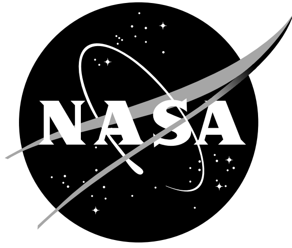
NASA/TM–2018–219824



Stochastic Reduced Order Models with Python (SROMPy)

James E. Warner
Langley Research Center, Hampton, Virginia

April 2018

NASA STI Program... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

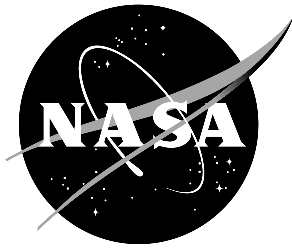
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at `http://www.sti.nasa.gov`
- E-mail your question to `help@sti.nasa.gov`
- Phone the NASA STI Information Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2018–219824



Stochastic Reduced Order Models with Python (SROMPy)

James E. Warner
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2018

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an offical endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

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 can be used to propagate uncertainty through a user-defined computational model to estimate statistics of a given quantity of interest. This report is meant to introduce the SROMPy module and briefly demonstrate its capabilities. A simple example of a spring-mass system with a random input is included to illustrate the practicality of the SROM approach to uncertainty quantification and relative ease of applying it with SROMPy. The example includes a comparison with a solution obtained using classical Monte Carlo simulation, demonstrating the similarities and advantages of using the SROM approach.

1 Introduction

Stochastic Reduced Order Models with Python (SROMPy[1]) is a software package developed to enable user-friendly use of the stochastic reduced order model (SROM) approach for uncertainty quantification [1]. A SROM is a low-dimensional discrete approximation to a random quantity that enables efficient and non-intrusive stochastic computations [2]. 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 [3]. Once a SROM is constructed, the software can be used to propagate uncertainty through a user-defined computational model to estimate statistics of a given quantity of interest. This report introduces the SROMPy package and its capabilities, while providing a brief review of the SROM theory that it implements.

The SROM concept was originally proposed by Grigoriu in 2009 [2] and then further developed by Warner et al. [4] and Grigoriu [5]. The use of SROMs has primarily focused on propagating uncertainty through computational models, including the determination of effective conductivities for random microstructures [6], the quantification of uncertainty in intergranular corrosion rates [7] and laser weld reliability [8], and the estimation of random linear dynamic system states [9]. However, there have been more recent efforts to extend their applicability to inverse [10] and design [11] problems as well. The primary strength of SROMs, as demonstrated in the referenced works, is their ability to represent a target random quantity with low dimensionality and to subsequently solve uncertainty propagation problems in a fraction of the computation time required by traditional Monte Carlo simulation. Furthermore, the approach is practical and straightforward to employ, given that it is a *non-intrusive* method, i.e., it does not require modification of the computational

[1]Publicly available at https://github.com/nasa/SROMPy

model being analyzed; the model can be used simply as a "black box".

SROMPy is the first open-source software library that implements the SROM approach for uncertainty quantification. The goal of this report is to provide an overview of SROMPy's features and capabilities and to demonstrate its usage on a simple example problem[2]. As the first document that discusses SROMPy, the report also represents a citable source for future research that leverages the software. While the functionality of SROMPy is summarized and illustrated within, this report is not meant to serve as the package's user documentation, which can be found with the source code [1] itself. The report is also not intended to provide an all-inclusive description of SROM theory, which can be found instead by consulting the references herein.

A short background on SROM theory is first provided in the following section. Here, the definition of a SROM is given, along with how it is constructed to model a given random quantity and how it is used to propagate uncertainty through a computational model. An overview of the SROMPy software package is then presented, including descriptions of submodules for representing target random quantities, constructing and using SROMs, and postprocessing results. Next, details of using SROMPy to propagate uncertainty through a simple model of a spring-mass system with random spring stiffness are illustrated. Finally, the report is concluded in the summary section.

2 Background - SROM Theory

SROMs can be viewed as a "smart" Monte Carlo method for uncertainty quantification. The approach efficiently discretizes the stochastic space and significantly reduces the computational complexity associated with propagating uncertainty, relative to Monte Carlo simulation, while retaining the benefit of being a non-intrusive method. This section provides a brief overview of the definition and construction of SROMs, followed by their use in propagating uncertainty. The interested reader can consult the relevant references [2, 4, 8] for a more detailed discussion.

2.1 SROM Definition

Let $\mathbf{X} \in \Gamma \subset \mathbb{R}^d$ be a d-dimensional random vector with known probability, i.e., its statistics are known and available:

$$F_i(x_i) = P(X_i \leq x_i) \tag{1}$$

$$\mu_i(q) = E[X_i^q] \tag{2}$$

$$\mathbf{r} = E[\boldsymbol{X}\boldsymbol{X}^T], \tag{3}$$

where F_i and $\mu_i(q)$ are the marginal cumulative distribution function (CDF) and marginal moment of order q for component i of the random vector, respectively, and $\mathbf{r}$ is the (unscaled) correlation matrix. A SROM $\tilde{\mathbf{X}}$ for $\mathbf{X}$ is simply a finite collection

[2]Syntax provided in this report is for Version 1.0 of SROMPy. Slight modifications may be necessary for future versions.

