SANDIA REPORT

SAND2018-####
Unlimited Release
Printed Month, YYYY

UQTk Version 3.0.4 User Manual

Khachik Sargsyan, Cosmin Safta, Kenny Chowdhary, Prashant Rai, Xun Huan,
Sarah Castorena, Sarah de Bord, Xiaoshu Zeng, Bert Debusschere

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2018-####
Unlimited Release
Printed Month, YYYY

UQTk Version 3.0.4 User Manual

Khachik Sargsyan, Cosmin Safta, Kenny Chowdhary, Prashant Rai, Xun Huan,
Sarah Castorena, Sarah de Bord, Xiaoshu Zeng, Bert Debusschere

Abstract

The UQ Toolkit (UQTk) is a collection of libraries and tools for the quantification of uncertainty in numerical model predictions. Version 3.0.4 offers intrusive and non-intrusive methods for propagating input uncertainties through computational models, tools for sensitivity analysis, methods for sparse surrogate construction, and Bayesian inference tools for inferring parameters from experimental data. This manual discusses the download and installation process for UQTk, provides pointers to the UQ methods used in the toolkit, and describes some of the examples provided with the toolkit.

Contents

Chapter 1

Overview

The UQ Toolkit (UQTk) is a collection of libraries and tools for the quantification of uncertainty in numerical model predictions. In general, uncertainty quantification (UQ) pertains to all aspects that affect the predictive fidelity of a numerical simulation, from the uncertainty in the experimental data that was used to inform the parameters of a chosen model, and the propagation of uncertain parameters and boundary conditions through that model, to the choice of the model itself.

In particular, UQTk provides implementations of many probabilistic approaches for UQ in this general context. Version 3.0.4 offers intrusive and non-intrusive methods for propagating input uncertainties through computational models, tools for sensitivity analysis, methods for sparse surrogate construction, and Bayesian inference tools for inferring parameters from experimental data.

The main objective of UQTk is to make these methods available to the broader scientific community for the purposes of algorithmic development in UQ or educational use. The most direct way to use the libraries is to link to them directly from C++ programs. Alternatively, in the examples section, many scripts for common UQ operations are provided, which can be modified to fit the users' purposes using existing numerical simulation codes as a black-box.

The next chapter in this manual discusses the download and installation process for UQTk, followed by some pointers to the UQ methods used in the toolkit, and a description of some of the examples provided with the toolkit.

Chapter 2

Download and Installation

Requirements

The core UQTk libraries are written in C++, with some dependencies on FORTRAN numerical libraries. As such, to use UQTk, a compatible C++ and FORTRAN compiler will be needed. UQTk is installed and built most naturally on a Unix-like platform, and has been tested on Mac OS X and Linux. Installation and use on Windows machines under Cygwin is possible, but has not been tested extensively.

Many of the examples rely on Python, including NumPy, SciPy, and matplotlib packages for postprocessing and graphing. As such, Python version 2.7.x with compatible NumPy, SciPy, and matplotlib are recommended. Further the use of XML for input files requires the Expat XML parser library to be installed on your system. Note, if you will be linking the core UQTk libraries directly to your own codes, and do not plan on using the UQTk examples, then those additional dependencies are not required.

Download

The most recent version of UQTk, currently 3.0.4, can be downloaded from the following location:
`http://www.sandia.gov/UQToolkit`

After download, extract the tar file into the directory where you want to keep the UQTk source directory.

```
% tar -xzvf UQTk_v3.0.tgz
```

Make sure to replace the name of the tar file in this command with the name of the most recent tar file you just downloaded, as the file name depends on the version of the toolkit.

Directory Structure

After extraction, there will be a new directory `UQTk_v3.0` (version number may be different). Inside this top level directory are the following directories:

`config`	**Configuration files**
`cpp`	**C++ source code**
`app`	C++ apps
`lib`	C++ libraries
`tests`	Tests for C++ libraries
`dep`	**External dependencies**
`ann`	Approximate Nearest Neighbors library
`blas`	Netlib's BLAS library (linear algebra)
`cvode-2.7.0`	SUNDIALS' CVODE library (ODE solvers)
`dsfmt`	dsfmt library (random number generators)
`figtree`	Fast Improved Gauss Transform library
`lapack`	Netlib's LAPACK library (linear algebra)
`lbfgs`	lbfgs library (optimization)
`slatec`	Netlib's SLATEC library (general purpose math)
`doc`	**Documentation**
`examples`	**Examples with C++ libraries and apps**
`fwd_prop`	forward propagation with a heat transfer example
`kle_ex1`	Karhunen-Loeve expansion example
`line_infer`	calibrate parameters of a linear model
`muq`	interface between MUQ and UQTk
`num_integ`	quadrature and Monte Carlo integrations
`ops`	operations with Polynomial Chaos expansions
`pce_bcs`	construct sparse Polynomial Chaos expansions
`polynomial`	polynomial model fit with MCMC
`sensMC`	Monte-Carlo based sensitivity index computation
`surf_rxn`	surface reaction example for forward and inverse UQ
`uqpc`	construct Polynomial Chaos surrogates for multiple outputs/functions
`PyUQTk`	**Python scripts and interface to C++ libraries**
`array`	interface to array class
`bcs`	interface to Bayesian compressive sensing library
`dfi`	interface to data-free inference class
`inference`	Python Markov Chain Monte Carlo (MCMC) scripts
`kle`	interface to Karhunen-Loeve expansion class
`mcmc`	interface to MCMC class
`pce`	interface to Polynomial Chaos expansion class
`plotting`	Python plotting scripts
`pytests`	
`quad`	interface to Quad class

`sens`	Python global sensitivity analysis scripts
`tools`	interface to UQTk tools
`utils`	interface to UQTk utils

External Software and Libraries

The following software and libraries are required to compile UQTK

1. **C++/Fortran compilers**. Please note that C++ and Fortran compilers need to be compatible with each other. Please see the table below for a list of platforms and compilers we have tested so far. For OS X these compilers were installed either using MacPorts, or Homebrew, or directly built from source code.

Platform	**Compiler**
OS X 10.9	GNU 4.8, 4.9, 5.2, Intel 14.0.3
OS X 10.10	GNU 5.2, 5.3
OS X 10.11	GNU 5.2–4, 6.1, 6.2, Intel 14.0.3, OpenMPI 1.10
Linux x86_64 (Red Hat)	GNU 4.8
Linux ia-64 (Suse)	Intel 10.1.021

2. **CMake**. We switched to a CMake-based build/install configuration in version 3.0. The configuration files require a CMake version 2.6.x or higher.

3. **Expat library**. The Expat XML Parser is installed together with other XCode tools on OS X. It is also fairly common on Linux systems, with installation scripts available for several platforms. Alternatively this library can be downloaded from `http://expat.sourceforge.net`

PyUQTk

The following additional software and libraries are not required to compile UQTK, but are necessary for the full Python interface to UQTk called PyUQTk.

1. **Python, NumPy, SciPy, and Matplotlib**. We have successfully compiled PyUQTk with Python 2.6.x and Python 2.7.x, NumPy 1.8.1, SciPy 0.14.0, and Matplotlib 1.4.2. Note that we have not tried using Python 3 or higher. Note that it is important that all of these packages be compatible with each other. Sometimes, your OS may come with a default version of Python but not SciPy or NumPy. When adding those packages afterwards, it can be hard to get them to all be compatible with each other. To avoid issues, it is recommended to install Python, NumPy, and SciPy all from the same package manager (*e.g.* get them all through MacPorts or Homebrew on OS X).

2. **SWIG**. PyUQTk has been tested with SWIG 3.0.2.

Installation

We define the following keywords to simplify build and install descriptions in this section.

- *sourcedir* - directory containing UQTk source files, i.e. the top level directory mentioned in Section 2.
- *builddir* - directory where UQTk library and its dependencies will be built. This directory should not be the same as *sourcedir*.
- *installdir* - directory where UQTk libraries are installed and header and script files are copied

To generate the build structure, compile, test, and install UQTk:

(1) > mkdir *builddir*; cd *builddir*
(2) > cmake <flags> *sourcedir*
(3) > make
(4) > ctest
(5) > make install

Configuration flags

A (partial) list of configuration flags that can be set at step (2) above is provided below:

- **CMAKE_INSTALL_PREFIX** : set *installdir*.
- **CMAKE_C_COMPILER** : C compiler
- **CMAKE_CXX_COMPILER** : C++ compiler
- **CMAKE_Fortran_COMPILER** : Fortran compiler
- **IntelLibPath**: For Intel compilers: path to libraries if different than default system paths
- **PyUQTk**: If ON, then build PyUQTk's Python to C++ interface. Default: OFF

Several pre-set config files are available in the "*sourcedir*/config" directory. Some of these shell scripts also accept arguments, e.g. config-teton.sh, to switch between several configurations. Type, for example "config-teton.sh –help" to obtain a list of options. For a basic setup using default system settings for GNU compilers, see "config-gcc-base.sh". The user is encouraged to copy of one these script files and edit to match the desired configuration. Then, step no. 2 above (cmake <flags> *sourcedir*) should be replaced by a command executing a particular shell script from the command line, *e.g.*

(2) > `../UQTk_v3.0.1/config/config-gcc-base.sh`

In this example, the configuration script is executed from the build directory, while it is assumed that the configuration script still sits in the configuration directory, in this case version 3.0.1, of the UQTk source code tree.

If all goes well, there should be no errors. Two log files in the "config" directory contain the output for Steps (2) and (3) above, for compilation and installation on OS X 10.9.5 using GNU 4.8.3 compilers:

(2) > `../UQTk/config/config-teton.sh -c gnu -p ON >& cmake-mac-gnu.log`
(3) > `make >& make-gnu.log ; make install >>& make-gnu.log`

After compilation ends, the *installdir* will be contain the following sub-directories:

`PyUQTk`	Python scripts and, if PyUQTk=ON, interface to C++ classes
`bin`	app's binaries
`cpp`	tests for C++ libraries
`examples`	examples on using UQTk
`include`	UQTk header files
`lib`	UQTk libraries, including for external dependencies

To use the UQTk libraries, your program should link in the libraries in `installdir/lib` and add `installdir/include/uqtk` and `installdir/include/dep` directories to the compiler include path. The apps are standalone programs that perform UQ operations, such as response surface construction, or sampling from random variables. For more details, see the Examples section.

Installation example

In this section, we will take the user through the installation of UQTk and PyUQTk on a Mac OSX 10.11 system with the GNU compilers. The following example uses GNU 6.1 installed under `/opt/local/gcc61`. For the compilation of PyUQTk, we are using Python version 2.7.10 with SciPy 0.14.0, Matplotlib 1.4.2, NumPy 1.8.1, and SWIG 3.0.2. Note that you will need to install both swig and swig-python libraries if you install SWIG via Macports. If you install SWIG from source, you do not need to install a separate swig-python library.

It will be cleaner to keep the source directory separate from the build and install directories. For simplicity, we will create a `UQTk-build` directory in the same parent folder as the source directory, `UQTk` . While in the source directory, create the build directory and cd into it:

```
$ mkdir ../UQTk-build
$ cd ../UQTk-build
```

It is important to note that the CMake compilation uses the `cc` and `c++` defined compilers by default. This may not be the compilers you want when installing UQTk. Luckily, CMake allows you to specify which compilers you want, similar to autoconf. Thus, we type

```
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=$PWD/../UQTk-install \
        -DCMAKE_Fortran_COMPILER=/opt/local/gcc61/bin/gfortran-6.1.0 \
        -DCMAKE_C_COMPILER=/opt/local/gcc61/bin/gcc-6.1.0 \
        -DCMAKE_CXX_COMPILER=/opt/local/gcc61/bin/g++-6.1.0 ../UQTk
```

Note that this will configure CMake to compile UQTk without the Python interface. Also, we specified the installation directory to be `UQTk-install` in the same parent directory at `UQTk` and `UQTk-build`. Figure 2.1 shows what CMake prints to the screen. To turn on the

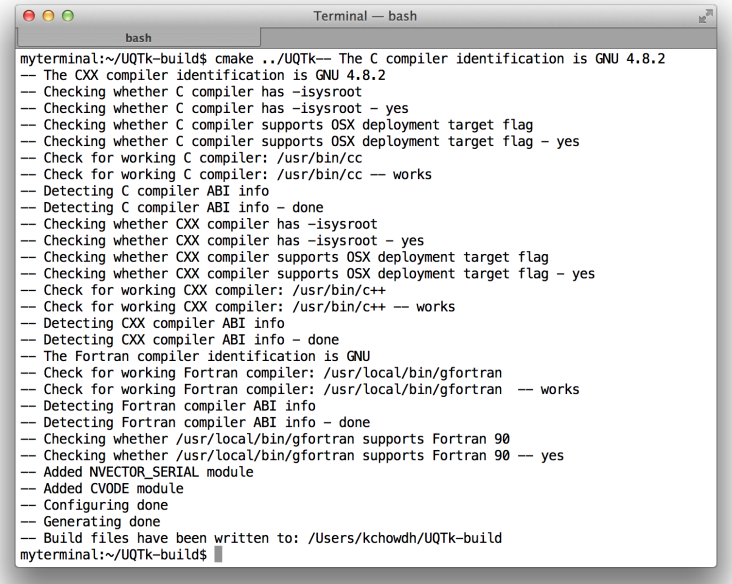


Figure 2.1: CMake configuration without the Python interface.

Python interface just set the CMake flag, `PyUQTk`, on, i.e.,

```
$ cmake -DPyUQTk=ON \
        -DCMAKE_INSTALL_PREFIX:PATH=$PWD/../UQTk-install \
        -DCMAKE_Fortran_COMPILER=/opt/local/gcc61/bin/gfortran-6.1.0 \
        -DCMAKE_C_COMPILER=/opt/local/gcc61/bin/gcc-6.1.0 \
        -DCMAKE_CXX_COMPILER=/opt/local/gcc61/bin/g++-6.1.0 ../UQTk
```

Figure 2.2 shows the additional output to screen after the Python interface flag is turned on.

