SROMPy Documentation

Release 1.0

James Warner

Feb 06, 2019

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

Stochastic reduced order models with Python (SROMPy) is a software package developed to enable user-friendly utilization of the stochastic reduced order model (SROM) approach for uncertainty quantification. A SROM is a low dimensional, discrete approximation to a random quantity that enables efficient and non-intrusive stochastic computations. With SROMPy, a user can easily generate a SROM to approximate a random variable or vector described by several different types of probability distributions using the Python programming language. Once a SROM is constructed, the software also contains functionality to propagate uncertainty through a user-defined computational model to estimate statistics of a given quantity of interest.

CHAPTER

TWO

EXAMPLE - SPRING MASS SYSTEM

This example provides a simple demonstration of SROMPy functionality. The goal is to estimate the maximum displacement of a spring-mass system with random stiffness using the SROM approach and compare the solution to Monte Carlo simulation. The example covers all steps for computing a solution using SROMs using SROMPy, including defining a random input parameter (spring stiffness) using a SROMPy target random variable, modeling the random input using a SROM, and propagating the uncertainty through a computational model (spring mass numerical integrator) to a quantity of interest (max. displacement). The full source code for this example can be found in the SROMPy repository: `/SROMPy/examples/spring_mass/run_spring_mass_1D.py`

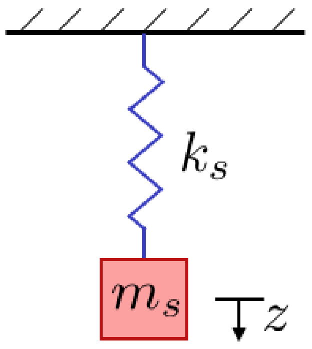


Fig. 1: Spring-mass system

2.1 Problem Specification

The governing equation of motion for the system is given by

$$m_s\ddot{z} = -k_s z + m_s g \tag{2.1}$$

where m_s is the mass, k_s is the spring stiffness, g is the acceleration due to gravity, z is the vertical displacement of the mass, and $\ddot{z}$ is the acceleration of the mass. The source of uncertainty in the system will be the spring stiffness, which is modeled as a random variable of the following form:

$$K_s = \gamma + \eta B \tag{2.2}$$

where γ and η are shift and scale parameters, respectively, and $B = \text{Beta}(\alpha, \beta)$ is a standard Beta random variable with shape parameters α and β. Let these parameters take the following values: $\gamma = 1.0N/m$, $\eta = 2.5N/m$, $\alpha = 3.0$, and $\beta = 2.0$. The mass is assumed to be deterministic, $m_s = 1.5kg$, and the acceleration due to gravity is $g = 9.8m^2/s$.

With uncertainty in an input parameter, the resulting displacement, Z, is a random variable as well. The quantity of interest in this example with be the maximum displacement over a specified time window, $Z_{max} = max_t(Z)$. It is assumed we have access to a computational model that numerically integrates the governing equation over this time window for a given sample of the random stiffness and returns the maximum displacement. The goal of this example will be to approximate the mean, $E[Z_{max}]$, and CDF, $F(z_{max})$, using the SROM approach with SROMPy and compare it to a Monte Carlo simulation solution.

2.2 Step 1: Define target random variable, initialize model, generate reference solution

Begin by importing the needed SROMPy classes as well as the SpringMassModel class that defines the spring mass model:

```python
import numpy as np
from spring_mass_model import SpringMassModel

#import SROMPy modules
from SROMPy.postprocess import Postprocessor
from SROMPy.srom import SROM, FiniteDifference as FD, SROMSurrogate
from SROMPy.target import SampleRandomVector, BetaRandomVariable
```

The first step in the analysis is to define the target random variable to represent the spring stiffness K_s using the `BetaRandomVariable` class in SROMPy:

```python
#Random variable for spring stiffness
stiffness_random_variable = BetaRandomVariable(alpha=3.,beta=2.,shift=1.,scale=2.5)
```

Next, the computational model of the spring-mass system is initialized:

```python
#Specify spring-mass system and initialize model:
m = 1.5
state0 = [0., 0.]
time_step = 0.01
model = SpringMassModel(m, state0=state0, time_step=time_step)
```

The source code for the spring mass model can be found in the SROMPy repository as well: `SROMPy/examples/spring_mass/spring_mass_model.py`

A reference solution using Monte Carlo simulation is now generated for comparison later on. This is done by sampling the random spring stiffness, evaluating the model for each sample, and then using the SROMPy `SampleRandomVector` class to represent the Monte Carlo solution for maximum displacement:

```python
#----------Monte Carlo------------------
#Generate stiffness input samples for Monte Carlo
num_samples = 5000
stiffness_samples = stiffness_random_variable.draw_random_sample(num_samples)

# Calculate maximum displacement samples using MC simulation.
displacement_samples = np.zeros(num_samples)
for i, stiff in enumerate(stiffness_samples):
    displacement_samples[i] = model.evaluate([stiff])

# Get Monte carlo solution as a sample-based random variable:
monte_carlo_solution = SampleRandomVector(displacement_samples)
```

