Common Model Library

written for

Antares Simulation Project

Model Documentation

MonteCarlo Model

Gary Turner

Odyssey Space Research

2020

MonteCarlo Model

Table of Contents

Revision History

Version	Date	Author	Purpose
1	April 2020	Gary Turner	Initial Version
2	March 2021	Gary Turner	Added verification

1 Introduction

The MonteCarlo Model is used to disperse the values assigned to variables at the start of a simulation. Dispersing the initial conditions and configurations for the simulation allows for robust testing and statistical analysis of the probability of undesirable outcomes, and measuring the confidence levels associated with achieving desirable outcomes.

Conventionally, most of the time we think about dispersing variables, we think about apply some sort of statistical distribution to the value. Most often, that is a normal or uniform distribution, but there may be situations in which other distributions are desired. In particular, this model provides an extensible framework allowing for any type of distribution to be applied to any variable.

For extensive analysis of safety-critical scenarios, where it is necessary to demonstrate high probability of success with high confidence, traditional MonteCarlo analyses require often many thousands of runs. For long duration simulations, it may not be feasible to run the number of simulations necessary to reach the high confidence of high success probability that is necessary to meet requirements. Typically, failure cases occur out near the edges of state-space, but most of the runs will be "right down the middle"; using conventional MonteCarlo techniques, most of these runs are completely unnecessary. With a Sequential-MonteCarlo configuration, a small number of runs can be executed, allowing for identification of problem areas, and a focussing of the distribution on those areas of state-space, thereby reducing the overall number of runs while adding complexity to the setup. While this model does not (at this time) provide a Sequential-MonteCarlo capability, the organization of the model has been designed to support external tools seeking to sequentially modify the distributions being applied to the dispersed variables, and generate new dispersion sets.

2 Requirements

1. The model shall provide common statistical distribution capabilities, including:
   (a) Uniform distribution between specified values
      i. as a floating-point value
      ii. as an integer value
   (b) Normal distribution, specified by mean and standard deviation
   (c) Truncated Normal Distribution, including
      i. symmetric and asymmetric truncations
      ii. it shall be possible to specify truncations by:
         A. some number of standard deviations from the mean,
         B. a numerical difference from the mean, and
         C. an upper and lower limit
2. The model shall provide an extensible framework suitable for supporting other statistical distributions
3. The model shall provide the ability to assign a common value to all runs:
   (a) This value could be a fixed, user-defined value
   (b) This value could be a random assignment, generated once and then applied to all runs
4. The model shall provide the capability to read values from a pre-generated file instead of generating its own values
5. The model shall provide the ability to randomly select from a discrete data set, including:
   (a) enumerations,
   (b) character-strings,
   (c) boolean values, and
   (d) numerical values
6. The model shall provide the capability to compute follow-on variables, the values of which are a function of one or more dispersed variables with values generated using any of the methods in requirements 1-5.
7. The model shall provide a record of the generated distributions, allowing for repeated execution of the same scenario using exactly the same conditions.
8. The model shall provide summary data of the dispersions which have been applied, including:
   (a) number of dispersions
   (b) types of dispersions
   (c) correlations between variables

3 Model Specification

3.1 Code Structure

The model can be broken down into its constituent classes; there are two principle components to the model – the variables, and the management of the variables.

3.1.1 Variable Management (MonteCarloMaster)

MonteCarloMaster is the manager of the MonteCarlo variables. This class controls how many sets of dispersed variables are to be generated; for each set, it has the responsibility for

- instructing each variable to generate its own dispersed value
- collecting those values and writing them to an external file

3.1.2 Dispersed Variables (MonteCarloVariable)

MonteCarloVariable is an abstract class that forms the basis for all dispersed variables. The following classes inherit from MonteCarloVariable:

- **MonteCarloVariableFile** will extract a value for its variable from a specified text file. Typically, a data file will comprise some number of rows and some number of columns of data. Each column of data represents the possible values for one variable. Each row of data represents a correlated set of data to be applied to several variables; each data-set generation will be taken from one line of data. Typically, each subsequent data-set will be generated from the next line of data; however, this is not required.
- In some situations, it is desirable to have the next line of data to be used for any given data set be somewhat randomly chosen. This has the disadvantageous effect of having some data sets being used more than others, but it supports better cross-population when multiple data files are being used.
    - For example, if *file1* contained 2 data sets and *file2* contained 4 data sets, then a sequential sweep through these file would set up a repeating pattern with line 1 of *file2* always being paired with line 1 of *file1*. For example, in 8 runs, we would get this pattern of line numbers from each run:
        - (1,1), (2,2), (1,3), (2,4), (1,1), (2,2), (1,3), (2,4)
    - If the first file was allowed to skip a line, the pattern can produce a more comprehensive combination of data:
        - (1,1), (1,2), (2,3), (1,4), (2,1), (2,2), (2,3), (1,4)
- **MonteCarloVariableFixed** provides fixed-values to a variable for all generated data-sets. The values can be represented as a *double*, *int*, or *STL-string*.
- **MonteCarloVariableRandom** is the base class for all variables being assigned a random value. The values can be represented as a *double*, *int*, or *STL-string*. There are several subclasses:
    - **MonteCarloVariableRandomNormal** provides a variable with a value dispersed according to a normal distribution specified by its mean and standard deviation.
    - **MonteCarloVariableRandomUniformInt** provides a variable with a value dispersed according to a uniform distribution specified by its upper and lower bounds. This class represents a discrete distribution, providing an integer value.
    - **MonteCarloVariableRandomUniform** provides a variable with a value dispersed according to a uniform distribution specified by its upper and lower bounds. This class represents a continuous distribution.

- **MonteCarloVariableRandomStringSet** represents a discrete variable, drawn from a set of STL-strings. The class inherits from *MonteCarloVariableRandomUniform*; this distribution generates a continuous value in [0,1) and scales and casts that to an integer index in {0, …, *size*-1} where *size* is the number of available strings from which to choose.

  Note – an astute reader may question why the discrete *MonteCarloVariableRandomStringSet* inherits from the continuous *MonteCarloVariableRandomUniform* rather than from the discrete *MonteCarloVariableRandomUniformInt.* The rationale is based on the population of the vector of selectable strings in this class. It is desirable to have this vector be available for population outside the construction of the class, so at construction time the size of this vector is not known. However, the construction of the *MonteCarloVariableRandomUniformInt* requires specifying the lower and upper bounds, which would be *0* and *size-1* respectively. Because *size* is not known at construction, this cannot be specified. Conversely, constructing a *MonteCarloVariableRandomUniform* with bounds at [0,1) still allows for scaling to the eventual size of the strings vector.

  - **MonteCarloVariableRandomBool** is a simple implementation of a *MonteCarloVariableRandomStringSet* in which the two available strings are {“true”,false}

- **MonteCarloVariableSemiFixed** utilizes a two-step process. First, a seed-variable has its value generated, then that value is copied to this variable. The seed-variable could be a “throw-away” variable, used only to seed this value, or it could be an instance of another dispersed variable. Once the value has been copied to this instance, it is retained in this instance for all data sets. The seed-variable will continue to generate a new value for each data set, but they will not be seen by this variable after that first set.

  The seed-variable can be any type of *MonteCarloVariable*, but note that not all types of *MonteCarloVariable* actually make sense to use in this context. Most of the usable types are specialized types of *MonteCarloVariableRandom*.

  However, restricting the seed-variable in such a way would limit the extensibility of the model. All *MonteCarloVariableRandom* types use the C++ <random> library for data generation. Limiting the *MonteCarloVariableSemiFixed* type to be seeded only by something using the <random> library violates the concept of free-extensibility. Consequently, the assigned value may be extracted from any *MonteCarloVariable* type. The only constraint is that the command generated by the seed-variable includes an “=” symbol; everything to the right of that symbol will be assigned to this variable.

- **MonteCarloPythonLineExec** provides a line of executable Python code that can be used to compute the value of this variable. So rather than generating an assignment statement, e.g.

  ```
  var_x = 5
  ```

  when the *MonteCarloMaster* processes an instance of this class, it will use a character string to generate an instruction statement, e.g.

  ```
  var_x = math.sin(2 * math.pi * object.circle_fraction)
  ```

  (in this case, the character string would be *“math.sin(2 * math.pi * object.circle_fraction)”* and *object.circle_fraction* could be a previously-dispersed variable).

  A particularly useful application of this capability is in generating systematic data sweeps across a domain, as opposed to random distributions within a domain. These are commonly implemented as a for-loop, but we can use the *MonteCarloPythonLineExec* to generate them internally. The first data assignment made in each file is to a run-number, which can be used as an index. The example shown below will generate a sweep across the domain [20,45) in steps of 2.5.

  ```
  object.sweep_variable = (monte_carlo.master.monte_run_number % 10) * 2.5 + 20
  ```

  - **MonteCarloPythonFileExec** is used when simple assignments and one-line instructions are insufficient, such as when one generated-value that feeds into an analytical algorithm to generate multiple other values. With this class, the execution of the Python file generated by *MonteCarloMaster* will hit a call to execute a file as specified by this class. This is an oddity among the bank of *MonteCarloVariable* implementations. In all other implementations, the identifying *variable_name* is used to identify the variable whose value is to be assigned (or computed). With the *MonteCarloPythonFileExec* implementation, the *variable_name* is hijacked to provide the name of the file to be executed.

3.2 Mathematical Formulation

No mathematical formulation. The random number generators use the C++ <random> library.

4 User's Guide

The model is written in C++; it must be instantiated and constructed as with any other C++ model.