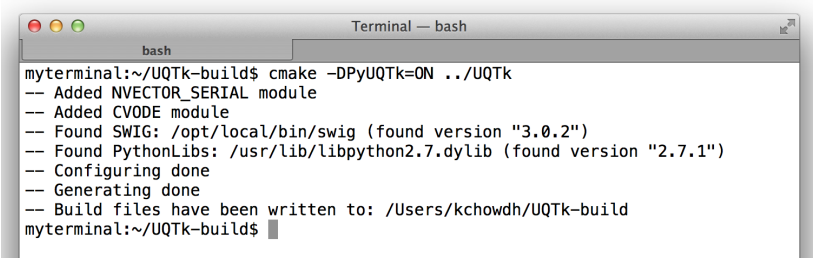


Figure 2.2: CMake configuration with the Python interface.

If the CMake command has executed without error, you are now ready to build UQTk. While in the build directory, type

```
$ make
```

or, for a faster compilation using `N` parallel threads,

```
$ make -j N
```

where one can replace `N` with the number of virtual cores on your machine, *e.g.* 8. This will build in the `UQTK-build/` directory. The screen should look similar to Figure 2.3 with or without the Python interface when building.

To verify that the build was successful, run the `ctest` command from the `UQTK-build/` directory to execute C++ and Python (only if building PyUQTk) test scripts.

```
$ ctest
```

The output should look similar to Figure 2.4.

If the Python tests fail, even though the compilation went well, a common issue that the configure script may have found a different version of the Python libraries than the one is used when you issue Python from the command line. To avoid this, specify the path to your Python executable and libraries to the configuration process. For example (on OS X):

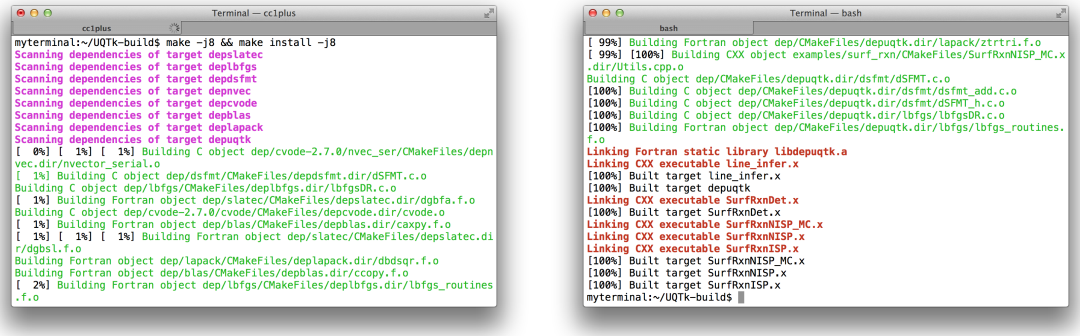


Figure 2.3: Start and end of build without Python interface.

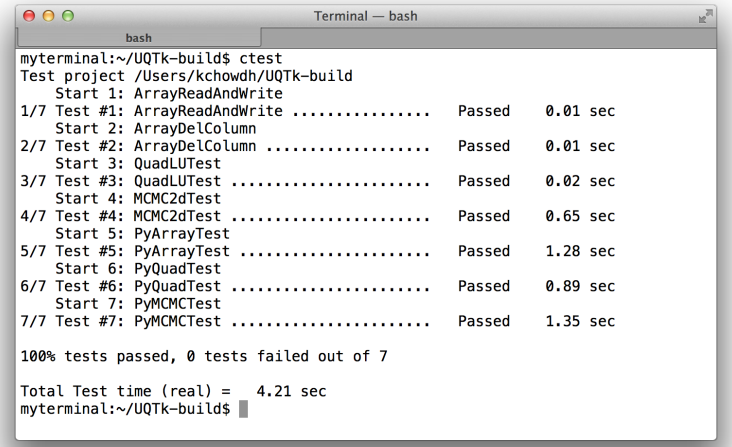


Figure 2.4: Result of ctest after successful build and install. Note that if you do not build PyUQTk, those tests will not be run.

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=$PWD/../UQTk-install \
      -DCMAKE_Fortran_COMPILER=/opt/local/gcc61/bin/gfortran-6.1.0 \
      -DCMAKE_C_COMPILER=/opt/local/gcc61/bin/gcc-6.1.0 \
      -DCMAKE_CXX_COMPILER=/opt/local/gcc61/bin/g++-6.1.0 ../UQTk
      -DPYTHON_EXECUTABLE:FILEPATH=/opt/local/bin/python \
      -DPYTHON_LIBRARY:FILEPATH=/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/libpython2.7.dylib \
      -DPyUQTk=ON \
      ../UQTk
```

If all looks good, you are now ready to install UQTk. While in the build directory, type

```
$ make install
```

which installs the libraries, headers, apps, examples, and such in the specified installation directory. Additionally, if you are building the Python interface, the install command will copy over the python scripts and SWIG modules (`*.so`) over to `PyUQTk/`.

As a reminder, commonly used configure options are illustrated in the scripts that are provided in the "*sourcedir*/config" folder.

Chapter 3

Theory and Conventions

UQTk implements many probabilistic methods found in the literature. For more details on the methods, please refer to the following papers and books on Polynomial Chaos methods for uncertainty propagation [4, 18], Karhunen-Loève (KL) expansions [8], numerical quadrature (including sparse quadrature) [14, 3, 10, 32, 7], Bayesian inference [31, 5, 19], Markov Chain Monte Carlo [6, 9, 11, 12], Bayesian compressive sensing [1], and the Rosenblatt transformation [24].

Below, some key aspects and conventions of UQTk Polynomial Chaos expansions are outlined in order to connect the tools in UQTk to the broader theory.

Polynomial Chaos Expansions

- The default ordering of PCE terms in the multi-index in UQTk is the canonical ordering for total order truncation
- The PC basis functions in UQTk are not normalized
- The Legendre-Uniform PC Basis type is defined on the interval [-1, 1], with weight function 1/2

Chapter 4

Source Code Description

For more details on the actual source code in UQTk, HTML documentation is also available in the `doc_cpp/html` folder.

Applications

The following command-line applications are available (source code is in `cpp/app`)

- ◎ `generate_quad` : Quadrature point/weight generation
- ◎ `gen_mi` : Polynomial multiindex generation
- ◎ `gp_regr` : Gaussian process regression
- ◎ `lr_regr` : Low-rank regression
- ◎ `model_inf` : Model parameter inference
- ◎ `pce_eval` : PC evaluation
- ◎ `pce_quad` : PC generation from samples
- ◎ `pce_resp` : PC projection via quadrature integration
- ◎ `pce_rv` : PC-related random variable generation
- ◎ `pce_sens` : PC sensitivity extraction
- ◎ `pdf_cl` : Kernel Density Estimation
- ◎ `regression` : Linear parametric regression
- ◎ `sens` : Sobol sensitivity indices via Monte-Carlo sampling

Below we detail the theory behind all the applications. For specific help in running an app, type `app_name -h`.

◎ generate_quad:

This utility generates isotropic quadrature (both full tensor product or sparse) points of given dimensionality and type. The keyword options are:

Quadrature types: `-g <quadType>`

- `LU` : Legendre-Uniform
- `HG` : Gauss-Hermite
- `LG` : Gamma-Laguerre
- `SW` : Stieltjes-Wiegert
- `JB` : Beta-Jacobi
- `CC` : Clenshaw-Curtis
- `CCO` : Clenshaw-Curtis Open (endpoints not included)
- `NC` : Newton-Cotes (equidistant)
- `NCO` : Newton-Cotes Open (endpoints not included)
- `GP3` : Gauss-Patterson
- `pdf` : Custom PDF

Sparsity types: `-x <fsType>`

- `full` : full tensor product
- `sparse` : Smolyak sparse grid construction

Note that one can create an equidistant multidimensional grid by using 'NC' quadrature type and 'full' sparsity type.

◎ gen_mi:

This utility generates multi index set of a given type and dimensionality. The keyword options are:

Multiindex types: `-x <mi_type>`

- `TO` : Total order truncation, *i.e.* $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$, where $\alpha_1 + \cdots + \alpha_d = ||\boldsymbol{\alpha}||_1 \leq p$, for given order p and dimensionality d. The number of multiindices is $N_{p,d}^{TO} = (p + d)!/(p!d!)$.

- `TP` : Tensor product truncation, *i.e.* $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$, where $\alpha_i \leq p_i$, for $i = 1, \ldots, d$. The dimension-specific orders are given in a file with a name specified as a command-line argument (`-f`). The number of multiindices is $N_{p_1,\ldots,p_d}^{TP} = \prod_{i=1}^{d}(p_i + 1)$.

- `HDMR` : High-Dimensional Model Representation, where, for each k, k-variate multiindices are truncated up to a given order. That is, if $||\boldsymbol{\alpha}||_0 = k$ (*i.e.* the number of non-zero elements is equal to k), then $||\boldsymbol{\alpha}||_1 \leq p_k$, for $k = 1, \ldots, k_{max}$. The variate-specific orders p_k are given in a file with a name specified as a command-line argument (`-f`). The number of multiindices constructed in this way is $N_{p_0,\ldots,p_{k_{max}}}^{HDMR} = \sum_{k=0}^{k_{max}}(p_k + k)!/(p_k!k!)$.

◎ `gp_regr:`

This utility performs Gaussian process regression [23], in particular using the Bayesian perspective of constructing GP emulators, see e.g. [13, 21]. The data is given as pairs $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, where $x \in \mathbb{R}^d$. The function to be found, $f(x)$ is endowed with a Gaussian prior with mean $\boldsymbol{h}(x)^T\boldsymbol{c}$ and a predefined covariance $C(x, x') = \sigma^2 c(x, x')$. Currently, only a squared-exponential covariance is implemented, *i.e.* $c(x, x') = e^{-(x-x')^T B(x-x')}$. The *mean trend* basis vector $\boldsymbol{h}(x) = (L_0(x), \ldots, L_{K-1}(x))$ consists of Legendre polynomials, while $\boldsymbol{c}$ and σ^2 are *hyperparameters* with a normal inverse gamma (conjugate) prior

$$p(\boldsymbol{c}, \sigma^2) = p(\boldsymbol{c}|\sigma^2)p(\sigma^2) \propto \frac{e^{-\frac{(\boldsymbol{c}-\boldsymbol{c}_0)^T V^{-1}(\boldsymbol{c}-\boldsymbol{c}_0)}{2\sigma^2}}}{\sigma} \frac{e^{-\frac{\beta}{\sigma^2}}}{\sigma^{2(\alpha+1)}}.$$

The parameters $\boldsymbol{c}_0, V^{-1}$ and B are fixed for the duration of the regression. Conditioned on $y_i = f(x_i)$, the posterior is a student-t process

$$f(x)|\mathcal{D}, \boldsymbol{c}_0, V^{-1}, B, \alpha, \beta \sim \text{St-t}(\mu^*(x), \hat{\sigma}c^*(x, x'))$$

with mean and covariance defined as

$$\begin{aligned} \mu^*(x) &= \boldsymbol{h}(x)^T\hat{\boldsymbol{c}} + \boldsymbol{t}(x)^T A^{-1}(\boldsymbol{y} - H\hat{\boldsymbol{c}}), \\ c^*(x, x') &= c(x, x') - \boldsymbol{t}(x)^T A^{-1}\boldsymbol{t}(x') + [\boldsymbol{h}(x)^T - \boldsymbol{t}(x)^T A^{-1}H]V^*[\boldsymbol{h}(x')^T - \boldsymbol{t}(x')^T A^{-1}H]^T, \end{aligned}$$

where $\boldsymbol{y}^T = (y^{(1)}, \ldots, y^{(N)})$ and

$$\begin{aligned} \hat{\boldsymbol{c}} &= V^*(V^{-1}\boldsymbol{c}_0 + H^T A^{-1}\boldsymbol{y}) & \hat{\sigma}^2 &= \frac{2\beta + \boldsymbol{c}_0^T V^{-1}\boldsymbol{c}_0 + \boldsymbol{y}^T A^{-1}\boldsymbol{y} - \hat{\boldsymbol{c}}^T (V^*)^{-1}\hat{\boldsymbol{c}}}{N + 2\alpha - K - 2} \\ \boldsymbol{t}(x)^T &= (c(x, x^{(1)}), \ldots, c(x, x^{(N)})) & V^* &= (V^{-1} + H^T A^{-1}H)^{-1} \\ H &= (\boldsymbol{h}(x^{(1)})^T, \ldots, \boldsymbol{h}(x^{(N)})^T) & A_{mn} &= c(x^{(m)}, x^{(n)}) \end{aligned} \tag{4.1}$$

Note that currently the commonly used prior $p(\boldsymbol{c}, \sigma^2) \propto \sigma^{-2}$ is implemented which is a special case with $\alpha = \beta = 0$ and $c_0 = \mathbf{0}$, $V^{-1} = 0_{K\times K}$. Also, a small *nugget* of size 10^{-6} is added to the diagonal of matrix A for numerical purposes, playing a role of 'data noise'. Finally, the covariance matrix B is taken to be diagonal, with the entries either fixed or found before the regression by maximizing marginal posterior [21]. More flexibility in trend basis and covariance structure selection is a matter of current work.

The app builds the Student-t process according to the computations detailed above, and evaluates its mean and covariance at a user-defined grid of points x.

◎ `lr_regr`:

This module constructs a canonical low rank approximation of a function in a black box setting given input/output samples.

Canonical-tensor decomposition: A univariate function $u(x)$ can be written approximately as

$$u(x) \approx \tilde{u}(x) = \sum_{j=0}^{p} v_j \phi_j(x), \tag{4.2}$$

where $\phi_j(x)$ is the jth basis function and v_j is the jth expansion coefficient, for $j = 0, \ldots, p$ with some $p > 0$. Likewise, a multivariate function $u(\boldsymbol{x})$ can be expanded as

$$u(\boldsymbol{x}) \approx \tilde{u}(\boldsymbol{x}) = \sum_{j_1=0}^{p_1} \cdots \sum_{j_m=0}^{p_m} v_{j_1,\ldots,j_m} \phi_{j_1}^{(1)}(x_1) \cdots \phi_{j_m}^{(m)}(x_m), \tag{4.3}$$

where $\phi_{j_i}^{(i)}(x_i)$ is the j_ith basis function in the ith coordinate, x_i. The number of expansion coefficients $\{v_{j_1,\ldots,j_m}\}$ is $\prod_{i=1}^{m}(p_i + 1)$ or an $O(p_1^m)$ quantity, if $p_1 = \cdots = p_m$. This exponential increase in the number of unknowns with dimension is a manifestation of the curse of dimensionality.

A low-rank approximation instead expands $u(\boldsymbol{x})$ in the form

$$u(\boldsymbol{x}) \approx \tilde{u}(\boldsymbol{x}) = \sum_{k=1}^{r} \prod_{i=1}^{m} w_k^{(i)}(x_i), \tag{4.4}$$

with each univariate function $w_k^{(i)}(x_i)$ being represented, in analogy to Eq. (4.2), as

$$w_k^{(i)}(x_i) = \sum_{j_i=0}^{p_i} w_{k,j_i}^{(i)} \phi_{j_i}^{(i)}(x_i). \tag{4.5}$$

Thus a low-rank approximation of $u(\boldsymbol{x})$ is given as

$$u(\boldsymbol{x}) \approx \sum_{k=1}^{r} \prod_{i=1}^{m} \left\{ \sum_{j_i=0}^{p_i} w_{k,j_i}^{(i)} \phi_{j_i}^{(i)}(x_i) \right\}. \tag{4.6}$$

The number of expansion coefficients $\{w_{k,j_i}^{(i)}\}$ is dramatically reduced to $r \sum_{i=1}^{m}(p_i+1)$, which is an $O(rmp_1)$ quantity, if $p_1 = \cdots = p_m$, and is linear with dimension m. The value of r and its scaling with m is dependent on problem and can only be assessed from applications as demonstrated below. Next, we describe an algorithm, which is based on alternating least squares, to determine the coefficients $\{w_{k,j_i}^{(i)}\}$.

Alternating-least-squares algorithm: Before explaining the alternating-least-squares (ALS) algorithm, we first review the standard least-squares method to determine the coefficients $\{v_j\}$ in Eq. (4.2). Suppose that we have S sample points of x, $\{\mathrm{x}^s | s = 1, \ldots, S\}$, at which we evaluate $u(x)$. Defining an S-by-$(p+1)$ matrix $\boldsymbol{\Phi}$ by

$$\boldsymbol{\Phi} = \begin{bmatrix} \phi_0(\mathrm{x}^1) & \ldots & \phi_p(\mathrm{x}^1) \\ \vdots & \ddots & \vdots \\ \phi_0(\mathrm{x}^S) & \ldots & \phi_p(\mathrm{x}^S) \end{bmatrix}, \tag{4.7}$$

we can express Eq.(4.2) on the sample points as

$$\mathbf{u} \approx \boldsymbol{\Phi}\mathbf{v}, \tag{4.8}$$

where $\mathbf{u}$ and $\mathbf{v}$ are column vectors defined as $(\mathbf{u})_i = u(\mathrm{x}^i)$, and $(\mathbf{v})_j = v_j$. The least-squares method solves for $\mathbf{v}$ that minimizes the variance,

$$\|\mathbf{u} - \boldsymbol{\Phi}\mathbf{v}\|_2^2, \tag{4.9}$$

where $\|\cdot\|_2$ is L_2 norm of a vector. Hence, the coefficients $\{v_i\}$ are obtained by performing the minimization

$$\min_{\mathbf{v}} \|\mathbf{u} - \boldsymbol{\Phi}\mathbf{v}\|_2^2, \tag{4.10}$$

which has a closed-form solution,

$$\mathbf{v} = \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^T\mathbf{u}, \tag{4.11}$$

in the case of real valued basis functions.

In a low-rank approximation of a multivariate function, we determine the expansion coefficients $\{w_{k,j_i}^{(i)}\}$ by minimizing the variance in the m-dimensional space,

$$\min_{\{w\}} \|u - \tilde{u}\|_2^2, \tag{4.12}$$

where $\tilde{u}$ is written as Eq. (4.6). The ALS algorithm consists in performing the standard least-squares determination of expansion coefficients $\{w_{k,j_l}^{(l)}\}$ for one coordinate (say, $l = i$) at a time, while holding others (all l except i) fixed, and repeating it for all coordinates cyclically until convergence.

One least-squares iteration for the ith coordinate is carried out as follows. Let the column vector [of length $r(p_i + 1)$] in the matrix of expansion coefficients corresponding to the ith coordinate be

$$\mathbf{z}^{(i)} = \begin{bmatrix} \mathbf{w}_1^{(i)} \\ \vdots \\ \mathbf{w}_r^{(i)} \end{bmatrix}, \tag{4.13}$$

where $\mathbf{w}_k^{(i)} = [w_{k,0}^{(i)}, \ldots, w_{k,p_i}^{(i)}]^T$ is a column vector of length $p_i + 1$. We also define an S-by-$r(p_i + 1)$ matrix $\mathbf{\Phi}^{(i)}$ as

$$\mathbf{\Phi}^{(i)} = \left[\mathbf{\Phi}_1^{(i)} \cdots \mathbf{\Phi}_r^{(i)} \right], \tag{4.14}$$

with

$$\mathbf{\Phi}_k^{(i)} = \begin{bmatrix} c_{k,1}^{(i)} \phi_0^{(i)}(\mathrm{x}_i^1) & \ldots & c_{k,1}^{(i)} \phi_{p_i}^{(i)}(\mathrm{x}_i^1) \\ \vdots & \ddots & \vdots \\ c_{k,S}^{(i)} \phi_0^{(i)}(\mathrm{x}_i^S) & \ldots & c_{k,S}^{(i)} \phi_{p_i}^{(i)}(\mathrm{x}_i^S) \end{bmatrix}, \tag{4.15}$$

where

$$c_{k,s}^{(i)} = \prod_{l=1, l\neq i}^{m} w_k^{(l)}(\mathrm{x}_l^s) \tag{4.16}$$

$$= \prod_{l=1, l\neq i}^{m} \left[\phi_0^{(l)}(\mathrm{x}_l^s) \ldots \phi_{p_l}^{(l)}(\mathrm{x}_l^s) \right] \cdot \mathbf{w}_k^{(l)}, \tag{4.17}$$

is the part of the multivariate function held fixed in this iteration.

According to Eq. (4.11), we find

$$\mathbf{z}^{(i)} = \left(\mathbf{\Phi}^{(i)^T}\mathbf{\Phi}^{(i)}\right)^{-1}\mathbf{\Phi}^{(i)^T}\mathbf{u}. \tag{4.18}$$

Starting with some initial guess of $\mathbf{z}^{(i)}$ for all i's ($1 \leq i \leq m$), we iterate the least-squares determination of $\mathbf{z}^{(i)}$ for one (the ith) dimension at a time, until the L_2 norm of difference of $\mathbf{z}^{(i)}$ in consecutive iterations falls below a small tolerance or the maximum iteration count is reached.

Implementation: The syntax of the main script is