of samples $\{\tilde{\mathbf{x}}^{(1)},...,\tilde{\mathbf{x}}^{(m)}\}$ and corresponding probabilities $(p^{(1)},...,p^{(m)})$ such that $p^{(k)} = P(\mathbf{X} = \tilde{\mathbf{x}}^{(k)})$, $p^{(k)} \geq 0\ \forall k$, and $\sum_{k=1}^{m} p^{(k)} = 1$ [2]. Here, m is referred to as the SROM size. With these SROM parameters specified, the statistics of $\tilde{\mathbf{X}}$ corresponding to those of $\mathbf{X}$ given in Equations (1) - (3) are

$$\tilde{F}_i(x_i) = \sum_{k=1}^{m} p^{(k)} \mathbf{1}\left(\tilde{x}_i^{(k)} \leq x_i\right) \tag{4}$$

$$\tilde{\mu}_i(q) = \sum_{k=1}^{m} p^{(k)} (\tilde{x}_i^{(k)})^q \tag{5}$$

$$\tilde{r}(i,j) = \sum_{k=1}^{m} p^{(k)} \tilde{x}_i^{(k)} \tilde{x}_j^{(k)}, \tag{6}$$

where $\mathbf{1}$(condition) is the indicator function, evaluating to 1 if the condition is true and 0 otherwise.

2.2 SROM Construction

The defining SROM parameters (samples and probabilities) are chosen such that $\tilde{\mathbf{X}}$ is an optimal representation of $\mathbf{X}$ in a statistical sense. This is done through the solution of the following optimization problem:

$$\tilde{\mathbf{X}} \equiv \underset{\{\tilde{\mathbf{x}}\},\mathbf{p}}{\text{argmin}} \left(\sum_{i=1}^{3} \alpha_i e_i(\{\tilde{\mathbf{x}}\}, \mathbf{p}) \right) \tag{7}$$
$$\text{s.t.} \sum_{k=1}^{m} p^{(k)} = 1 \text{ and } p^{(k)} \geq 0,\ k = 1,...,m,$$

where e_1, e_2, and e_3 quantify the error between the SROM and target CDFs, moments, and correlation matrix, respectively, and α_i are weighting factors. A typical error metric used is the sum-of-squares error (SSE) function [4], i.e.,

$$e_1^{\text{SSE}}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{n} \left(\frac{\tilde{F}_i(x_i^j) - F_i(x_i^j)}{C_1^{ij}} \right)^2 \tag{8}$$

$$e_2^{\text{SSE}}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \frac{1}{2} \sum_{i=1}^{d} \sum_{q=1}^{\bar{q}} \left(\frac{\tilde{\mu}_i(q) - \mu_i(q)}{C_2^{iq}} \right)^2 \tag{9}$$

$$e_3^{\text{SSE}}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=i+1}^{d} \left(\frac{\tilde{r}_{ij} - r_{ij}}{C_3^{ij}} \right)^2, \tag{10}$$

where $\{x_i^j\}_{j=1}^n$ is a set of preselected grid points over the range of x_i and $\bar{q}$ is the maximum moment order considered. Here, either relative ($C_1^{ij} = F_i(x_i^j)$, $C_2^{iq} = \mu_i(q)$, and $C_3^{ij} = r_{ij}$) or absolute ($C_1^{ij} = C_2^{iq} = C_3^{ij} = 1$) errors can be used. Other

choices for objective function include maximum errors [8]:

$$e_1^{\max}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \max_{1\le i\le d} \max_{1\le j\le n} \left| \frac{\tilde{F}_i(x_i^j) - F_i(x_i^j)}{C_1^{ij}} \right| \tag{11}$$

$$e_2^{\max}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \max_{1\le i\le d} \max_{1\le q\le \bar{q}} \left| \frac{\tilde{\mu}_i(q) - \mu_i(q)}{C_2^{iq}} \right| \tag{12}$$

$$e_3^{\max}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \max_{1\le i\le d} \max_{i< j\le d} \left| \frac{\tilde{r}_{ij} - r_{ij}}{C_3^{ij}} \right|, \tag{13}$$

and mean errors:

$$e_1^{\text{mean}}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \frac{1}{dn} \sum_{i=1}^{d} \sum_{j=1}^{n} \left| \frac{\tilde{F}_i(x_i^j) - F_i(x_i^j)}{C_1^{ij}} \right| \tag{14}$$

$$e_2^{\text{mean}}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \frac{1}{dq} \sum_{i=1}^{d} \sum_{q=1}^{\bar{q}} \left| \frac{\tilde{\mu}_i(q) - \mu_i(q)}{C_2^{iq}} \right| \tag{15}$$

$$e_3^{\text{mean}}(\{\tilde{\mathbf{x}}\}, \mathbf{p}) = \frac{2}{d(d-1)} \sum_{i=1}^{d} \sum_{j=i+1}^{d} \left| \frac{\tilde{r}_{ij} - r_{ij}}{C_3^{ij}} \right|. \tag{16}$$

An advantage of using the SSE objective function with Equations (8)-(10) is its differentiability , which provides an analytical gradient that can be used with optimization software to solve Equation (7) more efficiently.