In a Trick simulation, that means adding content to an existing S_module, or creating a new S_module. Examples of such an S_module are presented in this User's Guide.

4.1 What to expect

This role played by this model can be easily misunderstood, so let's start there.

This model generates Python files containing assignments to variables.

That's it!! It does not manage MonteCarlo runs. It does not execute any simulations. When it runs, it creates the requested number of Python files and exits.

This design is deliberate; we want the model to generate the instruction sets that will allow execution of a set of dispersed configurations. At that point, the simulation should cease, returning control to the user to distribute the execution of those configurations according to whatever distribution mechanism they desire. This could be:

- something really simple, like a wild-card, *<executive> MONTE_RUN_test/RUN*/monte_input.py*
- a batch-script,
- a set of batch-scripts launching subsets onto different machines,
- a load-management service, like SLURM
- any other mechanism tailored to the user's currently available computing resources.

The intention is that the model runs very early in the simulation sequence. If the model is inactive (as when running a regular, non-MonteCarlo run), it will take no action. But when this model is activated, the user should expect the simulation to terminate before it starts on any propagation.

When a simulation executes with this model active, the only result of the simulation will be the generation of files containing the assignments to the dispersed variables. The simulation should be expected to terminate at t=0.

4.1.1 Trick Users

The model is currently configured for users of the *Trick* simulation engine. The functionality of the model is almost exclusively independent of the chosen simulation engine, with the exceptions being the shutdown sequence, and the application of units information in the variables.

Found at the end of the *MonteCarloMaster::execute()* method, the following code:

```
exec_terminate_with_return(0, __FILE__, __LINE__,message.c_str());
```

is a Trick instruction set to end the simulation.

Found at the end of *MonteCarloVariable::insert_units()*, the following code:

```
  // TODO: Pick a unit-conversion mechanism
  //       Right now, the only one available is Trick:
  trick_units( pos_equ+1);
```

provides the call to

```
MonteCarloVariable::trick_units(
    size_t insertion_pt)
{
  command.insert(insertion_pt, " trick.attach_units(\"" + units + "\",");
  command.append(")");
```

which appends Trick instructions to interpret the generated value as being represented in the specified units.

The rest of the User's Guide will use examples of configurations for Trick-simulation input files.

4.1.2 Non-Trick Users

To configure the model for simulation engines other than Trick, the Trick-specific content identified above should be replaced with equivalent content that will result in:

- the shutdown of the simulation, and
- the conversion of units from the type specified in the distribution specification to the type native to the variable to which the generated value is to be assigned.

While the rest of the User's Guide will use examples of configurations for Trick-simulation input files, understand that these are mostly just C++ or Python code setting the values in this model to make it work as desired. Similar assignments would be required for any other simulation engine.

4.2 MonteCarlo Manager (MonteCarloMaster)

4.2.1 Instantiation

The instantiation of *MonteCarloMaster* would typically be done directly in the S_module. The construction of this instance takes a single argument, a STL-string describing its own location within the simulation data-structure.

The MonteCarloMaster class has a single public-interface method call, MonteCarloMaster::execute(). This method has 2 gate-keeping flags that must be set (the reason for there being 2 will be explained later):

- active
- generate_dispersions

If either of these flags is false (for reference, *active* is constructed as false and *generate_dispersions* is constructed as true) then this method returns with no action. If both are true, then the model will generate the dispersions, write those dispersions to the external files, and shut down the simulation.

An example S-module

```
class MonteCarloSimObject : public Trick::SimObject
{
 public:
  MonteCarloMaster master;  //    <--- master is instantiated
  MonteCarloSimObject(std::string location)
  :
    master(location) // <--- master is constructed with this STL-string
  {
    P_MONTECARLO ("initialization") master.execute(); // <--- the only function
call
  }
};
MonteCarloSimObject monte_carlo("monte_carlo.master"); // <--- location of “master”
                                                            is passed as an
                                                            argument
```

4.2.2 Configuration

The configuration of the *MonteCarloMaster* is something to be handled as a user-input to the simulation without requiring re-compilation; as such, it is typically handled in a Python input file. There are two sections for configuration:

- modifications to the regular input file, and
- new file-input or other external monte-carlo initiation mechanism

4.2.2.1 Modifications to the regular input file

A regular input file sets up a particular scenario for a nominal run. To add monte-carlo capabilities to this input file, the following code should be inserted somewhere in the file:

```
if monte_carlo.master.active:

  # insert non-mc-variable MC configurations like logging

  if monte_carlo.master.generate_dispersions:
    exec(open(“Modified_data/monte_variables.py").read())
```

Let’s break this down, because it explains the reason for having 2 flags:

		generate_dispersions	
		true	false
active	true	Generate dispersions for this scenario, but do not run the scenario.	Run scenario with monte-carlo configuration and pre-generated dispersions
	false	Regular (non-monte-carlo) run	

1. If the master is inactive, this content is passed over and the input file runs just as it would without this content
2. Having the master active flag set to true instructs the simulation that the execution is intended to be part of a monte-carlo analysis. Now there are 2 types of executions that fit this intent:
   (a) The generation of the dispersion files
   (b) The execution of this run with the application of previously-generated dispersions

Any code to be executed for case (a) must go inside the *generate_dispersions* gate. Any code to be executed for case (b) goes *inside* the *active* gate, but *outside* the *generate_dispersions* gate.

You may wonder why this distinction is made. In many cases, it is desirable to make the execution for monte-carlo analysis subtly different to that for regular analysis. One commonly used distinction is logging of data; the logging requirement may differ between a regular run and one as part of a monte-carlo analysis (typically, monte-carlo runs execute with reduced logging). By providing a configuration area for a monte-carlo run, we can support these distinctions.

Note – any code to be executed for only non-monte-carlo runs can be put in an *else:* block. For example, this code will set up one set of logging for a monte-carlo run, and another for a non-monte-carlo run of the same scenario:

```
if monte_carlo.master.active:

  exec(open("Log_data/log_for_monte_carlo.py").read())

  if monte_carlo.master.generate_dispersions:
    exec(open("Modified_data/monte_variables.py").read())

else:
  exec(open("Log_data/log_for_regular.py").read())
```

3. If the **generate_dispersions** flag is also set to true, the *MonteCarloMaster*::*execute*() method will execute, generating the dispersion files and shutting down the simulation.

4.2.2.2 Initiating MonteCarlo

Somewhere outside this file, the **active** and **generate_dispersion** flags must be set. This can be performed either in a separate input file or via a command-line argument. Unless the command-line argument capability is already supported, by far the easiest mechanism is to create a new input file that subsequently reads the existing input file:

```
monte_carlo.master.activate("RUN_1")
exec(open("RUN_1/input.py").read())
```

The activate method takes a single string argument, representing the name of the run. This must be exactly the same name as the directory containing the original input file, "*RUN_1*" in the example above. This argument is used in 2 places (<*argument*> in these descriptions refers to the content of the argument string):

- In the creation of a *MONTE_*<*argument*> directory. This directory will contain some number of sub-directories identified as, for example, RUN_01, RUN_02, RUN_03, etc. each of which will contain one of the generated dispersion files.
- In the instructions written into the generated dispersion files to execute the content of the input file found in <*argument*>.

4.2.2.3 Additional Configurations

There are additional configurations instructing the *MonteCarloMaster* on the generation of the new dispersion files. Depending on the use-case, these could either be embedded within the "*if monte_carlo.master.generate_dispersions:*" block of the original input file, or in the secondary input file (or command-line arguments if configured to do so).

- **Number of runs** is controlled with a single statement, e.g.

```
monte_carlo.master.set_num_runs(10)
```

- **Generation of meta-data**. The meta-data provides a summary of which variables are being dispersed, the type of dispersion applied to each, the random seeds being used, and correlation between different variables. This is written out to a file called *MonteCarlo_Meta_data_output* in the *MONTE_** directory.

```
monte_carlo.master.generate_meta_data = True
```

- **Changing the name of the automatically-generated monte-directory**. By default, this takes the form "MONTE_<run_name>" as assigned in the *MonteCarloMaster::activate(...)* method. The *monte_dir* variable is public and can be reset after activation and before the *MonteCarloMaster::execute()* method runs. This is particularly useful if it is desired to compare two distribution sets for the same run.

```
monte_carlo.master.monte_dir = "MONTE_RUN_1_vers2"
```

- **Changing the input file name**. It is expected that most applications of this model will run with a typical organization of a Trick simulation. Consequently, the original input file is probably named *input.py*, and this is the default setting for the *input_file_name* variable. However, to support other cases, this variable is public and can be changed at any time between construction and the execution of the *MonteCarloMaster::execute()* method.

```
monte_carlo.master.input_file_name = "modified_input.py"
```

- **Padding the filenames of the generated files**. By default, the generated RUN directories in the generated MONTE_* directory will have their numerical component padded according to the number of runs. When:
  - between 1 - 10 runs are generated, the directories will be named RUN_0, RUN_1, …
  - between 11-100 runs are generated, the directories will be named RUN_00, RUN_01, …
  - between 101-1000 runs are generated, the directories will be named RUN_000, RUN_001, …
  - etc.