2.3 Step 2: Construct SROM for the input

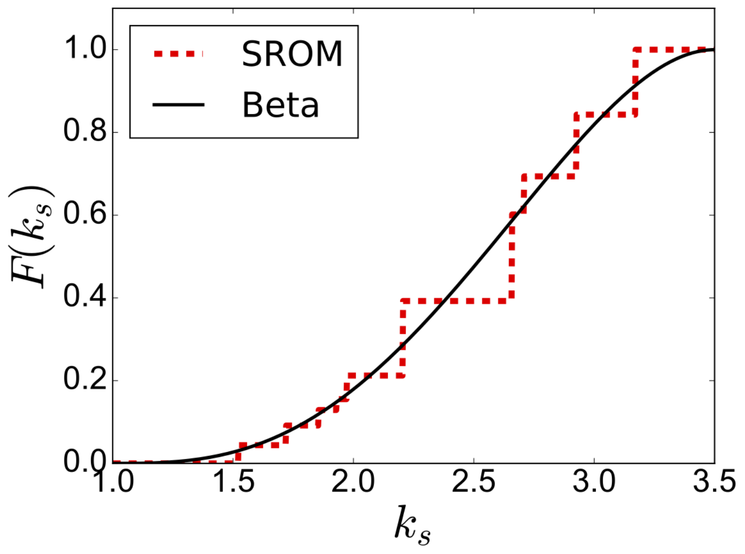
A SROM, $\tilde{K}_s$ is now formed to model the random stiffness input, K_s, with SROMPy. The following code initializes the SROM class for a model size of 10 and uses the optimize function to set the optimal SROM parameters to represent the random spring stiffness:

```
#Generate SROM for random stiffness
sromsize = 10
dim = 1
input_srom = SROM(sromsize, dim)
input_srom.optimize(stiffness_random_variable)
```

The CDF of the resulting SROM can be compared to the original Beta random variable for spring stiffness using the SROMPy `Postprocessor` class:

```
#Compare SROM vs target stiffness distribution:
pp_input = Postprocessor(input_srom, stiffness_random_variable)
pp_input.compare_CDFs()
```

This produces the following plot:



2.4 Step 3: Evaluate model for each SROM sample:

Now output samples of maximum displacement must be generated by running the spring-mass model for each stiffness sample from the input SROM, i.e.,

$\tilde{z}^{(k)}_{max} = \mathcal{M}(\tilde{k}^{(k)}_s)$ for $k = 1, ..., m$

Note that this is essentially a Monte Carlo simulation step, but with far fewer model evaluations using the SROM method (10 versus 5000)

This is done with the following code:

```
#run model to get max disp for each SROM stiffness sample
srom_displacements = np.zeros(srom_size)
(samples, probabilities) = input_srom.get_params()
for i, stiff in enumerate(samples):
    srom_displacements[i] = model.evaluate([stiff])
```

Here, the spring-mass model is executed for each of the 10 optimal stiffness samples found in Step 2, and the corresponding maximum displacements are stored for the next step.

2.5 Step 4: Form SROM surrogate model for output

2.5.1 Approach a) Piecewise-constant approximation

The simplest way to propagate uncertainty using SROMs is to form a piecewise-constant approximation that directly uses the model outputs obtained in Step 3 and the input SROM probabilities found in Step 2. This is done by constructing a new SROM for the model output (max. displacement) as follows:

```
# Form new SROM for the max disp. solution using samples from the model.
output_srom = SROM(srom_size, dim)
output_srom.set_params(srom_displacements, probabilities)
```

The mean of the output can now be estimated using the SROM and the SROMPy `compute_moments` function and compared to Monte Carlo as follows:

```
#Compare mean estimates for output:
print "Monte Carlo mean estimate: ", np.mean(displacement_samples)
print "SROM mean estimate: ", output_srom.compute_moments(1)[0][0]
```

The max. displacement CDF estimate using SROMs can be compared to the Monte Carlo solution using the SROMPy `Postprocessor` as follows:

```
#Compare solutions
pp_output = Postprocessor(output_srom, monte_carlo_solution)
pp_output.compare_CDFs(variablenames=[r'$Z_{max}$'])
```

This produces the following comparison plot:

