__init__.py File Used For Simulation Module

Introduction to Modular

Curtis Ogle

August 8, 2014

1 Introduction

Modular is a python package which provides a framework for arbitrary simulation. It provides a collection of features for use with simulation code written in compliance with the modular_core simulation module specification (found below.) The major features of this framework include a GUI as well as an underlying API accessible without the use of a GUI, multicore processing, hyperdimensional parameter sweeping, post processing analytic routines, fitting routines, several useful forms of output including a plotting window provided by Modular, a simple and flexible pipeline for parsing simulation configuration parameters of an ensemble, and intelligent data handling to expedite simulation workflows and avoid memory limitations. This is a guide for various use cases of Modular from installation to adding new simulation modules with all of the above features.

2 Acquisition

The primary python package associated with Modular is "modular_core". This package can be found at:

http://github.com/ctogle/modular

To begin using Modular, the user will likely also want the "stringchemical" module which performs Gillespie simulations. This package can be found at:

http://github.com/ctogle/stringchemical

3 Installation

Before installing Modular, it is necessary to install python and various dependencies (python packages) which are used by Modular. The installation process

for python and python packages will vary for various computers, so a list of packages is provided below which the user is responsible for installing.

python 2.7
appdirs
numpy
scipy
pyside
matplotlib
pyopenGL

Once these are installed, packages associated with Modular can be installed. The installation process is identical for modular_core and simulation modules. Run the following command in the "src" folder of each package after downloading them from the links provided in the "Acquisition" section:

python setup.py build install –user

4 Testing

After modular_core and the stringchemical simulation module have been installed it is a good idea to run the test script found in the "tests" directory of each package. Use the following command in the "tests" directory for modular_core:

python test_ensemble.py

This should produce a series of "*.pkl" files in the "tests" directory which can be viewed using the "plot_pkls.py" script also found in this directory using the command:

python plot_pkls.py

5 Running Modular

When modular_core has been installed, Modular can be started by using the following command in a directory containing the "modular.py" script:

python modular.py

5.1 Using The GUI

Simulations are handled by an "ensemble" object within Modular. The GUI is intended to provide the necessary controls to manage multiple ensembles and the various parameters that define them. The first thing to do once modular.py has been called is to add an ensemble. Each ensemble is assigned one simulation module which it uses with the various features of Modular. When only one simulation module has been installed, the ensemble will default to this module. If more options are available, a dialog will prompt the user to select a simulation module.

After the ensemble is made, a new tab will appear which corresponds to the new ensemble which was just added. A configuration file can now be parsed. Configuration files for ensembles carry the file extension ".mcfg". Clicking the "Parse mcfg File" button will prompt the user to select such a file. When the file is chosen, it will be parsed automatically. For simplicity, assume that in this context the chosen configuration file contains all the information the ensemble needs to run properly. Now clicking the "Run Ensemble" button should run whatever simulations were defined in the mcfg file.

5.2 Using The API

In the above section, we made an ensemble object, parsed an mcfg file, and ran it. This is very easy to accomplish in python code by using the modular_core API. The following code snippet demonstrate this.

```
python
import modular_core.libsimcomponents as lsc
ensem_manager = lsc.ensemble_manager()
ensem = ensem_manager.add_ensemble()
mcfg = '/path/to/a/file.mcfg'
did_run = ensem.run_mcfg(mcfg)
if did_run: ensem.produce_output()
else: print 'ensemble failed to run!'
```