  Specification of a minimum padding width is supported. For example, it might be desired to create 3 runs with names RUN_00000, RUN_00001, and RUN_00002, in which case the minimum-padding should be specified as 5 characters.

```
monte_carlo.master.minimum_padding = 5
```

- **Changing the run-name**. For convenience, the run-name is provided as an argument in the *MonteCarloMaster::activate(...)* method. The *run_name* variable is public, and can be reset after activation and before the *MonteCarloMaster::execute()* method runs. Because this setting determines which run is to be launched from the dispersion files, resetting *run_name* has limited application – effectively limited to correcting an error, which could typically be more easily corrected directly.

```
monte_carlo.master.run_name = "RUN_2"
```

4.3 MonteCarlo Variables (MonteCarloVariable)

The instantiation of the *MonteCarloVariable* instances is typically handled as a user-input to the simulation without requiring re-compilation. As such, these are usually implemented in Python input files. This is not a requirement, and these instances can be compiled as part of the simulation build. Both cases are presented.

4.3.1 Instantiation and Registration

For each variable to be dispersed, an instance of a *MonteCarloVariable* must be created, and that instance registered with the *MonteCarloMaster* instance:

1. Identify the type of dispersion desired
2. Select the appropriate type of *MonteCarloVariable* to provide that dispersion.
3. Create the new instance using its constructor.
4. Register it with the *MonteCarloMaster* using the *MonteCarloMaster::add_variable(MonteVarloVariable&)* method

4.3.1.1 Python input file implementation for Trick:

When the individual instances are registered with the master, it only records the address of those instances. A user may create completely new variable names for each dispersion, or use a generic name as illustrated in the example below. Because these are typically created within a Python function, it is important to add the *thisown=False* instruction on each creation to prevent its destruction when the function returns.

```
mc_var = trick.MonteCarloVariableRandomUniform( "object.x_uniform", 0, 10, 20)
mc_var.thisown = False
monte_carlo.master.add_variable(mc_var)

mc_var = trick.MonteCarloVariableRandomNormal( "object.x_normal", 0, 0, 5)
mc_var.thisown = False
monte_carlo.master.add_variable(mc_var)
```

4.3.1.2 C++ implementation in its own class:

In this case, the instances do have to be uniquely named.

Note that the registering of the variables could be done in the class constructor rather than in an additional method (*process_variables*), thereby eliminating the need to store the reference to MonteCarloMaster. In this case, the *generate_dispersions* flag is completely redundant because the variables are already registered by the time the input file is executed. Realize, however, that doing so does carry the overhead of registering those variables with the *MonteCarloMaster* **every** time the simulation starts up. This can a viable solution when there are only a few MonteCarloVariable instances, but is generally not recommended; using an independent method (*process_variables*) allows restricting the registering of the variables to be executed only when generating new dispersions.

```
class MonteCarloVarSet {
 private:
  MonteCarloMaster & master;
 public:
  MonteCarloVariableRandomUniform x_uniform;
  MonteCarloVariableRandomNormal  x_normal;
  …

  MonteCarloVarSet( MonteCarloMaster & master_)
    :
    master(master_),
    x_uniform("object.x_uniform", 0, 10, 20),
    x_normal ("object.x_normal", 0, 0, 5),
    ...
  { };

  void process_variables() {
    master.add_variable(x_uniform);
    master.add_variable(x_normal);
    ...
  }
};
```

4.3.1.3 C++ implementation within a Trick S-module:

Instantiating the variables into the same S-module as the master is also a viable design pattern. However, this can lead to a very long S-module so is typically only recommended when there are few variables. As with the C++ implementation in a class, the variables can be registered with the master in the constructor rather than in an additional method, with the same caveats presented earlier.

```
class MonteCarloSimObject : public Trick::SimObject
{
 public:
  MonteCarloMaster                master;
  MonteCarloVariableRandomUniform x_uniform;
  MonteCarloVariableRandomNormal  x_normal;
  ...

  MonteCarloSimObject(std::string location)
  :
    master(location),
    x_uniform("object.x_uniform", 0, 10, 20),
    x_normal ("object.x_normal", 0, 0, 5),
    ...
  { };

  void process_variables() {
    master.add_variable(x_uniform);
    master.add_variable(x_normal);
    ...
  };

  {
    P_MONTECARLO ("initialization") master.execute();
  }
};
MonteCarloSimObject monte_carlo("monte_carlo.master");
```

4.3.2 Input-file Access

If using the Python input file to create these instances, the execution of the configuration of these variables must be contained inside the *generate_dispersions* gate in the input file:

```
if monte_carlo.master.active:
  if monte_carlo.master.generate_dispersions:
    exec(open("Modified_data/monte_variables.py").read())
```

(where *monte_variables.py* is the file containing the *mc_var* = … content described earlier)

If using a (compiled) C++ implementation with a method to process the registration, that method call must be contained inside the *generate_dispersions* gate in the input file:

```
if monte_carlo.master.active:
  if monte_carlo.master.generate_dispersions:
    monte_carlo_variables.process_variables()
```

If using a (compiled) C++ implementation with the registration conducted at construction, the *generate_dispersions* flag is not used in the input file.

```
if monte_carlo.master.active:
  # add only those lines such as logging configuration
```

4.3.3 Configuration

For all variable-types, the *variable_name* is provided as the first argument to the constructor. This variable name must include the full address from the top level of the simulation. After this argument, each variable type differs in its construction arguments and subsequent configuration options.

4.3.3.1 MonteCarloVariable

MonteCarloVariable is an abstract class; its instantiable implementations are presented below. There is one important configuration for general application to these implementations, the setting of units. In a typical simulation, a variable has an inherent unit-type; these are often SI units, but may be based on another system. Those native units may be different to those in which the distribution is described. In this case, assigning the generated numerical value to the variable without heed to the units mismatch would result in significant error.

set_units(std::string units)

This method specifies that the numerical value being generated is to be interpreted in the specified units.

Notes

- if it is known that the variable's native units and the dispersion units match (including the case of a dimensionless value), this method is not needed.
- This method is not applicable to all types of *MonteCarloVariable;* use with *MonteCarloVariableRandomBool* and *MonteCarloPython** is considered undefined behavior.

4.3.3.2 MonteCarloVariableFile

The construction arguments are:

1. variable name
2. filename containing the data
3. column number containing data for this variable
4. (optional) first column number. This defaults to 1, but some users may want to zero-index their column numbers, in which case it can be set to 0.

There is no additional configuration beyond the constructor.

4.3.3.3 MonteCarloVariableFixed

The construction arguments are:

1. variable name
2. value to be assigned

Additional configuration for this model includes the specification of the maximum number of lines to skip between runs.

max_skip

This public variable has a default value of 0 – meaning that the next run will be drawn from the next line of data, but this can be adjusted.

4.3.3.4 MonteCarloVariableRandomBool

The construction arguments are:

1. variable name
2. seed for random generator

There is no additional configuration beyond the constructor.

4.3.3.5 MonteCarloVariableRandomNormal

The construction arguments are:

1. variable name
2. seed for random generator, defaults to 0
3. mean of distribution, defaults to 0
4. standard-deviation of distribution, defaults to 1.

The normal distribution may be truncated, and there are several configuration settings associated with truncation. Note that for all of these truncation options, if the lower truncation bound is set to be larger than the upper truncation bound, the generation of the dispersed value will fail and the simulation will terminate without generation of files. If the upper and lower bound are set to be equal, the result will be a forced assignment to that value.

TruncationType

This is an enumerated type, supporting the specification of the truncation limits in one of three ways:

- **StandardDeviation**. The distribution will be truncated at the specified number(s) of standard deviations away from the mean.
- **Relative**. The distribution will be truncated at the specified value(s) relative to the mean value.
- **Absolute**. The distribution will be truncated at the specified value(s).

max_num_tries

The truncation is performed by repeatedly generating a number from the unbounded distribution until one is found that lies within the truncation limits. This *max_num_tries* value determines how many attempts may be made before the algorithm concedes. It defaults to 10,000. If a value has not been found within the specified number of tries, an error message is sent and the value is calculated according to the following rules:

- For a distribution truncated at only one end, the truncation limit is used.
- For a distribution truncated at both ends, the midpoint value between the two truncation limits is used.

truncate(double limit, TruncationType)

This method provides a symmetric truncation, with the numerical value provided by *limit* being interpreted as a number of standard-deviations either side of the mean, a relative numerical value from the mean, or an absolute value.

The value *limit* should be positive. If a negative value is provided, it will be negated to a positive value.

The use of TruncationType Absolute and this method requires a brief clarification because this may result in an asymmetric distribution. In this case, the distribution will be truncated to lie between (-limit, limit) which will be asymmteric for all cases in which the mean is non-zero.

truncate(double min, double max, TruncationType)

This method provides a more general truncation, with the numerical value provided by *min* and *max* being interpreted as a number of standard-deviations away from the mean, a relative numerical value from the mean, or an absolute value.

Unlike the previous method, the numerical arguments (*min* and *max*) may be positive or negative, and care must be taken especially when specifying *min* with TruncationType StandardDeviation or Relative. Realize that a positive value of *min* will result in a lower bound with value above that of the mean; *min* does not mean “distance to the left of the mean”, it means the smallest acceptable value relative to the mean.

truncate_low(double limit, TruncationType)

This method provides a one-sided truncation. All generated values will be above the *limit* specification.

truncate_high(double limit, TruncationType)

This method provides a one-sided truncation. All generated values will be below the *limit* specification.

untruncate()

This method removes previously configured truncation limits.

4.3.3.6 MonteCarloVariableRandomStringSet

The construction arguments are:

1. variable name
2. seed for random generator

This type of *MonteCarloVariable* contains a STL-vector of STL-strings containing the possible values that can be assigned by this generator. This vector is NOT populated at construction time and must be configured.

add_string(std::string new_string)

This method adds the specified string (*new_string*) to the vector of available strings.