2.5.2 Approach b) Piecewise-linear approximation

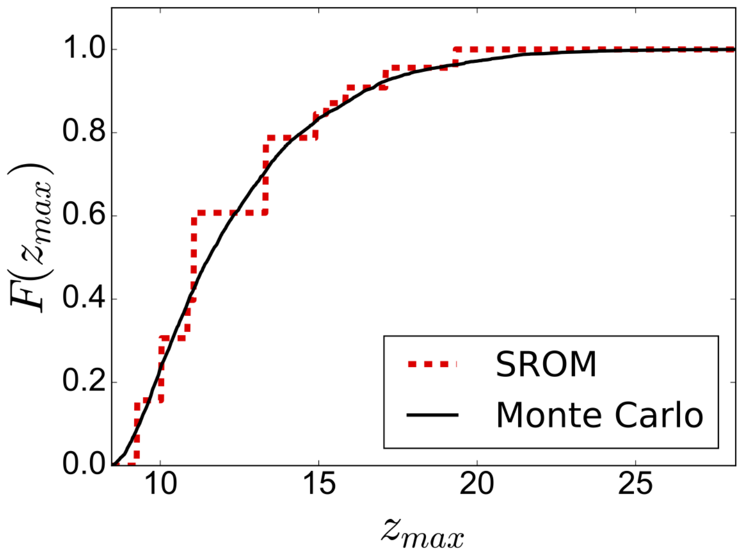
Now a more accurate piecewise-linear SROM surrogate model is formed to estimate the CDF of the maximum displacement. To do so, gradients must be calculated using finite difference and provided to the SROMSurrogate class upon initialization.

The finite different gradients are calculated with the help of the FiniteDifference class (FD), requiring extra model evaluations for perturbed inputs:

```
#Perturbation size for finite difference
stepsize = 1e-12
samples_fd = FD.get_perturbed_samples(samples, perturb_vals=[stepsize])

# Run model to get perturbed outputs for FD calc.
perturbed_displacements = np.zeros(srom_size)
for i, stiff in enumerate(samples_fd):
```

(continues on next page)



(continued from previous page)

```
    perturbed_displacements[i] = model.evaluate([stiff])
gradient = FD.compute_gradient(srom_displacements, perturbed_displacements,
                               [step_size])
```

A piecewise-linear surrogate model can now be constructed and then sampled to approximate the CDF of the maximum displacement:

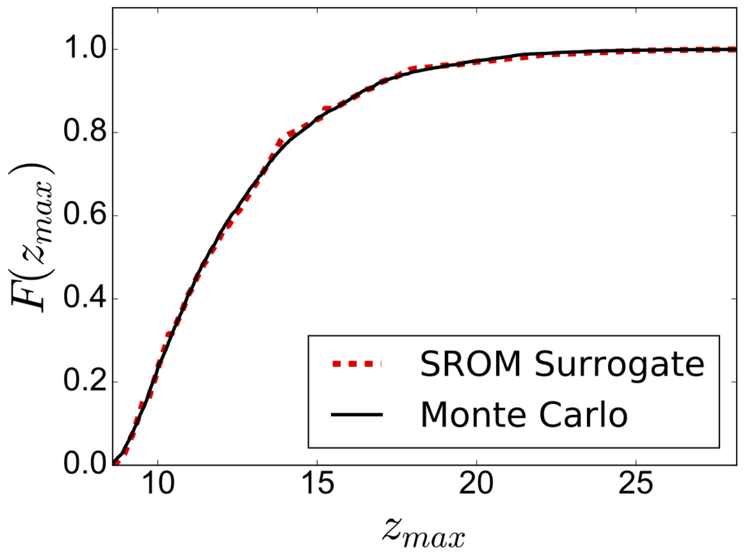
```
#Initialize piecewise-linear SROM surrogate w/ gradients:
surrogate_PWL = SROMSurrogate(input_srom, srom_displacements, gradient)

#Use the surrogate to produce max disp samples from the input stiffness samples:
output_samples = surrogate_PWL.sample(stiffness_samples)

#Represent the SROM solution as a sample-based random variable:
solution_PWL = SampleRandomVector(output_samples)
```

Finally, the new piece-wise linear CDF approximation is compared to the Monte Carlo solution:

```
#Compare SROM piecewise linear solution to Monte Carlo
pp_pwl = Postprocessor(solution_PWL, monte_carlo_solution)
pp_pwl.compare_CDFs(variablenames=[r'$Z_{max}$'])
```


1.0
0.8
0.6
0.4
0.2
0.0
10
15
20
25
$F(z_{max})$
z_{max}
SROM Surrogate
Monte Carlo

CHAPTER

THREE

SOURCE CODE DOCUMENTATION

Documentation for the primary SROMPy classes.

3.1 SROM Module Documentation

class `SROMPy.srom.`**`SROM`**(*size*, *dim*)

This is the primary SROMPy class for defining and utilizing a stochastic reduced order model (SROM). Main capability is optimizing for the defining SROM parameters to model a given target random quantity. Other functions provided to calculate SROM statistics, set/get defining parameters directly, and store/load SROM to/from file.

Parameters

- **`size`** (*`int`*) – SROM size
- **`dim`** (*`int`*) – dimension of random quantity being modeled

`compute_cdf`(*x_grid*)

Computes the SROM marginal CDF values in each dimension.

Parameters **`x_grid`** (*`Numpy array.`*) – Grid of points to compute CDF values on. If 1d array is provided, the same points are used to evaluate CDF in each dimension. If 2d array is provided, calculates CDF values on different points, but must have same # points for each dimension. Size is (# grid pts) x (dim) or (# grid pts) x (1).

Returns: Numpy array of CDF values at x_grid points. Size is (# grid pts) x (dim).

