MonteCarloGenerate (MCG)

Model Verification Documentation

Originally written by Gary Turner (Odyssey Space Research) 2020

Integrated into Trick Open Source in 2023 by

Isaac Reaves (NASA) in 2022

Dan Jordan (NASA) in 2023

See the official Trick documentation for MCG details and the user guide

MonteCarloGenerate Model (Verification Only)

1 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.

1.1 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

1.1.1 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

1.1.2 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
```

1.1.3 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.

1.1.3.1 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

1.1.3.2 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

1.1.3.3 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

1.1.4 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

1.1.5 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
```

1.1.6 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 (ft)	value	9.954870507	11.32489560	8.225020001	9.078215205	8.612318016
converted (m)	value	3.034244531	3.451828181	2.506986096	2.767039994	2.625034531
logged value		3.034244531	3.451828181	2.506986096	2.767039994	2.625034531

1.1.7 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

1.1.8 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

1.1.9 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.

1.1.10 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

1.1.10.1 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

1.1.10.2 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.

1.1.10.3 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

1.1.11 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

1.1.12 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

1.1.13 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

1.2 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.

1.2.1 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.

1.2.2 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):

1.2.3 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:

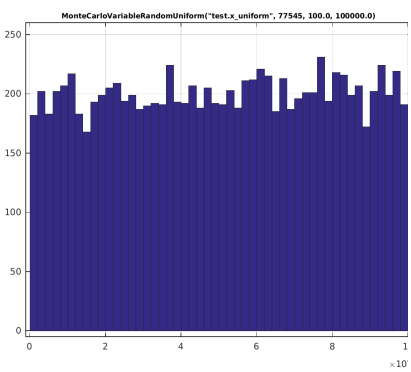
run number	x_file_lookup[0]	x_file_lookup[1]	x_file_lookup[2]	line numbers			lines skipped		
0	1	16	31	1	1	1			
1	2	17	32	2	2	2	0	0	0
2	4	20	36	4	5	6	1	2	3
3	5	22	38	5	7	8	0	1	1
4	7	24	41	7	9	11	1	1	2

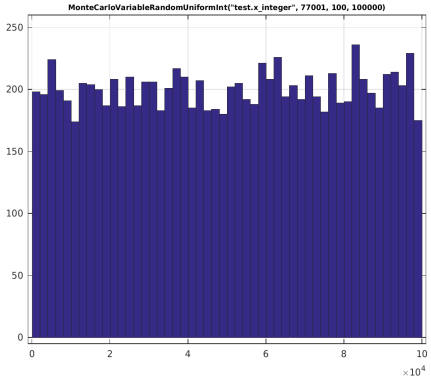
1.3 Distribution Analyses

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

1.3.1 Uniform Distribution

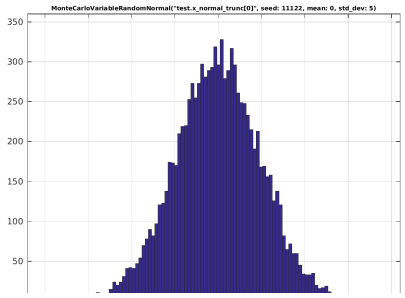
Here we test the distribution of both continuous (left) and discrete (right) variables.