4.3.3.7 MonteCarloVariableRandomUniform

The construction arguments are:

1. variable name
2. seed for random generator, defaults to 0
3. lower-bound of distribution, default to 0
4. upper-bound for distribution, defaults to 1

There is no additional configuration beyond the constructor.

4.3.3.8 MonteCarloVariableRandomUniformInt

The construction arguments are:

1. variable name
2. seed for random generator, defaults to 0
3. lower-bound of distribution, default to 0
4. upper-bound for distribution, defaults to 1

There is no additional configuration beyond the constructor.

4.3.3.9 MonteCarloVariableSemiFixed

The construction arguments are:

1. variable name
2. reference to the MonteCarloVariable whose generated value is to be used as the fixed value.

There is no additional configuration beyond the constructor.

4.3.3.10 MonteCarloPythonLineExec

The construction arguments are:

1. variable name
2. an STL-string providing the Python instruction for the computing of the value to be assigned to the specified variable.

There is no additional configuration beyond the constructor.

4.3.3.11 MonteCarloPythonFileExec

The construction argument is:

1. name of the file to be executed from the generated input file.

There is no additional configuration beyond the constructor.

4.4 Information on the Generated Files

This section is for informational purposes only to describe the contents of the automatically-generated dispersion files. Users do not need to take action on any content in here.

The generated files can be broken down into 3 parts:

- Configuration for the input file. These two lines set the flags such that when this file is executed, the content of the original input file will configure the run for a monte-carlo analysis but without re-generating the dispersion files.

```
monte_carlo.master.active = True
monte_carlo.master.generate_dispersions = False
```

- Execution of the original input file. This line opens the original input file so that when this file is executed, the original input file is also executed automatically.

```
exec(open('RUN_1/input.py').read())
```

- Assignment to simulation variables. This section always starts with the assignment to the run-number, which is also found in the name of the run, so RUN_0 gets a 0, RUN_1 gets a 1, etc. This value can be used, for example, to generate data sweeps as described in section MonteCarloPythonLineExec above.

```
monte_carlo.master.monte_run_number = 0
object.test_variable1 = 5
object.test_variable1 = 1.23456789
...
```

4.5 Extension

The model is designed to be extensible and while we have tried to cover the most commonly used applications, complete anticipation of all use-case needs is impossible. The most likely candidate for extension is in the area of additional distributions. In this case:

- A new distribution should be defined in its own class.
- That class shall inherit from MonteCarloVariable or, if it involves a random generation using a distribution found in the C++ <random> library, from MonteCarloVariableRandom.
- The class shall provide its own generate_assignment() method. This shall:
  - Populate the *command* variable inherited from *MonteCarloVariable*. This is the STL string representing the content that the *MonteCarloMaster* will place into the generated dispersion files.
  - Call the insert_units() method inherited from MonteCarloVariable
  - Set the *command_generated* flag to true if the command has been successfully generated.

5 Verification

The verification of the model is provided in directory *verif/SIM_verif.*

This verification package comprises runs categorized into several sections:

- **RUN_nominal** contains an example of each type of assignment available to the model. This is the primary test
- **RUN_random*** contains a more in-depth look at the random variables, including consideration of the generated distribution.
- **RUN_file*** considers the different configurations of using data read from a file.
- **RUN_generate_meta_data_early** tests the consequence of generating meta data before the assignments have been prepared
- **RUN_remove_variable** tests the ability to remove a variable from distribution.
- **RUN_WARN*** test the misconfigurations that should lead to warning messages.
- **RUN_ERROR*** test the misconfigurations that should lead to error messages.
- **IO*** test problems associated with reading or writing from the specified files.
- **FAIL*** test the misconfigurations that should lead to terminal failure.

5.1 RUN_nominal

Execution of RUN_nominal/input.py results in:

- generation of the *MONTE_RUN_nominal* directory, which contains:
  - RUN*. A set of configurations for each monte-carlo run. For unit-test purposes this contains only 2 runs; for this verification exercise the number of runs is increased to 20. Each directory contains:
    - *monte_input_a.py.* This is the input file that would be provided to the S_main to execute the specific scenario. Note - This file is typically called *monte_input.py* for normal usage of the model; it is generally referred to as the monte-input file.
    - *monte_variables.* A list of the dispersed variables; note that this is not a comprehensive set of every assignment generated by the model, only those assignments that can be expressed as a value assigned to a variable. Each run has a copy of the same file.
    - *monte_values.* A list of the values assigned to each variable identified in *monte_variables* for this specific run.
  - *monte_variables.* A copy of the file found in each RUN*.
  - *monte_values_all_runs.* A concatenation of all the *RUN_*/monte_values* files.
- Execution of the generated *monte-input.py* files
  - This generates a file *log_test_data.csv* in each RUN*, which contains the values of the variables located in the simulation as generated in that scenario. This is a more comprehensive set than that found in *monte_variables* because it includes assignments to variables that cannot be expressed as simple direct assignments. The values of those variables included in *monte_variables* should match those found in *log_test_data.csv.*

In this section, we will consider each assignment generation one at a time. For each generation type, the same presentation format will be used:

- the generation command will be provided, highlighted in yellow. e.g.:

mc_var = trick.MonteCarloVariableRandomUniform("test.x_uniform", 0, 10, 20)

- The relevant lines from the monte-input files is shown, highlighted in teal. Note – this is a concatentation of lines taken from each monte-input file, typically 1 line per file; these lines do not exist as a block anywhere in *MONTE_RUN_nominal.*
  - In most cases, these lines are simple assignments of the form:

    test.x_uniform = 11.31537787738761
  - In some cases, they are Python instructions:

    test.x_line_command = test.x_integer * test.x_uniform
- Where the data in *log_test_data.csv* is required to confirm expected execution of the monte-input file contents, these logged outputs will be presented, highlighted in orange.
- x_line_command (logged) 0 0 24.37918372 16.78864717 19.34692896
- Where messages or content are broadcast to stdout, these will be included highlighted in green:
- RUN_000: Standalone_function received a value of 10.7052

5.1.1 Uniform Distribution

mc_var = trick.MonteCarloVariableRandomUniform("test.x_uniform", 0, 10, 20)

RUN_000: test.x_uniform = 15.92844616516683
RUN_001: test.x_uniform = 18.44265744256598
RUN_002: test.x_uniform = 18.5794561998983
RUN_003: test.x_uniform = 18.47251737384331
RUN_004: test.x_uniform = 16.23563696496108
RUN_005: test.x_uniform = 13.84381708373757
RUN_006: test.x_uniform = 12.97534605357234
RUN_007: test.x_uniform = 10.56712975933164
RUN_008: test.x_uniform = 12.72656294741589
RUN_009: test.x_uniform = 14.77665111744646
RUN_010: test.x_uniform = 18.12168726649071
RUN_011: test.x_uniform = 14.79977171525567
RUN_012: test.x_uniform = 13.92784793294977
RUN_013: test.x_uniform = 18.36078769044391
RUN_014: test.x_uniform = 13.37396161647289
RUN_015: test.x_uniform = 16.48171876577458
RUN_016: test.x_uniform = 13.68241537367044
RUN_017: test.x_uniform = 19.5715515451334
RUN_018: test.x_uniform = 11.40350777604188
RUN_019: test.x_uniform = 18.70087251269727

Logged data matches Monte-input data

5.1.2 Normal Distribution

mc_var = trick.MonteCarloVariableRandomNormal("test.x_normal", 2, 10, 2)

RUN_000: test.x_normal = 9.954870507417668
RUN_001: test.x_normal = 11.32489560639709
RUN_002: test.x_normal = 8.225020001340255
RUN_003: test.x_normal = 9.078215205210737
RUN_004: test.x_normal = 8.61231801622891
RUN_005: test.x_normal = 10.72611121241102
RUN_006: test.x_normal = 14.56731728448253
RUN_007: test.x_normal = 9.489924230632374

```
RUN_008: test.x_normal = 11.10848764602406
RUN_009: test.x_normal = 11.9273239842278
RUN_010: test.x_normal = 15.29213437867134
RUN_011: test.x_normal = 9.912272360185705
RUN_012: test.x_normal = 8.068760643545902
RUN_013: test.x_normal = 11.75732773681084
RUN_014: test.x_normal = 5.508250256729741
RUN_015: test.x_normal = 12.23915038957531
RUN_016: test.x_normal = 7.89126396796938
RUN_017: test.x_normal = 7.982216956806577
RUN_018: test.x_normal = 9.864955835300426
RUN_019: test.x_normal = 9.837149685311553
```

Logged data matches Monte-input data

5.1.3 Truncated Normal Distribution

There are several methods by which a normal distribution can be truncated; these are explored here and in more detail in section 5.3.2.