Note:

- Increasing the number of grid points can significantly slow down the SROM optimization problem.
- Providing a 2d array for x_grid can specify a different range of values for each dimension, but must use the same number of pts.

`compute_corr_mat`()

Returns the SROM correlation matrix as (dim x dim) numpy array

srom_corr = sum_{k=1}^m [x^(k) * (x^(k))^T] * p^(k)

`compute_moments`(*max_order*)

Calculates and returns SROM moments.

Parameters **`max_order`** (*`int`*) – Maximum order of moments to return

Returns (max_order x dim) size Numpy array with SROM moments for each dimension.

get_params()

Returns: tuple of SROM sample & probability arrays. Samples array has size (SROM size x dim) and probability array has length (SROM size)

The sample/probability arrays have the following convention (srom sample index as rows, components of sample as columns):

Samples:

[[x_1^(1), x_2^(1), . . . , x_d^(1)],
[x_1^(2), x_2^(2), . . . , x_d^(2)],
. . . .
[x_1^(m), x_2^(m), . . . x_d^(m)]]

Probabilities:

[p^(1), p^(2), . . . , p^(m)]^T

load_params(*infile='srom_params.txt'*, *delimiter=' '*)

Load SROM parameters from file.

Parameters

- **infile** (*string*) – input file name containing SROM parameters
- **delimiter** (*string*) – delimiter used in input file (default - whitespace)

Returns: None. Sets sample/probability member variables.

Assumes input file has the following format (samples in each row with prob after):

x_1^(1), x_2^(1), . . . , x_d^(1), p^(1)
x_1^(2), x_2^(2), . . . , x_d^(2), p^(2)
. . . .
x_1^(m), x_2^(m), . . . x_d^(m), p^(m)

The dimension of the samples and probabilities arrays must be compatible with the SROM size and dimension that was used to initialize the SROM class.

optimize(*target_random_variable*, *weights=None*, *num_test_samples=50*, *error='SSE'*, *max_moment=5*, *cdf_grid_pts=100*, *tolerance=None*, *options=None*, *method=None*, *joint_opt=False*)

Optimize for the SROM samples & probabilities to best match the target random vector statistics. The main functionality provided by the SROM class. Solves SROM the optimization problem and sets the samples and probabilities for the SROM object to the optimized values.

Parameters

- **target_random_variable** (*SROMPy target object (AnalyticRandomVector, SampleRandomVector, or random variable class)*) – the target random quantity (variable/vector) being modeled by the SROM.

- **weights** (*1d Numpy array (length = 3)*) – relative weights specifying importance of matching CDFs, moments, and correlation of the target during optimization. Default is equal weights [1,1,1].
- **num_test_samples** (*int*) – Number of sample sets (iterations) to run optimization.
- **error** (*string*) – Type of error metric to use in objective ("SSE", "MAX", "MEAN").
- **max_moment** (*int*) – Max. number of target moments to consider matching
- **cdf_grid_pts** (*int*) – Number of points to evaluate CDF error on
- **tolerance** (*float*) – tolerance for scipy optimization algorithm (TODO)
- **options** (*dict*) – scipy optimization algorithm options (TODO)
- **method** (*string*) – method used for scipy optimization (TODO)
- **joint_opt** (*bool*) – Flag to optimize jointly for samples & probabilities.

Returns: None. Sets samples/probabilities member variables.

Assumes the targetRV object has been properly initialized beforehand. The optimization for SROM samples & probabilities is currently performed sequentially - a random set of samples are first drawn and the probabilities are then optimization for those samples. The input "num_test_samples" is the number of random sample sets this is performed for before terminating. The random sample set and optimal probabilities found that produce the lowest objective function value are used as the optimal parameters. The joint_opt input flag can specify to do the optimization over samples and probabilities simultaneously.

save_params (*outfile='srom_params.txt'*, *delimiter=' '*)
Write the SROM parameters to file.

Parameters

- **outfile** (*string*) – output file name
- **delimiter** (*string*) – delimiter used in output file (default - whitespace)

Returns: None. Produces output file.

Writes output file with the following format (samples in each row with prob after):

x_1^(1), x_2^(1), . . . , x_d^(1), p^(1)
x_1^(2), x_2^(2), . . . , x_d^(2), p^(2)
. . . .
x_1^(m), x_2^(m), . . . x_d^(m), p^(m)

set_params (*samples*, *probabilities*)
Set defining SROM parameters - samples & corresponding probabilities.

Parameters

- **samples** (*2d Numpy array, size - (SROM size) x (dim)*) – Array of SROM samples
- **probabilities** (*1d Numpy array, size - (SROM size) x 1*) – Array of SROM probabilities

The sample/probability arrays have the following convention (srom sample index as rows, components of sample as columns):

Samples:

[[x_1^(1), x_2^(1), . . . , x_d^(1)],
[x_1^(2), x_2^(2), . . . , x_d^(2)],
. . . .
[x_1^(m), x_2^(m), . . . x_d^(m)]]

Probabilities:

[p^(1), p^(2), . . . , p^(m)]^T

class `SROMPy.srom.`**`SROMSurrogate`**(*input_srom*, *output_samples*, *output_gradients=None*)

SROMPy class that provides a closed-form surrogate model for a model output that can be sampled as a means of efficiently propagating uncertainty. Enables both a piecewise-constant model and a piecewise-linear model, if gradient information is provided.

