StochKit2 Developer's Guide

This document is intended to help developers extend StochKit2 or incorporate StochKit2 into other software packages.

Table of Contents

Introduction

StochKit2 is the first major update to the popular StochKit software package. StochKit2 provides command-line executables for running stochastic simulations using variants of Gillespie's Stochastic Simulation Algorithm and Tau-leaping. Enhancements in StochKit2 include:

- Improved solvers including efficient implementations of the SSA Direct Method, Optimized Direct Method [Cao et al. 2004], Logarithmic Direct Method, a Constant-Time Algorithm [Slepoy et al. 2008], and an Adaptive Explicit Tau-leaping method
- Automatic parallelism
- Event-handling for the Direct SSA
- An updated command-line interface that is more user-friendly and provides more options
- Improved support for extending StochKit functionality

StochKit2 is intended for two audiences: 1) practicing scientists who wish to study the behavior of their biochemical models by running stochastic simulations, and 2) software developers who wish to extend the functionality of StochKit or incorporate StochKit into their software.

This document is targeted toward audience 2 above (Developers). This document is provided as a guide only and is not comprehensive. Readers of this guide should be familiar with the StochKit2 Manual, especially the Glossary of terms.

Users in audience 1 above looking for instructions on how to use StochKit2 should see the StochKit2 Manual (StochKit2_manual.pdf) included with the StochKit2 distribution.

A note to Windows users: This document is targeted primarily toward the Linux/UNIX/Mac OS X version of StochKit2. Developers using Windows should work within the Visual Studio framework, using the existing projects as examples. An empty custom driver template is provided in custom_drivers\custom_project\empty_project.

Overview

The remainder of this document will be divided into 4 main sections:

1. Adding a feature to the existing StochKit2 drivers
2. Creating a new solver
3. Creating a new driver
4. Incorporating StochKit2 into additional software packages

Adding a feature to the existing StochKit2 drivers

StochKit2 has two primary drivers: "ssa" and "tau_leaping". The executables in the main StochKit directory are scripts which first set an environment variable named "STOCHKIT_HOME" to the path of the StochKit directory, then call the driver program executable of the same name (e.g. "ssa" or "tau_leaping") in the bin directory. The ssa driver (src/drivers/ssa.cpp) examines the model and chooses which particular ssa executable to call (e.g. ssa_direct for the direct method implementation of ssa, etc.). There are various executables with different features: "_mixed" for models with custom propensities that must be compiled, "_small" uses dense stoichiometry vectors for small models, "_serial" runs on one processor and is called by the non-serial driver. NOTE: to run one of these executables directly, you must set an environment variable named STOCHKIT_HOME to the path to your StochKit directory.

These executables all take a model, simulation time, number of realizations, and number of intervals (among other things) as parameters. The "number of intervals" determines how many times during each realization that output should be recorded. A value of 0 (zero, the default) means data will only be stored at the end time of each realization. A value of 1 means data will be stored at the initial time and end time of each realization. A value of 2 means data will be stored at the start time, at end time divided by 2 and the end time, etc. So, data is stored at "number of intervals" + 1 evenly spaced time points.

To change the behavior of the existing StochKit2 drivers, you typically want to modify one or more of the following files:

- src/drivers/**ParallelIntervalSimulation.h** and .cpp
  The drivers (see src/drivers/ssa_direct.cpp for example) themselves are very simple. They create an instance of the ParallelIntervalSimulation class named "parallelDriver", call parallelDriver.run and parallelDriver.mergeOutput. The run method calls the appropriate serial driver once for each processor available on the system. A lot of work is done to pass the appropriate parameters to the serial driver (see the section on CommandLineInterface below). The mergeOutput method takes the output from the serial runs, combines it and writes the final output to file.
- src/drivers/**SerialIntervalSimulationDriver.h**
  The serial drivers (see src/drivers/ssa_direct_serial.cpp for example) are also simple. They create an instance of the SerialIntervalSimulation class named "driver", call the appropriate "create solver" and "call simulate" methods, and then call a "write output" method. The implementation of the SerialIntervalSimulationDriver class ensures that, if the compiled serial driver executable is called correctly, the serial driver will create the output files that the parallel driver needs to merge together to construct the final output.
  The output from the serial driver is typically written to a hidden directory named <output directory>/.StochKit/.parallel/
  Messages (errors and warnings) generated by the serial driver are typically written to log files in the hidden <output directory>/.StochKit/ directory when called by the parallel driver.
- src/utility/**CommandLineInterface.h** and .cpp
  The CommandLineInterface class uses the Boost program_options library. The constructor sets up the options (note that there are "visible" options that appear when running with --help and "hidden" options that are not intended to be called by users directly). The other important method is "parse" which parses the command line parameters and sets the CommandLineInterface instance's member values appropriately. Accessor methods are provided and used by the parallel and serial interval simulation drivers.
- src/output/**StandardDriverOutput.h**
  The standard driver output class, like the other output classes has two important methods: initialize (which must be called at the beginning of a simulation) and record (which is called at each interval for which data is to be