```
lr_regr -x <xfile> -y<yfile> -b <basistype> -r <rank>  -t <xcheckfile>
-o <order> -i<maxiter> -s<strpar> -v %-l<dblpar>
```

- `-x <xfile>` : A file containing input sample points $\{\mathrm{x}^s | s = 1, \ldots, S\}$ at which the function was evaluated (matrix of size $S \times m$). Default is `xdata.dat`
- `-y <yfile>` : A file containing output sample points $u(\mathrm{x}^s)$ (A vector of length S). Default is `ydata.dat`
- `-b <basistype>` : Type of basis $\phi_{j_i}^{(i)}$. Current implementation allows only one basis type for all dimensions. There are two options.
  - `PC` corresponds to Polynomial Chaos basis. Type of polynomial chaos is indicated by `-s` option (see below)
  - `POL` corresponds to monomial basis i.e. $1, x, x^2 \ldots$
- `-r <rank>` : An integer as Maximum rank of approximation (i.e. r in Eq. (4.4))
- `-t <xcheckfile>` : A file containing input sample points at which the approximation is tested for validation or plotting purposes. The output of low rank surrogate evaluation is stored in `ycheck_k.dat` files where $1 \leq k \leq r$. If `xcheckfile.dat` is not provided, `xdata.dat` is used instead.
- `-o <order>` : An integer as order of basis function (i.e. p_i in Eq. (4.5)). In the current implementation, we use the same order in all dimensions. The default order is 4.
- `-i <maxiter>` : An Integer for maximum iterations in ALS. The default value is 50.
- `-s <strpar>` : A string for type of polynomial chaos (for `PC` basis). The default used here is Legendre basis for standard uniform measure.
- `-v` : Verbosity flag to control display on screen during execution. Do not use it if you want only the bare minimum.

◎ `model_inf:`

This utility perform Bayesian inference for several generic types of models. Consider a dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{L}$ of pairs of $\boldsymbol{x}$-$\boldsymbol{y}$ measured values from some unknown 'truth' function $g(\cdot)$, i.e. $y^{(i)} = g(\boldsymbol{x}^{(i)})$+meas.errors. For example, $\boldsymbol{y}^{(i)}$ can be measurements at spatial locations $\boldsymbol{x}^{(i)}$, or at time instances $\boldsymbol{x}^{(i)}$, or $\boldsymbol{x}^{(i)} = i$ simply enumerating several observables. We call elements of $\boldsymbol{x} \in \mathbb{R}^S$ *design or controllable* parameters. For simplicity, assume $y^{(i)}$ is a scalar, but the code accepts multiple replica data for each $\boldsymbol{x}^{(i)}$. Assume, generally, that g is not deterministic, i.e. the vector of measurements $y^{(i)}$ at each i contains R instances/replicas/measurements of the true output $g(\boldsymbol{x})$. Furthermore, consider a model of interest $f(\boldsymbol{\lambda}; \boldsymbol{x})$ as a function of *model parameters* $\boldsymbol{\lambda} \in \mathbb{R}^D$ producing a single output. We are interested in calibrating the model $f(\boldsymbol{\lambda}; \boldsymbol{x})$ with respect to model parameters $\boldsymbol{\lambda}$, seeking an approximate match of the model to the truth:

$$f(\boldsymbol{\lambda}; \boldsymbol{x}) \approx g(\boldsymbol{x}). \tag{4.19}$$

The full error budget takes the following form

$$y^{(i)} = f(\boldsymbol{\lambda}; x^{(i)}) + \delta(x^{(i)}) + \epsilon_i, \tag{4.20}$$

where $\delta(x)$ is the model discrepancy term, and ϵ_i is the measurement error for the i-th data point. The most common assumption for the latter is an *i.i.d* Gaussian assumption with vanishing mean

$$\epsilon_i \sim N(0, \sigma^2), \text{ for all } i = 1, \ldots, L. \tag{4.21}$$

Concerning model error $\delta(x)$, we envision three scenarios:

- when the model discrepancy term $\delta(x)$ is ignored, one arrives at the *classical* construction $y^{(i)} - f(\lambda; x^{(i)}) \sim N(0, \sigma^2)$ with likelihood described below in Eq. (4.29).
- when the model discrepancy $\delta(x)$ is modeled explicitly as a Gaussian process with a predefined, typically squared-exponential covariance term with parameters either fixed apriori or inferred as hyperparameters, together with λ. This approach has been established in [16], and is referred to as "Kennedy-O'Hagan", *koh* approach.
- embedded model error approach is a novel strategy when model error is embedded into the model itself. For detailed discussion on the advantages and challenges of the approach, see [27]. This method leads to several likelihood options (keywords *abc, abcm, gausmarg, mvn, full, marg*), many of which are topics of current research and are under development. In this approach, one augments some of the parameters in λ with a probabilistic representation, such as multivariate normal, and infers parameters of this representation instead. Without loss of generality, and for the clarity of illustration, we assumed that the *first* M components of $\boldsymbol{\lambda}$ are augmented with a random variable.

  One embedding option is the first-order Gauss-Hermite PC expansion. In other words, $\boldsymbol{\lambda}$ is augmented by a multivariate normal random variable as

$$\boldsymbol{\lambda} \to \boldsymbol{\Lambda} = \boldsymbol{\lambda} + A(\boldsymbol{\alpha})\vec{\xi}, \tag{4.22}$$

where

$$A(\boldsymbol{\alpha}) = \begin{bmatrix} \alpha_{11} & 0 & 0 & \dots & 0 \\ \alpha_{21} & \alpha_{22} & 0 & \dots & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{M1} & \alpha_{M2} & \alpha_{M3} & \dots & \alpha_{MM} \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}_{D \times M}, \text{ and } \vec{\xi} = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_M \end{bmatrix} \tag{4.23}$$

Here $\vec{\xi}$ is a vector of independent identically distributed standard normal variables, and $\boldsymbol{\alpha} = (\alpha_{11}, \dots, \alpha_{MM})$ is the vector of size $M(M+1)/2$ of all non-zero entries in the matrix A. The set of parameters describing the random vector $\boldsymbol{\Lambda}$ is $\hat{\boldsymbol{\lambda}} = (\boldsymbol{\lambda}, \boldsymbol{\alpha})$ The full data model then is written as

$$y^{(i)} = f(\boldsymbol{\lambda} + A(\boldsymbol{\alpha})\vec{\xi}; x^{(i)}) + \epsilon_i \tag{4.24}$$

or

$$y^{(i)} = f_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}; \vec{\xi}) + \sigma^2 \xi_{M+i}, \tag{4.25}$$

where $f_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}; \vec{\xi})$ is a random process induced by this model error embedding. The mean and variance of this process are defined as $\mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x})$ and $\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x})$, respectively. To represent this random process and allow easy access to its first two moments, we employ a non-intrusive spectral projection (NISP) approach to propagate uncertainties in f via Gauss-Hermite PC expansion,

$$y^{(i)} = \sum_{k=0}^{K-1} f_{ik}(\boldsymbol{\lambda}, \boldsymbol{\alpha}) \Psi_k(\vec{\xi}) + \sigma^2 \xi_{M+i}, \tag{4.26}$$

for a fixed order p expansion, leading to $K = (p+M)!/(p!M!)$ terms.

The parameter estimation problem for $\boldsymbol{\lambda}$ is now reformulated as a parameter estimation for $\hat{\boldsymbol{\lambda}} = (\boldsymbol{\lambda}, \boldsymbol{\alpha})$. This inverse problem is solved via Bayesian machinery. Bayes' formula reads

$$\underbrace{p(\hat{\boldsymbol{\lambda}}|\mathcal{D})}_{\text{posterior}} \propto \underbrace{p(\mathcal{D}|\hat{\boldsymbol{\lambda}})}_{\text{likelihood}} \underbrace{p(\hat{\boldsymbol{\lambda}})}_{\text{prior}}, \tag{4.27}$$

where the key function is the likelihood function

$$\mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = p(\mathcal{D}|\hat{\boldsymbol{\lambda}}) \tag{4.28}$$

that connects the prior distribution of the parameters of interest to the posterior one. The options for the likelihood are given further in this section. For details on the likelihood construction, see [27]. To alleviate the invariance with respect to sign-flips, we use a prior that enforces $\alpha_{Mi} > 0$ for $i = 1, \dots, M$. Also, one can either fix σ^2 or infer it together with $\hat{\boldsymbol{\lambda}}$.

Exact computation of the potentially high-dimensional posterior (4.27) is usually problematic, therefore we employ Markov chain Monte Carlo (MCMC) algorithm for sampling from the posterior. Model f and the exact form of the likelihood are determined using command line arguments. Below we detail the currently implemented model types.

Model types: `-f <modeltype>`

- `prop` : for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^1$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \boldsymbol{\lambda}\boldsymbol{x}$.
- `prop_quad` : for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^2$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \boldsymbol{\lambda}_1\boldsymbol{x} + \boldsymbol{\lambda}_2\boldsymbol{x}^2$.
- `exp` : for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^2$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = e^{\boldsymbol{\lambda}_1 + \boldsymbol{\lambda}_2\boldsymbol{x}}$.
- `exp_quad` : for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^3$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = e^{\boldsymbol{\lambda}_1 + \boldsymbol{\lambda}_2\boldsymbol{x} + \boldsymbol{\lambda}_3\boldsymbol{x}^2}$.
- `const` : for any $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{\lambda} \in \mathbb{R}^1$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \boldsymbol{\lambda}$.
- `linear` : for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^2$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \boldsymbol{\lambda}_1 + \boldsymbol{\lambda}_2\boldsymbol{x}$.
- `bb` : the model is a 'black-box' executed via system-call of a script named `bb.x` that takes files `p.dat` (matrix $R \times D$ for $\boldsymbol{\lambda}$) and `x.dat` (matrix $L \times S$ for $\boldsymbol{x}$) and returns output `y.dat` (matrix $R \times L$ for f). This effectively simulates $f(\boldsymbol{\lambda}; \boldsymbol{x})$ at any R values of $\boldsymbol{\lambda}$ and L values of $\boldsymbol{x}$.
- `heat_transfer1` : a custom model designed for a tutorial case of a heat conduction problem: for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^1$, the model is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \frac{\boldsymbol{x} d_w}{A_w \boldsymbol{\lambda}} + T_0$, where $d_w = 0.1$, $A_w = 0.04$ and $T_0 = 273$.
- `heat_transfer2` : a custom model designed for a tutorial case of a heat conduction problem: for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^2$, the model is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \frac{\boldsymbol{x} Q}{A_w \lambda_1} + \lambda_2$, where $A_w = 0.04$ and $Q = 20.0$.
- `frac_power` : a custom function for testing. For $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^4$, the function is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \lambda_0 + \lambda_1 x + \lambda_2 x^2 + \lambda_3 (x+1)^{3.5}$.
- `exp_sketch` : exponential function to enable the sketch illustrations of model error embedding approach, for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^2$, the model is defined as $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \lambda_2 e^{\lambda_1 x} - 2$.
- `inp` : a function that produces the input components as output. That is $f(\boldsymbol{\lambda}; \boldsymbol{x}^{(i)}) = \lambda_i$, for $\boldsymbol{x} \in \mathbb{R}^1$ and $\boldsymbol{\lambda} \in \mathbb{R}^d$, assuming exactly d values for the design variables $\boldsymbol{x}$ (these are usually simply indices $x_i = i$ for $i = 1, \ldots, d$).
