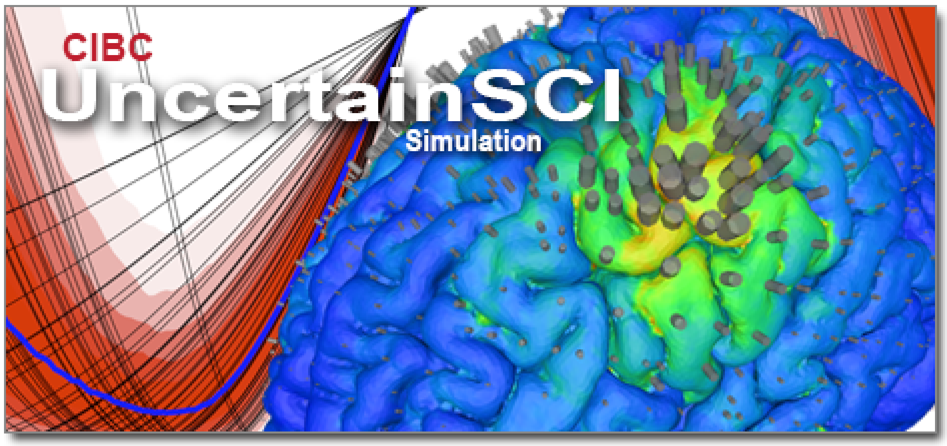
UncertainSCI

Release 1.0.1

Jake Bergquist, Dana Brooks, Zexin Liu, Rob MacLeod, Akil Naray

Oct 12, 2023

CONTENTS:



UncertainSCI source code

CHAPTER ONE

ABOUT UNCERTAINSCI

UncertainSCI [1] is a Python-based toolkit that harnesses modern techniques to estimate model and parametric uncertainty, with a particular emphasis on needs for biomedical simulations and applications. This toolkit enables non-intrusive integration of these techniques with well-established biomedical simulation software.

Currently implemented in UncertainSCI is Polynomial Chaos Expansion (PCE) with a number of input distributions. For more information about these techniques, see: [2, 3, 4, 5, 6]. For studies using UncertainSCI, see: [7, 8, 9, 10, 11, 12]

1.1 Getting Started with UncertainSCI

1.1.1 System Requirements

Requires Python 3 and modules listed in `requirements.txt`

1.1.2 Getting UncertainSCI

The easyiest way to get UncertainSCI is to use pip. Just run `pip install UncertainSCI` or `python -m pip install UncertainSCI` and pip will download and install the latest version of UncertainSCI. This will also try to download and install the relevent dependencies too.

To get pip, see its documentation.

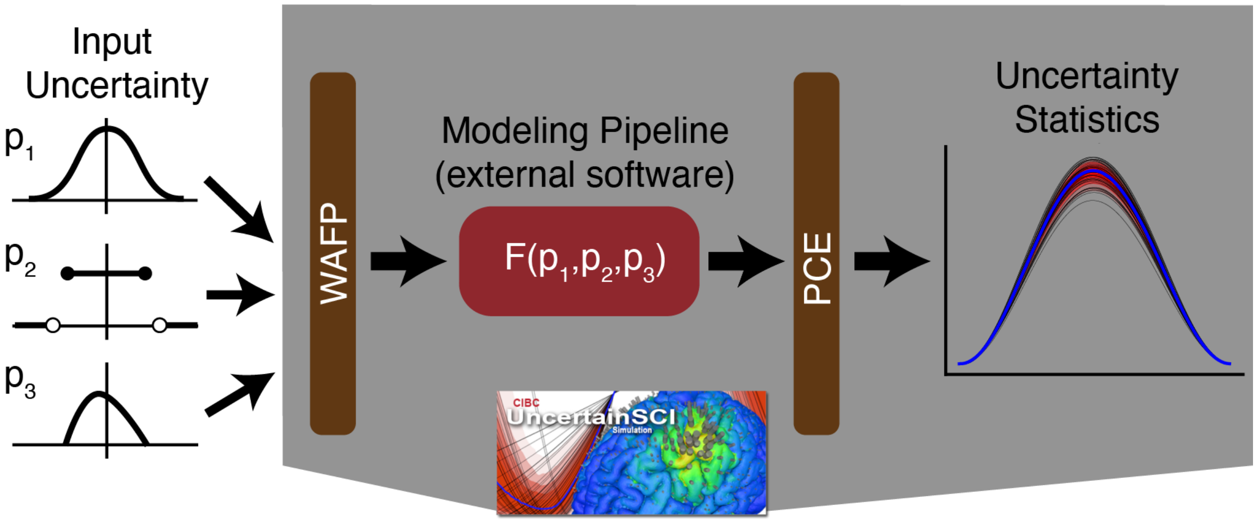
The source code for UncertainSCI can be downloaded from the Github page.

Installing UncertainSCI From Source

UncertainSCI can be built from source code using the `setup.py` script. To call this script, navagate to the source root directory in a terminal window and run the command `pip install .` or `python -m pip install .`

1.1.3 UncertainSCI Overview

Users can evaluate the effect of input uncertainty on forward models with UncertainSCI's implementation of polynomial Chaos expansion (PCE). The pipeline for the process is shown in the following image:



Before Using UncertainSCI

In order to run UncertainSCI, it must be supplied with the input parameter distributions, a model that will run the parameters in question, and a way to collect model solutions for use in PCE. The model itself can be implemented in Python or in another software, as appropriate, as parameter sets and solutions can pass between UncertainSCI and modeling software via hard disk. Models with high resolution solutions should could be cost prohibitive to run with PCE, so users should consider derived metrics or solutions subsets (regions of interest) for UQ.

Users must chose the input distributions for their modeling pipeline, which may significantly impact the output distributions. The indendent application will determine the best distributions to use, yet we suggest that the users use all information available to inform their choices. Published literature and previously generated data are, of course, more credible, but users may need to rely on their own intuition and observed trends with the targeted model.

PCE computation time

Since UncertainSCI uses PCE to compute UQ, it is worth noting the impact of some PCE parameters on computational time. Mostly, the time needed to compute UQ via PCE is limited by the evalutated model, especially in bioelectric field and other 3D simulation applications. PCE attempts to reduce computational cost by limiting the number of parameter samples are needed to estimated the model output distributions, as the fewer parameter sets needed, the few times the model needs to be run. Since PCE is estimating a polynomial to represent the output distribution, some user choices will affect the sample size

Number of parameters modeled in the PCE will affect the number of samples needed to estimate the model uncertainty by increasing the dimensionality of the estimated polynomial. Higher demensions require more sampling points to accurately capture, so will lead to higher computation times. However, parameters must be evaluated in the PCE together to determine the effect of their interaction in the targeted model.

polynomial (PCE) order affects the PCE samples needed for UQ by defining the complexity captured by PCE. As in 1D, polynomials with higher order are able to capture higher variability within the domain. Therefore, models with high complexity, i.e., significant response to variation in the parameter space, should use higher polynomial orders. However, more parameter samples are required to estimate higher polynomials, increasing the number of times the model must be run.

Distribution type of the parameters may effect the number of samples, but to a minor level when compared to number of parameters and polynomial order.

Running UncertainSCI

Running UncertainSCI for Sample Generation

With the model setup, the user will need to setup the input parameter distribution using UncertainSCI's distribution datatype. While there are a few distribution types to choose from, the Beta distribution is a common choice. If we had three input parameters, we can define a different beta distribution for each, thus:

```
from UncertainSCI.distributions import BetaDistribution

Nparams = 3
# Three independent parameters with different Beta distributions
p1 = BetaDistribution(alpha=0.5, beta=1.)
p2 = BetaDistribution(alpha=1., beta=0.5)
p3 = BetaDistribution(alpha=1., beta=1.)
```

For the default case used in here, the range of each parameter is `[0 , 1]`.

After the input parameters are set, we can generate a parameter sampling that will most efficiently estimate the UQ. In order to do this, the user must create a PCE object with the distributions. However, the user must also define a polynomial order. This defined as an integer:

```
# # Polynomial order
order = 5
```

Now we can create the PCE object and generate parameter samples:

```
from UncertainSCI.pce import PolynomialChaosExpansion

# Generate samples first, then manually query model, then give model output to pce.
pce = PolynomialChaosExpansion(distribution=[p1, p2, p3], order=order, plabels=plabels)
pce.generate_samples()
```

To use the samples generated by this PCE instantiation, users can access it through the object:

```
import numpy as np # output is np array
pce.samples
```

Which returns a numpy array of size `MxN` where `N` is the number of parameters and `M` is the number of samples (determined by uncertainSCI).

Running Parameter Samples for Model Outputs

While this is heavily dependent on the modeling pipeline, the generated samples can be save to disk to run in an external pipeline:

```
np.savetxt(filename,pce.samples)
```

or called within python:

```
for ind in range(pce.samples.shape[0]):
    model_output[ind, :] = model(pce.samples[ind, :])
```

If the samples are saved to disk to run asyncronously, they will need to be added to a PCE object if the original one is destroyed. This happens when it is run as a script, then rerun with data and is acheived with:

```
pce.set_samples(np.loadtxt(filename))
```

instead of running `pce.generate_samples()`. Similarly, the output results from the simulation will need to be a numpy array of `MxP` where `M` is the number of parameter sets, and `P` is the size of the solution array. 2D and 3D solutions can be flattened into a 1D array and collated into this solution array, then reshaped for visualization.

PCE and the Output Statistics

With the appropriate distributions and samples added to the PCE object and model output collect, the estimator and output statistics can be generated. First, the PCE must be built:

```
pce.build(model_output=model_output)
```

Then, output statistics can be return:

```
mean = pce.mean()
stdev = pce.stdev()
global_sensitivity, variable_interactions = pce.global_sensitivity()
quantiles = pce.quantile([0.25, 0.5, 0.75]) #  0.25, median, 0.75 quantile
```

There are also built in ploting functions (with matplotlib) for 1D data:

```
from matplotlib import pyplot as plt
mean_stdev_plot(pce, ensemble=50)
quantile_plot(pce, bands=3, xvals=x, xlabel='$x$')
piechart_sensitivity(pce)

plt.show()
```

The API documentation explains the implementation of UncertainSCI in more detail.

1.1.4 Running UncertainSCI Demos

There are a number of demos included with UncertainSCI to test it's installation and to demonstrate its use. The *previous description* can be found as a notebook with some more details here and as a script in `demos/build_pce.py`.

The demo scripts can be a way to quickly test the installation. Make sure that UncertainSCI is *installed*, then simply call the script with python using the command `python demos/build_pce.py`. Other demos can be run similarily.

We have included a number of demos and tutorials to teach users how to use UncertainSCI with various examples.

1.2 Support

1.2.1 Questions?

If you have questions, please ask them on our public discussion board: https://github.com/SCIInstitute/UncertainSCI/discussions

Or, you can ask them on our mailing list uncertainsci@sci.utah.edu. To subscribe:

- send an email to sympa@sci.utah.edu with `subscribe uncertainsci` in the body of the email.
- You will receive an email requesting confirmation of the subscription.
- Reply (no text needed) to the email to confirm submission.

Now you're on the list and will receive updates and questions. You can also submit questions to the list once subscribed

To unsubscribe from the mailing list:

- send an email to sympa@sci.utah.edu with `unsubscribe uncertainsci` in the body of the email.
- You will receive a confirmation email confirming the unsubcription.

You can also submit questions to our private support email at cibc-contact@sci.utah.edu

1.2.2 Bugs and Requests

If you find any bugs or have any feature requests, please create an issue on GitHub: https://github.com/SCIInstitute/UncertainSCI/issues

1.3 Tutorials

1.3.1 Simple Example Showing UQ With PCE

This example shows some basic functionality of UncertainSCI using the polynomial Chaos emulators (PCE) to quantify uncertainty in a 1D function. It is essentially identical to `demos/build_pce.py` and uses a 1D Laplacian ODE as a model with three parameters

[1]:
```
import numpy as np
from matplotlib import pyplot as plt

from UncertainSCI.distributions import BetaDistribution
from UncertainSCI.model_examples import laplace_ode_1d
from UncertainSCI.pce import PolynomialChaosExpansion
```

(continues on next page)

(continued from previous page)

```
from UncertainSCI.vis import piechart_sensitivity, quantile_plot, mean_stdev_plot
```

Model Parameter setup

The first step in running UncertainSCI is to specify the parameter distributions and the capacity of the PCE model (polynomial order).

Distributions

[2]:
```
# Number of parameters
Nparams = 3

# Three independent parameters with different Beta distributions
p1 = BetaDistribution(alpha=0.5, beta=1.)
p2 = BetaDistribution(alpha=1., beta=0.5)
p3 = BetaDistribution(alpha=1., beta=1.)

plabels = ['a', 'b', 'z']
```

Polynomial Order

How complicated the model is

[3]:
```
# # Polynomial order
order = 5
```

Forward Model

$$-\frac{d}{dx}a(x,p)\cdot\frac{d}{dx}u(x,p) = f(x)$$

with x in $[-1,1]$ discretized with N points, where $a(x,p)$ is a Fourier-Series-parameterized diffusion model with the variables $p_j = [a,b,z]$. See the `laplace_ode_1d` method or `UncertainSCI/model_examples.py` for details.

[4]:
```
N = 100
x, model = laplace_ode_1d(Nparams, N=N)
```

Running PCE in UncertainSCI

The steps to running PCE in UncertainSCI are - create PCE object - generate parameter samples - run parameter sets in the forward model - give model output to PCE - compute output statistics

Create PCE object and Build Sample Set

With the parameter distribution created and polynomial order set, we can create the PCE object and generate a parameter set that will efficiently and accurately estimate the output distribution of the model. UncertainSCI adds an extra 10 parameter sets as a precaution.

[5]:
```
pce = PolynomialChaosExpansion(distribution=[p1, p2, p3], order=order, plabels=plabels)
pce.generate_samples()

print('This queries the model {0:d} times'.format(pce.samples.shape[0]))
```

```
Precomputing data for Jacobi parameters (alpha,beta) = (-0.500000, 0.000000)...Done
This queries the model 66 times
```

Run Parameter Sets in the Forward Model

We query each parameter set in sequence to run in our Laplacian ODE model and collect the results

[6]:
```
model_output = np.zeros([pce.samples.shape[0], N])
for ind in range(pce.samples.shape[0]):
    model_output[ind, :] = model(pce.samples[ind, :])
```

Build PCE from Model Outputs

With the collated model solutions, the output distributions can be estimated using PCE

[7]:
```
pce.build(model_output=model_output)
```

[7]:
```
array([1.75571628e-37, 5.64860235e-17, 2.10743598e-16, 4.30180091e-16,
       6.73615485e-16, 8.98426649e-16, 1.06808811e-15, 1.15833221e-15,
       1.16057868e-15, 1.08209494e-15, 9.43231789e-16, 7.72726181e-16,
       6.02299430e-16, 4.61625174e-16, 3.74362183e-16, 3.55551159e-16,
       4.10403164e-16, 5.34387196e-16, 7.14489737e-16, 9.31478341e-16,
       1.16290358e-15, 1.38643597e-15, 1.58302359e-15, 1.73934375e-15,
       1.84914225e-15, 1.91328322e-15, 1.93860602e-15, 1.93592175e-15,
       1.91761943e-15, 1.89536658e-15, 1.87829599e-15, 1.87191182e-15,
       1.87777401e-15, 1.89387162e-15, 1.91549769e-15, 1.93639635e-15,
       1.94995954e-15, 1.95029022e-15, 1.93300605e-15, 1.89571844e-15,
       1.83817752e-15, 1.76211869e-15, 1.67087823e-15, 1.56886365e-15,
       1.46096875e-15, 1.35201633e-15, 1.24629417e-15, 1.14722754e-15,
       1.05720583e-15, 9.77557484e-16, 9.08648189e-16, 8.50065649e-16,
       8.00849025e-16, 7.59724059e-16, 7.25312441e-16, 6.96294722e-16,
       6.71517851e-16, 6.50048858e-16, 6.31183955e-16, 6.14427000e-16,
       5.99452329e-16, 5.86065283e-16, 5.74169862e-16, 5.63747820e-16,
```

(continues on next page)

(continued from previous page)

```
       5.54848130e-16, 5.47581156e-16, 5.42108762e-16, 5.38620514e-16,
       5.37288102e-16, 5.38194279e-16, 5.41239598e-16, 5.46038218e-16,
       5.51822165e-16, 5.57379403e-16, 5.61052914e-16, 5.60824129e-16,
       5.54493189e-16, 5.39951481e-16, 5.15520386e-16, 4.80308188e-16,
       4.34519740e-16, 3.79646212e-16, 3.18469931e-16, 2.54843985e-16,
       1.93246402e-16, 1.38157999e-16, 9.33612879e-17, 6.12921066e-17,
       4.25845893e-17, 3.59260824e-17, 3.82825975e-17, 4.54770963e-17,
       5.30202417e-17, 5.70278062e-17, 5.50298773e-17, 4.64948566e-17,
       3.29537537e-17, 1.77021507e-17, 5.15427668e-18, 0.00000000e+00])
```

Compute Output Statistics

With the PCE built, statistics can be calculated for each point in the solution space (x).