Conventions:

- m denotes the SROM size (superscripts). di denotes the dimension of the SROM input (subscripts). do denotes dimension of SROM output (subscripts).
- The output samples array has the following layout (m x d0):

  [[y^(1)_1, y_2^(1), . . . , y_do^(1)],
  [y_1^(2), y_2^(2), . . . , y_d0^(2)],
  . . . .
  [y_1^(m), y_2^(m), . . . y_d0^(m)]]

- The gradients array has the following layout (m x di):

  [[dy(x^{(1)})/dx_1, . . . , dy(x^{(1)})/dx_di],
  . . . , . . . , . . .
  [dy(x^{(m)})/dx_1, . . . , dy(x^{(m)})/dx_di]]

Note

- the order of the output samples array must match the order of the samples array from the input SROM!
- If gradients array is provided, the piecewise-linear surrogate model is implemented. Otherwise, the piecewise-constant surrogate is used.

`compute_cdf`(*x_grid*)

Computes the SROM marginal CDF values in each dimension.

Parameters **`x_grid`** (*`Numpy array.`*) – Grid of points to compute CDF values on. If 1d array is provided, the same points are used to evaluate CDF in each dimension. If 2d array is provided, calculates CDF values on different points, but must have same # points for each dimension. Size is (# grid pts) x (dim) or (# grid pts) x (1).

Returns: Numpy array of CDF values at x_grid points. Size is (# grid pts) x (dim).

Note:

- Increasing the number of grid points can significantly slow down the SROM optimization problem.
- Providing a 2d array for x_grid can specify a different range of values for each dimension, but must use the same number of pts.

compute_moments(*max_order*)

Calculates and returns SROM moments.

Parameters **max_order** (`int`) – Maximum order of moments to return

Returns (max_order x dim) size Numpy array with SROM moments for each dimension.

sample(*input_samples*)

Generates output samples from the SROM surrogate corresponding to the provided input samples.

Parameters **input_samples** (`2d Numpy array.`) – samples of inputs to draw output samples for

Returns: 2d Numpy array of output samples corresponding to input samples

Convention:

- N - number of samples. di - dimension of the input. do - dimension of the output.
- input samples array has following layout (N x di):

  [[x^(1)_1, . . . , x^(1)_di],
  . . . , . . . , . . .
  [x^(N)_1, . . . , x^(N)_di]]

- surrogate output samples has following layout (N x do):

  [[y^(1)_1, . . . , y^(1)_do],
  . . . , . . . , . . .
  [y^(N)_1, . . . , y^(N)_do]]

Note that the samples are drawn from a piecewise-linear SROM surrogate when gradients are provided to the constructor of this class, and drawn from a piecewise-constant SROM surrogate if not.

class SROMPy.srom.**FiniteDifference**

Class that contains static methods for assisting in computing gradients needed to implement the piecewise-linear SROM surrogate using the finite difference method.

static compute_gradient(*outputs*, *perturbed_outputs*, *perturbation_values*)

Calculates gradients based on original sample outputs, perturbed sample outputs, and the size or perturbations.

NOTE - it is being assumed here and other places that the output is scalar

inputs: (mx1 array) outputs = | y(x^(1))|

. . . |
y(x^(m))|

(mxd array) perturbed_outputs = | y(x^(1) + delta_1), . . . , y(x^(1)+ delta_d)|

. . . , . . . , . . . |
y(x^(m) + delta_1), . . . , y(x^(m)+delta_d)|

(dx1 array) perturbed_values = [delta_1, . . . , delta_d]

outputs: (mxd array) gradients = | dy(x^{(1)})/dx_1, . . . , dy(x^{(1)})/dx_d |

. . . , . . . , . . . |
dy(x^{(m)})/dx_1, . . . , dy(x^{(m)})/dx_d |

static get_perturbed_samples(*samples*, *perturbation_factor=None*, *perturbation_values=None*)

Returns the perturbed SROM samples that must be run through model to estimate gradients with finite difference.

input: samples: np array (m x d) - original input srom samples perturbation_factor: float - if specified, computes the

perturbation size in each dimension as (max_i - min_i)*perturbation_factor. max/min_i are the max/min sample values in dim. i.

perturbation_values: list of float - if specified uses the values in the array for perturbations in each dimension.