An additional strength of the SROM approach is that $\tilde{\mathbf{X}}$ can be formed even if the probability law of $\mathbf{X}$ is unknown and there is only access to a collection of N independent, equally likely samples $\{\hat{\mathbf{x}}^{(k)}\}_{k=1}^{N}$ of the vector. In this case, the empirical estimators for the statistics of $\mathbf{X}$ can be used in the optimization problem in Equation (7):

$$\hat{F}_i(x_i) = \frac{1}{N} \sum_{k=1}^{N} \mathbf{1}\left(\hat{x}_i^{(k)} \le x_i\right) \tag{17}$$

$$\hat{\mu}_i(q) = \frac{1}{N} \sum_{k=1}^{N} (\hat{x}_i^{(k)})^q \tag{18}$$

$$\hat{r}(i,j) = \frac{1}{N} \sum_{k=1}^{N} \hat{x}_i^{(k)} \hat{x}_j^{(k)} \tag{19}$$

2.3 SROMs for Propagating Uncertainty

SROMs are typically used as a smart Monte Carlo method for uncertainty propagation. Here, the components of the random vector, $\mathbf{X}$, represent some uncertain parameters to a deterministic model, $\mathcal{M}$. Using the SROM approach, one can efficiently estimate the statistics of a quantity of interest, $\mathbf{Y} \in \Gamma' \subset \mathbb{R}^{d'}$, that is dependent on $\mathbf{X}$ through the model:

$$\mathbf{Y} = \mathcal{M}(\mathbf{X}) \tag{20}$$

After a SROM, $\tilde{\mathbf{X}}$, is generated according to the developments in Section 2.2, $\mathbf{Y}$ can be estimated by first evaluating the model for each SROM sample

$$\tilde{\mathbf{y}}^{(k)} = \mathcal{M}(\tilde{\mathbf{x}}^{(k)}), \text{ for } k = 1, ..., m. \tag{21}$$

The collection of output samples, $\{\tilde{\mathbf{y}}^{(k)}\}_{k=1}^{m}$, and original probabilities $\mathbf{p}$ from $\tilde{\mathbf{X}}$ now define a new SROM, $\tilde{\mathbf{Y}}$, for the model output. Using $\tilde{\mathbf{Y}}$, the statistics of $\mathbf{Y}$ can be directly estimated using expressions analogous to Equations (4)-(6). For instance, the distributions and moments of the output can be approximated using the solutions from Equation (21) as

$$P(Y_i \le y_i) \approx P(\tilde{Y}_i \le y_i) = \sum_{k=1}^{m} p^{(k)} \mathbf{1}\left(\tilde{y}_i^{(k)} \le y_i\right) \tag{22}$$

$$E[Y_i^q] \approx E[\tilde{Y}_i^q] = \sum_{k=1}^{m} p^{(k)} (\tilde{y}_i^{(k)})^q \tag{23}$$

2.3.1 SROM Surrogate Models

In addition to directly estimating statistics of the output as described in the previous sections, SROMs can also be used to generate closed-form surrogate models for the output that can be efficiently sampled [5]. For example, the SROM-based distribution and moment estimates in Equations (22) and (23) can be seen as a result of constructing the following piecewise constant approximation, denoted by the subscript C, of the output:

$$\tilde{\mathbf{Y}}_C(\mathbf{X}) = \sum_{k=1}^{m} \mathbf{1}\left(\mathbf{X} \in \Gamma_k\right) \tilde{\mathbf{y}}^{(k)} \tag{24}$$

where $\{\Gamma_k,\ k = 1, ..., m\}$ is a partition of Γ such that $P(\mathbf{X} \in \Gamma_k) = p^{(k)}$. Specifically, $\{\Gamma_k\}$ is a Voronoi tessellation of Γ with centers at samples $\tilde{\mathbf{x}}^{(k)}$ of the SROM $\tilde{\mathbf{X}}$ [5,8]. In practice, however, the partition does not have to be constructed explicitly. Instead, a given sample of $\mathbf{X}$ is simply allocated to a particular cell depending on its distance to the SROM samples.

The expression in Equation (24) effectively provides a piecewise constant approximation to the model output $\mathbf{Y}$ that can be sampled. Building on this relatively crude approach, Grigoriu also proposed a SROM-based surrogate model that constructs a piecewise linear response surface, denoted by the subscript L, to map samples of $\mathbf{X}$ to the output via the truncated Taylor expansion [5]:

$$\tilde{\mathbf{Y}}_L(\mathbf{X}) = \sum_{k=1}^{m} \mathbf{1}\left(\mathbf{X} \in \Gamma_k\right) \left[\tilde{\mathbf{y}}^{(k)} + \nabla\tilde{\mathbf{y}}^{(k)} \cdot (\mathbf{X} - \mathbf{x}^{(k)})\right], \tag{25}$$

where the $\nabla\tilde{\mathbf{y}}^{(k)}$ denotes the gradient of the output with respect to the components of $\mathbf{X}$ evaluated at sample k, and is computed numerically with finite difference. Equation (25) represents a more accurate surrogate model for the output $\mathbf{Y}$ but

with the added expense for computing gradients, requiring $m(d+1)$ model evaluations versus m required (via Equation (21)) for the piecewise constant surrogate in Equation (24). Note that Equation (25) can be generalized to higher order approximations by including addition terms of the Taylor expansion.

3 SROMPy Functionality

This section provides an overview of how SROMPy allows users to apply SROM theory to solve uncertainty quantification problems. The first step in the solution process is defining the target random quantities that will be modeled by SROMs; this is described in the first subsection. Then, the SROMPy functionality for generating SROMs to represent these random quantities is described, followed by their use in propagating uncertainty through a computational model. Finally, the SROMPy features for postprocessing results are briefly discussed.

3.1 Target Random Quantities

One of the strengths of the SROM approach to uncertainty propagation is that it can be used whether the user has an explicit analytical representation of the random quantity, $\mathbf{X}$, being modeled or only a collection of independent samples that describes it. The following subsections describe modeling random quantities from both cases in SROMPy.