5.1.3.1 Truncated by Standard Deviations from Mean

Distribution ~N(10,2) truncated to lie with 0.5 standard deviations from the mean should produce a distribution with values in the range (9, 11)

```
mc_var = trick.MonteCarloVariableRandomNormal ( "test.x_normal_trunc[1]", 2, 10, 2)
mc_var.truncate(0.5, trick.MonteCarloVariableRandomNormal.Relative)
```

```
RUN_000: test.x_normal_trunc[0] = 9.954870507417668
RUN_001: test.x_normal_trunc[0] = 9.078215205210737
RUN_002: test.x_normal_trunc[0] = 10.72611121241102
RUN_003: test.x_normal_trunc[0] = 9.489924230632374
RUN_004: test.x_normal_trunc[0] = 9.912272360185705
RUN_005: test.x_normal_trunc[0] = 9.864955835300426
RUN_006: test.x_normal_trunc[0] = 9.837149685311553
RUN_007: test.x_normal_trunc[0] = 9.507649693417006
RUN_008: test.x_normal_trunc[0] = 10.56080947072924
RUN_009: test.x_normal_trunc[0] = 10.19631186831437
RUN_010: test.x_normal_trunc[0] = 9.804159469162096
RUN_011: test.x_normal_trunc[0] = 10.10165106830803
RUN_012: test.x_normal_trunc[0] = 9.057434611329896
RUN_013: test.x_normal_trunc[0] = 10.12373334458601
RUN_014: test.x_normal_trunc[0] = 9.661517044726127
RUN_015: test.x_normal_trunc[0] = 10.00965925367215
RUN_016: test.x_normal_trunc[0] = 9.952858400406081
RUN_017: test.x_normal_trunc[0] = 10.30478920309146
RUN_018: test.x_normal_trunc[0] = 10.27803333258882
RUN_019: test.x_normal_trunc[0] = 9.366018370198786
```

Logged data matches Monte-input data

5.1.3.2 Truncated by Difference from Mean

Distribution ~N(10,2) truncated to lie within [-0.5, +0.7] from the mean should produce a distribution with values in the range (9.5, 10.7)

```
mc_var = trick.MonteCarloVariableRandomNormal ( "test.x_normal_trunc[1]", 2, 10, 2)
mc_var.truncate(-0.5, 0.7, trick.MonteCarloVariableRandomNormal.Relative)
```

```
RUN_000: test.x_normal_trunc[1] = 9.954870507417668
RUN_001: test.x_normal_trunc[1] = 9.912272360185705
RUN_002: test.x_normal_trunc[1] = 9.864955835300426
RUN_003: test.x_normal_trunc[1] = 9.837149685311553
```

```
RUN_004: test.x_normal_trunc[1] = 9.507649693417006
RUN_005: test.x_normal_trunc[1] = 10.56080947072924
RUN_006: test.x_normal_trunc[1] = 10.19631186831437
RUN_007: test.x_normal_trunc[1] = 9.804159469162096
RUN_008: test.x_normal_trunc[1] = 10.10165106830803
RUN_009: test.x_normal_trunc[1] = 10.12373334458601
RUN_010: test.x_normal_trunc[1] = 9.661517044726127
RUN_011: test.x_normal_trunc[1] = 10.00965925367215
RUN_012: test.x_normal_trunc[1] = 9.952858400406081
RUN_013: test.x_normal_trunc[1] = 10.30478920309146
RUN_014: test.x_normal_trunc[1] = 10.27803333258882
RUN_015: test.x_normal_trunc[1] = 10.03792379373566
RUN_016: test.x_normal_trunc[1] = 10.20959040504398
RUN_017: test.x_normal_trunc[1] = 10.65930415393322
RUN_018: test.x_normal_trunc[1] = 10.06884635542625
RUN_019: test.x_normal_trunc[1] = 10.50899736993173
```

Logged data matches Monte-input data

5.1.3.3 Truncated by Specified Bounds

Distribution ~N(10,2) truncated directly to to lie within range [9.9, 11.0]

```
mc_var = trick.MonteCarloVariableRandomNormal ( "test.x_normal_trunc[2]", 2, 10, 2)
mc_var.truncate(9.9,11, trick.MonteCarloVariableRandomNormal.Absolute)
```

```
RUN_000: test.x_normal_trunc[2] = 9.954870507417668
RUN_001: test.x_normal_trunc[2] = 10.72611121241102
RUN_002: test.x_normal_trunc[2] = 9.912272360185705
RUN_003: test.x_normal_trunc[2] = 10.56080947072924
RUN_004: test.x_normal_trunc[2] = 10.19631186831437
RUN_005: test.x_normal_trunc[2] = 10.10165106830803
RUN_006: test.x_normal_trunc[2] = 10.12373334458601
RUN_007: test.x_normal_trunc[2] = 10.00965925367215
RUN_008: test.x_normal_trunc[2] = 9.952858400406081
RUN_009: test.x_normal_trunc[2] = 10.30478920309146
RUN_010: test.x_normal_trunc[2] = 10.27803333258882
RUN_011: test.x_normal_trunc[2] = 10.03792379373566
RUN_012: test.x_normal_trunc[2] = 10.20959040504398
RUN_013: test.x_normal_trunc[2] = 10.65930415393322
RUN_014: test.x_normal_trunc[2] = 10.86586573576455
RUN_015: test.x_normal_trunc[2] = 10.87710866037394
RUN_016: test.x_normal_trunc[2] = 10.06884635542625
RUN_017: test.x_normal_trunc[2] = 10.50899736993173
RUN_018: test.x_normal_trunc[2] = 10.80302599545891
RUN_019: test.x_normal_trunc[2] = 10.95710819942595
```

Logged data matches Monte-input data

5.1.4 Truncated on Left Only

Distribution ~N(10,2) truncated on the left only at 9.9 should produce a distribution with values in the range $[9.9, \infty)$

```
mc_var = trick.MonteCarloVariableRandomNormal ( "test.x_normal_trunc[3]", 2, 10, 2)
mc_var.truncate_low(9.9, trick.MonteCarloVariableRandomNormal.Absolute)
```

```
RUN_000: test.x_normal_trunc[3] = 9.954870507417668
RUN_001: test.x_normal_trunc[3] = 11.32489560639709
RUN_002: test.x_normal_trunc[3] = 10.72611121241102
RUN_003: test.x_normal_trunc[3] = 14.56731728448253
RUN_004: test.x_normal_trunc[3] = 11.10848764602406
RUN_005: test.x_normal_trunc[3] = 11.9273239842278
RUN_006: test.x_normal_trunc[3] = 15.29213437867134
RUN_007: test.x_normal_trunc[3] = 9.912272360185705
RUN_008: test.x_normal_trunc[3] = 11.75732773681084
RUN_009: test.x_normal_trunc[3] = 12.23915038957531
RUN_010: test.x_normal_trunc[3] = 15.67043193912775
```

```
RUN_011: test.x_normal_trunc[3] = 11.41102171943641
```

Logged data matches Monte-input data

5.1.5 Truncated on Right Only

Distribution ~N(10,2) truncated on the right only at 4.0 should produce a distribution with values in the range $(-\infty, 4.0]$

```
mc_var = trick.MonteCarloVariableRandomNormal ( "test.x_normal_trunc[4]", 2, 10, 2)
mc_var.truncate_high(4, trick.MonteCarloVariableRandomNormal.Absolute)
```

```
RUN_000: test.x_normal_trunc[4] = 3.772280419911035
RUN_001: test.x_normal_trunc[4] = 3.806042167887552
RUN_002: test.x_normal_trunc[4] = 2.291328792360356
RUN_003: test.x_normal_trunc[4] = 3.354371206758478
RUN_004: test.x_normal_trunc[4] = 2.901755006167329
RUN_005: test.x_normal_trunc[4] = 1.969547280869834
RUN_006: test.x_normal_trunc[4] = 1.398353722126554
RUN_007: test.x_normal_trunc[4] = 3.858009610124997
RUN_008: test.x_normal_trunc[4] = 3.691946831230261
RUN_009: test.x_normal_trunc[4] = 3.655165469770109
RUN_010: test.x_normal_trunc[4] = 3.780650644038053
RUN_011: test.x_normal_trunc[4] = 3.461168743091792
RUN_012: test.x_normal_trunc[4] = 3.914540451839235
```

Logged data matches Monte-input data

5.1.6 Dispersion in Non-native units

Any distribution could be used for this verification; a normal distribution is chosen arbitrarily. The distribution ~N(10,2), with units of feet, assigned to a variable with native units of meters.

```
mc_var = trick.MonteCarloVariableRandomNormal ( "test.x_normal_length", 2, 10, 2)
mc_var.units = "ft"
```

```
RUN_000: test.x_normal_length = trick.attach_units("ft", 9.954870507417668)
RUN_001: test.x_normal_length = trick.attach_units("ft", 11.32489560639709)
RUN_002: test.x_normal_length = trick.attach_units("ft", 8.225020001340255)
RUN_003: test.x_normal_length = trick.attach_units("ft", 9.078215205210737)
RUN_004: test.x_normal_length = trick.attach_units("ft", 8.61231801622891)
```

	RUN_000	RUN_001	RUN_002	RUN_003	RUN_004
generated value (ft)	9.954870507	11.324895606	8.225020001	9.078215205	8.612318016
converted value (m)	3.034244531	3.451828181	2.506986096	2.767039994	2.625034531
logged value	3.034244531	3.451828181	2.506986096	2.767039994	2.625034531

5.1.7 Discrete Integer (Uniform Distribution)

Note that the range for a discrete variable includes the limits, in this case the distribution is ~U(0,2) which should generate values 0, 1, or 2 for each run.

```
mc_var = trick.MonteCarloVariableRandomUniformInt ( "test.x_integer", 1, 0, 2)
```

```
RUN_000: test.x_integer = 1
RUN_001: test.x_integer = 2
RUN_002: test.x_integer = 2
RUN_003: test.x_integer = 2
RUN_004: test.x_integer = 0
RUN_005: test.x_integer = 0
RUN_006: test.x_integer = 0
```

```
RUN_007: test.x_integer = 2
RUN_008: test.x_integer = 0
RUN_009: test.x_integer = 0
RUN_010: test.x_integer = 0
RUN_011: test.x_integer = 1
RUN_012: test.x_integer = 0
RUN_013: test.x_integer = 1
RUN_014: test.x_integer = 1
RUN_015: test.x_integer = 2
RUN_016: test.x_integer = 1
RUN_017: test.x_integer = 2
RUN_018: test.x_integer = 1
RUN_019: test.x_integer = 2
```