- `pcl` : the model is a Legendre PC expansion that is linear with respect to coefficients $\boldsymbol{\lambda}$, *i.e.* $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \sum_{\boldsymbol{\alpha} \in \mathcal{S}} \lambda_{\boldsymbol{\alpha}} \Psi_{\boldsymbol{\alpha}}(\boldsymbol{x})$.
- `pcx` : the model is a Legendre PC expansion in both $\boldsymbol{x}$ and $\boldsymbol{\lambda}$, *i.e.* $\boldsymbol{z} = (\boldsymbol{\lambda}, \boldsymbol{x})$, and $f(\boldsymbol{\lambda}; \boldsymbol{x}) = \sum_{\boldsymbol{\alpha} \in \mathcal{S}} c_{\boldsymbol{\alpha}} \Psi_{\boldsymbol{\alpha}}(\boldsymbol{z})$

- `pc` : the model is a set of Legendre polynomial expansions for each value of $\boldsymbol{x}$: *i.e.* $f(\boldsymbol{\lambda}; \boldsymbol{x}^{(i)}) = \sum_{\boldsymbol{\alpha} \in \mathcal{S}} c_{\boldsymbol{\alpha},i} \Psi_{\boldsymbol{\alpha}}(\boldsymbol{\lambda})$.
- `pcs` : same as `pc`, only the multi-index set $\mathcal{S}$ can be different for each $\boldsymbol{x}^{(i)}$, *i.e.* $f(\boldsymbol{\lambda}; \boldsymbol{x}^{(i)}) = \sum_{\boldsymbol{\alpha} \in \mathcal{S}_i} c_{\boldsymbol{\alpha},i} \Psi_{\boldsymbol{\alpha}}(\boldsymbol{\lambda})$.

Likelihood construction is the key step and the biggest challenge in model parameter inference.

Likelihood types: `-l <liktype>`

- `classical` : No α, or $M = 0$. This is a classical, least-squares likelihood

$$\log \mathcal{L}_{\mathcal{D}}(\boldsymbol{\lambda}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - f(\boldsymbol{\lambda}; \boldsymbol{x}^{(i)}))^2}{2\sigma^2} - \frac{L}{2} \log (2\pi\sigma^2), \tag{4.29}$$

- `koh` : Kennedy-O'Hagan likelihood with explicit additive representation of model discrepancy [16].
- `full` : This is the exact likelihood

$$\mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = \pi_{\boldsymbol{h}_{\hat{\boldsymbol{\lambda}}}}(y^{(1)}, \ldots, y^{(L)}), \tag{4.30}$$

  where $\boldsymbol{h}_{\hat{\boldsymbol{\lambda}}}$ is the random vector with entries $f_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}; \vec{\xi}) + \sigma^2 \xi_{M+i}$. When there is no data noise, *i.e.* $\sigma = 0$, this likelihood is degenerate [27]. Typically, computation of this likelihood requires a KDE step for each $\hat{\boldsymbol{\lambda}}$ to evaluate a high-d PDF $\pi_{\boldsymbol{h}_{\hat{\boldsymbol{\lambda}}}}(\cdot)$.
- `marg` : Marginal approximation of the exact likelihood

$$\mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = \prod_{i=1}^{L} \pi_{\boldsymbol{h}_{\hat{\boldsymbol{\lambda}},i}}(y^{(i)}), \tag{4.31}$$

  where $\boldsymbol{h}_{\hat{\boldsymbol{\lambda}},i}$ is the i-th component of $\boldsymbol{h}_{\hat{\boldsymbol{\lambda}}}$. This requires one-dimensional KDE estimates performed for all N dimensions.
- `mvn` : Multivariate normal approximation of the full likelihood

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu}_{\hat{\boldsymbol{\lambda}}})^T \Sigma_{\hat{\boldsymbol{\lambda}}}^{-1} (\boldsymbol{y} - \boldsymbol{\mu}_{\hat{\boldsymbol{\lambda}}}) - \frac{L}{2} \log (2\pi) - \frac{1}{2} \log (\det \Sigma_{\hat{\boldsymbol{\lambda}}}), \tag{4.32}$$

  where mean vector $\boldsymbol{\mu}_{\hat{\boldsymbol{\lambda}}}$ and covariance matrix $\Sigma_{\hat{\boldsymbol{\lambda}}}$ are defined as $\boldsymbol{\mu}^i_{\hat{\boldsymbol{\lambda}}} = \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)})$ and ${\Sigma_{\hat{\boldsymbol{\lambda}}}}^{ij} = \mathbb{E}(\boldsymbol{h}_{\hat{\boldsymbol{\lambda}},i} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))(\boldsymbol{h}_{\hat{\boldsymbol{\lambda}},j} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(j)}))^T$, respectively.
- `gausmarg` : This likelihood further assumes independence in the gaussian approximation, leading to

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))^2}{2\left(\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}) + \sigma^2\right)} - \frac{1}{2} \sum_{i=1}^{L} \log 2\pi \left(\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}) + \sigma^2\right). \tag{4.33}$$

- `abcm` : This likelihood enforces the mean of $f_{\hat{\boldsymbol{\lambda}}}$ to match the mean of data

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))^2}{2\epsilon^2} - \frac{1}{2}\log(2\pi\epsilon^2), \tag{4.34}$$

- `abc` : This likelihood enforces the mean of $f_{\hat{\boldsymbol{\lambda}}}$ to match the mean of data and the standard deviation to match the average spread of data around mean within some factor γ

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))^2 + \left(\gamma|y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)})| - \sqrt{\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}) + \sigma^2}\right)^2}{2\epsilon^2} - \frac{1}{2}\log(2\pi\epsilon^2), \tag{4.35}$$

Input files:

For the complete list, type `model_inf -h`

- `-x <xdatafile>` : $L \times S$ matrix of $\boldsymbol{x}$
- `-y <ydatafile>` : $L \times E$ matrix of $\boldsymbol{y}$, usually $E = 1$, but one can provide more than one data point per design parameter $\boldsymbol{x}$
- `-t <xpredfile>` : $L' \times S$ matrix of $\boldsymbol{x}$ values used for posterior prediction, $L' \neq L$ in general. Defaults value (i.e. no flag given) is `xpredfile=xdatafile`. Most frequently, this is a file with a dense grid in the $\boldsymbol{x}$-space.

Output files:

- `fmeans.dat` : $L' \times 2$ mean predictions. The first column is the posterior mean, the second column is the MAP.
- `fvars.dat` : $L' \times 3$ prediction variance components. The first column is the posterior mean of the variance, the second column is the posterior variance of the mean, and the third column is the MAP of the variance.
- `pmeans.dat` : $d \times 2$ mean parameter values. The first column is the posterior mean, the second column is the MAP.
- `pvars.dat` : $d \times 3$ parameter variance components. The first column is the posterior mean of the variance, the second column is the posterior variance of the mean, and the third column is the MAP of the variance.
- `datavars.dat` : $L \times 2$ data variance values. The first column is the posterior mean, while the second column is MAP.

- `chain.dat` : The raw MCMC chain file of size $N_{MCMC} \times (d' + 3)$. The first column is simply the MCMC step number, the last two are the Metropolis-Hastings' ratio α and the log-posterior value, while the rest of the columns are the chain parameters. Chain dimensionality is d'.

- `pchain.dat` : $P \times d'$ 'thinned' posterior samples, where $P = int(N_{MCMC}/n_e)$, and the thinning factor n_e is given by the input `-n <every>`

- `mapparam.dat` : $d' \times 1$ vector of chain's MAP values

- `fmeans_sams.dat` : $L' \times P$ 'thinned' posterior samples of the mean predictions

- `parampccfs.dat` : $K \times P$ 'thinned' posterior samples of the input PC coefficients

◎ pce_eval:

This utility evaluates PC-related functions given input file `xdata.dat` and return the evaluations in an output file `ydata.dat`. It also provides gradient information in an output file `gdata.dat` for only LU PC function type. The keyword options are:

Function types: `-f <fcn_type>`

- `PC` : Evaluates the function $f(\vec{\xi}) = \sum_{k=0}^{K} c_k \Psi_k(\vec{\xi})$ given a set of $\vec{\xi}$, the PC type, dimensionality, order and coefficients.

- `PC_mi` : Evaluates the function $f(\vec{\xi}) = \sum_{k=0}^{K} c_k \Psi_k(\vec{\xi})$ given a set of $\vec{\xi}$, the PC type, multiindex and coefficients.

- `PCmap` : Evaluates 'map' functions from a germ of one PC type to another. That is PC1 to PC2 is a function $\vec{\eta} = f(\vec{\xi}) = C_2^{-1} C_1(\vec{\xi}_1)$, where C_1 and C_2 are the cumulative distribution functions (CDFs) associated with the PDFs of PC1 and PC2, respectively. For example, HG→LU is a map from standard normal random variable to a uniform random variable in $[-1, 1]$.

◎ pce_quad:

This utility constructs a PC expansion from a given set of samples. Given a set of N samples $\{x^{(i)}\}_{i=1}^{N}$ of a random d-variate vector $\vec{X}$, the goal is to build a PC expansion

$$\vec{X} \simeq \sum_{k=0}^{K} \boldsymbol{c}_k \Psi_k(\vec{\xi}), \tag{4.36}$$

where d is the stochastic dimensionality, i.e. $\vec{\xi} = (\xi_1, \ldots, \xi_d)$. We use orthogonal projection method, *i.e.*

$$\boldsymbol{c}_k = \frac{\langle \vec{X}\Psi_k(\vec{\xi})\rangle}{\langle \Psi_k^2(\vec{\xi})\rangle} = \frac{\langle \vec{G}(\vec{\xi})\Psi_k(\vec{\xi})\rangle}{\langle \Psi_k^2(\vec{\xi})\rangle}. \tag{4.37}$$

The denominator can be precomputed analytically or numerically with high precision. The key map $\vec{G}(\vec{\xi})$ in the numerator is constructed as follows. We employ the Rosenblatt transformation, constructed by shifted and scaled successive conditional cumulative distribution functions (CDFs),

$$\begin{aligned}
\eta_1 &= 2F_1(X_1) - 1 \\
\eta_2 &= 2F_{2|1}(X_2|X_1) - 1 \\
\eta_3 &= 2F_{3|2,1}(X_3|X_2, X_1) - 1 \\
&\vdots \\
\eta_d &= 2F_{d|d-1,\ldots,1}(X_d|X_{d-1}, \ldots, X_1) - 1.
\end{aligned} \tag{4.38}$$

maps any joint random vector to a set of independent standard Uniform[-1,1] random variables. Rosenblatt transformation is the multivariate generalization of the well-known CDF transformation, stating that $F(X)$ is uniformly distributed if $F(\cdot)$ is the CDF of random variable X. The shorthand notation is $\vec{\eta} = \vec{R}(\vec{X})$. Now denote the shifted and scaled univariate CDF of the 'germ' ξ_i by $H(\cdot)$, so that by the CDF transformation reads as $\vec{H}(\vec{\xi}) = \vec{\eta}$. For example, for Legendre-Uniform PC, the germ itself is uniform and $H(\cdot)$ is identity, while for Gauss-Hermite PC the function $H(\cdot)$ is shifted and scaled version of the normal CDF. Now, we can write the connection between $\vec{X}$ and $\vec{\xi}$ by

$$\vec{R}(\vec{X}) = \vec{H}(\vec{\xi}), \qquad \text{or} \qquad \vec{X} = \underbrace{\vec{R}^{-1} \circ \vec{H}}_{\vec{G}}(\vec{\xi}) \tag{4.39}$$

While the computation of $\vec{H}$ is done analytically or numerically with high precision, the main challenge is to estimate $\vec{R}^{-1}$. In practice the exact joint cumulative distribution $F(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d)$ is generally not available and is estimated using a standard Kernel Density Estimator (KDE) using the samples available. Given N samples $\{x^{(i)}\}_{i=1}^N$, the KDE estimate of its joint probability density function is a sum of N multivariate gaussian functions centered at each data point $\boldsymbol{x}^{(i)}$:

$$p_{\vec{X}}(\boldsymbol{x}) = \frac{1}{N\sigma^d(2\pi)^{d/2}} \sum_{i=1}^{N} \exp\left(-\frac{(\boldsymbol{x} - \boldsymbol{x}^{(i)})^T(\boldsymbol{x} - \boldsymbol{x}^{(i)})}{2\sigma^2}\right) \tag{4.40}$$

or

$$p_{\vec{X}_1,\ldots,\vec{X}_d}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d) = \frac{1}{N\sigma^d(2\pi)^{d/2}} \sum_{i=1}^{N} \exp\left(-\frac{(\boldsymbol{x}_1 - \boldsymbol{x}_1^{(i)})^2 + \cdots + (\boldsymbol{x}_d - \boldsymbol{x}_d^{(i)})^2}{2\sigma^2}\right), \tag{4.41}$$

where the *bandwidth* σ should be chosen to balance smoothness and accuracy, see [29, 30] for discussions of the choice of σ. Note that ideally σ should be chosen to be dimension-dependent, however the current implementation uses the same bandwidth for all dimensions.