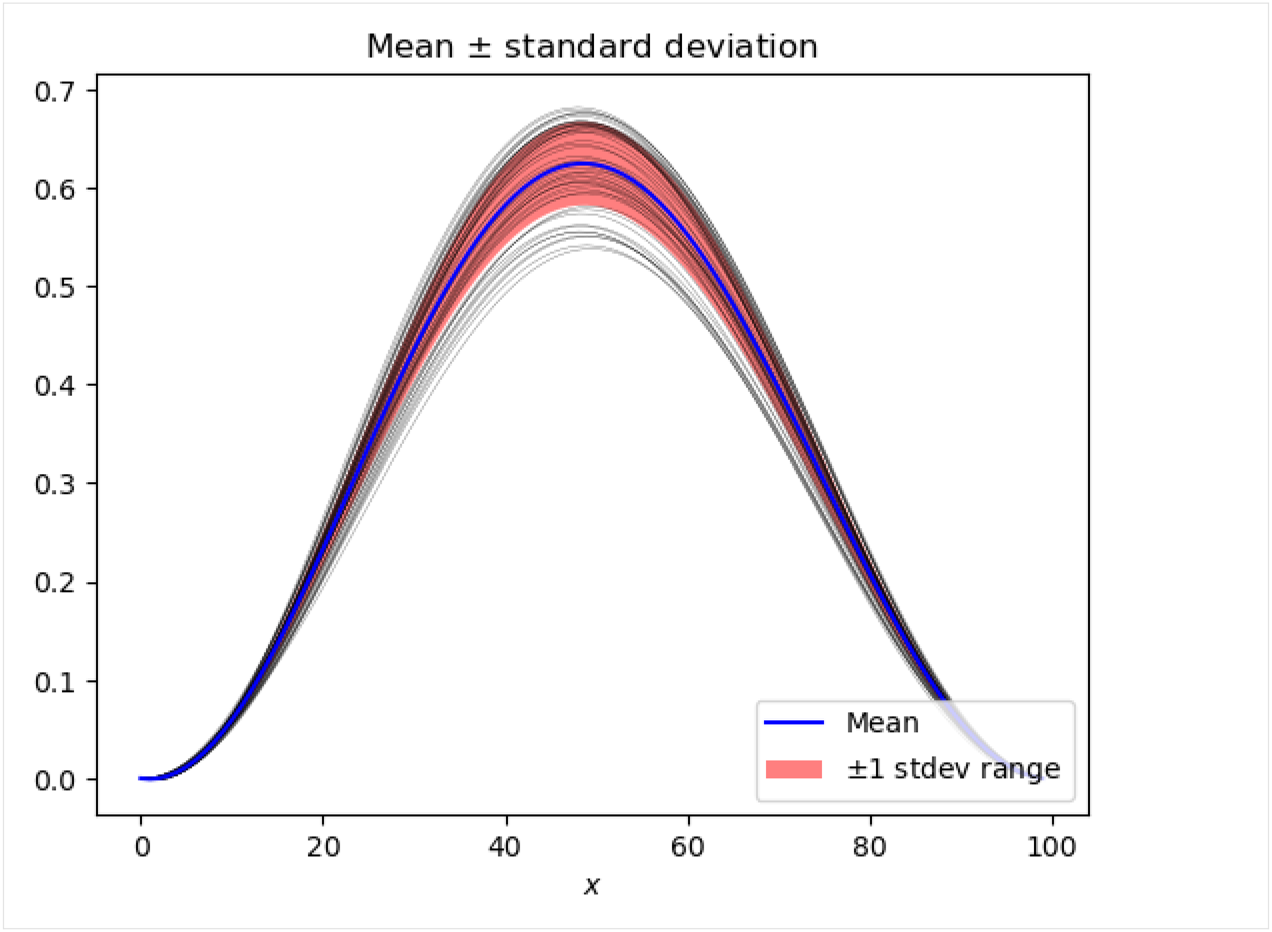
[8]:
```
## Postprocess PCE: statistics are computable:
mean = pce.mean()
stdev = pce.stdev()
global_sensitivity, variable_interactions = pce.global_sensitivity()
quantiles = pce.quantile([0.25, 0.5, 0.75]) #  0.25, median, 0.75 quantile
```

Output Visualizations

To help understand and interpret UQ statistics, we've included some plotting tools for 1D data.

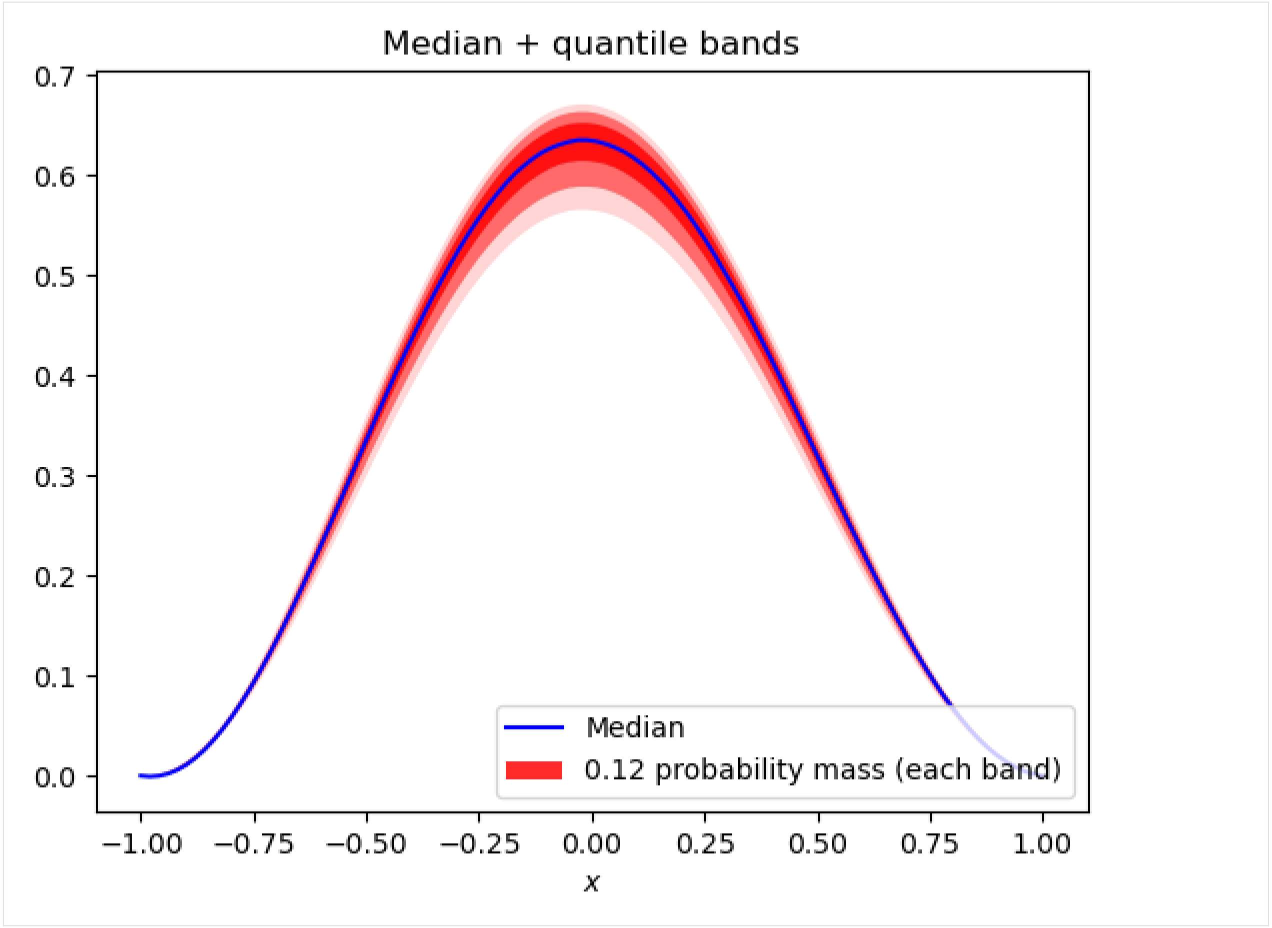
[9]:
```
mean_stdev_plot(pce, ensemble=50)
```

[9]:
```
<AxesSubplot:title={'center':'Mean $\\pm$ standard deviation'}, xlabel='$x$'>
```



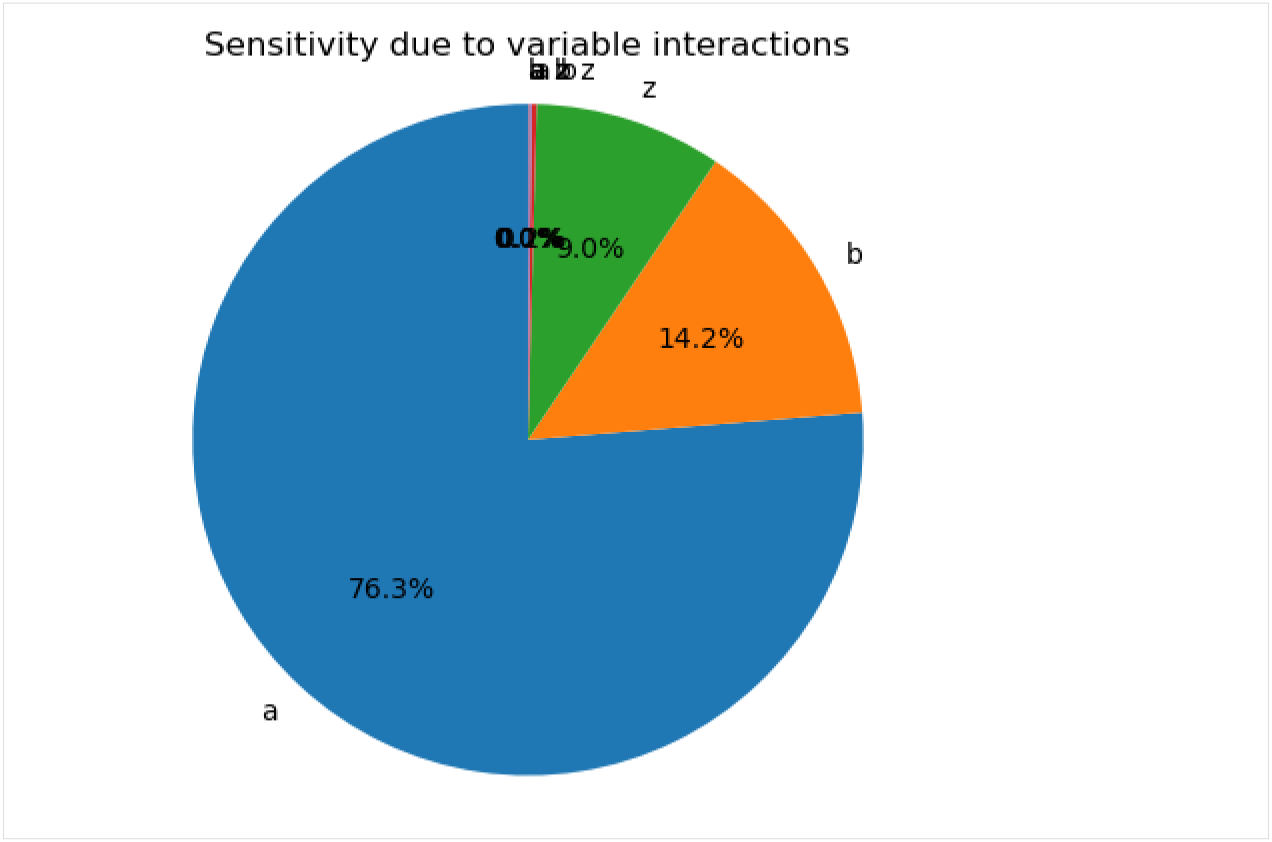
```
[10]: quantile_plot(pce, bands=3, xvals=x, xlabel='$x$')
```

```
[10]: <AxesSubplot:title={'center':'Median + quantile bands'}, xlabel='$x$'>
```



```
[11]: piechart_sensitivity(pce)
```

```
[11]: <AxesSubplot:title={'center':'Sensitivity due to variable interactions'}>
```



1.3.2 Demos

Simple example with boxplots

This project was supported by grants from the National Institute of Biomedical Imaging and Bioengineering (U24EB029012) from the National Institutes of Health.

Overview

This tutorial demonstrates basic construction of a `PolynomialChaosExpansion` (PCE) emulator, and generates box plots using these emulators. The code used in this document is in the `demos/simple_boxplots.py` demo file.

Building the emulator

UncertainSCI requires the user to specify

- a pointer that evaluates a forward model
- distributions for each input parameter to the model
- the polynomial order, which is a PCE expressivity parameter (alternatively, adaptive procedures can automatically select this)

The forward model

We will use the Ishigami function, a model built into UncertainSCI, as our forward model. The following imports this function

```
from UncertainSCI.model_examples import ishigami_function

# Parameters for function
a = 1
b = 0.05

f = ishigami_function(a,b)
```

This generates a function `f` of three parameters. See the `[Ishigami Function]` description for more information about this model.

Parameter distributions

A more complete discussion of creating parameter distributions is given in the `[Defining Random Parameters]` tutorial. Here we only present a simple setup. We will assume that our three parameters,

\begin{align} \mathbf{P} = (P_1, P_2, P_3) \in \mathbb{R}^3 \end{align}

are each independent and uniform on the interval \([-\pi, \pi]\). We set this up in UncertainSCI with the following:

```
from UncertainSCI.distributions import BetaDistribution, TensorialDistribution

## Set up parameter distributions
bounds = np.reshape(np.array([-np.pi, np.pi]), [2, 1])
p1 = BetaDistribution(alpha=1, beta=1, domain=bounds)
p2 = BetaDistribution(alpha=1, beta=1, domain=bounds)
p3 = BetaDistribution(alpha=1, beta=1, domain=bounds)

p = TensorialDistribution(distributions=[p1, p2, p3])
```

We have used the fact that a `BetaDistribution` with parameters `alpha=beta=1` corresponds to the uniform distribution. See `[Beta Distribution]` for more information about the Beta distribution. Note also that we have individually defined the parameters, and then combined them into a single three-dimensional random parameter using the `TensorialDistribution` command.

Defining polynomial order

There are many ways to set polynomial order in UncertainSCI. Here we describe a simple, manual approach. Using the command,

```
  index_set = TotalDegreeSet(dim=3, order=4)
```

would create an order-4 polynomial space in 3 dimensions (the number of parameters). In this particular demo, we will investigate the effect of increasing the polynomial order.

PCE emulators

We will create several PCE emulators, each corresponding to a different PCE order. We accomplish this as follows:

```
orders = [3, 4, 5, 6]
pces = []

for order in orders:
    index_set = TotalDegreeSet(dim=3, order=order)
    pce = PolynomialChaosExpansion(distribution=p, index_set=index_set)
    pce.build(model=f)
    pces.append(pce)
```

Each PCE object is instantiated by specifying the distribution `p` and the polynomial order via the `index_set` variable. The PCE is built from the model by calling `pce.build(model=f)`. This last command samples the model `f` as many times as needed for the chosen polynomial order. More advanced usage patterns allow finer control over how many samples of `f` are collected.

Building the PCE, which requires several forward model evaluations, is typically the most computationally costly portion of the pipeline. (In this particular demo it is not since the forward model is very cheap to evaluate.)