Contrary to when using the full GUI, using the API requires an explicit call to the produce_output method of the ensemble. The add_ensemble method of the ensemble class also accepts the keyword "module" which refers to which simulation module the new ensemble should use. The default simulation module is the gillespie simulator.

```
<end_criteria>
    time limit : 800

<capture_criteria>
    scalar increment : 20 : time

<variables>
    mu : 10.0
    K : 0.000000000000001
    lambda1 : 5
    lambda2 : 5
    gamma : 0.01

<functions>
    g : mu/(K + x1 + x2)

<reactions>
    nothing lambda1 -> 1 x1 : formation of x1 (rate is lambda1)
    nothing lambda2 -> 1 x2 : formation of x2 (rate is lambda2)
    1 x1 gamma -> nothing : dilution of x1 (rate is gamma)
    1 x2 gamma -> nothing : dilution of x2 (rate is gamma)
    1 x1 g -> nothing : degradation of x1 (rate is g1)
    1 x2 g -> nothing : degradation of x1 (rate is g2)

<species>
    x2 : 10
    x1 : 10

<plot_targets>
    time
    x2
    x1

<parameter_space>
    <product_space> 100
        lambda1 : value : 2.0-8.0;4.0
        lambda2 : value : 2.0-8.0;4.0
        mu : value : 8.0-12.0;2.0

<post_processes>
    x1, x2 correlation : correlation : 0 : x1 and x2 of time : 5 : ordered
    slices : slice from trajectory : 1 : all : -1
    reorg : reorganize data : 2 : all

<multiprocessing>
    workers : 8

<output_plans>
    3 : - : reorg_output : pkl : all
    1 : - : correlation_output : pkl : all
    2 : - : slices_output : none : all
    0 : - : ensemble_output : none : all

<ensemble>
    multiprocessing : on
    mapparameterspace : on
    fitting : off
    postprocessing : on
    trajectory_count : 100
```

Figure 1: Valid mcfg for chemical simulation module

6 Usage Of .mcfg Files

One key feature of the mcfg system in Modular is that every parameter needed to control how an ensemble is run can be parsed from an mcfg file. In general, neglecting a block of information should not cause problems but may lead to additional time spent configuring parameters in the GUI. Figure 1 is an example of a valid mcfg file for use with the gillespie simulation module, “chemical”.

6.1 Structure And Contents

Each block of information in an mcfg consists of a header, which is an expression appearing as “⟨header⟩”, and the lines of information up until the next header or the end of the file. The headers are either associated with features

of modular_core or with features of the simulation module for which the mcfg is expected to work. In the above example, the headers 'variables', 'functions', 'reactions', and 'species' pertain to features of the chemical simulation module. The remaining headers pertain to featuers of modular_core.

The first and second such headers are 'end_criteria' and 'capture_criteria'. These encode information for the conditions of ending a simulation and capturing data during simulation respectively. Not every simulator will make use of these but the infrastructure is present for arbitrary conditions for simulation end and data capture. A module can easily define and use subclasses of the criterion class provided by modular_core to provide arbitrary conditions for whatever the simulation requires.

The next header which associates with modular_core is 'plot_targets'. This declares which data should be collected and returned from each run of the simulation. In simulations with many possible plot targets, neglecting to capture data which won't be used later can have a profound effect on performance. It's best to include as few plot targets as are necessary.

Next is the 'parameter_space' header. This is used for both parameter sweeping and fitting routines, the latter of which is not demonstrated in the provided mcfg example. A parameter space consists of some number of axes, corresponding to parameters of the simulation, the values which are used for those axes, how the axes are combined to generate the various sets of parameters, and how many simulations should be performed for each set of parameters. The first of these which is specified is how the axes are combined. The header 'product_space' will tell the parser to create the parameter space using the cartesian product of all possible parameter values in the space. The alternatives to this are 'zip_space' and 'fitting_space'. The zip_space option will create a set of parameters by assuming the variations on each axis are one to one. It will then create the first set by using the first value on each axis, the second set using the second value on each axis, and so forth. The fitting_space option specifies that the parameter space is to be used by a fitting routine. In this case the information under the header defines the boundaries of the valid space in which to perform fitting. Adjacent to this header is the integer '100' which specifies the number of simulations to perform with each set of parameters. The remaining lines in this block specify the axes invidually.

The 'post_processes' block defines the post processes which are run on the data from the simulation. Each line represents a separate post process. The order in which post processes are specified matters as the output of one post process may be input to another.