3.1.1 Standard Random Variables

Random variables can be modeled directly for one-dimensional problems or aggregated to form the components of a random vector, as described in the following section. Currently, SROMPy supports random variables described by beta, gamma, and Gaussian distributions with the Python classes `BetaRandomVariable`, `GammaRandomVariable`, and `GaussianRandomVariable`, respectively.

While only a small subset of all possible probability distributions are currently available in SROMPy, it is straightforward to add classes to the package to model new random variables with SROMs. In order to be compatible with SROMPy, a class must be added that implements the following methods (in addition to an appropriate constructor method for initializing the object):

- `get_variance()`
  - Returns the variance of the random variable.
- `compute_moments(max_order)`
  - Returns array of non-central moments up to order `max_order`.
- `compute_CDF(x_grid)`
  - Returns array of CDF values at the points in `x_grid`.
- `compute_inv_CDF(x_grid)`

    - Returns array of inverse CDF values at the points in `x_grid`.

- `compute_pdf(x_grid)`
    - Returns array of probability density function values at the points in `x_grid`.

- `draw_random_sample(num_samples)`
    - Returns array of random samples with length `num_samples`.

For the random variables that are currently supported in SROMPy, the above methods are simply wrappers around the appropriate function calls supplied by the `SciPy` [12] Python module. Users should consult the `SciPy` documentation to see if the probability distribution they are adding to SROMPy exists there to adopt a similar approach.

3.1.2 Analytic Random Vector - `AnalyticRV`

The `AnalyticRV` class implements a translation random vector [13, 14] whose components follow standard analytic probability distributions. To model an analytic random vector in SROMPy, random variable objects (Section 3.1.1) representing each component of the random vector must first be properly initialized and created. A user can then create a `AnalyticRV` object by supplying these random variables as well as the correlations between them, using the following constructor:

- `AnalyticRV(random_variables, correlation_matrix)`,

where `random_variables` is a Python list of SROMPy random variable objects and `correlation_matrix` is a two-dimensional `NumPy` [15] array representing the scaled correlation between the components of the random vector. Note that this correlation matrix must be square and symmetric with size $d \times d$ and have entries between -1 and 1.

Once initialized, the primary functionality of `AnalyticRV` is to compute and return statistics of the random vector. These methods are listed below, along with the corresponding equations from Section 2.1:

- `compute_CDF(x_grid)` - implements Equation (1)

- `compute_moments(max_order)` - implements Equation (2)

- `compute_correlation()` - implements Equation (3).

There is also a `draw_sample(num_samples)` method to generate random vector samples that follow the specified distributions and correlations.

3.1.3 Sample-Based Random Vector - `SampleRV`

If an explicit probability law is not available to describe a random quantity and the user only has access to a collection of N independent, equally likely samples $\{\hat{\mathbf{x}}^{(k)}\}_{k=1}^{N}$, the `SampleRV` class in SROMPy can be used to model it. This class can be initialized from a `NumPy` array, `samples`, containing those independent samples:

- `SampleRV(samples)`,

where the `samples` array has size $N \times d$.

Similarly to the `AnalyticRV` class described previously, the primary functionality of `SampleRV` is to compute and return statistics of the random vector. The only difference is that sample-based estimators are used with `SampleRV`. These methods, along with their corresponding equations from Section 2.1, are listed below:

- `compute_CDF(x_grid)` - implements Equation (17)
- `compute_moments(max_order)` - implements Equation (18)
- `compute_correlation()` - implements Equation (19).

The `draw_sample(num_samples)` method randomly selects entries from the provided `samples` array that was used to initialize the random vector.

3.2 SROM Functionality

The goal of SROMPy is to allow users to easily model random quantities, $\mathbf{X}$, using SROMs, $\tilde{\mathbf{X}}$, and use them to efficiently propagate uncertainty through computational models, as described in Sections 2.2 and 2.3. The SROMPy classes `SROM` and `SROMSurrogate` that enable this functionality are now discussed.

3.2.1 `SROM`

The fundamental component of the SROMPy package is the `SROM` class, used to model a user-specified target random quantity. It is initialized by providing the SROM size, m, and the random vector dimension (1 for a scalar random variable), d:

- `SROM(size, dim)`

The primary role of the `SROM` class is to select the SROM parameters (samples and probabilities) such that the SROM is an optimal approximation of a target random quantity, using the `optimize` method:

- `optimize(targetRV)` - implements Equation (7),

where `targetRV` is an initialized standard random variable, analytic random vector, or sample-based random vector object, as introduced in Section 3.1. The `optimize` method also accepts many default arguments for fine tuning the optimization, including an `error` input for specifying which error metric to use in the objective function in Equation (7). The options are `SSE` (default) for the sum-of-squares error

functions in Equations (8)-(10), `MAX` for the maximum errors in Equations (11)-(13), and `MEAN` for the mean errors in Equations (14)-(16).

The statistics of the SROM can be computed using methods analogous to those of the target random quantities, listed below:

- `compute_CDF(x_grid)` - implements Equation (4)
- `compute_moments(max_order)` - implements Equation (5)
- `compute_correlation()` - implements Equation (6).

There are several additional `SROM` utility methods, including those for manually specifying and retrieving the SROM parameters (samples and probabilities) and saving and loading SROM parameters to and from file. See the user documentation in the source code [1] for more details.

After SROM parameters have been selected to approximate a random quantity using the `optimize` method, a user would need to write code to implement Equation (21) in order to facilitate uncertainty propagation (as described in Section 2.3). This would involve using the `SROM` method `get_params` to retrieve the optimal SROM samples, evaluating the computational model for each sample, and storing the corresponding outputs. Then, to directly estimate the statistics of the output, a new SROM is initialized using these output samples, and Equations (22) and (23) are employed using the statistics methods listed above. This process will be demonstrated in a numerical example in Section 4.