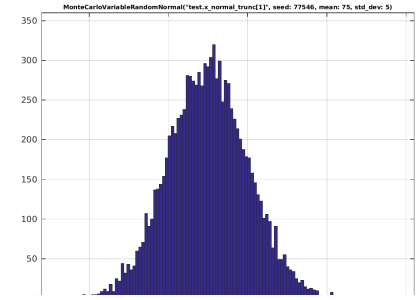


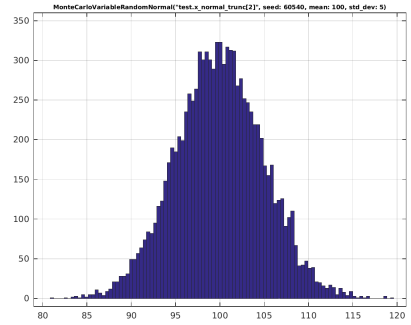


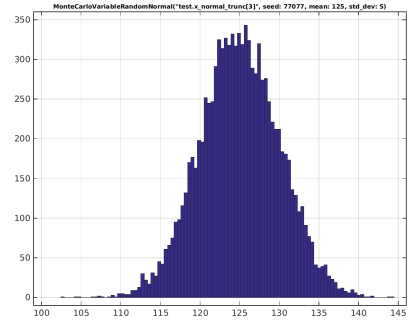
1.3.2 Normal Distribution

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.

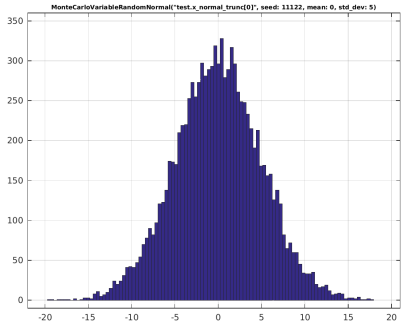
These 12 options will be investigated in more detail here, with graphical illustrations of the consequences of each. For each case, two plots are shown:

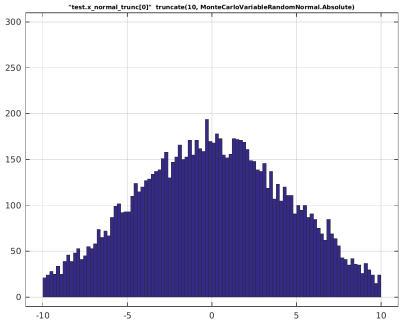
1. the plot on the left is that of a distribution with no truncation applied
2. the lot on the right is that of the same distribution with the truncation applied.

1.3.2.1 Truncated by Prescribed Range

symmetric (about 0)

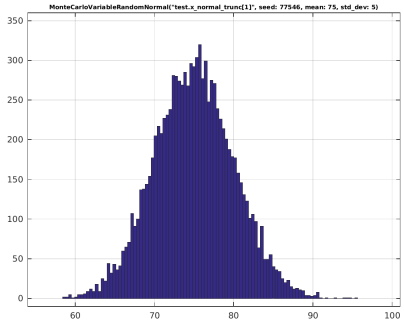
truncate(10, Absolute)

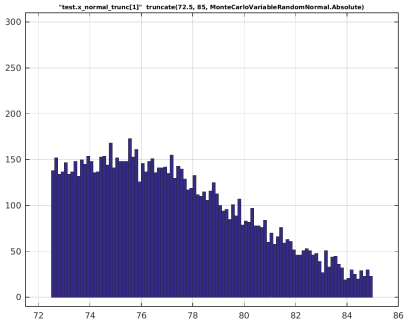




asymmetric, truncated on both sides

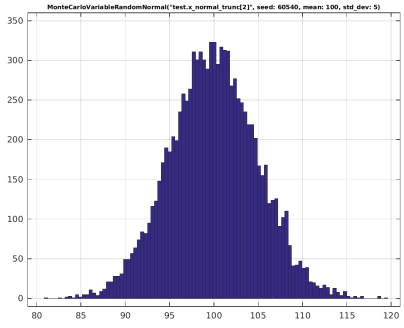
truncate(72.5, 85, Absolute)

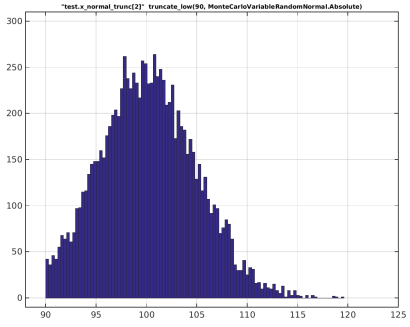




truncated on the left only

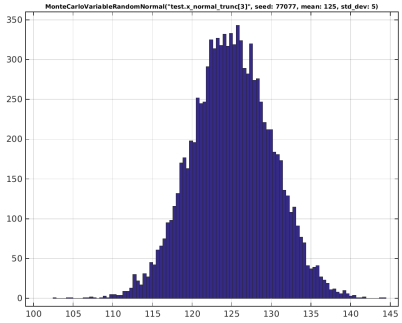
truncate_low(90, Absolute)