recorded). The standard driver output class also has methods to set what type of data to keep (e.g. stats or histograms) and for writing output to file (e.g. writeMeansToFile). Finally, the standard driver output class has a "merge" method to combine the output of multiple standard driver output objects. The merge function sometimes depends on additional helper files that were output during the simulation.

A typical modification might require adding a command line option by modifying CommandLineOptions, handling that option in both ParallelIntervalSimulation and SerialIntervalSimulation, and adding code in StandardDriverOutput to create additional output files. NOTE: some files contain sections wrapped in #ifdef MIXED, #ifdef EVENTS, or #ifdef WIN32 for code that deals with customized propensity models, models with events, or Windows-specific changes, respectively.

Creating a new solver

StochKit2 offers several solvers including the SSA "direct method", "logarithmic direct method", "optimized direct method", and "constant complexity method [of Slepoy et al.]" as well as an adaptive explicit tau-leaping method. To implement a new algorithm, follow these steps:

- Copy an existing solver implementation (e.g. SSA_ConstantTime.h and .ipp) and rename them.
- Modify the code as appropriate including rewriting the "selectReaction" and "fireReaction" methods. You will have to create additional data structures and helper functions as necessary. Update the "initialize" method which will be called at the beginning of each realization. There should be no need to update the simulate method.
- Create new drivers. Use an existing solver as a guide. For example, bin/tau_leaping determines properties of the model to then call tau_leaping_exp_adapt or tau_leaping_exp_adapt_mixed which then calls tau_leaping_exp_adapt_serial.
- Update the Makefiles. See Makefile and src/Makefile. Use existing drivers as a guide. For example, ssa_constant*. To create the dependencies for your make commands, see the files in hidden directory .make. Note that the process for handling "mixed" models is complicated because the executable first writes the custom propensity functions to a file, then compiles the code in that file to create a new executable, then calls that executable (see src/model_parse/Input_mixed_before_compile.h and Input_mixed_after_compile.h). See the subsection on "Model files and custom propensities" in the "Additional considerations" section below for more details.

Creating a new driver

If you need functionality that is different from the standard interval ensemble simulation that StochKit2 provides, you will have to write your own custom driver.

StochKit2 provides a custom driver example in the "custom_drivers" folder. This custom driver runs a single trajectory and outputs the state after each reaction up to an end time or until a maximum number of reactions has fired.

The most complicated aspect of this custom driver is the handling of "mixed" models. In single_trajectory.cpp, when we discover that the model contains customized propensities, we make a system call to "make single_trajectory_compiled". A look at the Makefile reveals that this call compiles single_trajectory_mixed.cpp. Some important points to note are that the file "CustomPropensityFunctions.h" is included (this file is created by single_trajectory.cpp by calling writeCustomPropensityFunctionsFile), and "-DMIXED" is used which leads to some code being included in some of the utility programs. Also, the files included by the Makefile require that the STOCHKIT_HOME environment variable be set (this is why the main executables are all shell scripts--they first must set the STOCHKIT_HOME environment variable, and then call the driver program).

Note: the custom driver example does not utilize multiple processors.

Incorporating StochKit2 into other software packages

It is expected that StochKit2 would be incorporated into other software packages by either: utilizing the executables provided with StochKit2 directly or by writing custom drivers and/or solvers.