Logged data matches Monte-input data

5.1.8 Discrete String (Uniform Distribution)

Three strings are provided from which to select randomly:

- “ABC”
- “DEF”
- ‘GHIJKL’

Note single quotes and double quotes are used to confirm the model supports both.

```
mc_var = trick.MonteCarloVariableRandomStringSet ( "test.x_string", 3)
mc_var.add_string("\"ABC\"")
mc_var.add_string("\"DEF\"")
mc_var.add_string("'GHIJKL'")
```

```
RUN_000: test.x_string = "ABC"
RUN_001: test.x_string = 'GHIJKL'
RUN_002: test.x_string = "ABC"
RUN_003: test.x_string = "DEF"
RUN_004: test.x_string = "DEF"
RUN_005: test.x_string = "ABC"
RUN_006: test.x_string = "ABC"
RUN_007: test.x_string = "ABC"
RUN_008: test.x_string = "ABC"
RUN_009: test.x_string = 'GHIJKL'
RUN_010: test.x_string = "ABC"
RUN_011: test.x_string = "DEF"
RUN_012: test.x_string = "ABC"
```

Logged data matches Monte-input data

5.1.9 Discrete Boolean (Uniform Distribution)

```
mc_var = trick.MonteCarloVariableRandomBool( "test.x_boolean", 4)
```

```
RUN_000: test.x_boolean = True
RUN_001: test.x_boolean = False
RUN_002: test.x_boolean = True
RUN_003: test.x_boolean = True
RUN_004: test.x_boolean = True
RUN_005: test.x_boolean = False
RUN_006: test.x_boolean = False
RUN_007: test.x_boolean = True
RUN_008: test.x_boolean = True
RUN_009: test.x_boolean = True
RUN_010: test.x_boolean = False
RUN_011: test.x_boolean = True
RUN_012: test.x_boolean = False
```

RUN_013: test.x_boolean = True
RUN_014: test.x_boolean = False
RUN_015: test.x_boolean = True
RUN_016: test.x_boolean = True
RUN_017: test.x_boolean = False
RUN_018: test.x_boolean = True
RUN_019: test.x_boolean = True

Matches Monte-input data, with True substituted by 1 and False substituted by 0.

5.1.10 Python Code Injection

This type of variable provides the ability to assign a value to a variable that is the output of a function or script

5.1.10.1 Line of Code

If 2 arguments are provided to *MonteCarloPythonLineExec*, the arguments are interpreted as:

- 1st argument is the assignment
- 2nd argument is the instruction

In this case, we multiply two previously generated values: test.x_integer, and test.x_uniform

```
mc_var = trick. MonteCarloPythonLineExec( "test.x_line_command",
                                          "test.x_integer * test.x_uniform")
```

RUN_000: test.x_line_command = test.x_integer * test.x_uniform
(identical for all runs)

	RUN_000	RUN_001	RUN_002	RUN_003	RUN_004
x_integer	1	2	2	2	0
x_uniform	15.928446165	18.442657443	18.579456199	18.472517374	16.235636965
product	15.928446165	36.885314886	37.158912398	36.945034748	0
x_line_comand (logged)	15.928446165	36.885314886	37.158912398	36.945034748	0

5.1.10.2 Execution of a Function

Where 1 argument is provided to *MonteCarloPythonLineExec*, it is interpreted as a command to be inserted into the monte-input file. In this case, we have a C++ function defined:

```
void standalone_function( double value)
{
   std::cout << "\nStandalone_function received a value of " << value << "\n";
   x_sdefine_routine_called = 1;
}
```

As a proxy for whether this was called during phase #2, two outputs are available:

- Output to screen
- The loggable variable x_sdefine_routine_called, which defaults to 0 and is only reset by this method.

```
mc_var = trick.MonteCarloPythonLineExec( "test.standalone_function( test.x_normal)")
```

RUN_000: test.standalone_function(test.x_normal)
(identical for all runs)

RUN_000: Standalone_function received a value of 9.9545
RUN_001: Standalone_function received a value of 11.3249
etc.

test.x_sdefine_routine_called has a value of 1 for all runs.

5.1.10.3 Execution of File or Script

File Modified_data/sample.py contains the following code:

```
test.x_file_command[0] = 1
test.x_file_command[1] = test.mc_master.monte_run_number
test.x_file_command[2] = test.x_file_command[0] + test.x_file_command[1]
```

As a proxy for whether this was called during phase #2, the x_file_command values can be logged.

```
mc_var = trick.MonteCarloPythonFileExec( "Modified_data/sample.py")
```

RUN_000: exec(open('Modified_data/sample.py').read())
(identical for all runs)

	RUN_000	RUN_001	RUN_002	RUN_003	RUN_004
monte_run_number	0	1	2	3	4
file_command[0] (logged)	1	1	1	1	1
file_command[1] (logged)	0	1	2	3	4
file_command[2] (logged)	1	2	3	4	5

5.1.11 Extraction From File

This test only operates with a sequential file read, with each subsequent run reading from the next line of the file. See section 5.2 for verification of random line selection logic.

3 values are extracted from columns 3, 2, and 1 in order from file Modified_data/datafile.txt:

```
0 1 2 3 4
# comment
10 11 12 13 14
            <---------   <blank line>
20 21 22 23 24
            <---------   <contains only white space>
30 31 32 33 34
```

Comments, blank lines, and lines containing only white-space should be skipped

When file-end is reached, the file should wrap back to the beginning

```
mc_var1 = trick.MonteCarloVariableFile( "test.x_file_lookup[0]",
                          "Modified_data/datafile.txt",
                          3)
…
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[1]",
                         "Modified_data/datafile.txt",
                         2)
…
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[2]",
                         "Modified_data/datafile.txt",
                         1)
```

RUN_000: test.x_file_lookup[0] = 2
RUN_000: test.x_file_lookup[1] = 1
RUN_000: test.x_file_lookup[2] = 0

```
RUN_001: test.x_file_lookup[0] = 12
RUN_001: test.x_file_lookup[1] = 11
RUN_001: test.x_file_lookup[2] = 10

RUN_002: test.x_file_lookup[0] = 22
RUN_002: test.x_file_lookup[1] = 21
RUN_002: test.x_file_lookup[2] = 20

RUN_003: test.x_file_lookup[0] = 32
RUN_003: test.x_file_lookup[1] = 31
RUN_003: test.x_file_lookup[2] = 30

RUN_004: test.x_file_lookup[0] = 2
RUN_004: test.x_file_lookup[1] = 1
RUN_004: test.x_file_lookup[2] = 0

RUN_005: test.x_file_lookup[0] = 12
RUN_005: test.x_file_lookup[1] = 11
RUN_005: test.x_file_lookup[2] = 10

RUN_006: test.x_file_lookup[0] = 22
RUN_006: test.x_file_lookup[1] = 21
RUN_006: test.x_file_lookup[2] = 20

RUN_007: test.x_file_lookup[0] = 32
RUN_007: test.x_file_lookup[1] = 31
RUN_007: test.x_file_lookup[2] = 30

RUN_008: test.x_file_lookup[0] = 2
RUN_008: test.x_file_lookup[1] = 1
RUN_008: test.x_file_lookup[2] = 0

RUN_009: test.x_file_lookup[0] = 12
RUN_009: test.x_file_lookup[1] = 11
RUN_009: test.x_file_lookup[2] = 10

RUN_010: test.x_file_lookup[0] = 22
RUN_010: test.x_file_lookup[1] = 21
RUN_010: test.x_file_lookup[2] = 20

RUN_011: test.x_file_lookup[0] = 32
RUN_011: test.x_file_lookup[1] = 31
RUN_011: test.x_file_lookup[2] = 30
```

run number	x_file_lookup[0]	x_file_lookup[1]	x_file_lookup[2]	(line number)
0	2	1	0	1
1	12	11	10	2
2	22	21	20	3
3	32	31	30	4
4	2	1	0	1
5	12	11	10	2
6	22	21	20	3
7	32	31	30	4
8	2	1	0	1

Note - last column (line number) is added for a reference

5.1.12 Assignment of Fixed Value

3 options implemented:

- int value = 7
- double value = 7.0
- std::string value = “7”

```
mc_var = trick.MonteCarloVariableFixed( "test.x_fixed_value_int", 7)
…
mc_var = trick.MonteCarloVariableFixed( "test.x_fixed_value_double", 7.0)
…
mc_var = trick.MonteCarloVariableFixed( "test.x_fixed_value_string", "\"7\"")
```

```
RUN_000: test.x_fixed_value_int = 7
RUN_000: test.x_fixed_value_double = 7
RUN_000: test.x_fixed_value_string = "7"
(identical for all runs)
```

All values are logged with value 7

5.1.13 Assignment of Semi-Fixed Value

A semi-fixed value is a value generated for the first run, and held at that value for all subsequent runs

This case takes the value of *mc_var1*, the local variable used in generating the values for *test.x_file_lookup[0]*

The variable should therefore match that of *test.x_file_lookup[0]* from RUN_000, and take this value for all runs.

```
mc_var = trick.MonteCarloVariableSemiFixed( "test.x_semi_fixed_value", mc_var1 )
```

```
RUN_000: test.x_semi_fixed_value =  2
(identical for all runs)
```

Note– test.x_file_lookup[0] = 2 for RUN_000

Logged data matches Monte-input data

5.2 Reading Values From a File

As with the earlier case in section 5.1.11, the data is read from the file with the following contents.

```
0 1 2 3 4
# comment
10 11 12 13 14
          <--------- <blank line>
20 21 22 23 24
          <--------- <contains only white space>
30 31 32 33 34
```

This test investigates the options for randomizing which values are drawn from the file.