3.2.2 `SROMSurrogate`

The `SROMSurrogate` class in SROMPy allows a user to construct a closed-form surrogate model that can be sampled for propagating uncertainty as described in Section 2.3.1. It provides an implementation of the piecewise constant, $\tilde{\mathbf{Y}}_C(\mathbf{X})$, and piecewise linear, $\tilde{\mathbf{Y}}_L(\mathbf{X})$, approximations to a model output given by Equations (24) and (25), respectively.

The class is initialized as follows:

- `SROMSurrogate(inputsrom, outputsamples, gradients=None)`

where `inputsrom` is the `SROM` object that was used to model the random model inputs, `outputsamples` is an array of the model outputs corresponding to each input SROM sample, and `gradients` is an array containing the gradient of the output with respect to each input sample. In terms of the notation from the SROM theory section, `inputsrom` represents $\tilde{\mathbf{X}}$, `outputsamples` represents $\{\tilde{\mathbf{y}}^{(k)}\}_{k=1}^m$ from Equation (21), and `gradients` represents $\{\nabla\tilde{\mathbf{y}}^{(k)}\}_{k=1}^m$ from Equation (25).

When initializing the `SROMSurrogate` class, the piecewise linear surrogate, $\tilde{\mathbf{Y}}_L(\mathbf{X})$, will be automatically used if the `gradients` argument is provided, whereas the piecewise constant surrogate, $\tilde{\mathbf{Y}}_C(\mathbf{X})$, will be used if not. Note that SROMPy also provides a `FiniteDifference` class that contains static methods to assist in computing the gradient, $\nabla\tilde{\mathbf{y}}$, using the finite difference method. More details can be found in the documentation in the source code [1].

Once a `SROMSurrogate` object is properly initialized, the primary class method is:

- `sample(newinputsamples)`,

which evaluates Equation (24) or (25) for the provided `newinputsamples` array and returns an array of the corresponding outputs. The output samples can subsequently be used to construct sample-based estimators of the true output statistics. These concepts will be illustrated in more detail in Section 4.

3.3 Postprocessing Results - `Postprocessor`

SROMPy provides a few simple utilities for comparing statistics of a SROM versus a target quantity it is approximating. The `Postprocessor` class that carries out these comparisons is initialized by providing a SROM object (`srom`) and the corresponding target random variable or vector object (`targetrv`) being modeled:

- `Postprocessor(srom, targetrv)`.

After initialization, the methods

- `compare_CDFs()`
- `compare_moments()`

can be used to generate plots comparing the SROM and target CDFs and a text output comparing the SROM and target moments along with associated errors, respectively. There are several optional formatting arguments for the `compare_CDFs()`; more details can be found in the user documentation. It is also straightforward for a user to extend this simple functionality or to write their own comparison codes based on the SROM and target statistics methods provided by SROMPy.

4 Example

SROMPy is now applied to an example of uncertainty propagation in a simple spring-mass system (Figure 1), demonstrating the functionality introduced in the previous section. The governing equation of motion for the system is given by

$$m_s \ddot{z} = -k_s z + m_s g, \tag{26}$$

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:

$$K_s = \gamma + \eta B \tag{27}$$

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

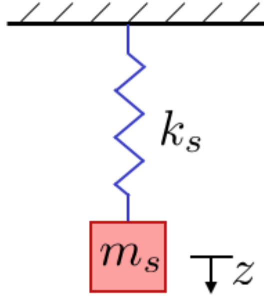


Figure 1. Spring-mass system.

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 $g = 9.8m^2/s$.

Since uncertainty has been introduced to the system, the resulting displacement, Z, is now random as well. The output of interest will be the maximum displacement over a time interval of 10 seconds. In terms of the general stochastic model in Equation (20), the input is $\mathbf{X} = [K_s]$, the output is $\mathbf{Y} = [\max(Z)] \equiv Z_{\max}$, and the computational model $\mathcal{M}$ numerically integrates Equation (26) starting from rest and then finds the maximum displacement over a time period of 10 seconds. SROMPy will be used to approximate the CDF, $F(z_{\max})$, of the maximum displacement using SROMs and compare it to the solution using Monte Carlo simulation. The highlights of the Python code used to carry out the analysis with be shown throughout this section, while it can be seen in its entirety in the Appendix. Note that the source code shown uses SROMPy Version 1.0, and syntax changes may be necessary for future versions of the package.

4.1 Step 1: Define target random variable and initialize model

The first step in the analysis is to define the target random variable in SROMPy that models the random spring stiffness, K_s, in Equation (27). Here, the `BetaRandomVariable` class (Section 3.1.1) is used to represent K_s:

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

Next, the computational model, $\mathcal{M}$, of the spring-mass system that carries out the numerical integration of Equation (26) is initialized:

```
#Specify spring-mass system and initialize model:
m = 1.5                             #deterministic mass
state0 = [0., 0.]                   #initial conditions at rest
t_grid = np.arange(0., 10., 0.1)    #time discretization
model = SpringMass_1D(m, state0, t_grid)
```

More details on the source code and implementation of the `SpringMass_1D` model can be found in the Appendix.