Querying the emulators

The PCE objects are computationally efficient emulators for evaluting `f(p)`. In particular, the following generates a random ensemble for the model output by querying an emulator `pce` at an input ensemble `pvals` for the parameter `p`:

```
ensembles = []
pvals = p.MC_samples(M=ensemble_size)

ensembles.append(pce.pce_eval(pvals))
```

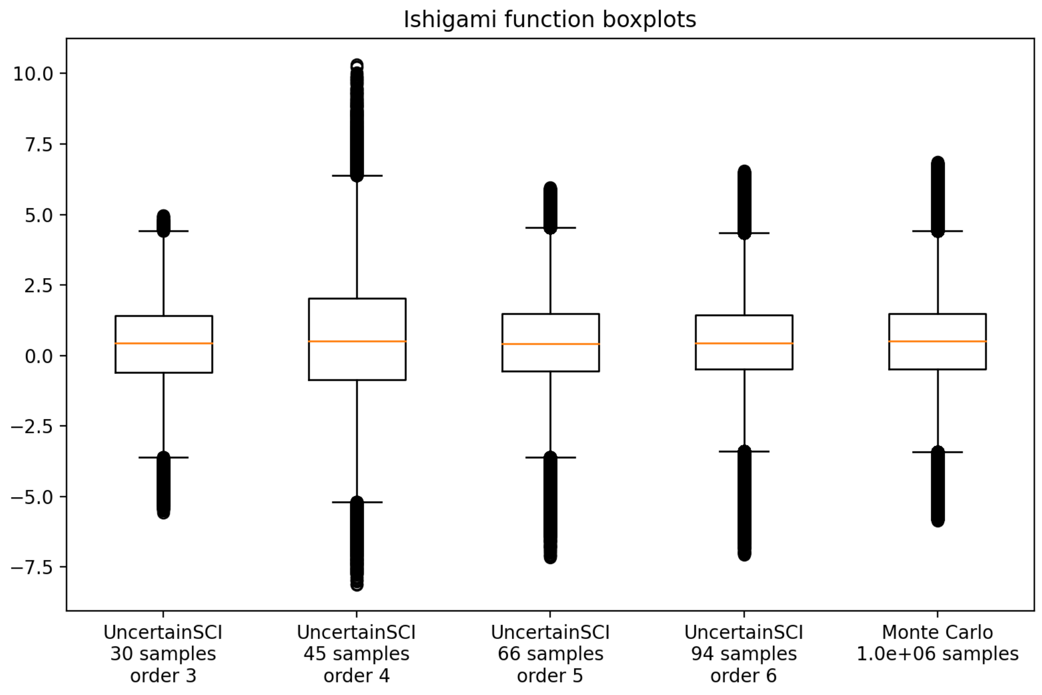
The number of samples required to build the PCE is relatively small (see the next section). However, the ensemble generated above does not require extra PCE samples from `f` to be generated. Thus, this ensemble sampling is very efficient.

Boxplots

We generate boxplots using standard Monte Carlo sampling and also sampling from the emulators. For this demo, we use an ensemble of size \(10^6\) to generate the boxplot visualizations. We compare the following two approaches.

- PCE emulators built using UncertainSCI with varying polynomial orders each collecting less than 100 forward model samples.
- A Monte Carlo (MC) approach that collects \(10^6\) forward model samples.

Boxplots are generated using the `matplotlib` boxplot feature.



PCE Statistics

Building PCE emulators

One of the main tools in *UncertainSCI* is the ability to build polynomial Chaos expansions. (See *Polynomial Chaos Expansions* for an overview of methodology.) In this first demo, we will investigate the `demos/build_pce.py` file and its output. The main goal of this demo is to build a PCE emulator for predicting variation in a model with respect to parameters.

To construct a PCE emulator, one needs to specify both a probability distribution on the parameters, and the expressivity of the PCE model.

In this example, we define a probability distribution over 3 parameters. We model these parmaeters as independent and uniform, each over the interval $[0, 1]$:

Listing 1: demos/build_pce.py: Specifying a probability distribution

```
# Number of parameters
dimension = 3

# Specifies 1D distribution on [0,1] (alpha=beta=1 ---> uniform)
alpha = 1.
beta = 1.
dist = BetaDistribution(alpha=alpha, beta=alpha, dim=dimension)
```

Here the `dimension` indicates that we have 3 random parameters. We have used the `BetaDistribution` class to define a Beta probability distribution (see *Distributions*) with `dimension` (3) independent variables. Since `alpha` and `beta` are scalars, they are assumed to apply for each dimension. The values `alpha=1` and `beta=1` correspond to the uniform density, and the default domain of the Beta distribution is $[0, 1]$.

Second, we need to specify the expressivity of the PCE model, which in turns will translate into how much data we'll need to gather from the model. Expressivity is defined by the type of polynomial space; for now, we'll impose a degree-5 total degree space:

Listing 2: demos/build_pce.py: Specifying expressivity of the PCE

```
order = 5
indices = TotalDegreeSet(dim=dimension, order=order)
```

See *Polynomial Chaos Expansions* for more about expressivity and polynomial spaces. With the distribution and the expressivity defined, we can instantiate the PCE object:

```
pce = PolynomialChaosExpansion(indices, dist)
```

We must now train the PCE on the model. For this simple, example, we take the model defined as

$$model(x,p) = \sin\left[\pi(p_1 + p_2^2 + p_3^3)x\right],$$

where x is one-dimensional physical space, and (p_1, p_2, p_3) are our three parameters. We create this modeling by specifying the discrete spatial grid over which x is defined. Our model is created via

Listing 3: demos/build_pce.py: Creating a parameterized model

```
N = int(1e2) # Number of degrees of freedom of model
model = sine_modulation(N=N)
```

This syntax implicity assumes `N` spatial grid points equispaced on the interval $[-1, 1]$. The function `model` takes as input a length-3 vector representing a parameter value for (p_1, p_2, p_3), and outputs a length-100 vector representing the value of the model output on the `N` spatial points at that parameter.

The simplest way to build the PCE emulator is to input the `model` to the `build` method of the `pce` object:

```
lsq_residuals = pce.build(model)
```

This call queries the `model` several times at different parameter locations, and uses this data to build a PCE emulator. The parameter locations along with the associated model data are accessible via

Listing 4: demos/build_pce.py: Accessing parameter locations and data from a built PCE emulator

```
parameter_samples = pce.samples
model_evaluations = pce.model_output
```

However, the main utility of having a now-built PCE emulator is that statistics (with respect to the parameters (p_1, p_2, p_3)) are easily computable. For example, the mean and standard deviation (which are functions of the spatial variable x) can be computed as

Listing 5: demos/build_pce.py: Computing the mean and standard deviation of a PCE emulator

```
mean = pce.mean()
stdev = pce.stdev()
```

More advanced operations are available. Variance-based sensitivity analysis can provide a means for ranking parameter importance. We can compute the so-called *total sensitivity index*, which measures the importance of each variable on a scale of 0 to 1, and also the *global sensitivity index*, which measures the relative importance that each *subset* of variables has to the overall variance:

Listing 6: demos/build_pce.py: Computing sensitivity indices

```
# Power set of [0, 1, 2]
variable_interactions = list(chain.from_iterable(combinations(range(dimension), r) for r␣
↪in range(1, dimension+1)))

total_sensitivity = pce.total_sensitivity()
global_sensitivity = pce.global_sensitivity(variable_interactions)
```

Finally, we can also compute quantiles (level sets of the cumulative distribution function) of the model output.

Listing 7: demos/build_pce.py: Computing quantiles and the median

```
Q = 4 # Number of quantile bands to plot
dq = 0.5/(Q+1)
q_lower = np.arange(dq, 0.5-1e-7, dq)[::-1]
q_upper = np.arange(0.5 + dq, 1.0-1e-7, dq)
quantile_levels = np.append(np.concatenate((q_lower, q_upper)), 0.5)

quantiles = pce.quantile(quantile_levels, M=int(2e3))
median = quantiles[-1,:]
```

The remainder of `build_pce.py` contains (a) simulations that compare PCE against methods using (much more expensive) Monte Carlo sampling, and (b) for visualizing the output. In particular, the following images are shown by running `build_pce.py`. **Note**: The procedures are randomized so that the output figures shown here may slightly vary with respect to results generated on a local machine.

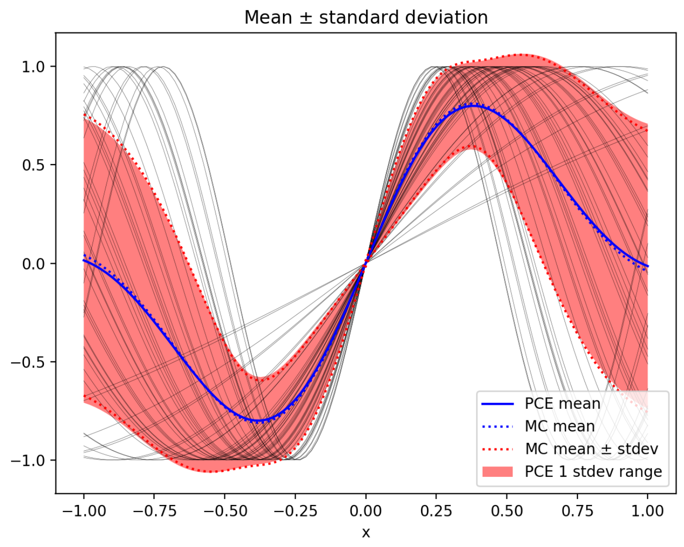


Fig. 1: Graphical output from `demos/build_pce.py` showing the predicted mean and standard deviation as a function of the spatial variable x, along with a comparison against Monte Carlo methods.

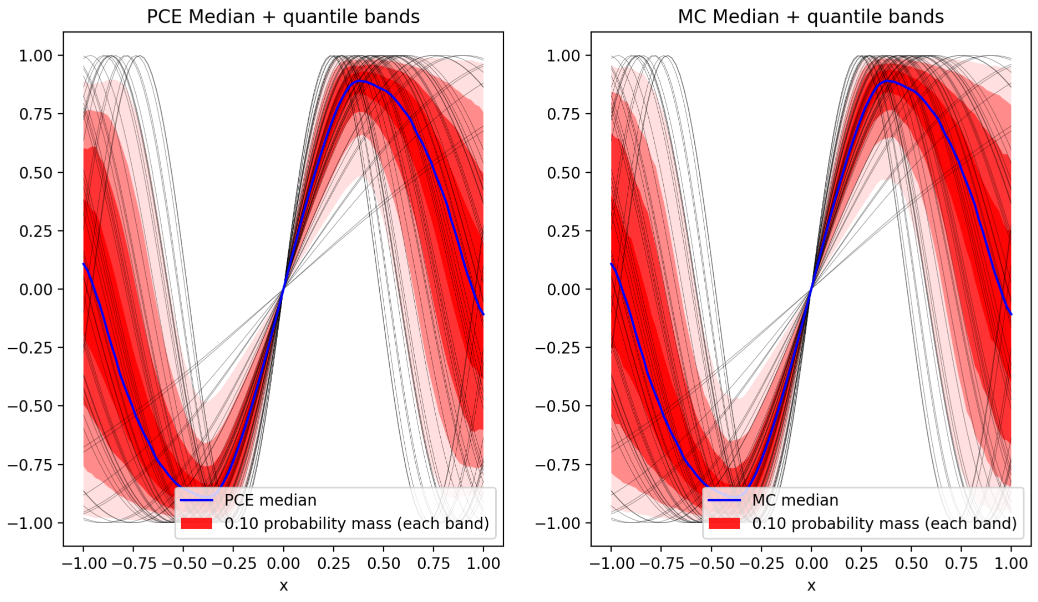


Fig. 2: Graphical output from `demos/build_pce.py` compared PCE output with medians and quantile bands against results from a more expensive Monte Carlo simulation.

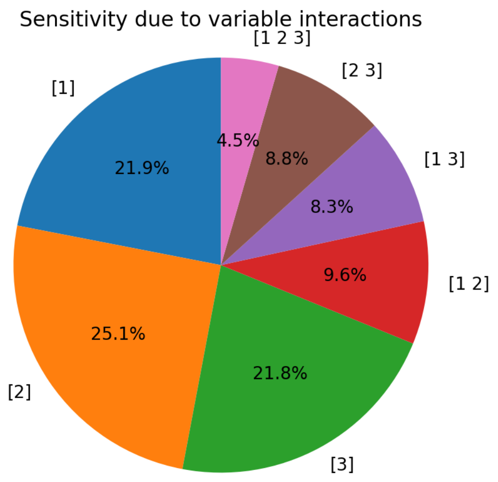


Fig. 3: Graphical output from `demos/build_pce.py` showing variance-based global sensitvity indices, measuring the relative importance of subsets of parameters.

1.3.3 Adaptive PCE

This notebook is an example of UncertainSCI's functionality to adaptively determine the needed polynomial order of a given model and parameter set. This example is equivalent to `demos/adapt_pce.py`

[1]:
```
from itertools import chain, combinations

import numpy as np
from matplotlib import pyplot as plt
import matplotlib.animation as animation

from UncertainSCI.distributions import BetaDistribution
from UncertainSCI.model_examples import KLE_exponential_covariance_1d, \
                                        laplace_ode, laplace_grid_x
from UncertainSCI.indexing import TotalDegreeSet
from UncertainSCI.pce import PolynomialChaosExpansion
from UncertainSCI.utils.version import version_lessthan

# some features work better with this option
%matplotlib notebook
```

Adaptive PCE Setup

Three things must be specified: - The physical model - A parameter distribution - The initial expressivity of the PCE (polynomial space)

Define Forward model

The model in this example is a 1D laplacian ODE. See the `laplace_ode_1d` method or `UncertainSCI/model_examples.py` for details.

$$-\frac{d}{dx}a(x,p)\cdot\frac{d}{dx}u(x,p) = f(x)$$

over x in $[-1,1]$, where $a(x,p)$ is a parameterized diffusion model:

$$a(x,p) = \bar{x} + \sum_{j=1}^{d}\lambda_j p_j \phi_j(x)$$

where d is the dimension, (λ_j, ϕ_j) are eigenpairs of the exponential covariance kernel,

$$K(s,t) = e^{-|s-t|/a}.$$

The p_j are modeled as random variables.

[2]:
```
# Number of parameters
dimension = 2

# Define diffusion coefficient
a = 1.
b = 1.  # Interval is [-b,b]
abar = lambda x: 1*np.ones(np.shape(x))
```

(continues on next page)

(continued from previous page)

```
KLE = KLE_exponential_covariance_1d(dimension, a, b, abar)

diffusion = lambda x, p: KLE(x, p)

N = int(1e2)  # Number of spatial degrees of freedom of model
left = -1.
right = 1.
x = laplace_grid_x(left, right, N)

model = laplace_ode(left=left, right=right, N=N, diffusion=diffusion)
```

Parameter Distributions

We are using two model parameters with the same beta distribution for this example.

[3]:
```


# Specifies 1D distribution on [0,1] (alpha=beta=1 ---> uniform)
alpha = 1.
beta = 1.
dist = BetaDistribution(alpha=alpha, beta=beta, dim=dimension)
```

Initial Expressivity of the PCE

This example uses a different setup than you might see in other examples. Other setups will be equivalent for many non-adaptive cases, but it is useful to operate with the index set in this example. Another important distinction that is necessary for the adaptive case is supplying the model as a lambda function to the PCE object.