The 'multiprocessing' block allows the user to specify the number of worker processes to utilize when multiprocessing is enable for the ensemble. The number of workers which is optimal depends on the computer being used.

The 'output_plans' block defines the output associated with the simulation, each post process, and each fitting routine. The first piece of information for each output plan is an integer identifying with which process the output plan is associated. Zero always corresponds to the simulation. If there are m post processes, then the integers 1 to m will correspond to each post process, in the order in which they are specified. The remaining output plans correspond similarly to each fitting routine. That is, if there are m post processes and n fitting routines, the fitting routines will utilize the range m + 1 to m + n in the order they appear in the mcfg file.

The final header corresponds to parameters of the ensemble object. The parameters specified here will directly affect how the ensemble is run. For instance, if the option 'mapparameterspace' is off the ensemble will not sweep parameters, whether or not a parameter space is specified or not. When parameter space mapping is enabled the data will have a different structure. Post processes, output plans, and multicore processing will all be aware of this and behave differently. For example, when a post process is run on data which was not generated by mapping a parameter space, there will be one plot to output for the analysis of the simulations run on one set of parameters. If the parameter space is mapped, then the post process will analyze the data associated with each unique set of parameters and provide data to output for each parameter set.

6.2 Generation Using The GUI

One can also generate an mcfg from the parameters specified in the GUI by using the 'Generate mcfg File' button. This is useful when the user does not know how to write an mcfg file.

7 Supporting New Simulation Modules

A simulation module is accountable for two things. It must be capable of performing one simulation and returning the relevant results and it must be capable of interfacing with modular_core. The first of these is the responsibility of the developer of the simulation module. The second of these is accomplished by complying with the simulation module specification which follows.

7.1 Loading/Unloading Simulation Modules

Once a simulation module has been installed as a python package, it is necessary to inform modular_core that the module is available for simulation. Use the following command to achieve this:

python modular.py –modules

```
1  #!/usr/bin/python2.7
2
3  import scripts.chemicallite as main
4
```

Figure 2: Contents of the __init__.py file found in the gillespie simulation module's top level directory

This will initiate a command line interface for loading and unloading simulation modules with respect to modular_core.

7.2 Modular's Simulation Module Specification

Simulation modules used with Modular must be installable as python packages. This typically means there is a folder which contains all source code for the package. This discussion will be about the contents of this folder. The source code folder should contain a directory whose name will be the top level namespace of the module. This is specified as a package for installation in the setup.py script. For the gillespie simulation module, this folder is called "chemical". Within this folder or its subfolders, there should be a python file which interfaces with modular_core. For the gillespie simulator, this file is called "chemicallite.py". This file is special to the simulation module and must be specified in the __init__.py file of the package as such. The contents of the __init__.py file for the gillespie simulator are shown in figure two. Importing chemicallite as the namespace "main" tells modular_core that this file interfaces the module with modular_core.

For this discussion we will refer to the file analogous to chemicallite.py as the "entry point" for a simulation module. The entry point for a simulation module must contain the attribute "module_name", and this name should match the name of the package which is the same as the top level namespace. The entry point also must contain a class called "sim_system" which should inherit from either of the two following classes:

import modular_core.libsimcomponents.sim_system
import modular_core.libsimcomponents.sim_system_external

An instance of this class will be created for every trajectory of the ensemble. It is responsible for running the simulation and returning data. In particular, the "data" attribute of the class is what will be used by the rest of modular_core. Thus as the end of the simulation, this attribute should contain all necessary data for plotting and future calculations.

To see an example of a very minimal simulation module which complies with modular_core, download the dummymodule simulation module from:

http://github.com/ctogle/dummymodule

This module makes very little use of the underlying features of modular_core.

For a more complicated example which uses basically all of the underlying features of, look at the "src" folder found in the stringchemical module at:

http://github.com/ctogle/stringchemical

8 Conclusion

Modular was written with extensibility and collaboration in mind. If there is any interest in new core features, simulation modules, or bug fixes please do not hesitate to say so:
cogle@vt.edu