine has to have a MATLAB Compiler installation/license available. The ideal situation is where each remote has a MATLAB Compiler available. It will be used once per machine, since the compiled code is replicated for each Simulator on that machine.

Cross–compilation is used in the case where a remote machine does not have a MATLAB Compiler available. In this case, a different remote with the MATLAB Compiler (let's call it the 'MC' machine) can be used by making use of the ***XCompileMachine*** class as seen in the code. If we assume that the MC machine is also being used as a NeuroManager remote, then the MATLAB Compiler will be simultaneously being used, once for itself, and once for each cross–compiling machine (say that there are N of these). This will require 1+N MATLAB licenses for that short compilation time, since NeuroManager compiles all machines in parallel for speed. Within NeuroManager there is currently no provision for cross–compiling in series. If there is a licensing failure, NeuroManager will not operate properly and compilation will take too long, at which point an inspection of the redirection files on the cross–compilation machine will show licensing failure text. Current solutions are 1. obtain more licenses, 2. wait until more licenses free up, 3. and limit the machines that use cross–compilation.

NeuroManager's current approach to do a compilation for each machine may appear to be wasteful, but in practice it allows the use of different MATLAB versions at the same time, which is a decided advantage. Future versions of NeuroManager may provide more sophisticated handling of this issue in order to limit licensing requirements while minimizing setup time.

3.12 The SineSim example: simulators without model files

One of the simulator examples is a trivial simulator called 'SineSim' which is entirely MATLAB and which plots a sine wave based on two input free parameters, frequency and duration. SineSim is essentially NeuroManager's 'Hello, World!' simulation. A class called ***SimSineSim.m*** (a subclass of ***SimNoModelMLOnly***) and its companion remote-side function `userSimulation()` form the sine wave and plot it. Supplied scripts, **SineSimSet_SS0X.m**, show how to set up and run this simulation, which result in a MATLAB figure similar to Figure 19, in which the simid and machine ids have been used in the title. There are several parts to the SineSimSet scripts; the script files themselves have comments indicating the parts as labeled here:

Part I defines static data on the host via **myNMStaticData.m**, including three directories on the host that NeuroManager will use to locate files. 'CoreDir' is where the Core files are. 'LocalMachineDir' is where the user can override the LocalMachine directory added to the MATLAB path upon installation, since 'LocalMachineDir' will come before the other on the MATLAB search path. 'SSHLibDir' is where the SSH library files are located. **myNMStaticData.m** also locates the private key file. On Windows machines, there will be two keyfiles: typically the same key in two formats: one in OpenSSH format with no extension, and one in Putty format with a '.ppk' extension (see Section 2.12.2 above). We have kept these files in our 'LocalMachine' directory with good success. Finally, **myNMStaticData.m** defines the user notification data mentioned above in Section 2.3.4.

Part II defines the remaining directories; those that are basically run–specific. 'CustomDir' is where the user–defined **userSimulation.m** for the desired simulator is to be found, plus any supporting files, plus any input data files. One may also keep any custom simulator class files here too; it is a good idea to keep the

simulator class file and its associated userSimulation file together in 'CustomDir'. 'ModelDir' is where the simulator model files are to be found, if it has them. Finally, 'ResultsDir' identifies an existing directory where the user wants simulation results to be placed. We typically have just used the same directory as that chosen for CustomDir which has worked quite well.

Part III is the call to the ***NeuroManager*** constructor, which references the data above and also has optional parameters for 'NotificationsType' ('BOTH', 'SMS', 'EMAIL', 'OFF', or 'NONE' — defaults to 'OFF'), for 'PollDelay' (the delay between loops of polling the simulators for changes), and for 'LogEchoFlag' (whether to echo the log to the MATLAB command window ('true') or not ('false')). The PollDelay currently defaults to 20 seconds and can be between 10 and 300 — the user will have to experiment to see what works best locally. Note that the PollDelay numeric value is such, and not a string, and isn't surrounded by quotes. The constructor call also provides the 'UseDualKey' option which can be set to 'true' or 'false', as described in Section 2.12.2.

Part IV calls the ***MachineSetConfig*** constructor and adds several machines to it, each with the parameters 1) Machine Type, 2) Number of Simulators desired on that machine, and 3) the existing work directory to be used on that specific machine. 'Machine Type' tells what SimMachine the function refers to; for example 'MYSERVER01' indicates that specific SimMachine. 'Number of Simulators' means how many simulators to run on that machine at the same time using the job submission approach defined in the job submission superclass for that machine. For example, 'MYSERVER01' is subclass to the ***NoSubMachine***, so its job submission approach is background process. In our lab, we found that 3-4 simulators resulted in optimal throughput on that specific machine. A number of '0' simulators means not to use that machine at all. 'Work directory' is just that; this directory should be empty and chosen carefully because its contents will be cleared by NeuroManager before and after simulations are run.

This call creates the 'Machine Set' referred to in our paper.

Part V tests communications and file transfer with all the machines that have been added to the Machine Set Config and that have nonzero Simulators. This process is quick and highly recommended because of the dynamic nature of remotely managed computational resources.

Part VI builds the requested simulators of the type named in the first parameter on the MachineSetConfig named in the second parameter. This process includes file uploads and parallel MATLAB compilation on all machines, and can take a couple of minutes or more depending on the user's request. NeuroManager provides log messages that help determine the progress in this part.

Part VII starts NeuroManager's process of handing simulations to the pool of simulators formed from all the simulators requested in the Machine Set Config, and concludes when the last simulation has been run and successfully downloaded. The simulations are found in the named simspec file (see SimSet Specification above). Note that the Simulation Type found in the SimSpec file must match that supplied in Part VII or NeuroManager will flag an error and halt.