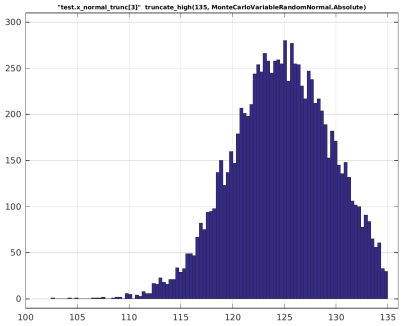




truncated on the right only

truncate_high(135, Absolute)

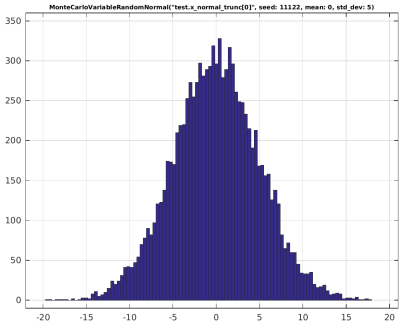


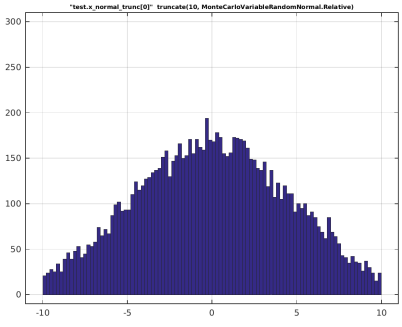


1.3.2.2 Truncated by Difference from Mean

symmetric (about the mean)

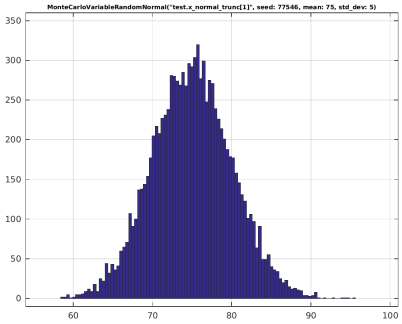
truncate(10,Relative)

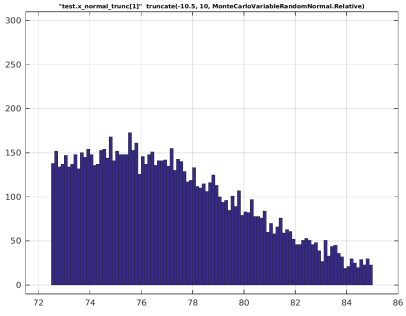




asymmetric, truncated on both sides

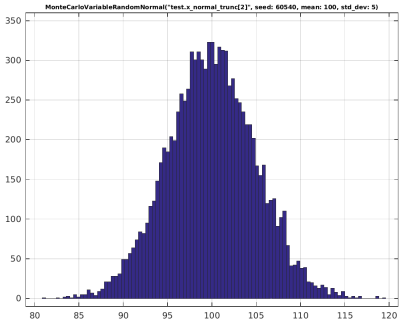
truncate(-2.5,10,Relative)

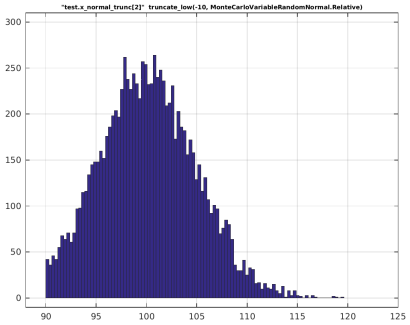




truncated on the left only

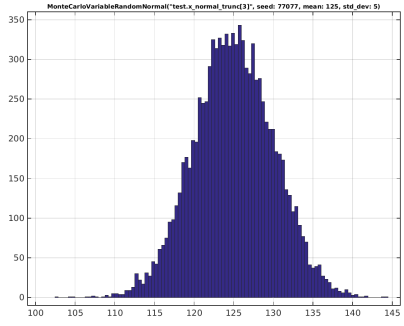
truncate_low(-10, Relative)