4.2 Step 2: Construct SROM for the input

To facilitate efficient uncertainty propagation, a SROM, $\tilde{K}_s$, must first be formed to model the random stiffness input, K_s, with SROMPy. This is done by initializing the `SROM` class (Section 3.2.1), and then calling the `optimize` function (Equation (7)) to determine the optimal parameters to match the previously defined `BetaRandomVariable` object. The code to implement this with a SROM size, $m = 10$, is shown below:

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

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

Here, the `input_srom` object is constructed to match the previously initialized target random variable, `stiffness_rv`. In the second block of code, the SROMPy `Postprocessor` (Section 3.3) class is used to compare the resulting SROM CDF with that of the target beta random variable, which can be seen in Figure 2(a).

4.3 Step 3: Propagate uncertainty using SROM

4.3.1 Approach (a). Estimate output statistics directly

The SROM generated to represent the random stiffness in SROMPy can now be used to propagate uncertainty through the spring-mass system model. In the first approach here, the distribution of $Z_{\max}$ is estimated directly by producing the SROM $\tilde{Z}_{\max}$ through the implementation of Equations (21) and (22) in Section 2.3. Equation (21) is carried out by executing the model to get the maximum displacement corresponding to each SROM sample of the spring stiffness. An output SROM is then formed using the resulting samples of $Z_{\max}$. The code to implement this is shown below:

```
#Run model to get max disp for each SROM stiffness sample
srom_disps = np.zeros(sromsize)
(samples, probs) = input_srom.get_params()
for i, stiff in enumerate(samples):
    srom_disps[i] = model.get_max_disp(stiff)

#Form new SROM for the max disp. solution using samples from the model
output_srom = SROM(sromsize, dim)
output_srom.set_params(srom_disps, probs)

#Compare solutions
pp_output = Postprocessor(output_srom, mc_solution)
pp_output.compare_CDFs()
```

Here, a new `SROM` object, `output_srom` is formed from the array of maximum displacement samples, `srom_disps`. The SROM approximation to the CDF of $Z_{\max}$

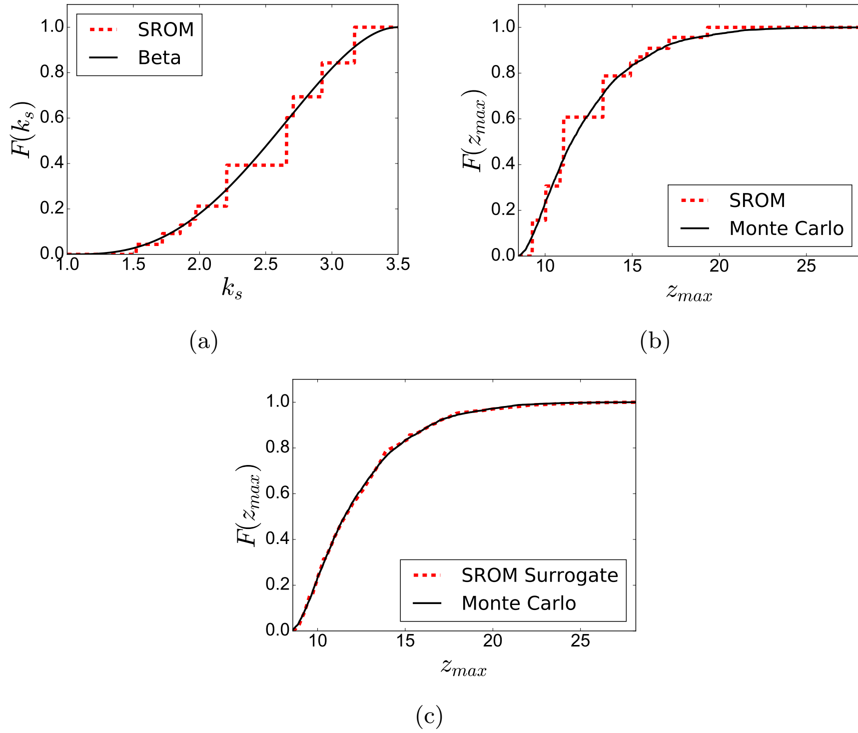


Figure 2. (a) SROM CDF versus the Beta random variable for spring stiffness. (b) SROM CDF versus the Monte Carlo solution for maximum displacement. (c) SROM surrogate model CDF versus the Monte Carlo solution for maximum displacement.

is then compared to that of a Monte Carlo simulation solution, `mc_solution`, with 5000 samples. The comparison of CDFs is shown in Figure 2(b), where reasonable agreement can be seen between SROM and Monte Carlo, despite the relatively crude piecewise constant SROM approximation. Note that the benefit of the SROM solution is that it requires only 10 model evaluations compared to the 5000 used for the Monte Carlo simulation solution. The code to generate `mc_solution` can be seen in the complete Python script provided in the Appendix.

4.3.2 Approach (b). Form SROM surrogate model for output

This section demonstrates an alternative approach for estimating the distribution of maximum displacement in the spring-mass system. Here, a SROM surrogate model is generated for $Z_{\max}$ using the input SROM $\tilde{K}_s$, as described in Section 2.3.1, that is then sampled in order to estimate the CDF. Specifically, the piecewise linear approximation in Equation (25) is used to approximate $Z_{\max}$ using the SROMPy `SROMSurrogate` class (Section 3.2.2). Equation (25) provides more accurate predictions relative to the piecewise constant estimates in the previous section, but requires gradient information and additional model evaluations.