-Must specify either perturbation_factor or perturbation_values

output: returns perturbed_samples: np array (m*d x d) samples = | x^(1)_1 + delta_1, . . . , x^(1)_d |

. . . , . . . , . . . |
x^(m)_1 + delta_1, . . . , x^(m)_d | . . . .
x^(1)_1, . . . , x^(1)_d + delta_d|
. . . , . . . , . . . |
x^(m)_1, . . . , x^(m)_d + delta_d|

3.2 Target Random Quantity Documentation

class SROMPy.target.**NormalRandomVariable**(*mean=0.0*, *std_dev=1.0*, *max_moment=10*)

Class for defining a normal random variable

compute_cdf(*x_grid*)
Returns numpy array of normal CDF values at the points contained in x_grid

compute_inv_cdf(*x_grid*)
Returns np array of inverse normal CDF values at pts in x_grid

compute_moments(*max_order*)
Returns moments up to order 'max_order' in numpy array.

compute_pdf(*x_grid*)
Returns numpy array of normal pdf values at the points contained in x_grid

draw_random_sample(*sample_size*)
Draws random samples from the normal random variable. Returns numpy array of length 'sample_size' containing these samples

generate_moments(*max_moment*)
Calculate & store moments to retrieve more efficiently later

get_variance()
Returns variance of normal random variable

class SROMPy.target.**BetaRandomVariable**(*alpha*, *beta*, *shift=0*, *scale=1*, *max_moment=10*)

Class for implementing a beta random variable

compute_cdf(*x_grid*)
Returns numpy array of beta CDF values at the points contained in x_grid

compute_inv_cdf(*x_grid*)
Returns np array of inverse beta CDF values at pts in x_grid

compute_moments(*max_order*)
Returns moments up to order 'max_order' in numpy array.

compute_pdf(*x_grid*)
Returns numpy array of beta pdf values at the points contained in x_grid

draw_random_sample(*sample_size*)
Draws random samples from the beta random variable. Returns numpy array of length 'sample_size' containing these samples

generate_moments(*max_moment*)
Calculate & store moments to retrieve more efficiently later

static get_beta_shape_params(*min_value*, *max_value*, *mean*, *variance*)
Returns the beta shape parameters (alpha, beta) and the shift/scale parameters that produce a beta random variable with the specified minimum value, maximum value, mean, and variance. Can be called prior to initialization of this class if only this info is known about the random variable being modeled. Returns a list of length 4 ordered [alpha, beta, shift, scale]

get_variance()
Returns variance of beta random variable

class SROMPy.target.**DiscreteRandomVector**(*samples*, *probabilities*, *max_moment=10*)
Discrete random vector. Defines a target that can be matched with a SROM that is created from samples and corresponding probabilities. Implements basic discrete statistics (similar to those of an SROM).

Parameters

- **samples** (`np array, size: (# samples x dim)`) – set of realizations/samples of the random vector
- **probabilities** (`np array, length = # samples`) – probabilties associated with each sample
- **max_moment** (`int`) – max. order moment to precompute and store

compute_cdf(*x_grid*)
Computes the marginal CDF values in each dimension.

Parameters x_grid (`Numpy array.`) – Grid of points to compute CDF values on. If 1d array is provided, the same points are used to evaluate CDF in each dimension. If 2d array is provided, calculates CDF values on different points, but must have same # points for each dimension. Size is (# grid pts) x (dim) or (# grid pts) x (1).

Returns: Numpy array of CDF values at x_grid points. Size is (# grid pts) x (dim).

Note:

- Increasing the number of grid points can significantly slow down the SROM optimization problem.
- Providing a 2d array for x_grid can specify a different range of values for each dimension, but must use the same number of pts.

compute_correlation_matrix()
Returns precomputed correlation matrix.

compute_moments(*max_order*)
Return precomputed moments up to specified order.

Parameters max_order (`int`) – Maximum order of moments to return

Returns (max_order x dim) size Numpy array with SROM moments for each dimension.

`draw_random_sample`(*sample_size*)
Randomly draws a sample of this random vector.

Parameters **`sample_size`** (`int`) – number of samples to return

sample_size must be smaller than total # of samples. For discrete random vector, we return a randomly selected # of samples

`class` `SROMPy.target.`**`GammaRandomVariable`**(*alpha*, *shift=0*, *scale=1*, *max_moment=10*)
Class for implementing a gamma random variable

`compute_cdf`(*x_grid*)
Returns numpy array of gamma CDF values at the points contained in x_grid

`compute_inv_cdf`(*x_grid*)
Returns np array of inverse gamma CDF values at pts in x_grid

`compute_moments`(*max_order*)
Returns moments up to order 'max_order' in numpy array.

`compute_pdf`(*x_grid*)
Returns numpy array of gamma pdf values at the points contained in x_grid

`draw_random_sample`(*sample_sz*)
Draws random samples from the gamma random variable. Returns numpy array of length 'sample_size' containing these samples

`generate_moments`(*max_moment*)
Calculate & store moments to retrieve more efficiently later

`get_variance`()
Returns variance of gamma random variable

`class` `SROMPy.target.`**`UniformRandomVariable`**(*min_val=0.0*, *max_val=0.0*, *max_moment=10*)
Class for defining a uniform random variable

`compute_cdf`(*x_grid*)
Returns numpy array of uniform CDF values at the points contained in x_grid.