[4]:
```
# Expressivity setup
order = 0
index_set = TotalDegreeSet(dim=dimension, order=order)
starting_indices = index_set.get_indices()
```

[5]:
```
# Building the PCE
pce = PolynomialChaosExpansion(index_set, dist)
pce.build(model=model)
Nstarting_samples = pce.samples.shape[0]
initial_accuracy = pce.accuracy_metrics.copy()
```

Running Adaptive PCE

With the initial PCE setup, we can run adaptive method. This will attempt to find the best order to represent the uncertainty for the given model and parameter distributions. The method will add new samples as needed to minimize the residual error between iterations. This may take a few minutes to run from scratch.

```
[39]: # pce.adapt_robustness(max_new_samples=50)
residuals, loocvs, added_indices, added_samples = pce.adapt_expressivity(max_new_
↪samples=100, add_rule=3)
```

```
new indices:       1,   new samples:      4
old residual: 1.418e-09,  old loocv: 4.334e-06
new residual: 1.424e-09,  new loocv: 4.259e-06
new indices:       1,   new samples:      4
old residual: 1.424e-09,  old loocv: 4.259e-06
new residual: 1.446e-09,  new loocv: 4.243e-06
new indices:       1,   new samples:      4
old residual: 1.446e-09,  old loocv: 4.243e-06
new residual: 9.789e-10,  new loocv: 2.748e-06
new indices:       1,   new samples:      4
old residual: 9.789e-10,  old loocv: 2.748e-06
new residual: 9.637e-10,  new loocv: 2.499e-06
new indices:       1,   new samples:      4
old residual: 9.637e-10,  old loocv: 2.499e-06
new residual: 6.000e-10,  new loocv: 2.085e-06
new indices:       1,   new samples:      4
old residual: 6.000e-10,  old loocv: 2.085e-06
new residual: 2.177e-10,  new loocv: 1.412e-06
new indices:       1,   new samples:      4
old residual: 2.177e-10,  old loocv: 1.412e-06
new residual: 9.205e-11,  new loocv: 9.741e-07
new indices:       1,   new samples:      4
old residual: 9.205e-11,  old loocv: 9.741e-07
new residual: 6.980e-11,  new loocv: 8.633e-07
new indices:       1,   new samples:      4
old residual: 6.980e-11,  old loocv: 8.633e-07
new residual: 6.651e-11,  new loocv: 8.321e-07
new indices:       1,   new samples:      4
old residual: 6.651e-11,  old loocv: 8.321e-07
new residual: 7.158e-11,  new loocv: 8.528e-07
new indices:       1,   new samples:      4
old residual: 7.158e-11,  old loocv: 8.528e-07
new residual: 4.889e-11,  new loocv: 6.559e-07
new indices:       1,   new samples:      4
old residual: 4.889e-11,  old loocv: 6.559e-07
new residual: 4.642e-11,  new loocv: 6.220e-07
new indices:       1,   new samples:      4
old residual: 4.642e-11,  old loocv: 6.220e-07
new residual: 4.520e-11,  new loocv: 6.129e-07
new indices:       1,   new samples:      4
old residual: 4.520e-11,  old loocv: 6.129e-07
new residual: 4.639e-11,  new loocv: 6.159e-07
new indices:       1,   new samples:      4
```

(continues on next page)

(continued from previous page)

```
old residual: 4.639e-11,  old loocv: 6.159e-07
new residual: 3.642e-11,  new loocv: 4.480e-07
new indices:      1,   new samples:      4
old residual: 3.642e-11,  old loocv: 4.480e-07
new residual: 3.216e-11,  new loocv: 3.926e-07
new indices:      1,   new samples:      4
old residual: 3.216e-11,  old loocv: 3.926e-07
new residual: 3.096e-11,  new loocv: 3.793e-07
new indices:      1,   new samples:      4
old residual: 3.096e-11,  old loocv: 3.793e-07
new residual: 2.029e-11,  new loocv: 3.257e-07
new indices:      1,   new samples:      4
old residual: 2.029e-11,  old loocv: 3.257e-07
new residual: 7.890e-12,  new loocv: 2.240e-07
new indices:      1,   new samples:      4
old residual: 7.890e-12,  old loocv: 2.240e-07
new residual: 2.909e-12,  new loocv: 1.355e-07
new indices:      1,   new samples:      4
old residual: 2.909e-12,  old loocv: 1.355e-07
new residual: 1.907e-12,  new loocv: 1.112e-07
new indices:      1,   new samples:      4
old residual: 1.907e-12,  old loocv: 1.112e-07
new residual: 1.733e-12,  new loocv: 1.058e-07
new indices:      1,   new samples:      4
old residual: 1.733e-12,  old loocv: 1.058e-07
new residual: 1.723e-12,  new loocv: 1.050e-07
new indices:      1,   new samples:      4
old residual: 1.723e-12,  old loocv: 1.050e-07
new residual: 1.540e-12,  new loocv: 9.896e-08
new indices:      1,   new samples:      4
old residual: 1.540e-12,  old loocv: 9.896e-08
new residual: 1.594e-12,  new loocv: 9.867e-08
```

Output Statistics

With the optimal polynomial order determined, computing the output statistics is as straightforward as other examples.

[40]:
```
# # Postprocess PCE: mean, stdev, sensitivities, quantiles
mean = pce.mean()
stdev = pce.stdev()
```

[41]:
```
# Set of subsets of [0, 1, ..., dimension-1] with at most 3 components
variable_interactions = list(chain.from_iterable(combinations(range(dimension), r) for r␣
↪in range(1, 3+1)))
```

[42]:
```
# "Total sensitivity" is a non-partitive relative sensitivity measure per
# parameter.
total_sensitivity = pce.total_sensitivity()
```

```
[43]: # "Global sensitivity" is a partitive relative sensitivity measure per set of
      # parameters.
      global_sensitivity = pce.global_sensitivity(variable_interactions)
```

```
[44]: # Quantile bands, similar to a box plot
      # used for PCE and MC
      Q = 3  # Number of quantile bands to plot
      dq = 0.5/(Q+1)
      q_lower = np.arange(dq, 0.5-1e-7, dq)[::-1]
      q_upper = np.arange(0.5 + dq, 1.0-1e-7, dq)
      quantile_levels = np.append(np.concatenate((q_lower, q_upper)), 0.5)
```

Monte Carlo Statistics

Here we compute the output statistics using Monte Carlo (MC) sampling to compare to the adaptive PCE method.

```
[45]: # # For comparison: Monte Carlo statistics
      M = 1000  # Generate MC samples
      p_phys = dist.MC_samples(M)
      output = np.zeros([M, N])

      for j in range(M):
          output[j, :] = model(p_phys[j, :])

      MC_mean = np.mean(output, axis=0)
      MC_stdev = np.std(output, axis=0)
      if version_lessthan(np,'1.15'):
          from scipy.stats.mstats import mquantiles
          MC_quantiles = mquantiles(output, quantile_levels, axis=0)
      else:
          MC_quantiles = np.quantile(output, quantile_levels, axis=0)
      MC_median = MC_quantiles[-1, :]
```

Visualize Statistics

Similar to other examples, we will plot the uncertainty over the domain of the domain of the model. This example shows how users might do so more manually instead of using the built in plotting functions

Plotting Setup

To plot our output statistics we are going to compute the quantile bands that we will visualize

```
[46]:
      quantiles = pce.quantile(quantile_levels, M=int(2e3))
      median = quantiles[-1, :]
```

Single Domain Plot Comparison

Plotting the mean and standard deviation for both PCE and MC. Each are plotted a different way to help differentiate them.

```
[49]: # # Visualization
V = 50  # Number of MC samples to visualize

# mean +/- stdev plot
plt.plot(x, output[:V, :].T, 'k', alpha=0.8, linewidth=0.2)
plt.plot(x, mean, 'b', label='PCE mean')
plt.fill_between(x, mean-stdev, mean+stdev, interpolate=True, facecolor='red',
                 alpha=0.5, label='PCE 1 stdev range')

plt.plot(x, MC_mean, 'b:', label='MC mean')
plt.plot(x, MC_mean+MC_stdev, 'r:', label='MC mean $\\pm$ stdev')
plt.plot(x, MC_mean-MC_stdev, 'r:')

plt.xlabel('x')
plt.title('Mean $\\pm$ standard deviation')

plt.legend(loc='lower right')
```

```
[49]: <matplotlib.legend.Legend at 0x7ff890449b50>
```

Side-by-Side Domain Plot Comparison

Here we plot similar data to the previous figure, but with two plots to allow the quantile bands to be shown for each method.

```
[50]: # quantile plot
plt.figure()


plt.subplot(121)
plt.plot(x, pce.model_output[:V, :].T, 'k', alpha=0.8, linewidth=0.2)
plt.plot(x, median, 'b', label='PCE median')

band_mass = 1/(2*(Q+1))

for ind in range(Q):
    alpha = (Q-ind) * 1/Q - (1/(2*Q))
    if ind == 0:
        plt.fill_between(x, quantiles[ind, :], quantiles[Q+ind, :],
                         interpolate=True, facecolor='red', alpha=alpha,
                         label='{0:1.2f} probability mass (each band)'.format(band_mass))
    else:
        plt.fill_between(x, quantiles[ind, :], quantiles[Q+ind, :],
                         interpolate=True, facecolor='red', alpha=alpha)

plt.title('PCE Median + quantile bands')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('x')
plt.legend(loc='lower right')

plt.subplot(122)
plt.plot(x, output[:V, :].T, 'k', alpha=0.8, linewidth=0.2)
plt.plot(x, MC_median, 'b', label='MC median')

for ind in range(Q):
    alpha = (Q-ind) * 1/Q - (1/(2*Q))
    if ind == 0:
        plt.fill_between(x, MC_quantiles[ind, :], MC_quantiles[Q+ind, :],
                         interpolate=True, facecolor='red',
                         alpha=alpha,
                         label='{0:1.2f} probability mass (each band)'
                         .format(band_mass))
    else:
        plt.fill_between(x, MC_quantiles[ind, :], MC_quantiles[Q+ind, :],
                         interpolate=True, facecolor='red', alpha=alpha)

plt.title('MC Median + quantile bands')
plt.xlabel('x')
plt.legend(loc='lower right')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
[50]: <matplotlib.legend.Legend at 0x7ff8b83d0e80>
```

Parameter Sensitivity Estimated with PCE

The pie chart shows the global sensitivity of the model to each parameter averaged over the domain. Global sensitivities can help differentiate model sensitivity from a single parameter and parameter interactions.

```
[51]: # Sensitivity pie chart, averaged over all model degrees of freedom
average_global_SI = np.sum(global_sensitivity, axis=1)/N

labels = ['[' + ' '.join(str(elem) for elem in [i+1 for i in item]) +
          ']' for item in variable_interactions]
_, ax = plt.subplots()
ax.pie(average_global_SI*100, labels=labels, autopct='%1.1f%%',
       startangle=90)
ax.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title('Sensitivity due to variable interactions')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
[51]: Text(0.5, 1.0, 'Sensitivity due to variable interactions')
```

Regression Plots

The regression plots in this section help illustrate the choices of the adaptive PCE method.

[56]:
```
# Index set plot
fig = plt.figure()

plt.subplot(121)

current_inds = plt.plot(starting_indices[:, 0], starting_indices[:, 1],
                        'k.', markersize=20, label='Current indices')[0]
rmargin = plt.plot([], [], 'bo', markersize=10, fillstyle='none',
                   label='Candidates')[0]
future_inds = plt.plot([], [], 'rx', markersize=15,
                       label='Chosen indices')[0]

plt.legend(frameon=False)

indices = pce.index_set.get_indices()
maxx, maxy = indices[:, 0].max(), indices[:, 1].max()
plt.axis([-0.5, maxx+1.5, -0.5, maxy+1.5])
plt.gca().set_yticks(range(maxy+2), minor=False)
plt.gca().set_xticks(range(maxx+2), minor=False)
plt.grid()
plt.xlabel('Degree in $p_1$')
plt.ylabel('Degree in $p_2$')

plt.subplot(122)
added_samples_plt = added_samples[:] # shallow copy
added_samples_plt.insert(0, Nstarting_samples)
Nsamples = np.cumsum(added_samples_plt)

plt.semilogy(Nsamples, residuals, 'k-', label='Residual')
plt.semilogy(Nsamples, loocvs, 'k--', label='LOOCV')
current_accuracy = plt.plot([], [], 'rx', markersize=15)[0]
plt.xlabel('Total model queries')
plt.legend(frameon=False)

def animation_init():
    return current_inds, future_inds, rmargin, current_accuracy

def animation_update(i):
    index_set = TotalDegreeSet(dim=dimension, order=order)
    for q in range(i):
        index_set.augment(added_indices[q])

    current_set = index_set.get_indices()
    reduced_margin = index_set.get_reduced_margin()

    current_inds.set_data(current_set[:, 0], current_set[:, 1])
    rmargin.set_data(reduced_margin[:, 0], reduced_margin[:, 1])
    future_inds.set_data(added_indices[i][:, 0], added_indices[i][:, 1])
    current_accuracy.set_data([Nsamples[i+1], Nsamples[i+1]],
```

(continues on next page)

(continued from previous page)

```
                              [residuals[i+1], loocvs[i+1]])

    return current_inds, future_inds, rmargin, current_accuracy

ani = animation.FuncAnimation(fig, animation_update,
                              np.arange(0, len(added_indices)),
                              interval=500, blit=True,
                              init_func=animation_init, repeat=False)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:

```

```

1.3.4 Built-in forward models

This project was supported by grants from the National Institute of Biomedical Imaging and Bioengineering (U24EB029012) from the National Institutes of Health.

Overview

UncertainSCI is distributed with simple forward models included. These forward models stem from a variety of applications in uncertainty quantification (UQ), and many are typical test problems that are used to validate UQ algorithms. These models are included with UncertainSCI so that users can test UQ procedures in a standalone way.

Algebraic models

Several simple algebraic models are described below. These are functions of explicit form that allow validation and debugging of UQ algorithms. In particular, statistics of many of these examples are explicitly computable.

Ishigami function

Given real parameters \((a,b) \in \mathbb{R}^2\), the Ishigami function is given by

\begin{align} f(\mathbf{p}) &= \left(1 + b p_3^4 \right) \sin p_1 + a \sin^2 p_2, & \mathbf{p} &= (p_1, p_2, p_3) \end{align}

Because this function has an explicit ANOVA decomposition, its partial variances are explicitly computable. If each parameters \(p_i\) is modeled as uniform random variable over \([-\pi, \pi]\), i.e., \(p_i \sim \mathcal{U}\left([-\pi, \pi]\right)\), then the variances are given by,

\begin{align} \mathrm{Mean}[f(\mathbf{p})] &= \frac{a}{2}, \\ \mathrm{Var}[f(\mathbf{p})] &= \frac{1}{2} + \frac{a^2}{8} + \pi^4 b \left(\frac{1}{5} + \frac{\pi^4 b}{18} \right), \\ \mathrm{Var}_1 [f(\mathbf{p})] &= \frac{1}{2} \left(1 + b \frac{\pi^4}{5}\right)^2, \\ \mathrm{Var}_2 [f(\mathbf{p})] &= \frac{a^2}{8}, \\ \mathrm{Var}_{13} [f(\mathbf{p})] &= \pi^8 b^4 \left(\frac{1}{18} - \frac{1}{50}\right) \end{align}

Borehole function

The Borehole function is given by

\begin{align} f(\mathbf{p}) = \frac{g_1(\mathbf{p})}{g_2(\mathbf{p}) g_3(\mathbf{p})}, \end{align} where \begin{align} g_1(\mathbf{p}) &= 2\pi p_3 (p_4 - p_6), & g_2(\mathbf{p}) &= \log(p_2/p_1), & g_3(\mathbf{p}) &= 1 + \frac{2 p_7 p_3}{g_2(\mathbf{p}) p_1^2 p_8} + \frac{p_3}{p_5} \end{align} where the 8 parameters collected in \(\mathbf{p}\) are \begin{align} \mathbf{p} = (p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8) = (r_w, r, T_u, H_u, T_l, H_l, L, K_w), \end{align} with physical interpretation and typical distributions as given below,

- \(r_w \sim \mathcal{U}([0.05, 0.15])\): Borehole radius [m]
- \(r \sim \mathcal{U}([100, 50,000])\): Borehole radius of influence [m]
- \(T_u \sim \mathcal{U}([63,070, 115,600])\): Upper acquifer transmissivity [m^2/year]
- \(H_u \sim \mathcal{U}([990, 1,100])\): Upper acquifer pentiometric head [m]
- \(T_l \sim \mathcal{U}([63.1, 116])\): Lower acquifer transmissivity [m^2/year]
- \(H_l \sim \mathcal{U}([700, 820])\): Lower acquifer pentiometric head [m]
- \(L \sim \mathcal{U}([1,120, 1,680])\): Borehole length [m]
- \(K_w \sim \mathcal{U}([9,885, 12,045])\): Borehole hydraulic conductivity [m/year]

The output function \(f\) models the water flow rate. [@citation-gupta1983]

Differential equation models

More complicated forward models are included in this section, which are solutions to parametric differential equations.

ODE boundary value problem

This model in this example is the solution to an ordinary differential equation (ODE) with random parameters $\mathbf{p}$. The solution \(u = u(x,\mathbf{p})\) depends on the spatial variable \(x\) lying on the compact domain \([x_-, x_+]\), and is governed by the ODE and boundary conditions,

\begin{align} -\frac{d}{dx} \left(a(x,\mathbf{p}) \frac{d}{dx} u(x,\mathbf{p}) \right) &= f(x), & x &\in (x_-, x_+) \\ u(x_-, \mathbf{p}) &= 0, & u(x_+, \mathbf{p}) &= 0. \end{align} Above, the forcing function \(f\) and the diffusion coefficient \(a(x, \mathbf{p})\) are specified functions. The diffusion coefficient must be strictly positive for every parameter value $\mathbf{p}$ to ensure the above ODE is well posed. One particular specification of the diffusion coefficient is as a (truncated) Karhunen-Loeve expansion (KLE), \begin{align} a(x,\mathbf{p}) &= a_0(x) + \sum_{j=1}^d p_j a_j(x), & \mathbf{p} &= (p_1, \ldots, p_d), \end{align} where the mean diffusion behavior \(a_0\) is given, and the functions \(a_j\) for \(j = 1, \ldots, d\) are (scaled) eigenfunctions of the integral operator associated to an assumed covariance kernel of a random field \(a(x)\). In UncertainSCI, one example of such a truncated KLE associated to an exponential covariance kernel can be created using the following:

```python
imoprt numpy as np
from UncertainSCI.model_examples import KLE_exponential_covariance_1d

xminus = -1.
xplus = 1.
abar = lambda x: 3*np.ones(np.shape(x))  # Constant a_0 equal to 3
d = 4

a = KLE_exponential_covariance_1d(d, xminus, xplus, abar)
```

The output a is a function with two inputs \(x\) and \(\mathbf{p}\). We discretize the \(x\) variable with \(N\) equispaced points on \([x_-, x_+]\) and use a finite-difference method to solve the ODE. The following code then creates the ODE model with this parametric diffusion coefficient:

```
from UncertainSCI.model_examples import laplace_grid_x, laplace_ode

N = 100
x = laplace_grid_x(xminus, xplus, N)

f = lambda x: np.pi**2 * np.cos(np.pi*x)

u = laplace_ode(left=xminus, right=xplus, N=N, diffusion=a, f=f)
```

This model can be queried using the syntax `u(p)`, where `p` is a `d`-dimensional parameter vector.

1.3.5 Using UncertainSCI with External Simulation Software

UncertainSCI's non-invasive methods and architecture allow it to be used with simulations run with a variety of software. The only requirements is to take parameter sets from UncertainSCI and to generate a set of solutions for UncertainSCI to use. This can be acheived with software that is implemented in Python or contains a Python API, or by creating a hard disk data passing system. We will include some examples on how to use these systems to integrate simulations with UncertainSCI.

We most often interface UncertainSCI with SCIRun, a simulation software we also produce, to UQ predictions on Bioelectric field simulations.

SCIRun/UncertainSCI ECG uncertainty due to cardiac postion

An uncertainty quantification (UQ) example computing the effect of heart position on boundary element method (BEM) ECG forward computations. This example is similar to the work of Swenson, etal. but is implemented in UncertainSCI and SCIRun.

Code and example data are found on GitHub: https://github.com/SCIInstitute/UQExampleBEMHeartPosition

1.3.6 Defining random parameters

This project was supported by grants from the National Institute of Biomedical Imaging and Bioengineering (U24EB029012) from the National Institutes of Health.

Overview

UncertainSCI propagates randomness in input parameters to the distribution of the model output. In order to accomplish this, a probability distribution on the input random parameters must be specified. This tutorial describes how to create random parameters for use in UncertainSCI.

Let \(\mathbf{p}\) be a \(d\)-dimensional random parameter, \begin{align} \mathbf{P} = (P_1, \ldots, P_d), \end{align} and assume that all parameters are independent. UncertainSCI specifies the joint distribution of \(\mathbf{P}\) by defining univariate distributions for each parameter \(P_i\) for \(i = 1, \ldots, d\). There are two main ways to specify these distributions:

- A univariate distribution for each \(P_i\) can be individually specified, and the joint distribution can be built from these univariate ones using the `TensorialDistribution` class.

- If \(P_i\) for every \(i = 1, \ldots, d\) each have distributions from the same parametric family (for example, if each \(P_i\) has an exponential distribution), then the joint distribution for \(P\) can be built from one call to the parametric family class.

We describe below first how to create univariate distributions of various types, and this is followed by examples of how to build joint (multivariate) distributions.

Types of distributions

If \(P\) is a scalar continuous random variable, we let \(f_P\) denote its probability density function (pdf). When \(P\) is discrete, we will still write a density \(f_P\) using Dirac (delta) masses located on the discrete support points of \(P\). In particular, \(\delta_y(p)\) denotes the Dirac delta, a function of \(p\), centered at \(p = y\).

Beta Distribution

Let \(P\) be a random variable with Beta distribution, having shape parameters \(\alpha\in (0,\infty)\) and \(\beta\in(0,\infty)\). The pdf for \(P\) is

\begin{align} f_P(p) &= \frac{ p^\alpha (1-p)^\beta}{B(\alpha,\beta)}, & p &\in (0, 1), \end{align} where \(B(\cdot,\cdot)\) is the Beta function, which is a normalization constant ensuring that \(f_P\) is a pdf.

The shape parameters \(\alpha\) and \(\beta\) dictate how the mass of \(P\) concentrates toward/away from the endpoints \(0\) and \(1\). Specializations of the Beta distribution include:

- Uniform distribution on \([0, 1]\): \(\alpha = \beta = 1\)
- Arcsine distribution on \([0, 1]\): \(\alpha = \beta = \frac{1}{2}\)
- Wigner semicircle distribution on \([0, 1]\): \(\alpha = \beta = \frac{3}{2}\)

Beta distributions are instantiated in UncertainSCI using the `distributions.BetaDistribution` class, which takes as inputs the shape parameters `alpha` and `beta`, corresponding to \(\alpha\) and \(\beta\), respectively. For example, the following creates a parameter P having Beta distribution with \((\alpha,\beta) = (1,1)\):

```
from UncertainSCI.distributions import BetaDistribution

P = BetaDistribution(alpha=1, beta=1)
```

Normal distribution

Let \(P\) be a random variable with a normal distribution, having mean \(\mu\in \mathbb{R}\) and variance \(\sigma^2\in(0,\infty)\). The pdf for \(P\) is

\begin{align} f_P(p) &= \frac{ 1}{\sigma \sqrt{2 \pi} } \exp \left(-\frac{(p-\mu)^2}{2 \sigma^2} \right), & p &\in \mathbb{R}. \end{align}

Normal distributions are instantiated in UncertainSCI using the `distributions.NormalDistribution` class, which takes as inputs the distribution statistics `mean` and `cov`, corresponding to \(\mu\) and \(\sigma^2\), respectively. For example, the following creates a parameter P having normal distribution with \((\mu,\sigma^2) = (1,3)\):

```
from UncertainSCI.distributions import NormalDistribution

P = NormalDistribution(mean=1, cov=3)
```

Exponential distribution

Let \(P\) be a random variable with an exponential distribution, having minimum \(p_0 \in \mathbb{R}\) and shape parameter \(\lambda \in (0, \infty)\). The pdf for \(P\) is

\begin{align} f_P(p) &= \lambda e^{-\lambda (p - p_0)}, & p &\in [p_0, \infty). \end{align}

Exponential distributions are instantiated in UncertainSCI using the `distributions.ExponentialDistribution` class, which takes as inputs the parameters `loc` and `lbd`, corresponding to \(p_0\) and \(\lambda\), respectively. For example, the following creates a parameter `P` having exponential distribution with \((p_0,\lambda) = (-1,2)\):

```python
from UncertainSCI.distributions import ExponentialDistribution

P = ExponentialDistribution(loc=-1, lbd=2)
```

Discrete uniform distribution

Let \(P\) be a random variable with a discrete uniform distribution, having \(n \in \mathbb{N}\) equally-spaced support points on \([0, 1]\). The pdf for \(P\) is

\begin{align} f_P(p) &= \sum_{j=1}^n \frac{1}{n} \delta_{\frac{j-1}{n-1}}(p) \end{align}

Discrete uniform distributions are instantiated in UncertainSCI using the `distributions.DiscreteUniformDistribution` class, which takes as inputs the parameter `n`, corresponding to the number of support points \(n\). For example, the following creates a parameter `P` having discrete uniform distribution with \(n = 17\):

```python
from UncertainSCI.distributions import DiscreteUniformDistribution

P = DiscreteUniformDistribution(n=17)
```

Non-parametric distributions

Coming soon. . . .

1.3.7 Template for Tutorial

This project was supported by grants from the National Institute of Biomedical Imaging and Bioengineering (U24EB029012) from the National Institutes of Health.

Authors:Smart people here

Overview

** Overview of the document **

Software Requirements

Overview Subsection

Chapter Name

** Overview text for the Chapter. In this case there will be examples of several types of content**

Section

Example section with subsection.

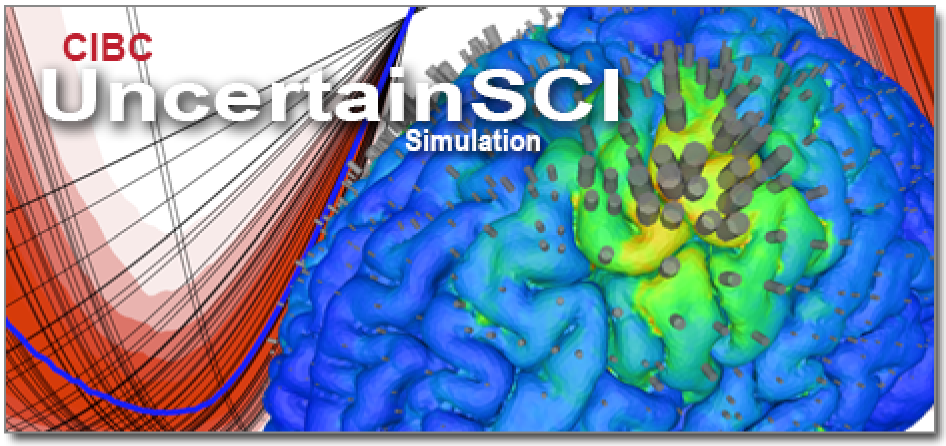
Use as many paragraphs as needed.

There are many markdown guides. Here are some examples: https://www.markdownguide.org/basic-syntax/ https://guides.github.com/features/mastering-markdown/

Subsection

Example Subsection. These can go to 6 #'s. Subsections are optional for table of contents and chapter scope.

Figures



Math

Math equations use MathJax. This requires the inclusion of this near the beginning of the document:

```
<script type="text/javascript" async
  src="https://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS_CHTML">
</script>
```

Example equation: \[x = {-b \pm \sqrt{b^2-4ac} \over 2a} \] $$ \frac{\partial \rho}{\partial t} + \nabla \cdot \vec{j} = 0 ,. \label{eq:continuity} $$

inline equations use the `\\(\mathbf{p}\\)` sytanx: \(\mathbf{p}\)

Citations

Sphinx has a built in citation manager for bibtex: sphinxcontrib-bibtex. Works well for RST, but we are still working on it for markdown. The sphinxcontrib-bibtex is built to run with rst in Sphinx. However, it can be used in markdown using the AutoStructify package.

The whole paragraph will need to be in the eval_rst block [2]. For multiple references: [2, 6]

add a bibliography section

~~~
```eval_rst
.. bibliography::
```
~~~

Snippets

Inline snippets `like this`. Muliple lines:

```
# # Define model
N = int(1e2)  # Number of degrees of freedom of model
left = -1.
right = 1.
x = np.linspace(left, right, N)
model = sine_modulation(N=N)
```

Links

Internal link: *Overview*

External link: https://www.markdownguide.org, or Markdown

Tables

Tables can be used with normal markdown syntax with the sphinx-markdown-tables package

```
| Syntax      | Description |
| ----------- | ----------- |
| Header      | Title       |
| Paragraph   | Text        |
```

Referencing Sphynx

To link the UncertainSCI API generated using Sphynx, Use this syntax: `[text](../api_docs/pce.html#polynomial-chaos-expansions)`

1.4 Developer Documentation

1.4.1 Contribution Guide

Thank you for you contributions to UncertainSCI! We welcome and appreciate and contributions, from reporting a bugs to code contributions. If you wish to contribute, please do so in the following ways.

Community Support

A great way to start contributing to UncertainSCI is to submit and answer questions on our [discussion board]https://github.com/SCIInstitute/UncertainSCI/discussions.

Other ways of contacting the communtity are located on our support page

Bugs and Features

We encourage users to report any bugs they find and request any features they'd like as a [GitHub issue]https://github.com/SCIInstitute/UncertainSCI/issues. If you would like to tackle any issues, please volunteer by commenting in the issue or [assigning yourself]https://docs.github.com/en/issues/tracking-your-work-with-issues/assigning-issues-and-pull-requests-to-other-github-users.

Make a Tutorial

If you have a tutorial you'd like to share, we'd love to have it. We have a Tutorial Tutorial to explain how to make and contribute tutorials.

Contribute Code

We appreciate to code maintenance and development that our community can provide. If you'd like to submit a bug fix, dependency update, or an added feature, please keep in mind the style guide, create a *fork* of the UncertainSCI repo, and use a *Pull Request* to add it to UncertainSCI.

It is best practice to make sure that there is [GitHub issue]https://github.com/SCIInstitute/UncertainSCI/issues to describe the required changes to the code, and having these issues documented become more important with the scope of the additions and changes. Possible additions can also be discussed on our [discussion board]https://github.com/SCIInstitute/UncertainSCI/discussions.

Fork Repo

With your own github account, go to the UncertainSCI Github page. Click the fork button on the upper right side of the page. It will ask you where to move the fork to, chose your own account. Once the repository is forked, clone it to your local machine with the following command.

```
$git clone https://github.com/[yourgithubaccount]/UncertainSCI.git
```

After the the code is cloned, navigate to the repository directory and add the upstream path to the original UncertainSCI repository.

```
$git remote add upstream https://github.com/SCIInstitute/UncertainSCI.git
```

You should be able to see both your and the original repository when you use the command:

```
$git remote -v
```

The fork is good to go, but you will need to sync the fork occasionally to keep up with the changes in the main repository. To sync your fork, use the following commands:

```
$git fetch upstream

$git checkout master

$git merge upstream/master
```

You should sync and merge your fork before you start a new module and before you create a pull request.It is a good practice to create a new branch in your fork for every module you will be adding. The command to create a new branch is:

```
$git checkout -b [branch_name]
```

Please see the Github help page for more information.

Pull Requests

With the contributions added to a branch on a fork of UncertainSCI, it is ready to create a [pull request]https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests. While this can be done in many ways, the easiest is probably through the web page of the forked repo. When navigating to the main page, it will usually display the a `contribute` button near the top of the page for recently updated branches. This is a shortcut to creating a pull request to the main repo and branch. Alternatively, a pull request can be created from the pull request tab of either the main repo or the fork. Before making a pull request, please make sure that you've tried your best to follow the style guide, and that the branch is up-to-date with the lastest master branch. Also, please update or add *testing* as appropriate.

Once the pull request is created, the maintainers of UncertainSCI will assign reviewers who will test and review the code to ensure that it meets the requirements of the style guide and is stable. It is best to limit the size of each pull request to facilitate review, so if there are major new additions, please add a [GitHub issue]https://github.com/SCIInstitute/UncertainSCI/issues to track the progress.

Syle Guide

If you are editing code, take a few minutes to look at the code around you and determine its style. Please try to keep the style of new code as similar as possible to unchanged code to avoid jarring inconsistencies in the style.

Python

- Indentation is 4 spaces per level for consistency with the rest of the code base. This may be revisited in the future. Do not use tabs.
- Text encoding: UTF-8 is preferred, Latin-1 is acceptable
- Comparisons:
  - To singletons (e.g. None): use 'is' or 'is not', never equality operations.
  - To Booleans (True, False): don't ever compare with True or False (for further explanation, see PEP 8).
- Prefix class definitions with two blank lines
- Imports
  - Grouped in order of scope/commonallity
    - Standard library imports
    - Related third party imports
    - Local apps/library specific imports
      - UncertainSCI application imports and local/module imports may be grouped independently.
  - One package per line (with or without multiple function/module/class imports from the package)
- Avoid extraneous whitespaces

Demos and Usecases

New demos and usecases are always welcome. Please add self-contained scripts demonstrating core functionality to the [demos folder]https://github.com/SCIInstitute/UncertainSCI/tree/master/demos. Demos that require external packages can be located in seperate repos, such as this [UQ BEM heart position usecase]https://github.com/SCIInstitute/UQExampleBEMHeartPosition

Testing

In addition to demos, please add unit testing to new function contributed to UncertainSCI using pytest. Unit test should be placed in the [test folder]https://github.com/SCIInstitute/UncertainSCI/tree/master/tests, which contains several tests to use as examples. To run the test, use the command:

```
pytest tests
```

1.4.2 Making Tutorials

This project was supported by grants from the National Institute of Biomedical Imaging and Bioengineering (U24EB029012) from the National Institutes of Health.

Authors:Jess Tate

Overview

This tutorial demonstrates how to use markdown to create new tutorials for UncertainSCI. It will walk through all the files needed and the basic structure needed expected for tutorials. Knowledge of Markdown, Github, Python, Sphinx, and Read the Docs will be useful. If you have questions, please ask.

Software requirements

UncertainSCI

To make a Tutorial for UncertainSCI, start with an up-to-date version of the code and documentation. Download the source code or clone the repository from github. We suggest *creating a fork* of the repository so that you can track your changes and create pull requests to the UncertainSCI repository. UncertainSCI requirements are found here

Dependencies and Development Tools

UncertainSCI uses Read the Docs and Sphinx to build and host tutorial documentation. This platform converts markdown files to html for web viewing using Sphinx, a Python Library. Testing the new documentation may require building the web pages locally for viewing. This will require installing Python, pip, Sphinx, Recommonmark, and other packages in the `docs/requirements.txt` file. More information can be found in the walkthorugh and on the Sphinx documentation.

Creating Your UncertainSCI Fork

With your own github account, go to the UncertainSCI Github page. Click the fork button on the upper right side of the page. It will ask you where to move the fork to, chose your own account. Once the repository is forked, clone it to your local machine with the following command.