If the StochKit2 executables cannot be used directly, it may be possible to just define a new output class (that implements initialize and record) if the standard interval ensemble simulation methods in the solvers can be used.

See the above sections on creating custom solvers and drivers. See also the next section on additional considerations for custom StochKit2 code.

Additional considerations

Model files and custom propensities (i.e. "mixed models")

The model file format is described in detail in the StochKit2 Manual.

From a developer's perspective, the important files for handling model files are in the src/model_parser directory. This directory contains classes for parsing the model file and creating the data structures (initial population, stoichiometry matrix, propensities functor, and dependency graph) for the solvers. It also contains a useful helper class named ModelTag that is used to identify important characteristics of the model.

When a model file contains a "mixed model" (reactions where Type is "customized"), the standard drivers first identify that it is a mixed model using the helper ModelTag class. The model is then parsed using the Input_mixed_before_compile class in the StandardDriverUtilities::compileMixed method. After parsing the model file, compileMixed writes a custom propensity functions file that contains C++ code for the custom propensities, then compiles that code into a new executable by calling "make <executable name>" (where <executable name> is passed to compileMixed and must have a corresponding line in the Makefile. See the line for ssa_direct_mixed_compiled in src/Makefile. Note that the generated code directory is passed as a parameter to make and is used to determine which CustomPropensityFunctions.h file to include and where to save the compiled executable). Also, "-DMIXED" is specified which is used by some utility programs. After the new executable is created, the mixed parallel driver calls the newly generated executable like it would any serial driver and merges the output.

MATRIX_STOICHIOMETRY and template classes

Many classes in StochKit2 are template classes which allow them to operate on any input type, as long as that type provides the interfaces (methods) that get called by the template class. In most StochKit2 drivers, the stoichiometric matrix is implemented as a (standard library) vector of (dense or sparse Boost uBlas) vectors. However, tau-leaping requires matrix-vector multiplication. To facilitate this functionality, tau-leaping is compiled with "-DMATRIX_STOICHIOMETRY" which defines a macro variable named "MATRIX_STOICHIOMETRY". When MATRIX_STOICHIOMETRY is defined, the stoichiometry type is a Boost uBlas compressed_matrix. Since the compressed_matrix class provides different functionality from a standard vector of uBlas vectors, most StochKit2 solver classes have preprocessor directives to check if MATRIX_STOICHIOMETRY defined which then includes the correct code as necessary.

Similar issues can arise when creating custom code in StochKit2 that uses data structures that aren't compatible with the expectations (interfaces) of the existing code. To get around this issue, programmers can insert preprocessor directives like with MATRIX_STOICHIOMETRY or write wrapper classes around their data structures to conform to the interfaces.

Propensities classes

The StochKit2 solvers take a "_propensitiesFunctorType" as a template argument. From a solver's perspective, the propensities class is expected to have an overloaded function call operator that takes a std::size_t reaction index and a _populationVectorType population as parameters and returns the propensity as a double. The standard driver propensities type is (usually) CustomPropensitySet which is basically just a vector of CustomSimplePropensity objects. A CustomSimplePropensity can be used for either a mass action reaction by setting the propensity PropensityFunction to one of the propensityX member functions or for a custom reaction by setting the propensity PropensityFunction to point to some other function.

Libraries

The version of the Boost library that appears in StochKit2 is Boost 1.42 that has been modified by deleting many of the files that are not used by StochKit2.

Histogram data file format

The histogram .dat files generated by the --keep-histograms option are plain text files that have the following format:
Line 1: <species id> <output time> <species index> <output time index>
Line 2: <lower bound> <upper bound> <bin width> <number of bins> <1/(bin width)>
Line 3: <data>

where <lower bound> is the minimum value that would be counted in the leftmost bin (note actual count in that bin may be zero), <upper bound> is maximum value that would be counted in the rightmost bin.

License

Use of StochKit2 is subject to the license agreement.

Contact information and bug reporting

The StochKit team welcomes your feedback on the software. Please send comments, questions, enhancement ideas, and suspected bugs via email to StochKit@cs.ucsb.edu. For bug reports, please include the error message, operating system and version of StochKit you are using.

References

1. Slepoy, A., Thompson, A.P., and Plimpton, S.J. J. Chem. Phys. **128**, 205101 (2008).