Now the conditional CDF is KDE-estimated by

$$\begin{aligned} F_{k|k-1,\ldots,1}(\boldsymbol{x}_k|\boldsymbol{x}_{k-1},\ldots,\boldsymbol{x}_1) &= \int_{-\infty}^{\boldsymbol{x}_k} p_{k|k-1,\ldots,1}(\boldsymbol{x}_k^{'}|\boldsymbol{x}_{k-1},\ldots,\boldsymbol{x}_1)d\boldsymbol{x}_k^{'} \\ &= \int_{-\infty}^{\boldsymbol{x}_k} \frac{p_{k,\ldots,1}(\boldsymbol{x}_k^{'},\boldsymbol{x}_{k-1},\ldots,\boldsymbol{x}_1)}{p_{k-1,\ldots,1}(\boldsymbol{x}_{k-1},\ldots,\boldsymbol{x}_1)}d\boldsymbol{x}_k^{'} \\ &\approx \frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{\boldsymbol{x}_k} \frac{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_k^{'}-\boldsymbol{x}_k^{(i)})^2}{2\sigma^2}\right)}{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)}d\boldsymbol{x}_k^{'} \end{aligned}$$

$$\begin{aligned} &= \int_{-\infty}^{\boldsymbol{x}_k} \frac{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)\times\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(\boldsymbol{x}_k^{'}-\boldsymbol{x}_k^{(i)})^2}{2\sigma^2}\right)}{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)}d\boldsymbol{x}_k^{'} \\ &= \frac{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)\times\Phi\left(\frac{\boldsymbol{x}_k-\boldsymbol{x}_k^{(i)}}{\sigma}\right)}{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)}, \end{aligned} \tag{4.42}$$

where $\Phi(z)$ is the CDF of a standard normal random variable. Note that the numerator in (4.42) differs from the denominator only by an extra factor $\Phi\left(\frac{\boldsymbol{x}_k-\boldsymbol{x}_k^{(i)}}{\sigma}\right)$ in each summand, allowing an efficient computation scheme.

The above Rosenblatt transformation maps the random vector $\boldsymbol{x}$ to a set of i.i.d. uniform random variables $\vec{\eta} = (\eta_1,\ldots,\eta_d)$. However, the formula (4.39) requires the inverse of the Rosenblatt transformation. Nevertheless, the approximate conditional distributions are monotonic, hence they are guaranteed to have an inverse function, and it can be evaluated rapidly with a bisection method.

With the numerical estimation of the map (4.39) available, we can proceed to evaluation the numerator of the orthogonal projection (4.37)

$$\langle\vec{G}(\vec{\xi})\Psi_k(\vec{\xi})\rangle = \int_{\vec{\xi}}\vec{G}(\boldsymbol{x})\Psi_k(\boldsymbol{x})\pi_{\vec{\xi}}(\vec{\xi})d\vec{\xi}, \tag{4.43}$$

where $\pi_{\vec{\xi}}(\vec{\xi})$ is the PDF of $\vec{\xi}$. The projection integral (4.43) is computed via quadrature integration

$$\int_{\vec{\xi}} \vec{G}(\vec{\xi})\Psi_k(\vec{\xi})\pi_{\vec{\xi}}(\vec{\xi})d\vec{\xi} \approx \sum_{q=1}^{Q} \vec{G}(\vec{\xi}_q)\Psi_k(\vec{\xi}_q)w_q = \sum_{q=1}^{Q} \vec{R}^{-1}(\vec{H}(\vec{\xi}_q))\Psi_k(\vec{\xi}_q)w_q, \tag{4.44}$$

where $(\vec{\xi}_q, w_q)$ are Gaussian quadrature point-weight pairs for the weight function $\pi_{\vec{\xi}}(\vec{\xi})$.

◎ `pce_resp`:

This utility performs orthogonal projection given function evaluations at quadrature points, in order to arrive at polynomial chaos coefficients for a Total-Order PC expansion

$$f(\vec{\xi}) \approx \sum_{||\boldsymbol{\alpha}||_1 \leq p} c_{\boldsymbol{\alpha}}\Psi_{\boldsymbol{\alpha}}(\vec{\xi}) \equiv g(\vec{\xi}). \tag{4.45}$$

The orthogonal projection computed by this utility is

$$c_{\boldsymbol{\alpha}} = \frac{1}{\langle\Psi^2_{\boldsymbol{\alpha}}\rangle}\int_{\vec{\xi}} f(\vec{\xi})\Psi_{\boldsymbol{\alpha}}(\vec{\xi})\pi_{\vec{\xi}}(\vec{\xi})d\vec{\xi} \approx \frac{1}{\langle\Psi^2_{\boldsymbol{\alpha}}\rangle}\sum_{q=1}^{Q} w_q f(\vec{\xi}^{(q)})\Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(q)}). \tag{4.46}$$

Given the function evaluations $f(\vec{\xi}^{(q)})$ and precomputed quadrature $(\vec{\xi}^{(q)}, w_q)$, this utility outputs the PC coefficients $c_{\boldsymbol{\alpha}}$, PC evaluations at the quadrature points $g(\vec{\xi}^{(q)})$ as well as, if requested by a command line flag, a quadrature estimate of the relative L_2 error

$$\frac{||f-g||_2}{||f||_2} \approx \sqrt{\frac{\sum_{q=1}^{Q} w_q(f(\vec{\xi}^{(q)}) - g(\vec{\xi}^{(q)}))^2}{\sum_{q=1}^{Q} w_q f(\vec{\xi}^{(q)})^2}}. \tag{4.47}$$

Note that the selected quadrature may not compute the error accurately, since the integrated functions are squared and can be higher than the quadrature is expected to integrate accurately. In such cases, one can use the `pce_eval` app to evaluate the PC expansion separately and compare to the function evaluations with an ℓ_2 norm instead.

◎ `pce_rv`:

This utility generates PC-related random variables (RVs). The keyword options are:

RV types: `-w <type>`

- `PC` : Generates samples of *univariate* random variable $\sum_{k=0}^{K} c_k\Psi_k(\vec{\xi})$ given the PC type, dimensionality, order and coefficients.

- `PCmi` : Generates samples of *univariate* random variable $\sum_{k=0}^{K} c_k \Psi_k(\vec{\xi})$ given the PC type, multiindex and coefficients.

- `PCvar` : Generates samples of *multivariate* random variable $\vec{\xi}$ that is the *germ* of a given PC type and dimensionality.

◎ `pce_sens:`

This utility evaluates Sobol sensitivity indices of a PC expansion with a given multiindex and a coefficient vector. It computes main, total and joint sensitivities, as well as variance fraction of each PC term individually. Given a PC expansion $\sum_{\mathbb{I}} c_{\boldsymbol{\alpha}} \Psi_{\boldsymbol{\alpha}}(\vec{\xi})$, the computed moments and sensitivity indices are:

- mean: $m = c_{\vec{0}}$

- total variance: $V = \sum_{\boldsymbol{\alpha} \neq \vec{0}} c_{\boldsymbol{\alpha}}^2 \langle \Psi_{\boldsymbol{\alpha}}^2 \rangle$

- variance fraction for the basis term $\boldsymbol{\alpha}$: $V_{\boldsymbol{\alpha}} = \frac{c_{\boldsymbol{\alpha}}^2 \langle \Psi_{\boldsymbol{\alpha}}^2 \rangle}{V}$

- main Sobol sensitivity index for dimension i: $S_i = \frac{1}{V} \sum_{\boldsymbol{\alpha} \in \mathbb{I}_i^S} c_{\boldsymbol{\alpha}}^2 \langle \Psi_{\boldsymbol{\alpha}}^2 \rangle$, where $\mathbb{I}_i^S$ is the set of multiindices that include *only* dimension i.

- total Sobol sensitivity index for dimension i: $S_i^T = \frac{1}{V} \sum_{\boldsymbol{\alpha} \in \mathbb{I}_i^T} c_{\boldsymbol{\alpha}}^2 \langle \Psi_{\boldsymbol{\alpha}}^2 \rangle$, where $\mathbb{I}_i^T$ is the set of multiindices that include dimension i, among others.

- joint-total Sobol sensitivity index for dimension pair (i, j): $S_{ij}^T = \frac{1}{V} \sum_{\boldsymbol{\alpha} \in \mathbb{I}_{ij}^T} c_{\boldsymbol{\alpha}}^2 \langle \Psi_{\boldsymbol{\alpha}}^2 \rangle$, where $\mathbb{I}_{ij}^T$ is the set of multiindices that include dimensions i and j, *among others*. Note that this is somewhat different from the conventional definition of joint sensitivity indices, which presumes terms that include *only* dimensions i and j.

◎ `pdf_cl:`

Kernel density estimation (KDE) with Gaussian kernels given a set of samples to evaluate probability distribution function (PDF). The procedure relies on approximate nearest neighbors algorithm with fast improved Gaussian transform to accelerate KDE by only computing Gaussians of relevant neighbors. Our tests have shown 10-20x speedup compared to Python's default KDE package. Also, the app allows clustering enhancement to the data set to enable cluster-specific bandwidth selection - particularly useful for multimodal data. User provides the samples' file, and either a) number of grid points per dimension for density evaluation, or b) a file with target points where the density is evaluated, or c) a file with a hypercube limits in which the density is evaluated.

◎ `regression:`

This utility performs regression with respect to a linear parametric expansions such as PCs or RBFs. Consider a dataset $(x^{(i)}, y^{(i)})_{i=1}^N$ that one tries to fit a basis expansion with:

$$y^{(i)} \approx \sum_{k=1}^{K} c_k P_k(x^{(i)}), \tag{4.48}$$

for a set of basis functions $P_k(x)$. This is a linear regression problem, since the object of interest is the vector of coefficients $\boldsymbol{c} = (c_1, \ldots, c_k)$, and the summation above is linear in $\boldsymbol{c}$. This app provides various methods of obtaining the expansion coefficients, using different kinds of bases.

The key implemented command line options are

Basis types: `-f <basistype>`

- `PC` : Polynomial Chaos bases of total-order truncation
- `PC_MI` : Polynomial Chaos bases of custom multiindex truncation
- `POL` : Monomial bases of total-order truncation
- `POL_MI` : Monomial bases of custom multiindex truncation
- `RBF` : Radial Basis Functions, see e.g. [22]

Regression methods: `-f <meth>`

- `lsq` : Bayesian least-squares, see [26] and more details below.
- `wbcs` : Weighted Bayesian compressive sensing, see [28].

Although the standard least squares is commonly used and well-documented elsewhere, we detail here the specific implementation in this app, including the Bayesian interpretation.

Define the data vector $\boldsymbol{y} = (y^{(1)}, \ldots, y^{(N)})$, and the *measurement matrix* $\boldsymbol{P}$ of size $N \times K$ with entries $\boldsymbol{P}_{ik} = P_k(x^{(i)})$. The regularized least-squares problem is formulated as

$$\arg\min_{\boldsymbol{c}} \underbrace{||\boldsymbol{y} - \boldsymbol{P}\boldsymbol{c}||^2 + ||\boldsymbol{\Lambda}\boldsymbol{c}||^2}_{R(\boldsymbol{c})} \tag{4.49}$$

with a closed form solution

$$\hat{\boldsymbol{c}} = \underbrace{(\boldsymbol{P}^T\boldsymbol{P} + \boldsymbol{\Lambda})^{-1}}_{\boldsymbol{\Sigma}} \boldsymbol{P}^T\boldsymbol{y} \tag{4.50}$$

where $\mathbf{\Lambda} = diag(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_K})$ is a diagonal matrix of non-negative regularization weights $\lambda_i \geq 0$.

The Bayesian analog of this, detailed in [26], infers coefficient vector $\boldsymbol{c}$ and data noise variance σ^2, given data $\boldsymbol{y}$, employing Bayes' formula

$$\overbrace{p(\boldsymbol{c}, \sigma^2|\boldsymbol{y})}^{\text{Posterior}} \propto \overbrace{p(\boldsymbol{y}|\boldsymbol{c}, \sigma^2)}^{\text{Likelihood}} \overbrace{p(\boldsymbol{c}, \sigma^2)}^{\text{Prior}} \tag{4.51}$$

The likelihood function is associated with *i.i.d.* Gaussian noise model $\boldsymbol{y} - \boldsymbol{P}\boldsymbol{c} \sim N(0, \sigma^2 \boldsymbol{I}_N)$, and is written as,

$$p(\boldsymbol{y}|\boldsymbol{c}, \sigma^2) \equiv L_{\boldsymbol{c},\sigma^2}(\boldsymbol{y}) = (2\pi\sigma^2)^{-\frac{N}{2}} \exp\left(-\frac{1}{2\sigma^2}||\boldsymbol{y} - \boldsymbol{P}\boldsymbol{c}||^2\right) \tag{4.52}$$

Further, the prior $p(\boldsymbol{c}, \sigma^2)$ is written as a product of a zero-mean Gaussian prior on $\boldsymbol{c}$ and an inverse-gamma prior on σ^2:

$$p(\boldsymbol{c}, \sigma^2) = \underbrace{\left(\prod_{k=1}^{K} \frac{\lambda_k}{2\pi}\right)^{\frac{1}{2}} \exp\left(-\frac{1}{2}||\mathbf{\Lambda}\boldsymbol{c}||^2\right)}_{p(\boldsymbol{c})} \underbrace{(\sigma^2)^{-\alpha-1} \exp\left(-\frac{\beta}{\sigma^2}\right)}_{p(\sigma^2)} \tag{4.53}$$

The posterior distribution then takes a form of normal-scaled inverse gamma distribution which, after some re-arranging, is best described as

$$p(\boldsymbol{c}|\sigma^2, \boldsymbol{y}) \sim MVN(\hat{\boldsymbol{c}}, \sigma^2 \mathbf{\Sigma}), \tag{4.54}$$