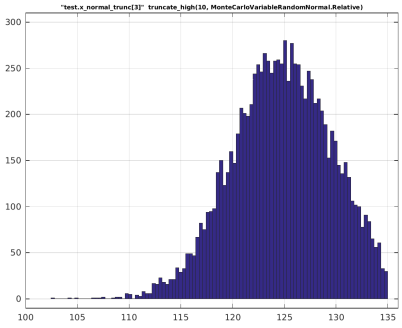




truncated on the right only

truncate_high(10, Relative)

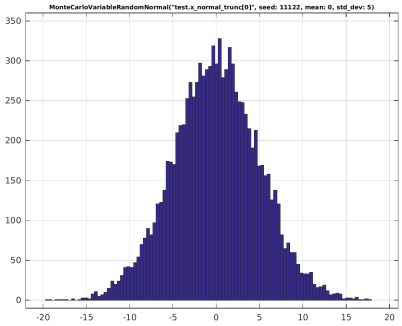


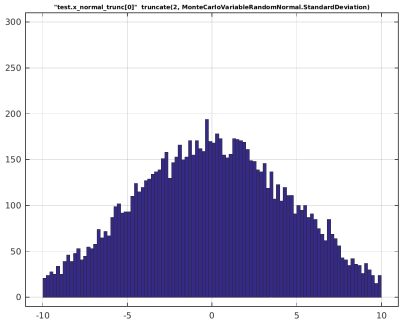


1.3.2.3 Truncated by Standard Deviations from Mean

symmetric (about the mean)

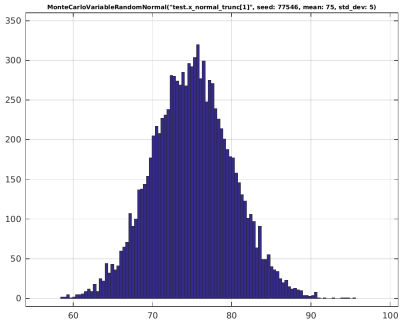
truncate(2, StandardDeviation)

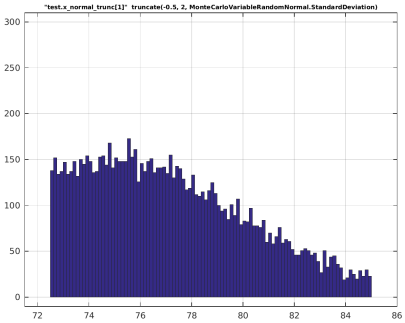




asymmetric, truncated on both sides

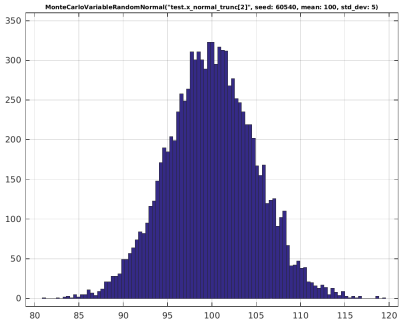
truncate(-0.5, 2.0, StandardDeviation)

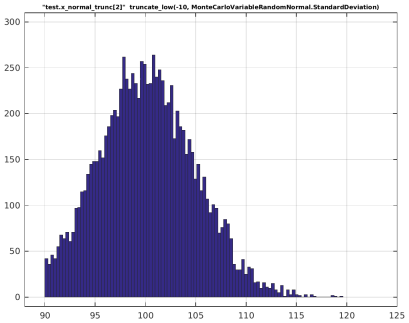




truncated on the left only

truncate_low(-2, StandardDeviation)





truncated on the right only

truncate_high(2, StandardDeviation)