```
$git clone https://github.com/[yourgithubaccount]/UncertainSCI.git
```

After the the code is cloned, navigate to the repository directory and add the upstream path to the original UncertainSCI repository.

```
$git remote add upstream https://github.com/SCIInstitute/UncertainSCI.git
```

You should be able to see both your and the original repositories when you use the command:

```
$git remote -v
```

The fork is good to go, but you will need to sync the fork occasionally to keep up with the changes in the main repository. To sync your fork, use the following commands:

```
$git fetch upstream

$git checkout master

$git merge upstream/master
```

You should sync and merge your fork before you start a new module and before you create a pull request.It is a good practice to create a new branch in your fork for every module you will be adding. The command to create a new branch is:

```
$git checkout -b [branch_name]
```

Please see the Github help page for more information.

Files Needed for a New Tutorial

This chapter will describe the files need to create a Tutorial for UncertainSCI.

Overview of Files Needed for a Tutorial

To make a new tutorial, a markdown file is required for the content of the tutorial. Other files, such as images, may also be included. In addition to the new files for the tutorial, a link to the new tutorial should be added to the *User Documents* file.

Markdown File

The main file needed for a new tutorial is a markdown file. The file should have an file ending of *.md* and should be located in the `UncertainSCI/docs/tutorials/` directory. There is a template file that can be used, or an existing tutorial like this one can be used.

If using math equations in a latex, a call to the mathjax server is required at the begining of the document:

```
<script type="text/javascript" async
  src="https://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS_CHTML">
</script>
```

Sphinx will build the menu and table of contents from the header names, and the document will be formated correctly if the proper markdown headings levels are used. For instance, the Document title should be the first content:

```
# Title
```

Then the chaper headings:

```
## Overview
## Chapter The First
## ETC.
```

An important part of the document is to acknowledge the authors and funding sources. Include text similar to this after the document title:

```
This project was supported by grants from the National Institute of Biomedical Imaging␣
↪and Bioengineering (U24EB029012) from the National Institutes of Health.

Authors:
Smart people here
```

Chapters and sections will appear in the table of contents with the page titles. Once the file is *added to the docs/tutorial/index.rst file*

Added Figures

Most tutorials require a number of screenshots and other images. Figures and images should be added in a folder for each tutorial in the `UncertainSCI/docs/user_docs/` folder. The figure folder should be named after the tutorial, for example, the images in this tutorial are in a folder called `HowToTutorial_figures`. Instructions on how to use the images and figures are found *here*

Additional Files

Additional files added to the `tutorials` folder should be minimized as much as possible. Example scripts should be located in the `UncertainSCI/examples` directory and example data will, generally, need separate hosting. However, if one or two files are needed, they may be added to the `UncertainSCI/docs/tutorials/` with a consistent naming scheme. Bibtex file with a matching name should be added in `UncertainSCI/docs/tutorials/`, yet if multiple additional files are needed, they should placed in a new folder indicating the tutorial: `UncertainSCI/docs/tutorials/[tutorial name]_files`.

Linking to New Tutorial

For the new tutorial to be visible on the tutorials page, add the filename to the list in the `UncertainSCI/docs/tutorials/index.rst` file.

Testing Documentation

This chapter describes how to test the look and content of the new tutorial. Test the generated github-pages with either a local Sphinx build or using the online build on Read the Docs.

Testing Locally

Testing the documentation locally involves building documentation website on your local machine using Sphinx and Recommonmark. These instructions are adapted from Read the Docs' help page.

Installing Shpinx

To install the Sphinx, make sure that a relatively recent version of Python 3. Installing sphinx and other dependencies is easiest with pip, just run

```
pip install -r requirements.txt
```

in the `UncertainSCI/docs` folder and the relavent python dependencies, including Sphinx, will be installed.

Alternatively, the dependencies listed in `UncertainSCI/docs/requirements.txt` file could be installed seperately.

Please ask if you have any questions.

Building Documentation

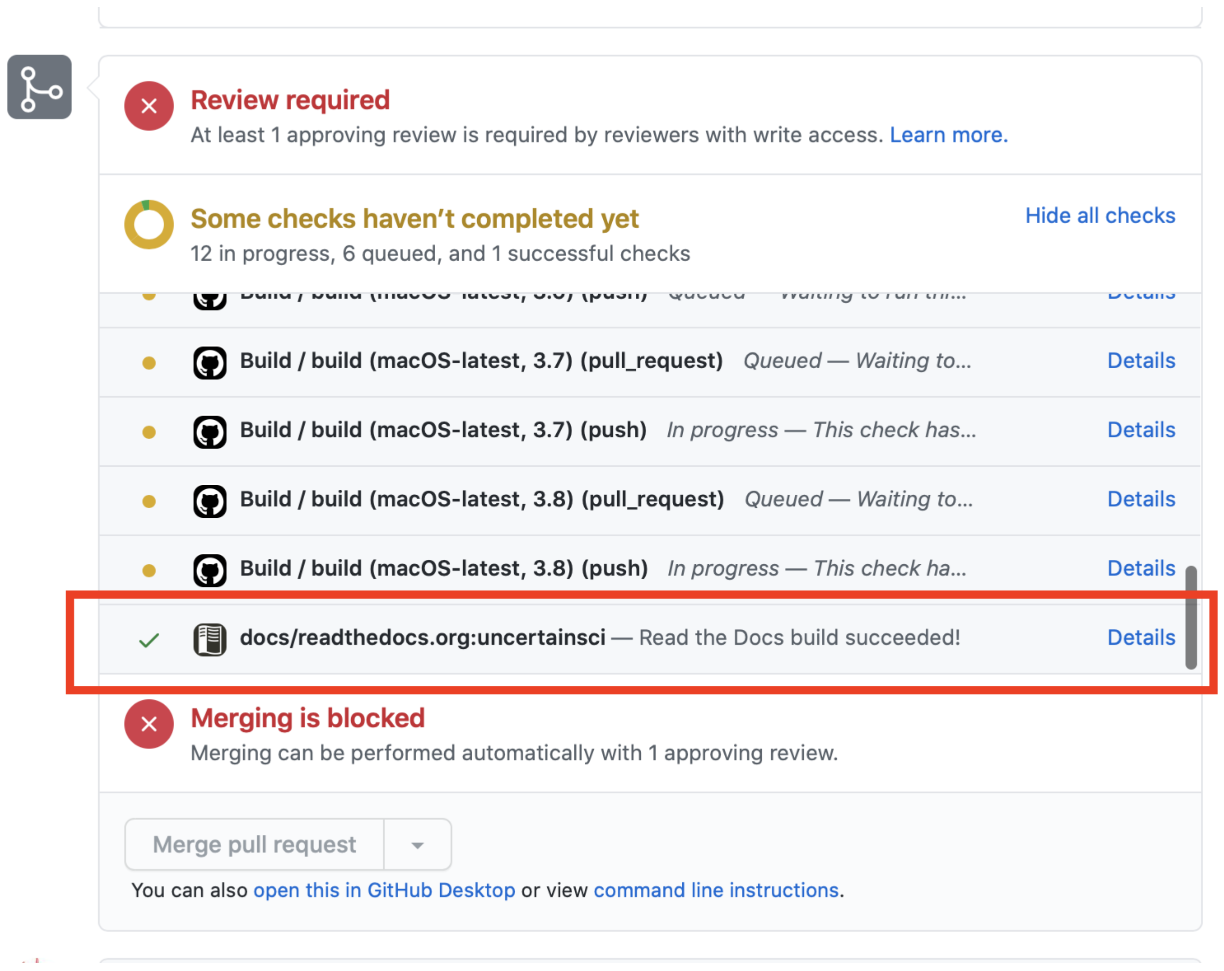
Once the relavent python packages are installed properly, such as with pip, building the documentation pages is as simple as running

```
make html
```

in the `UncertainSCI/docs` folder. A script will run and, assuming no errors, the html pages will be generated within the `UncertainSCI/docs/_build/` folder. Any of the html files can be tested by opening it in a browser.

Testing Online

In addition to building and testing the documentation pages locally, they can also be tested online with Read the Docs. Any branch of a github repo can be built one the Read the Docs page. Online builds can only be triggered deirectly on the readthe docs by those who have access. However, every pull request to the UncertainSCI repo will trigger an online build that can be viewed by anyone.


Review required
At least 1 approving review is required by reviewers with write access. Learn more.

Some checks haven't completed yet
12 in progress, 6 queued, and 1 successful checks

●	Build / build (macOS-latest, 3.7) (pull_request) *Queued — Waiting to...*	Details
●	Build / build (macOS-latest, 3.7) (push) *In progress — This check has...*	Details
●	Build / build (macOS-latest, 3.8) (pull_request) *Queued — Waiting to...*	Details
●	Build / build (macOS-latest, 3.8) (push) *In progress — This check ha...*	Details
✓	docs/readthedocs.org:uncertainsci — Read the Docs build succeeded!	Details

Merging is blocked
Merging can be performed automatically with 1 approving review.

You can also open this in GitHub Desktop or view command line instructions.

The documentation page can be viewd by clicking on the details link, shown in the image. Pull requests are required when submitting contributions to UncertainSCI.

Please ask if you have any questions.

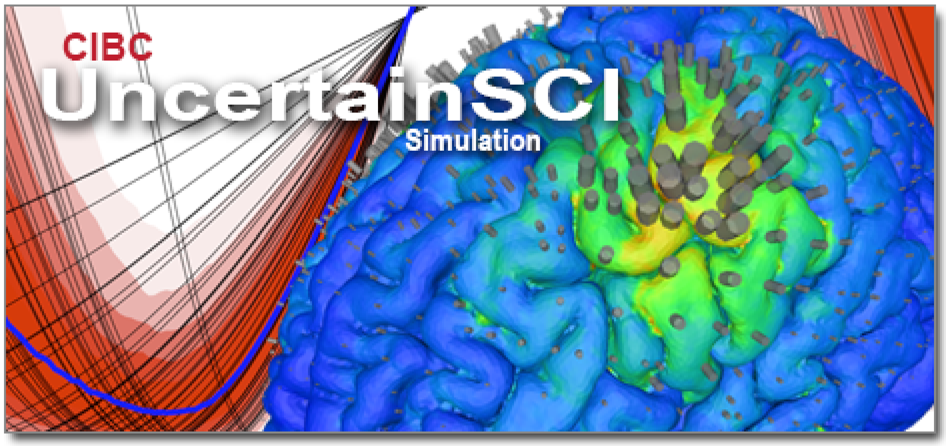
Adding Content

This chapter provides some examples of how to add some types of content that may be needed for a tutorial. For general Markdown info, see here and here.

Figures

Figures can be added fairly easily in Markdown, with a simple call to the location:

```
![Alt text](../_static/UncertainSCI.png "Title")
```



However, using a bit of html allows us to reference the figure easier:

```
<figure id="example">
<img src="../_static/UncertainSCI.png" alt="UncertainSCI example image">
<figcaption>Example for including an image in tutorial.</figcaption>
</figure>
```

And to reference (not working with Sphinx):

```
<a href="#example">reference the figure</a>
```

Math

Math equations can be used in Markdown using MathJax. Mathjax will convert LaTex format:

```
$$ \frac{\partial \rho}{\partial t} + \nabla \cdot \vec{j} = 0 \,. \label{eq:continuity}
↪$$
```

$$ \frac{\partial \rho}{\partial t} + \nabla \cdot \vec{j} = 0 ,. \label{eq:continuity} $$ It can also use MathJax specific tags:

```
\\[ x = {-b \pm \sqrt{b^2-4ac} \over 2a} \\]
```

\[x = {-b \pm \sqrt{b^2-4ac} \over 2a} \]

inline equations use the `\\(\mathbf{p}\\)` sytanx: \(\mathbf{p}\)

Citations

Since we are using Sphinx to build the documentation, we can use its citation manager, sphinxcontrib-bibtex. We will provide some exmples here, but for more information, refer to the sphinxcontrib-bibtex documentation.

Citations to include in the UncertainSCI docs can be included in the `UncertainSCI/docs/references.bib` file. For the keys, please use the convention: Initials of the contributor, colon, first three letters of the author (with apropriate capitalization), and the year. For example `JDT:Bur2020`. For multiple citations from the same author in the same year, lowercase letters can be appended to the key: `JDT:Bur2020a`.

After the reference has been added to `UncertainSCI/docs/references.bib`, the final step is to include the command in the appropriate place.

The sphinxcontrib-bibtex is built to run with rst in Sphinx. However, it can be used in markdown using the AutoStructify package. This will require using using an `eval_rst` block as follows:

~~~
```eval_rst
The whole paragraph will need to be in the eval_rst block :cite:p:`JDT:Bur2020`. For␣
↪multiple references: :cite:p:`JDT:Bur2020,gupta1983`
```
~~~

The whole paragraph will need to be in the eval_rst block [2]. For multiple references: [2, 6]

add a bibliography section

~~~
```eval_rst
.. bibliography:: ../references.bib
```
~~~

Snippets

Basic markdown `some snippet`

```
def function():
    return True
```

```
def function():
    return True
```

Links

Including links in Markdown is simple, just use <> or `[]()`. For example, an internal link for section *Adding Content* is :

```
[Adding Content](#adding-content)
```

When using internal links to sections, include the name of the section, all lower case and with `-` replacing spaces, and all special characters ommited. Linking to other pages in within the UncertainSCI documentation requires a relative path. Demos is:

```
[Demos](../tutorials/demos.html#demos)
```

Links to other websites can include the full URL. Using <> will show the URL, `[]()` will hide it with other text.

```
<https://www.markdownguide.org>
[Markdown](https://www.markdownguide.org)
```

https://www.markdownguide.org Markdown

Tables

Tables can be used with normal markdown syntax with the sphinx-markdown-tables package

```
| Syntax      | Description |
| ----------- | ----------- |
| Header      | Title       |
| Paragraph   | Text        |
```

Header 1	Header 2	Header 3
body row 1	column 2	column 3
body row 2	Cells may span columns.	

Referencing Sphinx

To link the UncertainSCI API generated using Sphinx, Use this syntax: `[text](../api_docs/pce.html#polynomial-chaos-expansions)`.

Content Guide

Try to be comprehensive, yet consise, and keep in mind the target audience for the tutorial. It is ok to write an advanced tutorial that builds on more basic knowledge, but please make this expectation clear and link to tutorials and materials that will help the reader develop the required understanding. Include code snippets, example scripts, screenshots, and videos as appropriate. Please use existing tutorials as try to match the style, flow, and level of detail they provide.

Supplemental Materials

Some things to consider including with the tutorial.

Example Scripts

Example scripts should be located in the `UncertainSCI/demos/` directory. Consider using one of the existing demos as a template and try to follow the coding standards outlined in the contribution guide.

Movies

Movies should be stored in a serperate place. We host most of ours on youtube or vimeo.

Youtube

Get the imbed link from the youtube video. This can be found by pressing the share button.

Vimeo

Get the imbed link from the vimeo video. This can be found by pressing the share button. There are more options if for some users who own the video. More info here.

With the embed link, just include the html into the markdown file.