`compute_inv_cdf`(*x_grid*)
Returns np array of inverse uniform CDF values at pts in x_grid

`compute_moments`(*max_order*)
Returns moments up to order 'max_order' in numpy array.

`compute_pdf`(*x_grid*)
Returns numpy array of uniform pdf values at the points contained in x_grid

`draw_random_sample`(*sample_size*)
Draws random samples from the uniform random variable. Returns numpy array of length 'sample_size' containing these samples

`generate_moments`(*max_moment*)
Calculate & store moments to retrieve more efficiently later

`get_variance`()
Returns variance of uniform random variable

`class` `SROMPy.target.`**`AnalyticRandomVector`**(*random_variables*, *correlation_matrix*)
Class for implementing a translation random vector for non-gaussian random vectors whose components are governed by analytic probability distributions and have known correlation.

Parameters

- **random_variables** (*list of SROMPy random variable objects*) – list of SROMPy target random variable objects defining each component of the random vector.
- **correlation_matrix** (*np array, size: dim x dim*) – specifies correlation between vector components.

random_variables list must have length equal to the random vector dimension. Each SROMPy random variable object in the list must be properly initialized and have compute_moments and compute_CDF functions implemented.

compute_cdf (*x_grid*)

Evaluates the precomputed/stored CDFs at the specified x_grid values and returns. x_grid can be a 1D array in which case the CDFs for each dimension are evaluated at the same points, or it can be a (num_grid_pts x dim) array, specifying different points for each dimension - each dimension can have a different range of values but must have the same # of grid pts across it. Returns a (num_grid_pts x dim) array of corresponding CDF values at the grid points

compute_correlation_matrix ()

Returns the correlation matrix

compute_moments (*max_*)

Calculate random vector moments up to order max_moment based on samples. Moments from 1,. . . ,max_order

draw_random_sample (*sample_size*)

Implements the translation model to generate general random vectors with non-gaussian components. Nonlinear transformation of a std gaussian vector according to method in S.R. Arwade 2005 paper.

random component sample: theta = inv_cdf(std_normal_cdf(normal_vec)) Theta = F^{-1}(Phi(G))

generate_gaussian_correlation ()

Generates the Gaussian correlation matrix that will achieve the covariance matrix specified for this random vector when using a translation random vector sampling approach. See J.M. Emery 2015 paper pages 922,923 on this procedure. Helper function - no inputs, operates on self._correlation correlation matrix and generates self._gaussian_corr

generate_unscaled_correlation ()

Generates the unscaled correlation matrix that is matched by the SROM during optimization. No inputs / outputs. INternally produces self._unscaled_correlation from self._correlation.

>> C_ij = E[X_i X_j]

get_correlation_entry (*k*, *j*, *rho_kj*)

Get the correlation between this random vector's k & j components from the correlation btwn the Gaussian random vector's k & j components. Helper function for generate_gaussian_correlation Need to integrate product of k/j component's inv cdf & a standard 2D normal pdf with correlation rho_kj. This is equation 6 in J.M. Emery et al 2015.

integrand_helper (*u*, *v*, *k*, *j*, *rho_kj*)

Helper function for numerical integration in the generate_gaussian_correlation() function. Implements the integrand of equation 6 of J.M. Emery 2015 paper that needs to be integrated w/ scipy Passing in values of the k^th and j^th component of the random variable - u and v - and the specified correlation between them rho_kj.

static verify_correlation_matrix (*corr_matrix*)

Do error checking on the provided correlation matrix, e.g., is it square? is it symmetric?

class SROMPy.target.**SampleRandomVector** (*samples*, *max_moment=10*)

Sample-based random vector. Defines a target random vector to match with an SROM based on a set of realizations of that random vector. Implements basic statistics to use in SROM optimization and comparisons.

Parameters

- **samples** (*np array, size: (# samples x dim)*) – set of realizations/samples of the random vector
- **max_moment** (*int*) – max. order moment to precompute and store

compute_cdf(*x_grid*)
Evaluates the precomputed/stored CDFs at the specified x_grid values and returns.

Parameters **x_grid** (*Numpy array.*) – Grid of points to compute CDF values on. If 1d array is provided, the same points are used to evaluate CDF in each dimension. If 2d array is provided, calculates CDF values on different points, but must have same # points for each dimension. Size is (# grid pts) x (dim) or (# grid pts) x (1).

Returns: Numpy array of CDF values at x_grid points. Size is (# grid pts) x (dim).

compute_correlation_matrix()
Returns precomputed correlation matrix.

compute_moments(*max_order*)
Return precomputed moments up to specified order.

Parameters **max_order** (*int*) – Maximum order of moments to return

Returns (max_order x dim) size Numpy array with SROM moments for each dimension.

draw_random_sample(*sample_size*)
Randomly draws a sample of this random vector.