In order to construct the piecewise linear SROM surrogate model in SROMPy, the gradient of $Z_{\max}$ with respect to each SROM stiffness sample, $\tilde{k}_s^{(k)}$, must first

be computed using finite difference:

$$\nabla \tilde{Z}_{\max}^{(k)} = \frac{\mathcal{M}(\tilde{k}_s^{(k)} + \delta) - \mathcal{M}(\tilde{k}_s^{(k)})}{\delta} \tag{28}$$

for $k = 1, ...10$. Here, δ is a small perturbation. The SROMPy code to implement Equation (28) is shown below:

```
#Get perturbed input srom samples to run through model for FD
stepsize = 1e-12
samples_fd = FD.get_perturbed_samples(samples,perturb_vals=[stepsize])

#Run model to get perturbed outputs for FD calc.
perturbed_disps = np.zeros(sromsize)
for i, stiff in enumerate(samples_fd):
    perturbed_disps[i] = model.get_max_disp(stiff)
gradient = FD.compute_gradient(srom_disps, perturbed_disps,[stepsize])
```

where `stepsize` represents the perturbation δ in Equation (28) and `FD` is the SROMPy `FiniteDifference` class that provides utilities to assist in calculating gradients with the finite difference method.

Now, the `SROMSurrogate` class can be initialized to implement Equation (25) using this gradient information. The surrogate model can then be used to generate samples of $Z_{\max}$ for new values of K_s *without needing to evaluate the spring-mass computational model*. The code below carries out this initialization and sampling, and then estimates the CDF of $Z_{\max}$ from its samples:

```
#Form SROM surrogate and draw samples from it:
surrogate_PWL = SROMSurrogate(input_srom, srom_disps,gradient)
stiffness_samples = stiffness_rv.draw_random_sample(5000)
output_samples = surrogate_PWL.sample(stiffness_samples)
solution_PWL = SampleRV(output_samples)

#Compare SROM piecewise linear solution to Monte Carlo
pp_pwl = Postprocessor(solution_PWL, mc_solution)
pp_pwl.compare_CDFs()
```

Here, the `SampleRV` class (Section 3.1.3) was used to generate a sample-based random variable object from the surrogate model samples that could then be used to estimate the distribution of $Z_{\max}$. Again, the `Postprocessor` class is used to compare the SROM CDF solution versus the Monte Carlo simulation solution, which can be seen in Figure 2(c). The SROM CDF using the piecewise linear approximation is significantly more accurate than the piecewise constant approximation in Figure 2(b), showing good agreement with the Monte Carlo solution. Note that the advantage of using the SROM surrogate in Equation (25) here is that it is an analytical expression and does not require additional model evaluations to produce output samples. For applications with computationally expensive models, this can provide substantial performance gains.

5 Summary

This report provides an introduction to the SROMPy Python package and its functionality while supplying a brief background on the corresponding SROM theory that it implements. SROMs are a practical and general tool for efficient uncertainty propagation and SROMPy is the first publicly available software library that enables the methodology. To help potential users understand the advantages of using SROMPy, a simple example of uncertainty propagation in a random spring-mass system was illustrated. Here, the distribution of an output of interest (maximum displacement) was calculated using SROMs when the spring stiffness was assumed to be a random variable. It was shown that this analysis could be carried out with relatively few lines of code by relying on SROMPy functionality. The report also explained how the package could be straightforwardly extended to handle random variables from new probability distributions that are not currently supported. The interested reader is referred to the SROMPy documentation that accompanies the source code [1] for more practical details of using the software, and to the citations herein for more background on SROM theory.

References

1. Warner, J. E.: Stochastic Reduced Order Models with Python (SROMPy), Version 1.0. `https://github.com/nasa/SROMPy`. 2018.

2. Grigoriu, M.: Reduced Order Models for Random Functions. Application to Stochastic Problems. *Applied Mathematical Modelling*, vol. 33, 2009, pp. 161–175.

3. Python Software Foundation: Python Language Reference, version 2.7. `www.python.org`. 2016.

4. Warner, J. E.; Grigoriu, M.; and Aquino, W.: Stochastic reduced order models for random vectors. Application to random eigenvalue problems. *Probabilistic Engineering Mechanics*, vol. 31, 2013, pp. 1–11.

5. Grigoriu, M.: A method for solving stochastic equations by reduced order models and local approximations. *Journal of Computational Physics*, vol. 231, 2011, pp. 6495–6513.

6. Grigoriu, M.: Effective Conductivity by Stochastic Reduced Order Models (SROMs). *Computational Materials Science*, vol. 50, 2010, pp. 138–146.

7. Sarkar, S.; Warner, J. E.; Aquino, W.; and Grigoriu, M.: Stochastic reduced order models for uncertainty quantification of intergranular corrosion rates. *Corrosion Science*, vol. 80, 2014, pp. 257–268.

8. Emergy, J. M.; Field, R. V.; Foulk, J. W.; Karlson, K. N.; and Grigoriu, M. D.: Predicting laser weld reliability with stochastic reduced-order models. *International Journal for Numerical Methods in Engineering*, vol. 103, 2015, pp. 914–936.

9. Grigoriu, M.: Linear Random Vibration by Stochastic Reduced-Order Models. *International Journal for Numerical Methods in Engineering*, vol. 82, 2010, pp. 1537–1559.

10. Warner, J. E.; Aquino, W.; and Grigoriu, M.: Stochastic reduced order models for inverse problems under uncertainty. *Computer Methods in Applied Mechanics and Engineering*, vol. 285, 2015, pp. 488–514.

11. Aguilo, M. A.; and Warner, J. E.: Multi-material structural topology optimization under uncertainty via a stochastic reduced order model approach. *Proceedings of the 28th Annual International Solid Freeform Fabrication Symposium - An Additive Manufacturing Conference*, Austin, TX, Aug 2017.