$$p(\sigma^2|\boldsymbol{y}) \sim IG\left(\underbrace{\alpha + \frac{N-K}{2}}_{\alpha^*}, \underbrace{\beta + \frac{R(\hat{\boldsymbol{c}})}{2}}_{\beta^*}\right) \tag{4.55}$$

where $\hat{\boldsymbol{c}}$ and $\mathbf{\Sigma}$, as well as the residual $R(\cdot)$ are defined via the classical least-squares problem (4.49) and (4.50). Thus, the mean posterior value of data variance is $\hat{\sigma}^2 = \frac{\beta + \frac{R(\hat{\boldsymbol{c}})}{2}}{\alpha + \frac{N-K}{2} - 1}$. Also, note that the residual can be written as $R(\hat{\boldsymbol{c}}) = \boldsymbol{y}^T \left(\boldsymbol{I}_N - \boldsymbol{P}\left(\boldsymbol{P}^T\boldsymbol{P} + \mathbf{\Lambda}\right)^{-1}\boldsymbol{P}^T\right)\boldsymbol{y}$. One can integrate out σ^2 from (4.53) to arrive at a multivariate t-distribution

$$p(\boldsymbol{c}|\boldsymbol{y}) \sim MVT\left(\hat{\boldsymbol{c}}, \frac{\beta^*}{\alpha^*}\mathbf{\Sigma}, 2\alpha^*\right) \tag{4.56}$$

with a mean $\hat{\boldsymbol{c}}$ and covariance $\frac{\alpha^*}{\alpha^*-2}\mathbf{\Sigma}$.

Now, the pushed-forward process at *new* values x would be, defining $\boldsymbol{P}(x) = (P_1(x), \ldots, P_k(x))$, a Student-t process with mean $\mu(x) = \boldsymbol{P}(x)\hat{\boldsymbol{c}}$, scale $C(x, x') = \frac{\beta^*}{\alpha^*}\boldsymbol{P}(x)\mathbf{\Sigma}\boldsymbol{P}(x')$ and degrees-of-freedom $2\alpha^*$.

Note that, currently, Jeffrey's prior for $p(\sigma^2) = 1/\sigma^2$ is implemented, which corresponds to the case of $\alpha = \beta = 0$. We are currently implementing more flexible user-defined input for α and β. In particular, in the limit of $\beta = \sigma_0^2\alpha \to \infty$, one recovers the case with a fixed, predefined data noise variance σ_0^2.

◎ `sens:`

This utility performs a series of tasks for for the computation of Sobol indices. Some theoretical background on the statistical estimators employed here is given in Chapter 5. This utility can be used in conjunction with utility `trdSpls` which generates truncated normal or log-normal random samples. It can also be used to generate uniform random samples by selecting a truncated normal distribution and a suitably large standard deviation.

In addition to the `-h` flag, it has the following command line options:

- `-a <action>:` Action to be performed by this utility
  - `splFO:` assemble samples for first order Sobol indices
  - `idxFO:` compute first order Sobol indices
  - `splTO:` assemble samples for total order Sobol indices
  - `idxTO:` compute total order Sobol indices
  - `splJnt:` assemble samples for joint Sobol indices
  - `idxJnt:` compute joint Sobol indices
- `-d <ndim>:` Number of dimensions
- `-n <ndim>:` Number of dimensions
- `-u <spl1>:` name of file holding the first set of samples, nspl×ndim
- `-v <spl2>:` name of file holding the second set of samples, nspl×ndim
- `-x <mev>:` name of file holding model evaluations
- `-p <pfile>:` name of file possibly holding a custom list of parameters for Sobol indices

Python Modules

Bayesian Evidence Estimation

This capability is currently within the UQTk `inference` Python module, and the file is located at `PyUQTk/inference/evidence_solvers.py`.

Let λ denote uncertain model parameters that we are interested in inferring, y the observation data, and $\mathcal{M}$ the assumed model. Bayes' Theorem for the parameter λ conditioned on using the model $\mathcal{M}$ is

$$p(\lambda|y,\mathcal{M}) = \frac{p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})}{p(y|\mathcal{M})}, \tag{4.57}$$

where, with some abuse of notation, $p(\cdot)$ denotes either probability density function (PDF) for a continuous random variable or probability mass function (PMF) for a discrete random variable. Here, $p(\lambda|\mathcal{M})$ is known as the prior, $p(y|\lambda,\mathcal{M})$ the likelihood, $p(\lambda|y,\mathcal{M})$ the posterior, and $p(y|\mathcal{M})$ the evidence.

The evidence is very important for Bayesian model selection. Given a candidate model $\mathcal{M}_k$, we can write Bayes' rule for the *model* as

$$p(\mathcal{M}_k|y) = \frac{p(y|\mathcal{M}_k)p(\mathcal{M}_k)}{p(y)}. \tag{4.58}$$

The ratio of model posteriors between models $\mathcal{M}_1$ and $\mathcal{M}_2$ is then

$$\frac{p(\mathcal{M}_1|y)}{p(\mathcal{M}_2|y)} = \frac{p(y|\mathcal{M}_1)p(\mathcal{M}_1)}{p(y|\mathcal{M}_2)p(\mathcal{M}_2)}. \tag{4.59}$$

If further assuming uniform prior across the models (i.e., $p(\mathcal{M}_1) = p(\mathcal{M}_2)$), it reduces to

$$\frac{p(\mathcal{M}_1|y)}{p(\mathcal{M}_2|y)} = \frac{p(y|\mathcal{M}_1)}{p(y|\mathcal{M}_2)}. \tag{4.60}$$

The RHS of (4.60), being the ratio of model likelihoods (which is also the ratio of evidence terms as defined in (4.57)) is called *Bayes factor* between $\mathcal{M}_1$ and $\mathcal{M}_2$.

Since it is often more numerically stable to work with log values of Bayes' Theorem terms, this module seeks to estimate the natural logarithm of the evidence, $\ln p(y|\mathcal{M})$, given a model $\mathcal{M}$. We describe the available functions below.

◎ `LikelihoodMC_PriorSamples:`

This function estimates the evidence via Monte Carlo marginalization of the likelihood using prior sampling:

$$p(y|\mathcal{M}) = \int_\lambda p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})\,d\lambda \approx \frac{1}{N}\sum_{i=1}^{N} p\left(y|\lambda^{(i)},\mathcal{M}\right). \tag{4.61}$$

Here $\lambda^{(i)} \sim p(\lambda|\mathcal{M})$ are samples drawn from the prior.

Notes: Requires likelihood values for prior samples. May be inefficient if posterior is very "small" compared to prior, adaptive importance sampling recommended.

Inputs:

- `ln_likelihood` — vector of N values of $\ln p(y|\lambda^{(i)}, \mathcal{M})$ corresponding to the prior samples $\lambda^{(i)}$

Outputs:

- $\ln p(y|\mathcal{M})$ estimate

◎ `ImportanceLikelihoodMC_PosteriorSamples:`

This function estimates the evidence via Monte Carlo marginalization of the likelihood using importance sampling:

$$p(y|\mathcal{M}) = \int_\lambda p(y|\lambda, \mathcal{M}) \frac{p(\lambda|\mathcal{M})}{p_b(\lambda|\mathcal{M})} p_b(\lambda|\mathcal{M})\, d\lambda \approx \frac{1}{N} \sum_{i=1}^{N} p\left(y|\lambda^{(i)}, \mathcal{M}\right) \frac{p\left(\lambda^{(i)}|\mathcal{M}\right)}{p_b\left(\lambda^{(i)}|\mathcal{M}\right)}. \quad (4.62)$$

Here $p_b(\lambda|\mathcal{M})$ is a biasing distribution. In this implementation, we choose it to be a Gaussian approximation to the posterior constructed using posterior sample moments, i.e., $p_b(\lambda|\mathcal{M}) = p_G(\lambda|y, \mathcal{M}) \sim \mathcal{N}(\tilde{\mu}_p, \tilde{\Sigma}_p)$ where $\tilde{\mu}_p$ and $\tilde{\Sigma}_p$ are sample mean and covariance computed from posterior samples. $\lambda^{(i)} \sim p_b(\lambda|\mathcal{M})$ are samples drawn from this biasing distribution.

Notes: Requires posterior samples, and the ability to evaluate prior and likelihood PDFs at new points.

The function works in two stages. The first stage involves constructing the biasing distribution and generating samples from that distribution.

Stage 1 inputs:

- `posterior_samples` — array of posterior samples (each row is a sample)
- `n_importance_samples` — number samples requested from the biasing distribution
- `stage` — set to 1 for stage 1

Stage 1 outputs:

- `importance_samples` — array of samples from the biasing distribution (each row is a sample)
- `importance_samples_ln_PDF` — vector of $\ln p_b(\lambda^{(i)}|\mathcal{M})$ values corresponding to these samples

At this point, the user needs to externally compute and provide the ln-prior and ln-likelihood values for these samples and pass them back into the function. The second stage can then estimate the ln-evidence.

Stage 2 inputs:

- `ln_prior` — vector of $\ln p(\lambda^{(i)}|\mathcal{M})$ values corresponding to the biasing samples generated in stage 1
- `ln_likelihood` — vector of $\ln p(y|\lambda^{(i)}|\mathcal{M})$ values corresponding to the biasing samples generated in stage 1
- `ln_importance_input` — pass back in the output `importance_samples_ln_PDF` generated from stage 1 without modifications
- `stage` — set to 2 for stage 2

Stage 2 outputs:

- $\ln p(y|\mathcal{M})$ estimate

◎ `PosteriorGaussian_PosteriorSamples:`

This function estimates the evidence via Gaussian approximation using posterior sample moments:

$$p(y|\mathcal{M}) = \frac{p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})}{p(\lambda|y,\mathcal{M})} \approx \frac{p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})}{\tilde{p}(\lambda|y,\mathcal{M})}. \tag{4.63}$$

Here, $\tilde{p}(\lambda|y,\mathcal{M})$ is an estimate to the posterior constructed from a Gaussian approximation using posterior sample moments, i.e., $p_b(\lambda|\mathcal{M}) = p_G(\lambda|y,\mathcal{M}) \sim \mathcal{N}(\tilde{\mu}_p, \tilde{\Sigma}_p)$ where $\tilde{\mu}_p$ and $\tilde{\Sigma}_p$ are sample mean and covariance computed from posterior samples. The above expression is valid for any λ, and we can evaluate it for each posterior sample we already have; the function returns the mean value of 4.63 evaluated for all such samples.

Notes: Requires posterior samples, and the prior and likelihood PDF values for those samples.

Inputs:

- `posterior_samples` — array of posterior samples (each row is a sample)
- `ln_prior` — vector of $\ln p(\lambda^{(i)}|\mathcal{M})$ values corresponding to the posterior samples
- `ln_likelihood` — vector of $\ln p(y|\lambda^{(i)}|\mathcal{M})$ values corresponding to the posterior samples

Outputs:

- $\ln p(y|\mathcal{M})$ estimate

◎ `Harmonic_PosteriorSamples:`

This function estimates the evidence via the Harmonic approximation formula:

$$p(y|\mathcal{M}) \approx \left\{ \frac{1}{N} \sum_{i=1}^{N} \frac{1}{p\left(y|\lambda^{(i)}, \mathcal{M}\right)} \right\}^{-1}. \tag{4.64}$$

Here $\lambda^{(i)} \sim p(\lambda|y, \mathcal{M})$ are samples from the posterior.

Notes: Requires likelihood values for posterior samples. Poor numerical stability observed, often yields NaN.

Inputs:

- `ln_likelihood` — vector of $\ln p(y|\lambda^{(i)}|\mathcal{M})$ values corresponding to the posterior samples

Outputs:

- $\ln p(y|\mathcal{M})$ estimate

Chapter 5

Examples

The primary intended use for UQTk is as a library that provides UQ functionality to numerical simulations. To aid the development of UQ-enabled simulation codes, some examples of programs that perform common UQ operations with UQTk are provided with the distribution. These examples can serve as a template to be modified for the user's purposes. In some cases, *e.g.* in sampling-based approaches where the simulation code is used as a black-box entity, the examples may provide enough functionality to be used directly, with only minor adjustments. Below is a brief description of the main examples that are currently in the UQTk distribution. For all of these, make sure the environment variable `UQTK_INS` is set and points upper level directory of the UQTk install directory, *e.g.* the keyword *installdir* described in the installation section. This path also needs to be added to environment variable `PYTHONPATH`.

Elementary Operations

Overview

This set of examples is located under `examples/ops`. It illustrates the use of UQTk for elementary operations on random variables that are represented with Polynomial Chaos (PC) expansions.

Description

This example can be run from the command-line:

```
./Ops.x
```

followed by

```
./plot_pdf.py samples.a.dat
./plot_pdf.py samples.loga.dat
```

to plot select probability distributions based on samples from Polynomial Chaos Expansions (PCE) utilized in this example.

Another example compares the Taylor series to the integration approach for computing the natural logarithm of a PCE:

```
./LogComp.x
```

followed by

```
./plot_logs.py
```

to plot the comparison in the pdf of the natural log of a.

The script `test_all.sh` runs through all of these commands.

Ops.x step-by-step

- Wherever relevant the PCSet class implements functions that take either "double *" arguments or array container arguments. The array containers, named "Array1D", "Array2D", and"Array3D", respectively, are provided with the UQTk library to streamline the management of data structures.

1. Instantiate a PCSet class for a 2nd order 1D PCE using Hermite-Gauss chaos.

```
int ord = 2;
int dim = 1;
PCSet myPCSet("ISP",ord,dim,"HG");
```

2. Initialize coefficients for HG PCE expansion $\hat{a}$ given its mean and standard deviation:

```
double ma = 2.0; // Mean
double sa = 0.1; // Std Dev
myPCSet.InitMeanStDv(ma,sa,a);
```

$$\hat{a} = \sum_{k=0}^{P} a_k \Psi_k(\xi), \quad a_0 = \mu, \quad a_1 = \frac{\sigma}{\sqrt{\langle \psi_1^2 \rangle}}, \quad a_2 = a_3 = \ldots = 0$$

3. Initialize $\hat{b} = 2.0\psi_0(\xi) + 0.2\psi_1(\xi) + 0.01\psi_2(\xi)$ and subtract $\hat{b}$ from $\hat{a}$:

```
b[0] = 2.0;
b[1] = 0.2;
b[2] = 0.01;
myPCSet.Subtract(a,b,c);
```

The subtraction is a term by term operation: $c_k = a_k - b_k$

4. Product of PCE's, $\hat{c} = \hat{a} \cdot \hat{b}$:

```
myPCSet.Prod(a,b,c);
```

$$\hat{c} = \sum_{k=0}^{P} c_k \Psi_k(\xi) = \left(\sum_{k=0}^{P} a_k \Psi_k(\xi) \right) \left(\sum_{k=0}^{P} b_k \Psi_k(\xi) \right)$$

$$c_k = \sum_{i=0}^{P} \sum_{j=0}^{P} C_{ijk} a_i b_j, \quad C_{ijk} = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}$$

The triple product C_{ijk} is computed and stored when the PCSet class is instantiated.

5. Exponential of a PCE, $\hat{c} = \exp(\hat{a})$ is computed using a Taylor series approach

```
myPCSet.Exp(a,c);
```

$$\hat{c} = \exp(\hat{a}) = \exp(a_0) \left(1 + \sum_{n=0}^{N_T} \frac{\hat{d}^n}{n!} \right) \tag{5.1}$$

where

$$\hat{d} = \hat{a} - a_0 = \sum_{k=1}^{P} a_k \tag{5.2}$$

The number of terms N_T in the Taylor series expansion are incremented adaptively until an error criterion is met (relative magnitude of coefficients compared to the mean) or the maximum number of terms is reached. Currently, the default relative tolerance and maximum number of Taylor terms are 10^{-6} and 500. This values can be changed by the user using public PCSet methods `SetTaylorTolerance` and `SetTaylorTermsMax`, respectively.

6. Division, $\hat{c} = \hat{a}/\hat{b}$:

```
myPCSet.Div(a,b,c);
```

Internally the division operation is cast as a linear system, see item 4, $\hat{a} = \hat{b} \cdot \hat{c}$, with unknown coefficients c_k and known coefficients a_k and b_k. The linear system is sparse and it is solved with a GMRES iterative solver provided by NETLIB

7. Natural logarithm, $\hat{c} = \log(\hat{a})$:

```
myPCSet.Log(a,c);
```

Currently, two methodologies are implemented to compute the logarithm of a PCE: Taylor series expansion and an integration approach. For more details see Debusschere *et. al.* [4].

8. Draw samples from the random variable $\hat{a}$ represented as a PCE:

```
myPCSet.DrawSampleSet(aa,aa_samp);
```

Currently "Ops.x" draws sample from both $\hat{a}$ and $\log(\hat{a})$ and saves the results to files "samples.a.dat" and "samples.loga.dat", respectively.

9. The directory contains a python script that computes probability distributions from samples via Kernel Density Estimate (KDE, also see Lecture #1) and generates two plots, "samples.a.dat.pdf" and "samples.loga.dat.pdf", also shown in Fig. 5.1.

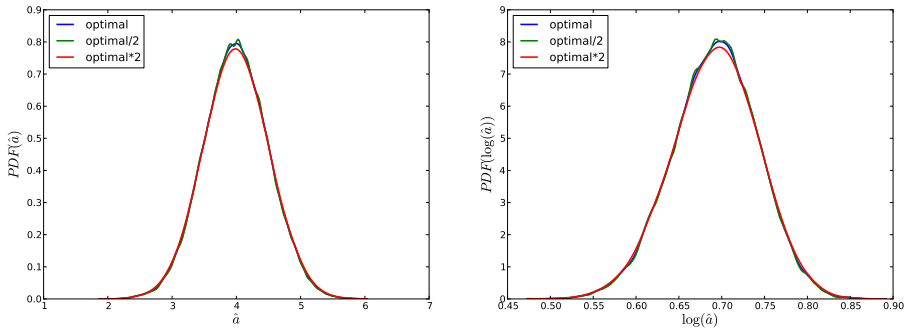


Figure 5.1: Probability densities for $\hat{a}$ and $\log(\hat{a})$ computed via KDE. Results generated using several KDE bandwidths. This feature is available in the Python's SciPy package starting with version 0.11

Polynomial Fitting

Overview

This example is located in `polynomial`. It contains codes to generate a random polynomial data with noise, fit a set of polynomial models to the data using Markov Chain Monte Carlo, comparing the models to each other using model evidence, calculate the derivatives of the models with uncertainties, and produce other plots about the model fits.

Implementation

This workflow has 3 main steps:

1. Getting the data from a random polynomial
   - Ran in `get_data.py`

- Picks random coefficients for a third order polynomial, randomly picks 15 points, and adds gaussian noisy.
- Relevant flags include:
  - `--ix <input.xml>` the name of the input xml file. Default is `<input.xml>`
  - `-g` flag to show a plot with the chosen polynomial and the data points
  - `-e` flag to run with the same coefficients used in this example

2. Fitting the model to the data

   - Ran in `fit.py`
   - Uses Markov Chain Monte Carlo (MCMC) to fit the models to the data
   - Relevant flags include:
     - `--ix <input.xml>` the name of the input xml file. Default is `input.xml`
     - `-w <output_file>` the name of the output file. Results will be printed to this file along with the command line. Default is `output.txt`.

3. Postprocessing

   - Ran in `post.py`
   - Makes various types of plots and performs various calculations from the MCMC results.
   - Relevant flags include:
     - `--ix <input.xml>` the name of the input xml file. Default is `<input.xml>`
     - `-p` flag to show the posterior plots
     - `-g` flag to show the parameter graphs
     - `-d` flag to calculate the derivatives and their uncertainties, and to make a plot.
     - `-v <verbosity>` verbosity level. Default is 1
     - `--interactive` flag to show plots interactively. Default is False
     - `--jpeg` flag to save all plots as .jpg. Default is to save as .pdf
     - `--evidence` flag to calculate the evidence values of each model and to make a plot of all
   - Plots to view:
     - `polynomial_all_fits.pdf` shows the fits of all the models, along with the true solution and the data used to fit the model.
     - `polynomial_all_fits_with_error.pdf` shows the fits of all the models with error bars visualizing standard deviation.
     - `polynomial_all_fits_with_error_shaded.pdf` shows the fits of all the models with shaded regions visualizing standard deviation, the true solution, and the data used to fit the model.

  - `polynomial_derivatives.pdf` shows the derivatives of all the models, with mean and standard deviation.
  - `polynomial_importance_evidence.pdf` shows the log evidence values of all the models as calculated using Importance samping.
  - `*_parameter_graphs.pdf` shows the MCMC chains of all the parameters, after the burnin and with the stride.
  - `*_model_data_agreement_xy_with_real.pdf` shows the model with the MAP parameters, the real polynomial, and the data points.

Other relevant files include:

- `input.xml`
  - The input xml file where all relevant information for the fitting is stored.
- `tools.py`
  - File where all tools for fitting are stored.
  - Most notable is the class for the models.
- `graph_tools.py`
  - File where all helper functions to plot different graphs are stored.
  - These functions are general enough to be used for a variety of applications.
- `full_run.sh`
  - Example of entire workflow run. Has all necessary flags to run the complete example

Example Outputs

This example run will do all components of `full_run.sh` step by step. You can run each part individually or `full_run.sh` to perform all components at once.

Start in `/run`

- `../scripts/get_data.py --ix input.xml -g -e` For the example, the coefficients are fixed to $[0, 0.15, -0.65, 0.5]$, running without the `-e` flag will give 4 random integers in the range $[-10, 10]$ for the coefficients. It will then choose 15 random points from the range $[0, 1]$ and add gaussian noise. Fig (5.2) shows the sample graph of the polynomial and chosen data points. From the sample output shown in Fig (5.3). You can see the files that the outputs are stored in and the real coefficients of the polynomial.

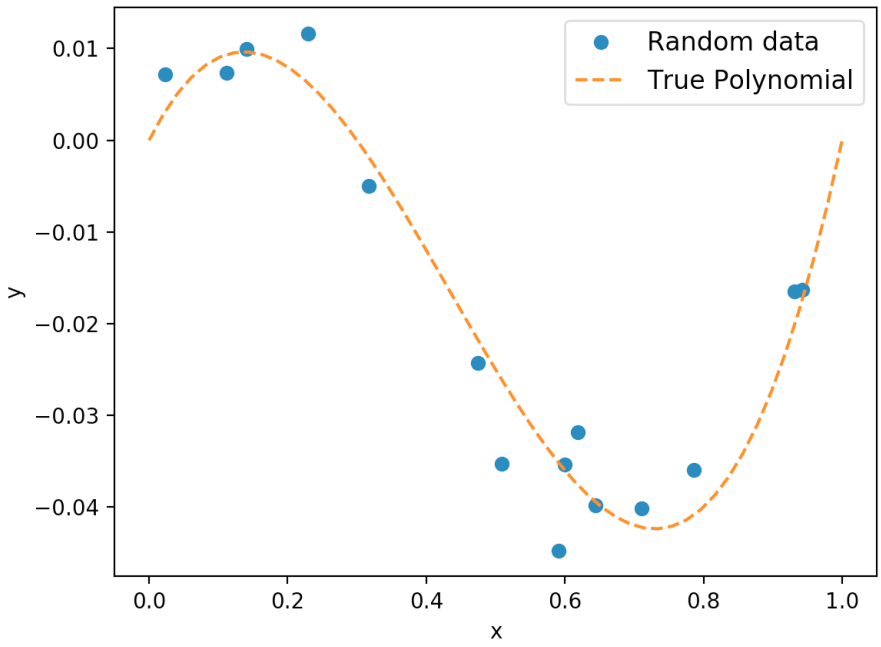


Figure 5.2: Example output of get_data.py -g -e

```
Read in the file input.xml for run settings.
Data saved to  x_y_data.csv
coefficients saved to  coeff.csv
coefficients =  [0, 0.15, -0.65, 0.5]
```

Figure 5.3: Command line output of get_data.py