Parameters **sample_size** (*int*) – number of samples to return

sample_size must be smaller than total # of samples. For sample-based random vector, we return a randomly selected # of samples

generate_cdfs()
Calculate & store marginal CDFs for each dimension of the random vector. Stores a linear interpolator of the CDF for each dim Uses trick from : http://stackoverflow.com/questions/3209362/

%20how-to-plot-empirical-cdf-in-matplotlib-in-python

to calculate CDF from samples

generate_correlation()
Calculates and stores sample-based correlation matrix for random vector

generate_moments(*max_moment*)
Calculate & store random vector moments up to order max_moment based on samples. Moments from 1,. . . ,max_order

generate_statistics(*max_moment*)
Precompute & store moments, CDFs, correlation matrix of the samples so that they can be returned quickly later

get_plot_cdfs()
Get CDF values for plotting (without using interpolant) - returns tuple with x_grid & CDF values arrays

3.3 Postprocessor Documentation

class SROMPy.postprocess.**Postprocessor**(*srom*, *target_random_vector*)

Class for comparing an SROM vs the target random vector it is modeling. Capabilities for plotting CDFs/pdfs and tabulating errors in moments, correlations, etc.

compare_cdfs(*variable='x'*, *plot_dir='.'*, *plot_suffix='CDFcompare'*, *show_figure=True*, *save_figure=True*, *variable_names=None*, *x_limits=None*)

Generates plots comparing the srom & target cdfs for each dimension of the random vector.

inputs: variable, str, name of variable being plotted plot_suffix, str, name for saving plot (will append dim & .pdf) plot_dir, str, name of directory to store plots show_figure, bool, show or not show generated plot save_figure, bool, save or not save generated plot variable_names, list of strings, names of variable in each dimension

optional. Used for x axes labels if provided.

compare_pdfs(*variable='x'*, *plot_dir='.'*, *plot_suffix='pdf_compare'*, *show_figure=True*, *save_figure=True*, *variable_names=None*)

Generates plots comparing the srom & target pdfs for each dimension of the random vector.

inputs: variable, str, name of variable being plotted plot_suffix, str, name for saving plot (will append dim & .pdf) plot_dir, str, name of directory to store plots show_figure, bool, show or not show generated plot save_figure, bool, save or not save generated plot variable_names, list of strings, names of variable in each dimension

optional. Used for x axes labels if provided.

compare_random_variable_cdfs(*random_variable_1*, *random_variable_2*, *variable='x'*, *plot_dir='.'*, *plot_suffix='CDFscompare'*, *show_figure=True*, *save_figure=False*, *variable_names=None*, *x_limits=None*, *labels=None*)

Generates plots comparing CDFs from sroms of different sizes versus the target variable for each dimension of the vector.

inputs: random_variable_1, SampleRandomVector, target random variable object random_variable_2, SampleRandomVector, target random variable object variable, str, name of variable being plotted plot_suffix, str, name for saving plot (will append dim & .pdf) plot_dir, str, name of directory to store plots show_figure, bool, show or not show generated plot save_figure, bool, save or not save generated plot variable_names, list of strings, names of variable in each dimension

optional. Used for x axes labels if provided.

labels, list of str: names of random_variable_1 & random_variable_2

static compare_srom_cdfs(*size2srom*, *target*, *variable='x'*, *plot_dir='.'*, *plot_suffix='CDFscompare'*, *show_figure=True*, *save_figure=True*, *variable_names=None*, *y_limits=None*, *x_ticks=None*, *cdf_y_label=False*, *x_axis_padding=None*, *axis_font_size=30*, *label_font_size=24*, *legend_font_size=25*)

Generates plots comparing CDFs from sroms of different sizes versus the target variable for each dimension of the vector.

inputs: size2srom, dict, key=size of SROM (int), value = srom object target, TargetRV, target random variable object variable, str, name of variable being plotted plot_suffix, str, name for saving plot (will append dim & .pdf) plot_dir, str, name of directory to store plots show_figure, bool, show or not show generated plot save_figure, bool, save or not save generated plot variable_names, list of strings, names of variable in each dimension

optional. Used for x axes labels if provided.

cdf_y_label, bool, use "CDF" as y-axis label? If False, uses F(<variable_name>)

x_axis_padding, int, spacing between xtick labels and x-axis

`compute_moment_error`(*max_moment=4*)
Performs a comparison of the moments between the SROM and target, calculates the percent errors up to moments of order 'max_moment'. Optionally generates text file with the latex source to generate a table.

`generate_cdf_grids`(*cdf_grid_pts=1000*)
Generate numerical grids for plotting CDFs based on the range of the target random vector. Return x_grid variable with cdf_grid_pts along each dimension of the random vector.

`static plot_cdfs`(*x_grid*, *srom_cdf*, *x_target*, *target_cdf*, *x_label='x'*, *y_label='F(x)'*, *plot_name=None*, *show_figure=True*, *x_limits=None*)
Plotting routine for comparing a single srom/target cdf

`static plot_pdfs`(*samples*, *probabilities*, *x_target*, *target_pdf*, *x_label='x'*, *y_label='f(x)'*, *plot_name=None*, *show_figure=True*)
Plotting routine for comparing a single srom/target pdf

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX