

Short Introduction to SimElements 1.0

This is a short introduction to *SimElements*. Together with the docstring documentation it is intended as a kind of User's Guide.

Dynamic/Continuous Simulation

The basis for formulating and solving dynamic simulation problems is the *Dynamics* class (in the *dynamics* module). Objects created with *Dynamics* enable you to solve systems of ordinary differential equations (ODEs) according to the "stock-and-flow" formalism. There are two options: you may choose to let the stock or state vector be described using a dictionary or using a list (or a Python 'd' array). The dictionary option allows for more clear, text string identifier-based coding and is also faster than the traditional list approach. The list option may be preferable for large systems with large coefficient matrices (use the *Matrix* class of the *misclib/matrix* module for that!). Cf. the docstring documentation of the *Dynamics* class for details.

There are a number of ODE solver methods to choose from, two of which use adaptive time stepping. One of the methods - the matrix exponential - only works with the list option.

One of the major reasons for using dynamic simulation is the possibility to see what happens when a dynamic system is subjected to a abrupt (discontinuous) change or "shock". The *dynamics* class offers the *tiptoe* method for carrying the ODE solution to a time point immediately before the time when a drastic change occurs and then carry on with the solution from the exact time point of change with the status immediately before the change as a new set of initial conditions from the point of change - consult the docstring documentation of the *Dynamics* class for details. This feature is also valuable in combined continuous-discrete simulation. *SimElements* provides the user with a set of functions that may be used to describe discontinuous transients to which a dynamic system may be exposed (may be found in the *numlib/singfunc* module).

Stiff ODE systems may require implicit solvers. A number of methods for solving stiff systems may be found in the *StiffDynamics* class in the *stiffdyn* module. *StiffDynamics* is a subclass of the *Dynamics* class and inherits everything from it. Implicit solvers offer great stability but less good accuracy, generally speaking, and are much slower than the implicit

ones. Many of the solvers of the Dynamics class handle stiff equations very well, but there may be situations when an implicit solver must be used. The solver methods of StiffDynamics only work with the list (or 'd' array) option, but conversions may be used that will enable you to use the more text string-friendly dictionary option anyway, cf. the *stiffdict.py* file in the demo set.

The *StochDynamics* class (a subclass of the basic Dynamics class) in the *stochdyn* module allows you to add noise in the form of Wiener processes and generalized Poisson jump processes to your problem - see the docstring documentation.

Finally: one of the most important features of dynamic modeling is the possibility to introduce *delays*, so there is a module with that name. It contains two classes: one for setting up exponential delays (called *ExpDelay*), and one for setting up step delays or dead time delays (called *StepDelay*). Consult the docstring documentation of the dynamics and delay modules for details. The demo *expdelaydemo.py* shows how to impose exponential delays on a dynamic system described by ODEs.

You may be surprised at the large number of solver methods; the reason for the fairly large number is the prospect of running a dynamic/continuous simulation inside a Monte Carlo lopp - for uncertainty analysis purposes, for instance - when speed, accuracy and stability will have to be balanced against one another.

A very simple demo example showing how to use the basic features of SimElements for dynamic simulation may be downloaded from the SimElements web page as *host_parasite.py*.

Discrete Event Simulation

The basic module to use for discrete event simulation problems is the *eventschedule* module with the *EventSchedule* class. This implies that the basic approach to discrete event simulation of SimElements is the traditional event schedule or event queue approach. This approach is very general and may be used for any type of discrete simulation problem formulation. Objects punched out with the EventSchedule class use a heap for the event times and a dictionary for the corresponding events with the event times as keys. There may be situations when these data structures will limit the usability. For that purpose the *EventScheduleStack* class (from the *eventschstack* module) may be used where the event times and events are SimElements stacks (cf. the *misc/lib/stack* module) with the full

capabilities of the Python 'list' data structure. EventSchedule is normally faster, however, and should be used when possible.

SimElements also has a module for handling physical queues including methods for handling queue priorities, queue discipline, gathering of statistics etc.: the *line* module containing the *Line* class. It inherits everything that is in the EventSchedule class, so you may use Line directly for queues. The line module has a companion just like the eventschedule module: there is a *linestack* module (containing the *LineStack* class) to use when Python's 'list' capabilities are needed. (Line and LineStack both inherit from the ABCLine abstract base class, but don't bother with that, use Line or LineStack).

No discrete event simulation is possible in practice without Monte Carlo (deterministic problems rarely require simulation...). The basic modules are *genrandstrm* and *cumrandstrm* with their classes *GeneralRandomStream* and *CumulRandomStream*, respectively. They are designed to crank out streams of random numbers from a large set of probability distributions. The rule is one stream per instance. GeneralRandomStream is a multi-purpose random stream generator that may be used that is suited for sampling service times, consumption rates and other random type parameters. CumulRandomStream is specially designed for providing random number streams cumulatively and is preferably used for sampling arrival times and similar stuff - it keeps track of cumulative time internally. GeneralRandomStream may of course also be used for that but only for the increments - the calling program unit must keep track of cumulative time.

There is also an *invrandstrm* module with a class *InverseRandomStream* where all random number generation methods are based on the inverse of the cdf. The corresponding methods provided by GeneralRandomStream are faster and InverseRandomStream does not normally have to be used for discrete event simulation, but there are situations associated with uncertainty and sensitivity analysis where it performs better or is even required, cf. below. (GeneralRandomStream and InverseRandomStream both inherit from the abstract base class ABCRand, but you cannot use ABCRand in a standalone fashion).

SimElements does not provide any methods for variance reduction that are particularly suited for use with discrete event simulation - don't use Latin Hypercube Sampling from the *randstruct* module to generate arrival times or service times! Antithetic sampling (the same module) may be used - with care not to introduce unwanted correlations - but its efficiency is questionable for discrete event simulation.

The *statlib/stats* module provides a number of tools for the statistical post-processing of the raw output from a discrete event simulation, including several functions for bootstrapping. A couple of very basic methods for creating application-tailored bootstrapping procedures may be found in the *RandomStructure* class in the *randstruct* module.

A basic demo example showing how to set up a simulation of a M/M/1 queue (single-server, Poisson arrival, exponentially distributed service time) may be found in *mm1queue.py*.

Combined Continuous-Discrete Simulation

SimElements has the capabilities for continuous/dynamic as well as discrete event simulation. A modern programming language like Python provides excellent possibilities for combining the two, and SimElements does not add a lot of premeditated features for that end, even if the *tiptoe* method of the *Dynamics* class may be regarded as one. There is one module, however (inspired by GASP IV): the *crossing* module containing the *Crossing* class for handling external and internal state-events as well as scheduled and unscheduled time-events.

Uncertainty (and Sensitivity) Analysis

The random number generator classes and methods and the statistical post-processing functions of SimElements make it ideal for uncertainty analysis. SimElements also provides the analyst with a couple of methods for variance reduction: Latin Hypercube Sampling (LHS) and antithetic sampling, contained in the *randstruct* module with the *RandomStructure* class. LHS and antithetic variates both require that the basic generation of random variates be based on the inverse of the cdfs from which samples are drawn. The *InverseRandomStream* class accepts a stream of [0, 1] random numbers generated by the *RandomStructure* methods *lhs_sample* or *antithet_sample* as an input in place of the traditional seed used internally for generating random variates, cf. the docstrings documentation for details.

The ultimate aim of variance reduction methods is to reduce CPU time. Some of the generation methods of the *InverseRandomStream* class are notably slower than the corresponding methods of the *GeneralRandomStream* class, so great care must be taken not to increase the CPU time needed for a given statistical precision when using LHS or antithetic variates...

There are a couple of other situations when the use of synchronized random variate generation based on the cdf inverses is particularly desirable. One is the "common random numbers" sensitivity analysis setup for studying the effects of changing the type of input distributions or the values of the parameters of input distributions. The use of inverse-based generators will be much more efficient since it makes it possible to base the statistics of the resulting differences on a set of direct pair-wise realization-by-realization comparisons.

Another call for inverse-based random variate generation occurs when there is a need for introducing correlations among the input parameters - when the input parameters are not normal. SimElements' GeneralRandomStream class has a method for cranking out pair-wise correlated normal random variates. The RandomStructure class has a method for putting out batches of intercorrelated normal variates with a given correlation matrix. But when the distributions are other than the Gaussian, then Latin Hypercube Sampling imposing rank correlations among the parameters is the only practical alternative. And LHS requires inverse-based random variate generation!

statlib/stats may of course be used for the post-processing of results from uncertainty analyses as well!

See the *LHStest.py* and *antithest.py* demos for examples on how to get Latin Hypercube Sampling and antithetic variates going.

Optimization

SimElements offers some support for optimization as well. The *nelder_mead* function present in the *findmin* module is a Python implementation of the robust Nelder & Mead "downhill simplex method" (cf. the docstrings documentation). It should be remembered that optimization based on the results from a Monte Carlo procedure is a challenge to any optimization algorithm and that it requires small sampling variances in order to be reliable.

Miscellany

The supporting subpackages of SimElements contain some stuff that may be useful besides their being called by the classes at the top level of the package.

The *machdep/machnum* module contains a set of numerically related constants the values of which may be implementation-dependent. You

should run the auxiliary *machineparams.py* and the externally available 'machar' script before using SimElements and change the numerical values in the file *machdep/machnum.py* if necessary.

The *misclib* subpackage contains a number of modules introducing specially designed data structure classes (modules *deque*, *heap*, *matrix* and *stack*) that complement Python's built-in data structures. *misclib* also contains a module for user-specific handling of errors and warnings (*errwarn*), a module (*inandout*) for reading from and writing to tab-delimited files, a module with functions for manipulating iterable data structures (*iterables*), a module containing mathematical constants (*mathconst*), and a module (*numbers*) for assigning, checking and manipulating numbers.

The *numlib* subpackage contains a module (*matrixops*) for manipulating matrices punched out by the *Matrix* class defined in the *misclib/matrix* module, a module (*quadrature*) containing a function for Romberg integration, a module for equation solving (*solveq*), and a module for the numerical inversion of Laplace transforms (*talbot*). *specfunc* and *singfunc* contain functions representing functions with singularities and functions that need special numerical procedures for their computation, respectively.

Besides functions for post-processing of simulation results, the *statlib* subpackage also has some modules for computing the values of pdfs, cdfs and inverse cdfs (named *pdf*, *cdf* and *invcdf*), and a module dealing with binomial coefficients (*binco*).

The *xbuiltins* subpackage contains modules with functions that extend the capabilities of Python's built-in 'abs' and 'map' functions.

Finally:

The SimElements package may be used directly without recompiling and relinking of the Python source code. You may place the source directory anywhere on your computer. You will have to tell Python where to find the simelements directory, however, by appending the path name to Python's built-in sys.path list, such as:

```
from sys import path    # Slashes or backslashes or whatever...
path.append('<Full path name>/simelements')
```

Good luck!