12. Jones, E.; Oliphant, T.; Peterson, P.; et al.: SciPy: Open source scientific tools for Python. 2001.

13. Grigoriu, M.: *Applied Non-Gaussian Processes: Examples, Theory, Simulation, Linear Random Vibration, and Matlab Solutions*. Prentice Hall, Englewoods Cliffs, NJ, 1995.

14. Arwade, S. J.: Translation vectors with non-identically distributed components. *Probabilistic Engineering Mechanics*, vol. 20, 2005, pp. 158–167.

15. Van der Walt, S.; Colbert, S. C.; and Varoquaux, G.: The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, vol. 13, no. 2, 2011, pp. 22–30. URL http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37.

6 Appendix

6.1 Full Python script for SROMPy example

```
import numpy as np

from model import SpringMass_1D
from postprocess import Postprocessor
from srom import SROM, FiniteDifference as FD, SROMSurrogate
from target import SampleRV, BetaRandomVariable

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

#Specify spring-mass system:
m = 1.5                              #deterministic mass
state0 = [0., 0.]                    #initial conditions
t_grid = np.arange(0., 10., 0.1)     #time

#Initialize model
model = SpringMass_1D(m, state0, t_grid)

#----------Monte Carlo------------------
```

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

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

#Get Monte carlo solution as a sample-based random variable:
mc_solution = SampleRV(disp_samples)

#-------------SROM-----------------------

#generate SROM for random stiffness
sromsize = 10
dim = 1
input_srom = SROM(sromsize, dim)
input_srom.optimize(stiffness_rv)

#Compare SROM vs target stiffness distribution (See Fig 2a):
pp_input = Postprocessor(input_srom, stiffness_rv)
pp_input.compare_CDFs()

#Run model to get max disp for each SROM stiffness sample
srom_disps = np.zeros(sromsize)
(samples, probs) = input_srom.get_params()
for i, stiff in enumerate(samples):
    srom_disps[i] = model.get_max_disp(stiff)

#Form new SROM for the max disp. solution using samples from the model
output_srom = SROM(sromsize, dim)
output_srom.set_params(srom_disps, probs)

#Compare solutions (See Fig 2b)
pp_output = Postprocessor(output_srom, mc_solution)
pp_output.compare_CDFs()

#--------Piecewise LINEAR surrogate with gradient info-------

#Need to calculate gradient of output wrt input samples first

#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_disps = np.zeros(sromsize)
for i, stiff in enumerate(samples_fd):
    perturbed_disps[i] = model.get_max_disp(stiff)
gradient = FD.compute_gradient(srom_disps, perturbed_disps,[stepsize])

#Form SROM surrogate and draw samples from it:
surrogate_PWL = SROMSurrogate(input_srom, srom_disps, gradient)
stiffness_samples = stiffness_rv.draw_random_sample(num_samples)
```

```
output_samples = surrogate_PWL.sample(stiffness_samples)
solution_PWL = SampleRV(output_samples)

#Compare SROM piecewise linear solution to Monte Carlo (See Fig 2c)
pp_pwl = Postprocessor(solution_PWL, mc_solution)
pp_pwl.compare_CDFs()
```

6.2 Spring-Mass System Python Model

```
#Saved as "model.py"
import numpy as np
from scipy.integrate import odeint
#------------------------------------------------------
#Helper function to use scipy integrator in model class
def mass_spring(state, t, k, m):
    '''
    Return velocity/acceleration given velocity/position and values
    for stiffness and mass
    '''

    # unpack the state vector
    x = state[0]
    xd = state[1]

    g = 9.8 # metres per second

    # compute acceleration xdd
    xdd = ((-k*x)/m) + g

    # return the two state derivatives
    return [xd, xdd]

#------------------------------------------------------

class SpringMass_1D(object):
    '''
    Defines Spring Mass model with 1 free param (spring stiffness, k)
    '''
    def __init__(self, m=1.5, state0=None, time_grid=None):

        self._m = m

        #Give default initial conditions & time grid if not specified
        if state0 is None:
            state0 = [0.0, 0.0]
        if time_grid is None:
            time_grid = np.arange(0.0, 10.0, 0.1)

        self._state0 = state0
        self._t = time_grid

    def simulate(self, k=2.5):
        '''
        Simulate spring mass system for given spring constant. Returns
        state(position, velocity) at all points in time grid
```

```
        '''

        return odeint(mass_spring, self._state0,
                      self._t, args=(k, self._m))

    def get_max_disp(self, k=2.5):
        '''
        Returns the max displacement over the course of the simulation
        '''

        state = self.simulate(k)
        return max(state[:,0])
```

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
01-04-2018

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Stochastic Reduced Order Models with Python (SROMPy)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
James E. Warner

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
533127.02.16.07.06

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, Virginia 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L–12345

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2018–219824

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
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 can be used to propagate uncertainty through a user-defined computational model to estimate statistics of a given quantity of interest. This report is meant to introduce the SROMPy module and briefly demonstrate its capabilities. A simple example of a spring-mass system with a random input is included to illustrate the practicality of the SROM approach to uncertainty quantification and relative ease of applying it with SROMPy. The example includes a comparison with a solution obtained using classical Monte Carlo simulation, demonstrating the similarities and advantages of using the SROM approach.

15. SUBJECT TERMS
Uncertainty quantification, stochastic methods

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Information Desk (help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	24	(757) 864-9658