5.2.1 Sequential Lines

With this option, each subsequent run reads the next line from the file, wrapping back to the top of the file when it reaches the end. This has been investigated in section 5.1.11.

5.2.2 Random Lines with Linked Variables

This option uses the max_skip variable to allow some number of lines to be randomly passed over; the next run does not need to read the next line.

Note – when multiple variables are being drawn from the same file, each variable must be given the same *max_skip* value or an error will be flagged. It is not possible to draw two variables from different lines of the same file.

```
test.mc_master.set_num_runs(10)
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[0]", "Modified_data/datafile.txt", 3)
mc_var.max_skip = 3
…
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[1]", "Modified_data/datafile.txt", 2)
mc_var.max_skip = 3
…
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[2]", "Modified_data/datafile.txt", 1)
mc_var.max_skip = 3
```

run number	x_file_lookup[0]	x_file_lookup[1]	x_file_lookup[2]	(line number)	lines skipped
0	2	1	0	1	
1	12	11	10	2	0
2	12	11	10	2	3
3	32	31	30	4	1
4	22	21	20	3	2
5	32	31	30	4	0
6	2	1	0	1	0
7	32	31	30	4	2
8	22	21	20	3	2
9	22	21	20	3	3

Note - line number and number of skipped lines added for clarification):

5.2.3 Random Lines with Independent Variables

Must use independent data files if variables are not to be correlated (i.e. all extracted from the same line). It is not possible to have multiple variables drawn from different lines of one file.

```
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[0]", "Modified_data/single_col_1.txt", 0, 0)
mc_var.max_skip = 1
…
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[1]", "Modified_data/single_col_2.txt", 0, 0)
mc_var.max_skip = 2
…
mc_var = trick.MonteCarloVariableFile( "test.x_file_lookup[2]", "Modified_data/single_col_3.txt", 0, 0)
mc_var.max_skip = 3
```

Each file contains a single column of data:

- single_col_1.txt contains values 1 to 15
- single_col_2.txt contains values 16 to 30
- single_col_3.txt contains values 31 to 55

MONTE_RUN_file_skip2/monte_values_all_runs contains a summary of the assigned values. The line number and number of skipped lines added for clarification:

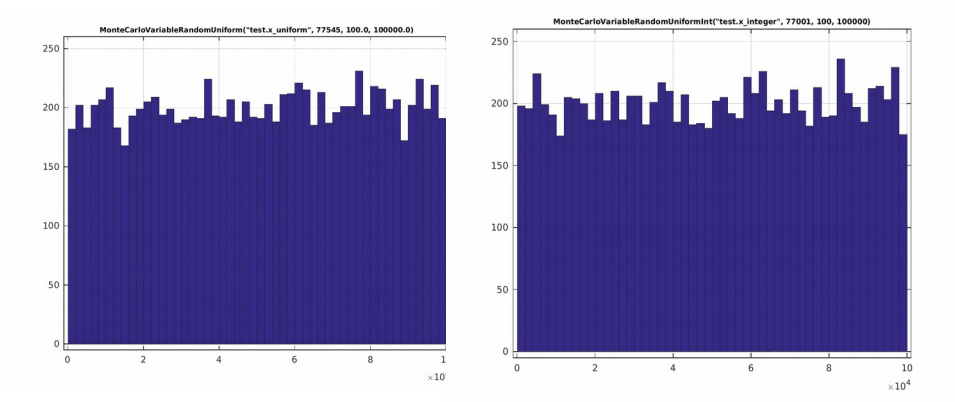
run number	x_file_lookup[0]	x_file_lookup[1]	x_file_lookup[2]	line numbers			lines skipped		
0	1	16	31	1	1	1			
1	2	17	32	2	2	2	0	0	0
2	4	20	36	4	5	6	1	2	3
3	5	22	38	5	7	8	0	1	1
4	7	24	41	7	9	11	1	1	2

5.3 Distribution Analyses

For these distributions, we increased the number of data points to 10,000 to get a better visualization of the distribution.

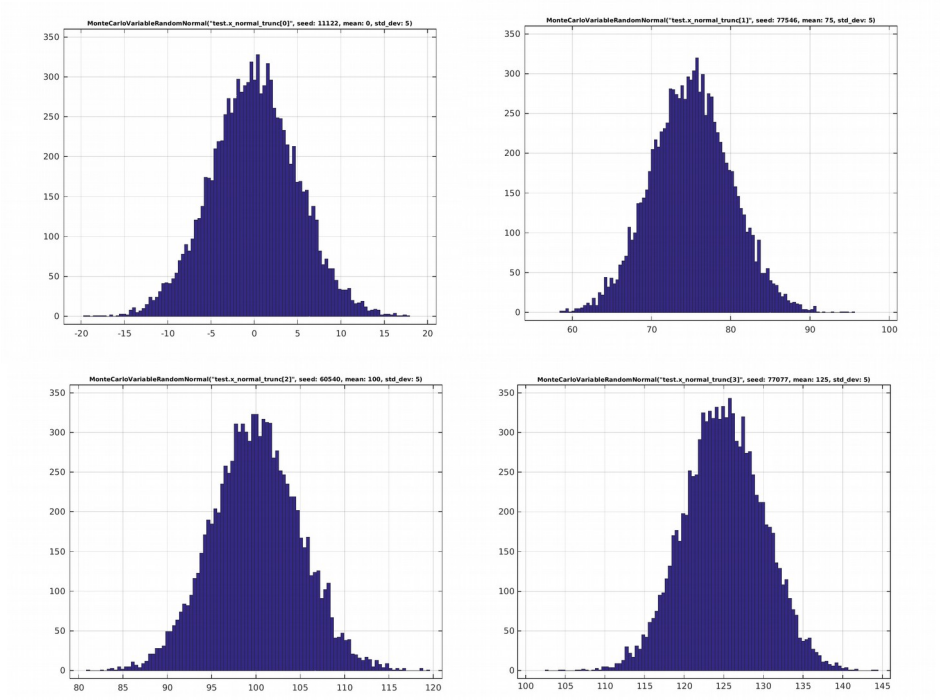
5.3.1 Uniform Distribution

Here we test the distribution of both continuous (left) and discrete (right) variables.



5.3.2 Normal Distribution

For the analysis of the normal distribution, we start with 4 unique distributions, illustrated below:



Any normal distribution may be truncated. As we saw in section 5.1.3, a normal distribution can be truncated according to one of 3 methods for specifying the range:

- a prescribed range (*Absolute*)
- some number of standard deviations relative to the mean (*StandardDeviation*)
- some a prescribed range relative to the mean (*Relative*)

For each of these options, there are 4 options for specifying how to truncate:

- symmetric (about the mean or about 0) – uses truncate(…) with 1 numerical argument; the truncation is applied within +- the value specified in the argument.
- asymmetric, truncated on both sides – uses truncate(…) with 2 numerical arguments; the first argument provides the left-truncation and the 2nd argument the right-truncation.
- truncated on the left only – uses truncate_low(…) with 1 numerical argument specifying the left-truncation.
- truncated on the right only – uses truncate_high(…) with 1 numerical argument specifying the right-truncation.

These 12 options will be investigated in more detail here, with graphical illustrations of the consequences of each.

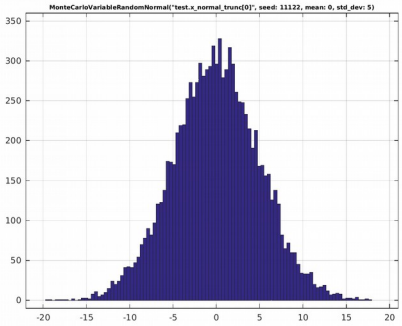
For each case, two plots are shown:

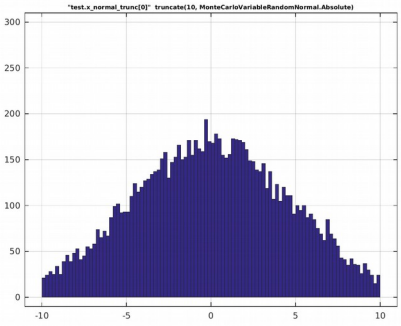
1. the plot on the left is that of a distribution with no truncation applied
2. the lot on the right is that of the same distribution with the truncation applied.