```
<iframe title="vimeo-player" src="https://player.vimeo.com/video/279319572" width="640"␣
↪height="360" frameborder="0" allowfullscreen></iframe>
```

Datasets

Datasets should be located serperately, unless the size is small. Please ask if you have any questions.

Bibliography

1.5 API Documentation

1.5.1 Polynomial Spaces

The expressivity of a polynomial Chaos expansion (PCE, see *Polynomial Chaos Expansions*) can be set by specifying a particular polynomial space.

class `UncertainSCI.indexing.`**`TotalDegreeSet`**(*dim=1*, *order=0*)

    `get_indices`()

class `UncertainSCI.indexing.`**`HyperbolicCrossSet`**(*dim=1*, *order=0*)

    `get_indices`()

class `UncertainSCI.indexing.`**`LpSet`**(*dim=1*, *order=0*, *p=1*)

    `get_indices`()

1.5.2 Distributions

Introduction

This module is used to define probability distributions. For computing polynomial Chaos expansions (see *Polynomial Chaos Expansions*), these will be input distributions into a forward model defining the stochastic variation of model parameters.

UncertainSCI currently supports the following types of random variables:

- Beta distributions (See `BetaDistribution`)
- Exponential distributions (See `ExponentialDistribution`)
- Normal distributions (See `NormalDistribution`)
- Discrete uniform distributions (See `DiscreteUniformDistribution`)

Tensorizations within a distribution are possible across these families by instantiating the distribution appropriately. Tensorizations across distributions is also possible, but requires individual instantiation of each distribution, followed by a constructor call to the TensorialDistribution class. (See `TensorialDistribution`)

E.g., a three-dimensional random variable $Y = (Y_1, Y_2, Y_3)$ can have independent components, with the distribution of Y_1 normal, that of Y_2 beta, and that of Y_3 exponential.

The distributions are located in the *distributions.py* file.

class UncertainSCI.distributions.**BetaDistribution**(*alpha=None*, *beta=None*, *mean=None*, *stdev=None*, *dim=None*, *domain=None*, *bounds=None*)

Constructs a Beta distribution object; supports multivariate distributions through tensorization. In one dimension, beta distributions have support on the real interval [0,1], with probability density function,

$$w(y; \alpha, \beta) := \frac{y^{\alpha-1}(1-y)^{\beta-1}}{B(\alpha, \beta)}, \qquad y \in (0,1),$$

where α and β are positive real parameters that define the distribution, and B is the Beta function. Some special cases of note:

- $\alpha = \beta = 1$: the uniform distribution
- $\alpha = \beta = \frac{1}{2}$: the arcsine distribution

To generate this distribution on a general compact interval $[a, b]$, set the domain parameter below.

Instead of (α, β), a mean μ and standard deviation :math:`\sigma` may be set. In this case, (μ, σ) must correspond to valid (i.e., positive) values of (α, β) on the interval [0,1], or else an error is raised.

Finally, this class supports tensorization: multidimensional distributions corresopnding to independent one-dimensional marginal distributions are supported. In the case of identically distributed marginals, the *dim* parameter can be set to the appropriate dimension. In case of non-identical marginals, an array or iterable can be input for $\alpha, \beta, \mu, \sigma$.

Parameters

- **alpha** (*float or iterable of floats, optional*) – Shape parameter
- **1.** (*associated to left-hand boundary. Defaults to*) –
- **beta** (*float or iterable of floats, optional*) – Shape parameter
- **1.** –
- **mean** (*float or iterable of floats, optional*) – Mean of the distribution.

- `None.` (*the distribution. Defaults to*) –
- `stdev` (*float or iterable of floats, optional*) – Standard deviation of
- `None.` –
- `dim` (*int, optional*) – Dimension of the distribution. Defaults to None.
- `domain` (numpy.ndarray or similar, of size 2 x *dim*, optional) – Compact
- `None` (*hypercube that is the support of the distribution. Defaults to*) –

`dim`

Dimension of the distribution.

Type
int

`alpha`

Shape parameter(s) alpha.

Type
float or np.ndarray

`beta`

Shape parameter(s) beta.

Type
float or np.ndarray

`polys`

Type
`JacobiPolynomials` or list thereof

`MC_samples`(*M=100*)

Returns M Monte Carlo samples from the distribution.

`cov`()

Returns the (auto-)covariance matrix of the distribution.

`mean`()

Returns the mean of the distribution.

`meanstdev_to_alphabeta`(*mu*, *stdev*)

Returns alpha, beta given an input mean (mu) and standard deviation (stdev) for a Beta distribution on the interval [0, 1].

`pdf`(*x*)

Evaluates the probability density function (pdf) of the distribution at the input locations x.

`stdev`()

Returns the standard deviation of the distribution, if the distribution is one-dimensional. Raises an error if called for a multivariate distribution.

`class UncertainSCI.distributions.ExponentialDistribution`(*flag=True*, *lbd=None*, *loc=None*, *mean=None*, *stdev=None*, *dim=None*)

`MC_samples`(*M=100*)

Returns M Monte Carlo samples from the distribution.

`cov()`

`mean()`

`meanloc_to_lbd(mu, loc)`

Returns lbd given an input mean (mu) and location (loc) for a Exponential distribution

`pdf(x)`

`stdev()`

Returns the standard deviation of the distribution, if the distribution is one-dimensional. Raises an error if called for a multivariate distribution.

`class UncertainSCI.distributions.NormalDistribution(mean=None, cov=None, dim=None)`

`MC_samples(M=100)`

Returns M Monte Carlo samples from the distribution.

`cov()`

`mean()`

`pdf(x)`

Evaluates the probability density function (pdf) of the distribution at the input locations x.

`stdev()`

Returns the standard deviation of the distribution, if the distribution is one-dimensional. Raises an error if called for a multivariate distribution.

`class UncertainSCI.distributions.DiscreteUniformDistribution(n=None, domain=None, dim=None)`

`MC_samples(M=100)`

Returns M Monte Carlo samples from the distribution.

`cov()`

`mean()`

`pmf(x)`

Evaluates the probability mass function (pmf) of the distribution

`stdev()`

Returns the standard deviation of the distribution, if the distribution is one-dimensional. Raises an error if called for a multivariate distribution.

`class UncertainSCI.distributions.TensorialDistribution(distributions=None, dim=None)`

`MC_samples(M=100)`

Returns M Monte Carlo samples from the distribution

1.5.3 Polynomial Families

The computational building blocks of polynomial Chaos expansions (PCE, see *Polynomial Chaos Expansions*) are methods for univariate orthogonal polynomial families. Contains classes/methods for general univariate orthogonal polynomial families. - evaluation - gauss quadrature - ratio evaluations - linear/quadratic measure modifications

`class UncertainSCI.opoly1d.OrthogonalPolynomialBasis1D`(*recurrence=[]*, *probability_measure=True*)

`apply_jacobi_matrix`(*v*)

Premultiplies the input array by the Jacobi matrix of the appropriate size for the polynomial family. Applies the Jacobi matrix across the first dimension of v.

Parameters
`v` (*ndarray*) – Input vector or array

Returns
Jv – J*v, where J is the Jacobi matrix of size v.shape[0].

Return type
ndarray

`canonical_connection`(*N*)

Returns the N x N matrix C, where row n of C contains expansion coefficients for p_n in the monomial basis. I.e.,

p_n(x) = sum_{j=0}^{n} C[n,j] x**j,

for n = 0, …, N-1.

`canonical_connection_inverse`(*N*)

Returns the N x N matrix C, where row n of C contains expansion coefficients for x^n in the orthonormal basis . I.e.,

x^n = sum_{j=0}^{n} C[n,j] p_j(x)

for n = 0, …, N-1.

`christoffel_function`(*x*, *k*)

Computes the normalized (inverse) Christoffel function lambda, defined as

lambda**2 = k / sum(p**2, axi=1),

where p is a matrix containing evaluations of an orthonormal polynomial family up to degree k-1, defined by the recurrence coefficients ab.

`derivative_expansion`(*s*, *N*, *K=None*)

Computes the coefficients

$$\sigma_{n,k}^{(s)} = \left\langle p_n^{(s)}, p_k \right\rangle,$$

where $p_n^{(s)}$ is the $s`thderivativeofthedegree- : math : `k$ orthonormal polynomial p_k. These are, equivalently, expansion coefficients of $p_n^{(s)}$ in the basis $\{p_k\}_k$.

Parameters

- `s` – The integer order of the derivative. Must be non-negative.
- `N` – Computes coefficients for $n \leq N$
- `K` – Computes coefficients for $k \leq K$ (optional). If not given, is set to N.

Returns
(N+1) x (K+1) numpy array containing coefficients.

Return type
C

`eval`(*x*, *n*, *d=0*)

`gauss_quadrature`(*N*)

Computes the N-point Gauss quadrature rule associated to the recurrence coefficients ab.

`gauss_radau_quadrature`(*N*, *anchor=0.0*)

Computes the N-point Gauss quadrature rule associated to the polynomial family, with a node at the specified anchor.

`jacobi_matrix`(*N*)

Returns the N x N jacobi matrix associated to the polynomial family.

`jacobi_matrix_driver`(*N*)

Returns the N x N jacobi matrix associated to the input recurrence coefficients ab. (Requires ab.shape[0] >= N+1.)

`leading_coefficient`(*N*)

Returns the leading coefficients for the first N polynomial basis elements.

`qpoly1d_eval`(*x*, *n*, *d=0*)

Evalutes Christoffel-function normalized polynomials. These are given by

q_k(x) = p_k(x) / sqrt(sum_{j=0}^{n-1} p_j^2), k = 0, …, n-1

The output is a x.size x n array

`r_eval`(*x*, *n*, *d=0*)

Evalutes ratios of orthonormal polynomials. These are given by

r_n(x) = p_n(x) / p_{n-1}(x), n >= 1

The output is a x.size x n.size array.

`recurrence`(*N*)

Returns the first N+1 orthogonal polynomial recurrence pairs. The orthonormal polynomial family satisfies the recurrence

p_{-1}(x) = 0 p_0(x) = 1/ab[0,1]

x p_n(x) = ab[n+1,1] p_{n+1}(x) + ab[n+1,0] p_n(x) + ab[n,1] p_{n-1}(x)
(n >= 0)

The value ab[0,0] is ignored and never used.

Recurrence coefficients ab, once computed, are stored as an instance variable. On subsequent calls to this function, the stored values are returned if the instance variable already contains the desired coefficients. If the instance variable does not contain enough coefficients, then a call to recurrence_driver is performed to compute the desired coefficients, and the output is stored in the instance variable.

Parameters
N (*positive integer*) – Maximum polynomial degree for desired recurrence coefficients

Returns
ab – (N+1) x 2 array of recurrence coefficients.

Return type
ndarray

`recurrence_driver`(*N*)

`s_eval`(*x*, *n*)

The output is a x.size x (n+1) array.

s_n(x) = p_n(x) / sqrt(sum_{j=0}^{n-1} p_j^2(x)), n >= 0

s_0(x) = p_0(x)

s_1(x) = 1 / b_1 * (x - a_1)

s_2(x) = 1 / (b_2 * sqrt(1+s_1^2)) * ((x - a_2)*s_1 - b_1)

Need {a_k, b_k} k up to n

`tuple_product`(*N*, *alpha*)

Computes integrals of polynomial products. Returns an N x N matrix C with entries

C[n,m] = < p_n p_m, p_alpha >,

where alpha is a vector of integers and p_alpha is defined

p_alpha = prod_{j=1}^{alpha.size} p_[alpha[j]](x),

The notation <., .> denotes the inner product under which the polynomial family is orthogonal.

Parameters

- `N` (*integer*) – Size of matrix to return
- `alpha` (*ndarray (1d)*) – Multi-index defining a polynomial product

Returns
C – Output N x N matrix containing integral values

Return type
ndarray

`tuple_product_generator`(*IC*, *ab=None*)

Helper function that increments indices for a polynomial product expansion.

IC is a vector with entries

IC[j] = < p_j, p_alpha >, j in range(N),

where N = IC.size. The notation < ., .> is the inner product under which the polynomial family is orthonormal. alpha is a multi-index of arbitrary shape with a polynomial defined by

p_alpha = prod_{j in range(alpha.size)} p_{alpha[j]}(x).

The value of alpha is not needed by this function.

This function returns an N x N matrix C with entries

C[n,j] = < p_n p_j, p_alpha >, j, n in range(N)

Parameters

- `IC` (*vector (1d array)*) – Values of input inner products
- `ab` (*ndarray, optional*) – Recurrence coefficients

Returns
C – Output coefficient expansion vector for beta

Return type
ndarray

Contains routines that specialize opoly1d things for classical orthogonal polynomial families - jacobi polys - hermite poly - laguerre polys

`class UncertainSCI.families.`**`JacobiPolynomials`**`(`*alpha=0.0*, *beta=0.0*, *domain=[-1.0, 1.0]*, ***options*`)`

Jacobi Polynomial family.

`eval_1d``(`*x*, *n*`)`

Evaluates univariate orthonormal polynomials given their three-term recurrence coefficients ab

`eval_nd``(`*x*, *lambdas*`)`

Evaluates tensorial orthonormal polynomials associated with the univariate recurrence coefficients ab

`fidistinv``(`*u*, *n*`)`

`fidistinv_jacobi_setup``(`*n*, *data*`)`

`idist``(`*x*, *n*, *M=50*`)`

Computes the order-n induced distribution at the locations x using M=10 quadrature nodes.

`idist_medapprox``(`*n*`)`

Computes an approximation to the median of the degree-n induced distribution.

`idistinv``(`*u*, *n*`)`

`recurrence_driver``(`*N*`)`

`class UncertainSCI.families.`**`HermitePolynomials`**`(`*alpha=2*, *rho=0.0*, ***options*`)`

`idist``(`*x*, *n*`)`

`idistinv``(`*u*, *n*`)`

`recurrence_driver``(`*N*`)`

`class UncertainSCI.families.`**`LaguerrePolynomials`**`(`*alpha=1.0*, *rho=0.0*, ***options*`)`

`idist``(`*x*, *n*`)`

`idist_medapprox``(`*n*`)`

`idistinv``(`*u*, *n*`)`

`recurrence_driver``(`*N*`)`

`class UncertainSCI.families.`**`DiscreteChebyshevPolynomials`**`(`*M=2*, *domain=[0.0, 1.0]*`)`

Class for polynomials orthonormal on [0,1] with respect to an M-point discrete uniform measure with support equidistributed on the interval.

`eval``(`*x*, *n*, ***options*`)`

`fidistinv``(`*u*, *n*`)`

Fast routine for idistinv. (In this case, the “slow” routine is already very fast, so this is just an alias for idistinv.)

`idist``(`*x*, *n*, *nugget=False*`)`

Evalutes the order-n induced distribution at the locations x.

Optionally, add a nugget to ensure correct computation on the support points.

idistinv(u, n)

Computes the inverse order-n induced distribution at the locations u.

recurrence_driver(N)

1.5.4 Polynomial Chaos Expansions

Introduction

In their most basic use, polynomial chaos expansions (PCE) create an emulator for a quantity that depends on a random variable. For simplicity, we will assume that this random variable is finite-dimensional. Let ξ denote a d-dimensional random variable. For some function $f : \mathbb{R}^d \to \mathbb{R}$, the PCE approach performs the approximation,

$$f(\xi) \approx f_N(\xi) := \sum_{n=1}^{N} \hat{f}_n \phi_n(\xi), \tag{1.1}$$

where $\{\phi_n\}_{n=1}^{\infty}$ are polynomial functions of the random variable ξ, and $\{hatf_n\}_{n=1}^{\infty}$ are coefficients. If such an emulator can be constructed, then statistics are evaluated as statistics of the emulator.

For example, the (approximation to the) mean is

$$\mathbb{E} f(\xi) \approx \sum_{n=1}^{N} \hat{f}_n \mathbb{E}[\phi_n(\xi)], \tag{1.2}$$

which can be efficiently evaluated by manipulation of the coefficients $\hat{f}_n$. The terms $\mathbb{E}[\phi_n(\xi)]$ can be evaluated exactly using properties of the ϕ_n polynomials.

All of the above extends to the case when the function f depends on other variables, such as space x or time t. For example, if $f = f(x, t, \xi)$, then the PCE approach becomes

$$f(x, t, \xi) \approx f_N(x, t, \xi) := \sum_{n=1}^{N} \hat{f}_n(x, t) \phi_n(\xi),$$

so that the coefficients depend on (x, t). Then the space- and time-varying expectation can be evaluated in a manner similar to (1.2).

For PCE approaches, most of the computation involves computing the coefficients $\{\hat{f}_n\}_{n=1}^{\infty}$. Non-intrusive PCE strategies accomplish this by computing values of f on a specific sampling grid or experimental design in stochastic space: $\{\xi_m\}_{m=1}^{M}$. The procedures used in UncertainSCI typically require the number of samples M to scale with the degrees of freedom in the emulator N.

In order to utilize the PCE approaches in UncertainSCI, two items must be provided:

1. The distribution of the random variable ξ. See *Distributions* for how to generate this distribution.
2. The type of polynomial functions in (1.1). This amounts to defining a particular polynomial subspace. See *Polynomial Spaces* for how to generate this subspace.

PolynomialChaosExpansion

`class UncertainSCI.pce.PolynomialChaosExpansion`(*index_set=None*, *distribution=None*, *order=None*, *plabels=None*, *sampling='greedy-induced'*, *training='wlsq'*, ***kwargs*)

Base polynomial chaos expansion class.

Provides interface to construct and manipulate polynomial chaos expansions.

Attributes:

coefficients: A numpy array of polynomial chaos expansion coefficients. indices: A MultiIndexSet instance specifying the polynomial

approximation space.

distribution: A ProbabilityDistribution instance indicating the
distribution of the random variable.

samples: The experimental or sample design in stochastic space.

`adapt_expressivity`(*max_new_samples=10*, ***chase_bulk_options*)

Adapts the PCE approximation by increasing expressivity. (Intended to combat residual error.)

`adapt_robustness`(*max_new_samples=10*, *verbosity=1*)

Adapts the PCE approximation by increasing robustness. (Intended to combat cross-validation error.)

`assert_pce_built`()

`augment_samples_idist`(*K*, *weights=None*, *fast_sampler=True*)

Augments random samples from induced distribution. Typically done via an adaptive refinement procedure. As such some inputs can be given to customize how the samples are drawn in the context of adaptivity:

K: how many samples to add (required) weights: a discrete probability distribution on

self.index_set.get_indices() that describes how the induced distrubtion is sampled. Default is uniform.

`build`(*model=None*, *model_output=None*, ***options*)

Builds PCE from sampling and approximation settings.

Parameters

- `model` – A pointer to a function with the syntax xi —> model(xi), which returns a vector corresponding to the model evaluated at the stochastic parameter value xi. The input xi to the model function should be a vector of size self.dim, and the output should be a 1D numpy array. If model_output is None, this is required. If model_output is given, this is ignored.
- `model_output` – A numpy.ndarray corresponding to the output of the model at the sample locations specified by self.samples. This is required if the input model is None.

Return type
None

`build_pce_wlsq`()

Performs a (weighted) least squares PCE surrogate using saved samples and model output.

`chase_bulk`(*delta=0.5*, *max_new_samples=None*, *max_new_indices=None*, *add_rule=None*, *mult_rule=None*, *verbosity=1*)

Performs adaptive bulk chasing, which (i) adds the most "important" indices to the polynomial index set, (ii) takes more samples, (iii) updates the PCE approximation, including statistics and error metrics.

Parameters

- `max_new_samples` (*int*) – Maximum number of new samples to add. Defaults to None.
- `max_new_indices` (*int*) – Maximum number of new PCE indices to add. Defaults to None.
- `add_rule` (*int*) – Specifies number of samples added as a function of number of added indices. Nsamples = Nindices + add_rule. Defaults to None.
- `mult_rule` (*float*) – Specifies number of samples added as a function of number of added indices. Nsamples = int(Nindices * add_rule). Defaults to None.

`check_distribution`()

`check_indices`()

`check_sampling`()

Determines if a valid sampling type has been specified.

`check_training`()

Determines if a valid training type has been specified.

`christoffel_weights`()

Generates sample weights associated to Christoffel preconditioning.

`eval`(*p*, *components=None*)

Evaluates the PCE at particular parameter locations.

Parameters

- `p` – An array (satisfying p.shape[1]==self.dim) containing a set of parameter points at which to evaluate the PCE prediction.
- `components` – An array of non-negative integers specifying which indices in the model output to compute. Other indices are ignored. If given as None (default), then all components are computed.

Returns

An array containing evaluations (predictions) from the PCE emulator. If the input components is None, this array is of size (self.p.shape[0] x self.coefficients.shape[1]). Otherwise, the second dimension is of size components.size.

Return type

numpy.ndarray

`generate_samples`(***kwargs*)

Generates sample/experimental design for use in PCE construction.

Parameters

`new_samples` (*array-like, optional*) – Specifies samples that must be part of the ensemble.

global_derivative_sensitivity(*dim_list*)

Computes global derivative-based sensitivity indices. For a PCE with respect to a d-dimensional random variable Z, then this senstivity index along dimension i is defined as

$$S_i \colon = E\left[p(Z)\right] = \int p(z)\omega(z)dz,$$

where $E[\cdot]$ it expectation operator, p is the PCE emulator, and ω is the probability density function for the random variable Z.

These sensitivity indices measure the average rate-of-change of the PCE response with respect to dimension i.

Parameters
`dim_lists` – A list-type iterable with D entries, containing dimensional indices in 0-based indexing. All entries must be between 0 and self.distribution.dim.

Returns

DxK array, where each row corresponds to the sensitivity index
S_i across all K features of the PCE model.

Return type
S

global_sensitivity(*dim_lists=None*, *interaction_orders=None*, *vartol=1e-16*)

Computes global sensitivity associated to dimensional indices dim_lists from PCE coefficients.

dim_lists should be a list of index lists. The global sensitivity for each index list is returned.

interaction_orders (list): Computes sensitivities corresponding to variable interactions for the specified orders. E.g., order 2 implies binary interactions, 3 is ternary interactions, [2,3] computes both of these orders.

The output is len(dim_lists) x self.coefficients.shape[1]

identify_bulk(*delta=0.5*)

Performs (adaptive) bulk chasing for refining polynomial spaces. Returns the indices associated with a delta-bulk of a OMP-type indicator.

integration_weights()

Generates sample weights associated to integration.

map_to_model_space(*p*)

Maps parameter values from standard space to model space.

Parameters
`p` (*array-like*) – Samples in standard space.

Returns
Samples in model space

Return type
q (numpy.ndarray)

map_to_standard_space(*q*)

Maps parameter values from model space to standard space.

Parameters
`q` (*array-like*) – Samples in model space.

Returns
Samples in standard space

Return type
p (numpy.ndarray)

`mean()`

Returns PCE mean.

Returns

A vector containing the PCE mean, of size equal to the size
of the vector of the model output.

Return type
numpy.ndarray

`pce_eval`(*p*, *components=None*)

Evaluates the PCE at particular parameter locations.

Parameters

- `p` – An array (satisfying p.shape[1]==self.dim) containing a set of parameter points at which to evaluate the PCE prediction.
- `components` – An array of non-negative integers specifying which indices in the model output to compute. Other indices are ignored. If given as None (default), then all components are computed.

Returns
An array containing evaluations (predictions) from the PCE emulator. If the input components is None, this array is of size (self.p.shape[0] x self.coefficients.shape[1]). Otherwise, the second dimension is of size components.size.

Return type
numpy.ndarray

`quantile`(*q*, *M=100*)

Computes q-quantiles using M-point Monte Carlo sampling.

`set_distribution`(*distribution*)

Sets type of probability distribution of random variable.

Parameters
`distribution` – A ProbabilityDistribution instance specifying the distribution of the random variable.

Return type
None

`set_indices`(*index_set*)

Sets multi-index set for polynomial approximation.

Parameters
`indices` – A MultiIndexSet instance specifying the polynomial approximation space.

Return type
None

`set_samples`(*samples*)

`set_weights()`

Sets weights based on assigned samples.

`stdev()`

Returns PCE standard deviation

Returns

A vector containing the PCE standard deviation, of size
equal to the size of the vector of the model output.

Return type
numpy.ndarray

`total_sensitivity`(*dim_indices=None*, *vartol=1e-16*)

Computes total sensitivity associated to dimensions dim_indices from PCE coefficients. dim_indices should be a list-type containing dimension indices.

The output is len(js) x self.coefficients.shape[1]

CHAPTER

TWO

CONTRIBUTORS

Jake Bergquist, Dana Brooks, Zexin Liu, Rob MacLeod, Akil Narayan, Sumientra Rampersad, Lindsay Rupp, Jess Tate, Dan White

CHAPTER

THREE

ACKNOWLEDGEMENTS

This project was supported by grants from the National Institute of Biomedical Imaging and Bioengineering (U24EB029012) from the National Institutes of Health.

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FIVE

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Akil Narayan, Zexin Liu, Jake Bergquist, Chantel Charlebois, Sumientra Rampersad, Lindsay Rupp, Dana Brooks, Dan White, Jess Tate, and Rob S MacLeod. UncertainSCI: uncertainty quantification for computational models in biomedicine and bioengineering. *Available at SSRN 4049696*, 2022.

[2] Kyle M. Burk, Akil Narayan, and Joseph A. Orr. Efficient sampling for polynomial chaos-based uncertainty quantification and sensitivity analysis using weighted approximate fekete points. *International Journal for Numerical Methods in Biomedical Engineering*, 36(11):e3395, 2020. doi:https://doi.org/10.1002/cnm.3395.

[3] Akil Narayan. Computation of induced orthogonal polynomial distributions. *Electronic Transactions on Numerical Analysis*, 50:71–97, 2018. arXiv:1704.08465 [math]. URL: https://epub.oeaw.ac.at?arp=0x003a184e, doi:10.1553/etna_vol50s71.

[4] L. Guo, A. Narayan, L. Yan, and T. Zhou. Weighted Approximate Fekete Points: Sampling for Least-Squares Polynomial Approximation. *SIAM Journal on Scientific Computing*, 40(1):A366–A387, 2018. arXiv:1708.01296 [math.NA]. URL: http://epubs.siam.org/doi/abs/10.1137/17M1140960, doi:10.1137/17M1140960.

[5] Albert Cohen and Giovanni Migliorati. Optimal weighted least-squares methods. *SMAI Journal of Computational Mathematics*, 3:181–203, 2017. arxiv:1608.00512 [math.NA]. doi:10.5802/smai-jcm.24.

[6] S.K. Gupta and W.V. Harper. Sensitivity/Uncertainty Analysis of a Borehole Scenario Comparing Latin Hypercube Sampling and Deterministic Sensitivity Approaches. Technical Report BMI/ONWI-516, Battelle Memorial Institute, Office of Nuclear Waste Isolation, 1983.

[7] Jake Bergquist, Brian Zenger, Lindsay Rupp, Akil Narayan, Jess Tate, and Rob MacLeod. Uncertainty quantification in simulations of myocardial ischemia. In *Computing in Cardiology*, volume 48. September 2021.

[8] Lindsay C Rupp, Zexin Liu, Jake A Bergquist, Sumientra Rampersad, Dan White, Jess D Tate, Dana H. Brooks, Akil Narayan, and Rob S. MacLeod. Using uncertainSCI to quantify uncertainty in cardiac simulations. In *Computing in Cardiology*, volume 47. September 2020.

[9] Lindsay C Rupp, Jake A Bergquist, Brian Zenger, Karli Gillette, Akil Narayan, Jess Tate, Gernot Plank, and Rob S. MacLeod. The role of myocardial fiber direction in epicardial activation patterns via uncertainty quantification. In *Computing in Cardiology*, volume 48. September 2021.

[10] Jess D. Tate, Wilson W. Good, Nejib Zemzemi, Machteld Boonstra, Peter van Dam, Dana H. Brooks, Akil Narayan, and Rob S. MacLeod. Uncertainty quantification of the effects of segmentation variability in ECGI. In *Functional Imaging and Modeling of the Heart*, pages 515–522. Springer-Cham, Palo Alto, USA, 2021. doi:https://doi.org/10.1007/978-3-030-78710-3_49.

[11] Jess Tate, Sumientra Rampersad, Chantel Charlebois, Zexin Liu, Jake Bergquist, Dan White, Lindsay Rupp, Dana Brooks, Akil Narayan, and Rob MacLeod. Uncertainty quantification in brain stimulation using uncertainSCI. *Brain Stimulation: Basic, Translational, and Clinical Research in Neuromodulation*, 14(6):1659–1660, January 2021. URL: https://doi.org/10.1016/j.brs.2021.10.226, doi:10.1016/j.brs.2021.10.226.

[12] Jess D Tate, Nejib Zemzemi, Shireen Elhabian, Beáta Ondru\u sová, Machteld Boonstra, Peter van Dam, Akil Narayan, Dana H Brooks, and Rob S MacLeod. Segmentation uncertainty quantification in cardiac propagation models. In *2022 Computing in Cardiology (CinC)*, volume 498, 1–4. 2022. doi:10.22489/CinC.2022.419.

[1] Akil Narayan, Zexin Liu, Jake Bergquist, Chantel Charlebois, Sumientra Rampersad, Lindsay Rupp, Dana Brooks, Dan White, Jess Tate, and Rob S MacLeod. UncertainSCI: uncertainty quantification for computational models in biomedicine and bioengineering. *Available at SSRN 4049696*, 2022.

[2] Kyle M. Burk, Akil Narayan, and Joseph A. Orr. Efficient sampling for polynomial chaos-based uncertainty quantification and sensitivity analysis using weighted approximate fekete points. *International Journal for Numerical Methods in Biomedical Engineering*, 36(11):e3395, 2020. doi:https://doi.org/10.1002/cnm.3395.

[3] Akil Narayan. Computation of induced orthogonal polynomial distributions. *Electronic Transactions on Numerical Analysis*, 50:71–97, 2018. arXiv:1704.08465 [math]. URL: https://epub.oeaw.ac.at?arp=0x003a184e, doi:10.1553/etna_vol50s71.

[4] L. Guo, A. Narayan, L. Yan, and T. Zhou. Weighted Approximate Fekete Points: Sampling for Least-Squares Polynomial Approximation. *SIAM Journal on Scientific Computing*, 40(1):A366–A387, 2018. arXiv:1708.01296 [math.NA]. URL: http://epubs.siam.org/doi/abs/10.1137/17M1140960, doi:10.1137/17M1140960.

[5] Albert Cohen and Giovanni Migliorati. Optimal weighted least-squares methods. *SMAI Journal of Computational Mathematics*, 3:181–203, 2017. arxiv:1608.00512 [math.NA]. doi:10.5802/smai-jcm.24.

[6] S.K. Gupta and W.V. Harper. Sensitivity/Uncertainty Analysis of a Borehole Scenario Comparing Latin Hypercube Sampling and Deterministic Sensitivity Approaches. Technical Report BMI/ONWI-516, Battelle Memorial Institute, Office of Nuclear Waste Isolation, 1983.

[7] Jake Bergquist, Brian Zenger, Lindsay Rupp, Akil Narayan, Jess Tate, and Rob MacLeod. Uncertainty quantification in simulations of myocardial ischemia. In *Computing in Cardiology*, volume 48. September 2021.

[8] Lindsay C Rupp, Zexin Liu, Jake A Bergquist, Sumientra Rampersad, Dan White, Jess D Tate, Dana H. Brooks, Akil Narayan, and Rob S. MacLeod. Using uncertainSCI to quantify uncertainty in cardiac simulations. In *Computing in Cardiology*, volume 47. September 2020.

[9] Lindsay C Rupp, Jake A Bergquist, Brian Zenger, Karli Gillette, Akil Narayan, Jess Tate, Gernot Plank, and Rob S. MacLeod. The role of myocardial fiber direction in epicardial activation patterns via uncertainty quantification. In *Computing in Cardiology*, volume 48. September 2021.

[10] Jess D. Tate, Wilson W. Good, Nejib Zemzemi, Machteld Boonstra, Peter van Dam, Dana H. Brooks, Akil Narayan, and Rob S. MacLeod. Uncertainty quantification of the effects of segmentation variability in ECGI. In *Functional Imaging and Modeling of the Heart*, pages 515–522. Springer-Cham, Palo Alto, USA, 2021. doi:https://doi.org/10.1007/978-3-030-78710-3_49.

[11] Jess Tate, Sumientra Rampersad, Chantel Charlebois, Zexin Liu, Jake Bergquist, Dan White, Lindsay Rupp, Dana Brooks, Akil Narayan, and Rob MacLeod. Uncertainty quantification in brain stimulation using uncertainSCI. *Brain Stimulation: Basic, Translational, and Clinical Research in Neuromodulation*, 14(6):1659–1660, January 2021. URL: https://doi.org/10.1016/j.brs.2021.10.226, doi:10.1016/j.brs.2021.10.226.

[12] Jess D Tate, Nejib Zemzemi, Shireen Elhabian, Beáta Ondru\u sová, Machteld Boonstra, Peter van Dam, Akil Narayan, Dana H Brooks, and Rob S MacLeod. Segmentation uncertainty quantification in cardiac propagation models. In *2022 Computing in Cardiology (CinC)*, volume 498, 1–4. 2022. doi:10.22489/CinC.2022.419.

PYTHON MODULE INDEX

INDEX