Part VIII removes the simulators from the Machine Set and cleans up all the remotes. *Side Note:* before this step is run the simulators are still available for running other SimSets; see Section 2.11.1 above.

Part IX ensures the remote machines are clear; shuts down the SSH agent (UNIX hosts only), closes out the log, and restores the former MATLAB path.

The SineSim example shows a case (SineSimRun04) where an input parameter is ill-formed, leading to an error in `userSimulation()` when it executes on the remote as part of the total **runSimulation** executable. In this case the simulation returns a FAILURE result and the SimSet Report will show the result. It is instructive to examine the files available to see how the error shows up for this class of error.

SineSim also provides an example of using the *Upload Simulation–Specific Data Files* Stage. The ***SimSineSim*** class overrides the definition of `defineSimulationInputDataFiles()` found in the ***Simulator*** class, so that instead of the file lists being empty, they now contain the name of a file to upload from the Custom Files directory and the name that it should have on the remote when it is placed in the simulation's input directory. Just to exercise this little demo, we use SineSim's `userSimulation()` to rename the file and copy it to the output for download. In a real scenario, however, `userSimulation()` would make use of the file as data for input to the simulation.

Another NeuroManager feature that the SineSim example demonstrates is the ability to

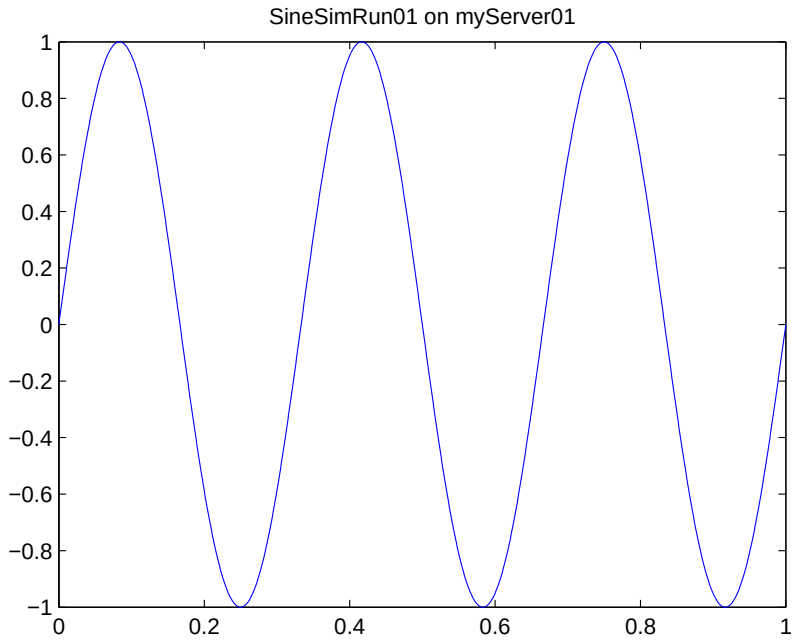

Figure 19: Example SineSim Output

attach a pdf file to user notifications. The class definition ***SimSineSim*** overrides the `postDownloadProcessingSimulatorSpecific()` method defined in the ***Simulator*** class. The override copies a pdf file produced on the remote to a name composed of the simid + 'AttachMe.pdf'. If notifications are turned on and if there is a file by that name in the simulation's host directory, that file will be attached to email and/or text messages upon completion of the simulation. This feature allows the user to see some results from the simulation without having to do anything but look at the text message or email.

Finally, the user will see that the SineSim directory also contains a file called **SimType.m**. This file defines only one type of simulator (SIM_SINESIM), and is an example of overriding the **SimType.m** file seen in the Core directory. The ***NeuroManager*** class adds the user's Custom directory to the MATLAB path after it adds the Core directory, so that the user's **SimType.m** will be seen before the one in Core. Labs that have a central installation of NeuroManager can use this feature so that the Core files don't have to be updated every time an individual researcher creates a new Simulator.

3.13 The GPUSim example

The GPUSim example is an example of using a simple MATLAB GPU operation within NeuroManager. We create a `gpuArray` from a time domain signal composed of two sinusoids, the frequency and amplitude of which are specified in the **GPUSimSpec.txt**. We plot the time signal and the spectrum on the remote. The remote also collects information on the GPU it is using; see the downloaded file named **GPUData.txt** in each simulation directory on the host. Note that the `userSimulation()` function checks for a GPU and acts so that the simulation fails gracefully with error message if none is present. NeuroManager does not check ahead of time to see that a given machine in the configuration has the proper GPU hardware; it is up to the user to ensure that the proper machines are chosen.

The GPUSim example is also a great place to see how to add a support function to **userSimulation.m**; in this case the file is **collectGPUData.m**, called from **userSimulation.m**. `collectGPUData()` was written so that it was easy to debug outside of NeuroManager (always a good policy). To use that support

file on the remote we simply added it to the ***SimGPUSim*** class as an additional custom file.

MATLAB GPU documentation (for v. R2014b) can be seen at http://www.mathworks.com/help/distcomp/gpu-computing.html.

References

Barrett DJ, Silverman RE, Byrnes RG (2005) SSH, The Secure Shell: The Definitive Guide. O'Reilly Media, Inc.

Miyasho T, Takagi H, Suzuki H, Watanabe S, Inoue M, Kudo Y, Miyakawa H (2001) Low-threshold potassium channels and a low-threshold calcium channel regulate Ca2+ spike firing in the dendrites of cerebellar Purkinje neurons: a modeling study. Brain Research 891(1-2):106–115