5.3.2.1 Truncated by Prescribed Range

symmetric (about 0)

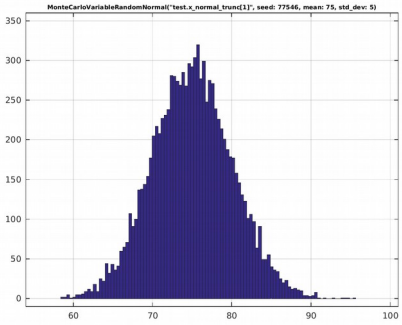
truncate(10, Absolute)

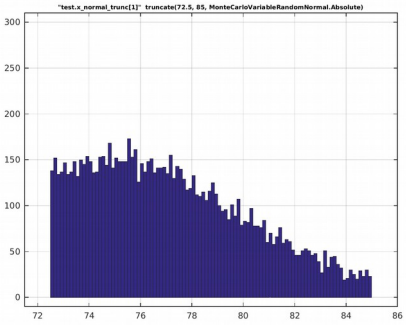




asymmetric, truncated on both sides

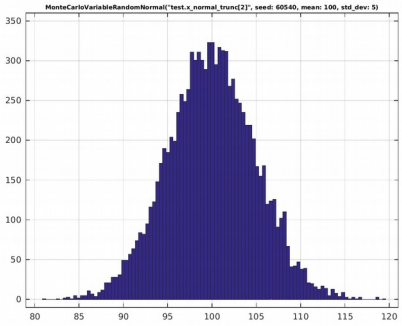
truncate(72.5, 85, Absolute)

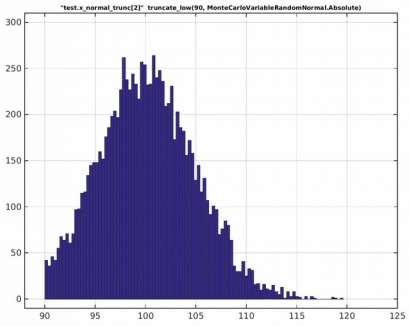




truncated on the left only

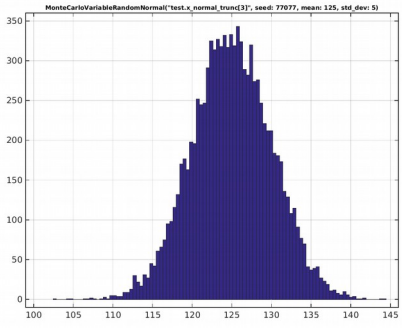
truncate_low(90, Absolute)

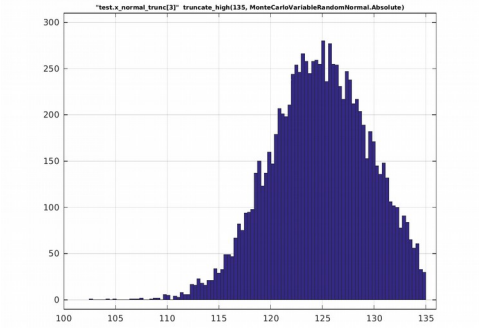




truncated on the right only

truncate_high(135, Absolute)

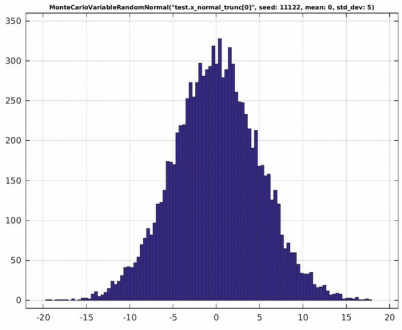


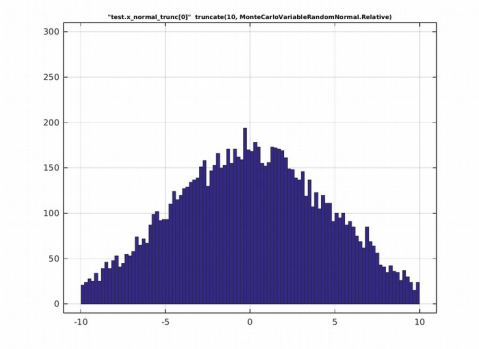


5.3.2.2 Truncated by Difference from Mean

symmetric (about the mean)

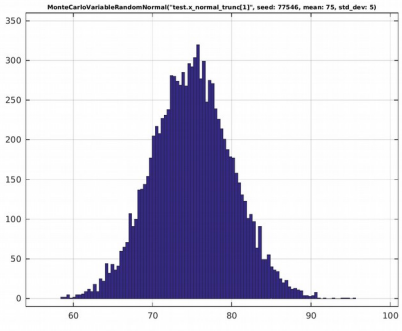
truncate(10,Relative)

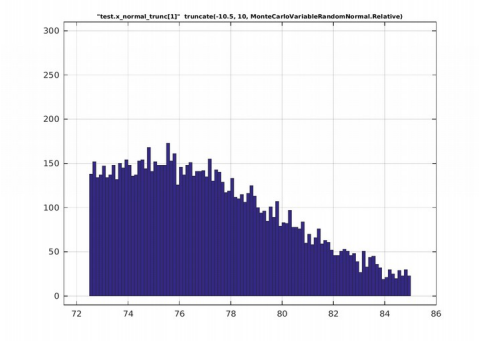




asymmetric, truncated on both sides

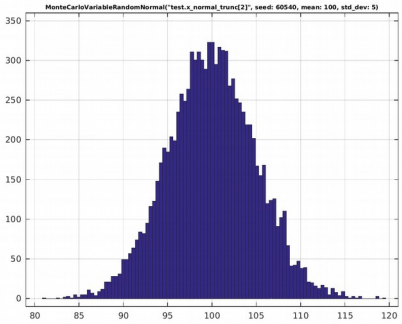
truncate(-2.5,10,Relative)

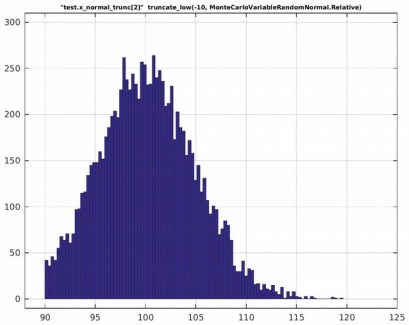




truncated on the left only

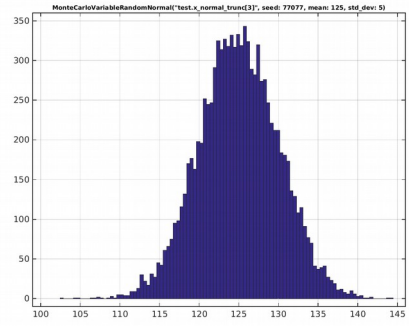
truncate_low(-10, Relative)

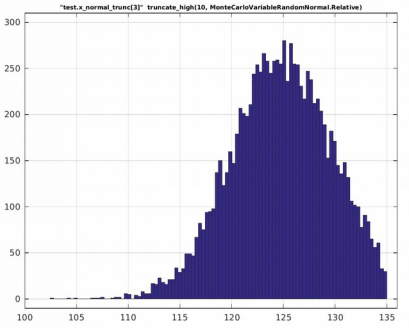




truncated on the right only

truncate_high(10, Relative)

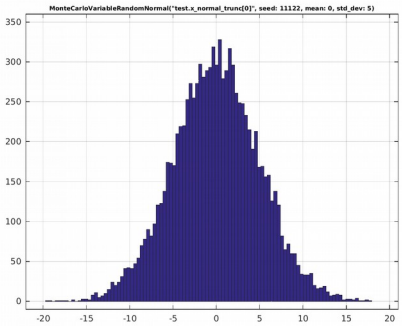


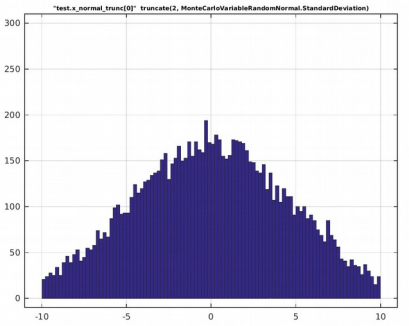


5.3.2.3 Truncated by Standard Deviations from Mean

symmetric (about the mean)

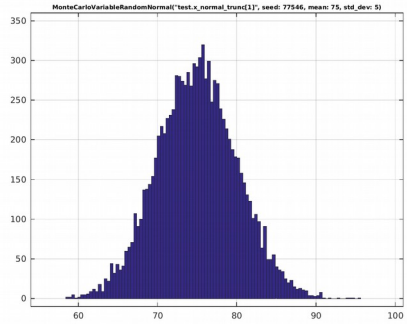
truncate(2, StandardDeviation)

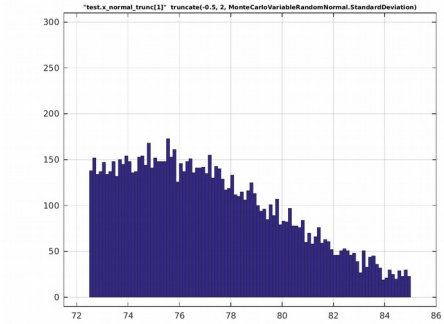




asymmetric, truncated on both sides

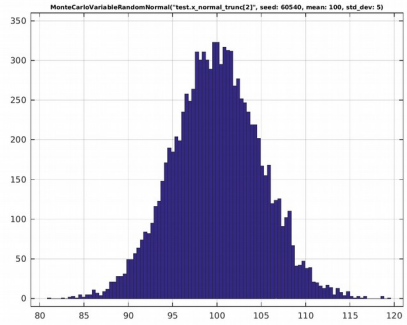
truncate(-0.5, 2.0, StandardDeviation)

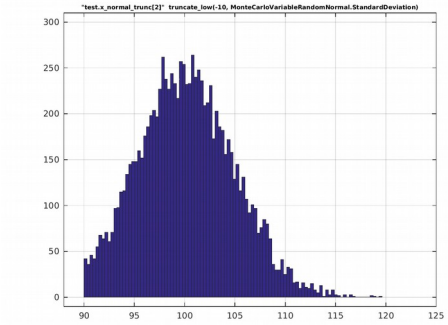




truncated on the left only

truncate_low(-2, StandardDeviation)





truncated on the right only

truncate_high(2, StandardDeviation)