```
Running for model model_A
Making object for  model_A
Scaling down the proposal at step 49
Scaling down the proposal at step 99
Scaling down the proposal at step 149
Scaling down the proposal at step 849
MCMC sample size: (50000, 2)
Overall acceptance rate: 0.42984
Fraction of samples outside of prior bounds: 0.0
MAP parameter set:
a :  1.278484e-02
b : -5.529389e-02
```

Figure 5.4: Command line output of fit.py

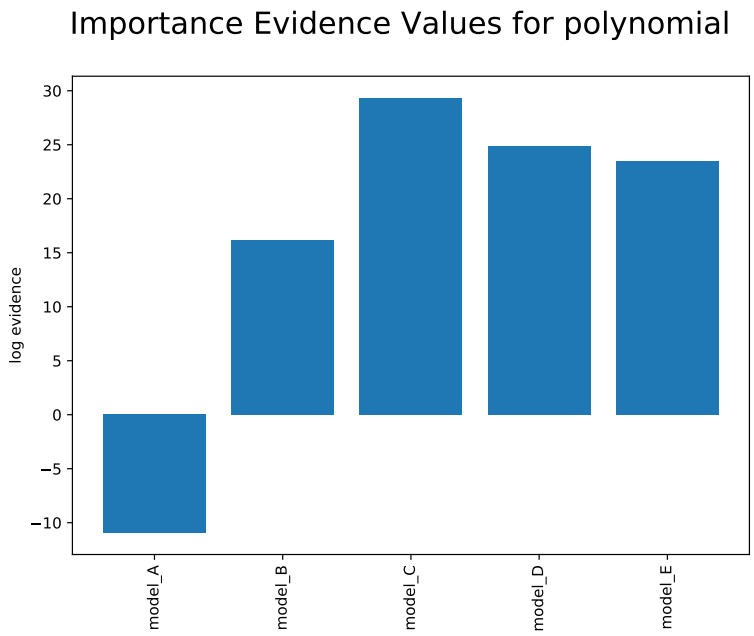


Figure 5.5: Importance Evidence Values for the Polynomial Model. As you can see, model C has the highest evidence value, implying the best fit. This is good because our true solution is of order 3.

- `../scripts/fit.py --ix input.xml -w output.txt`

  This will use MCMC to fit all the models to the data. Fig (5.4) shows a sample output for one of the models. A very similar output will also print out for all other tested models. This script will also produce the files `MCMC_samples_polynomial_mA.dat` for all models. These files store the MCMC sample that will be processed in the next stop.

- `../scripts/post.py -p -g --evidence -d`

  This will run the post processing with all of the common flag options. Many plots will be produced including fitting graphs, derivatives graphs, and evidence value graphs. Figs (5.5) and (5.6) show some examples. There are also many more types of graphs that are produced. See "Plots to view" in the implementation section for a description of all plots produced.

Troubleshooting

- If `get_data.py` does not produce a good example polynomial:
  - Try running `get_data.py` a few times
  - Try changing `error_level` in the .xml file, probably to a lower value
  - Try changing the `size_range` in the .xml file

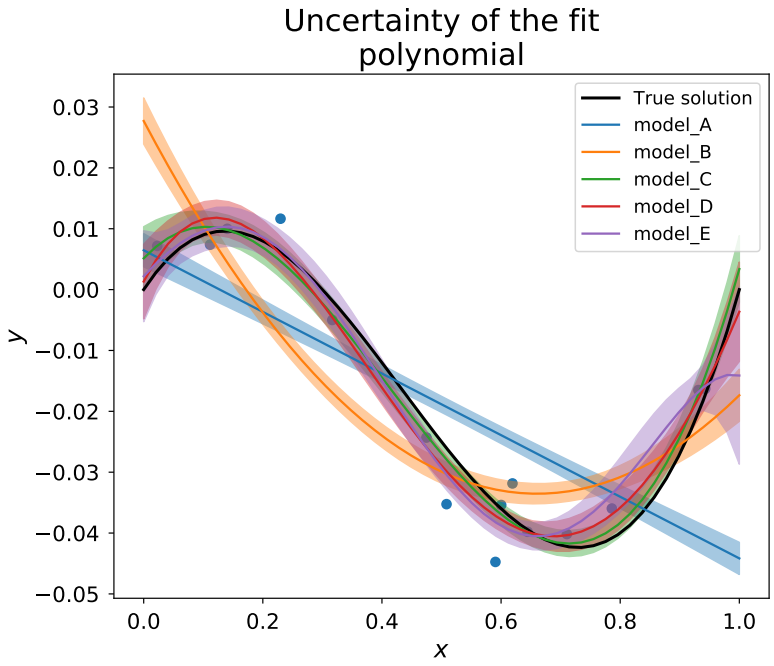


Figure 5.6: Here is the fitted models with uncertainties for all models. The shaded regions show 1 standard deviation. You can also see the true solution and the data used.

- If the MCMC chain is not mixing well, accepting too many/few samples:
  - This is a very common place that adjustments will need to be made. Because we are considering random data that is different each time, there many be a considerable amount of variability in the acceptance rates and mixing of the chains.
  - Try changing the value of `gamma`, increasing gamma will typically decrease your acceptance rate, and decreasing gamma will typically increase your acceptance rate.
  - Try increasing the number of samples, and making a longer burn-in period.
  - Try changing the initial stating point of the chain

Customizing the code to your model

To customize this workflow to your own model, you only need to change `input.xml` and `tools.py`.

In `input.xml`, you need to enter all relevant information about the case and the model. Follow the same format, and see comments in file for all necessary information.

In `tools.py`, you need to make a new class for your models. To make a model with the same format as the example models, all you need to do is make a child class of `model_letter`, with your desired prediction function. If desired, you can also add the `compute_derivative`

function to calculate the derivative of the model. This function can also be edited to calculate any other desired derived quantity. You also need to edit the `make_model_object` function in order to make the appropriate type of model object.

Forward Propagation of Uncertainty

Overview

- Located in `examples/surf_rxn`
- Several examples of propagating uncertainty in input parameters through a model for surface reactions, consisting of three Ordinary Differential Equations (ODEs). Two approaches are illustrated:
  - Direct linking to the C++ UQTk libraries from a C++ simulation code:
    - Propagation of input uncertainties with Intrusive Spectral Projection (ISP), Non Intrusive Spectral Projection (NISP) via quadrature , and NISP via Monte Carlo (MC) sampling.
    - For more documentation, see a detailed description below
    - An example can be run with `./forUQ_sr.py`
  - Using simulation code as a black box forward model:
    - Propagation of uncertainty in one input parameter with NISP quadrature approach.
    - For more documentation, see a detailed description below
    - An example can be run with `./forUQ_BB_sr.py`

Simulation Code Linked to UQTk Libraties

The example script `forUQ_sr.py`, provided with this example can perform parametric uncertainty propagation using three methods

- *NISP*: Non-intrusive spectral projection using quadrature integration
- *ISP*: Intrusive spectral projection
- *NISP_MC*: Non-intrusive spectral projection using Monte-Carlo integration

The command-line usage for this example is

```
./forUQ_sr.py <pctype> <pcord> <method1> [<method2>]  [<method3>]
```

The script requires the xml input template file *forUQ_surf_rxn.in.xml.templ*. In this template, the default setting for param_b is uncertain normal random variable with a standard deviation set to 10% of the mean.

The following parameters are defined at the beginning of the file:

- *pctype*: The type of PC, supports 'HG', 'LU', 'GLG', 'JB'
- *pcord*: The order of output PC expansion
- *methodX*: NISP, ISP or NISP_MC
- *nsam*: Number of samples requested for NISP Monte-Carlo (currently hardwired in the script)

Description of Non-Intrusive Spectral Projection utilities (*SurfRxnNISP.cpp and SurfRxnNISP_MC.cpp*)

$$f(\vec{\xi}) = \sum_k c_k \Psi_k(\vec{\xi}) \qquad\qquad c_k = \frac{\langle f(\vec{\xi})\Psi_k(\vec{\xi})\rangle}{\langle \Psi_k^2(\vec{\xi})\rangle}$$

$$\langle f(\vec{\xi})\Psi_k(\vec{\xi})\rangle = \int f(\vec{\xi})\Psi_k(\vec{\xi})\pi(\vec{\xi})d\vec{\xi} \approx \underbrace{\left[\sum_q f(\vec{\xi}_q)\Psi_k(\vec{\xi}_q)w_q\right]}_{NISP} \text{ or } \underbrace{\left[\frac{1}{N}\sum_s f(\vec{\xi}_s)\Psi_k(\vec{\xi}_s)\right]}_{NISP\_MC}$$

These codes implement the following workflows

1. Read XML file
2. Create a PC object with or without quadrature
   - NISP: `PCSet myPCSet("NISP",order,dim,pcType,0.0,1.0)`
   - NISP_MC: `PCSet myPCSet("NISPnoq",order,dim,pcType,0.0,1.0)`
3. Get the quadrature points or generate Monte-Carlo samples
   - NISP: `myPCSet.GetQuadPoints(qdpts)`
   - NISP_MC: `myPCSet.DrawSampleVar(samPts)`
4. Create input PC objects and evaluate input parameters corresponding to quadrature points
5. Step forward in time
   - Collect values for all input parameter samples

- Perform Galerkin projection or Monte-Carlo integration
- Write the PC modes and derived first two moments to files

Description of Intrusive Spectral Projection utility (*SurfRxnISP.cpp*)

This code implement the following workflows

1. Read XML file
2. Create a PC object for intrusive propagation
   `PCSet myPCSet("ISP",order,dim,pcType,0.0,1.0)`
3. Represent state variables and all parameters with their PC coefficients
   - $u \to \{u_k\}$, $v \to \{v_k\}$, $w \to \{w_k\}$, $z \to \{z_k\}$,
   - $a \to \{a_k\}$, $b \to \{b_k\}$, $c \to \{c_k\}$, $d \to \{d_k\}$, $e \to \{e_k\}$, $f \to \{f_k\}$.
4. Step forward in time according to PC arithmetics, *e.g.*
   $a \cdot u \to \{(a \cdot u)_k\}$ with

$$a \cdot u = \left(\sum_i a_i \Psi_i(\vec{\xi})\right)\left(\sum_j u_j \Psi_j(\vec{\xi})\right) = \sum_k \underbrace{\left(\sum_{i,j} a_i u_j \frac{\langle \Psi_i \Psi_j \Psi_k \rangle}{\langle \Psi_k^2 \rangle}\right)}_{(a \cdot u)_k} \Psi_k(\vec{\xi})$$

Postprocessing Utilities - time series

```
./plSurfRxnMstd.py NISP
./plSurfRxnMstd.py ISP
./plSurfRxnMstd.py NISP_MC
```

These commands plot the time series of mean and standard deviations of all three species with all three methods. Sample results are shown in Fig. 5.7.

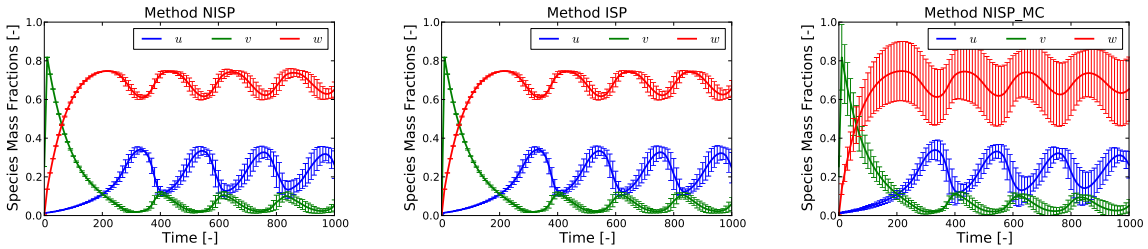


Figure 5.7: Time series of mean and standard deviations for u, v, and w with NISP, ISP, and NISP_MC, respectively.

Postprocessing Utilities - PDFs

```
./plPDF_method.py <species>  <qoi> <pctype> <pcord> <method1> [<method2>] [<method3>]
```

e.g.

```
./plPDF_method.py u ave HG 3 NISP ISP
```

This script samples the PC representations, then computes the PDFs of time-average (ave) or the final time value (tf) for all three species. Sample results are shown in Fig. 5.8.

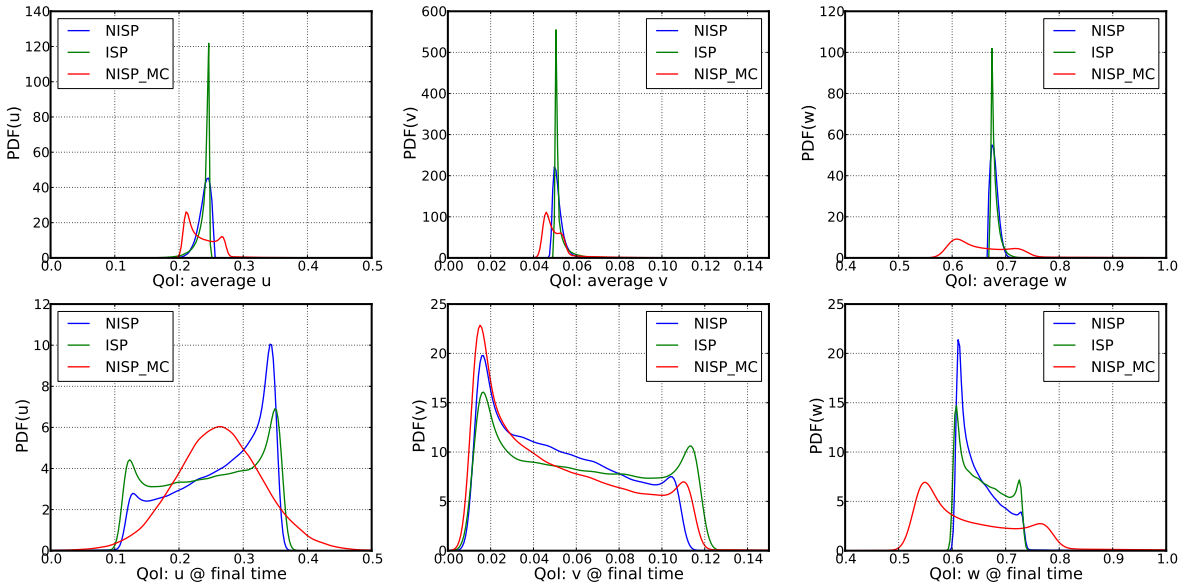


Figure 5.8: PDFs for u, v, and w; Top row shows results for average u, v, and w; Bottom row shows results corresponding to values at the last integration step (final time).

Simulation Code Employed as a Black Box

The command-line usage for the script implementing this example is given as

```
./forUQ_BB_sr.py --nom nomvals -s stdfac -d dim -l lev -o ord -q sp --npdf npdf
          --npces npces
```

The following parameters can be controlled by the user

- *nomvals*: List of nominal parameter values, separated by comma if more than one value, and no spaces. Default is one value, 20.75
- *stdfac*: Ratio of standard deviation/nominal parameter values. Default value: 0.1
- *dim*: number of uncertain input parameters. Currently this example can only handle $dim = 1$
- *lev*: No. of quadrature points per dimension (for full quadrature) or sparsity level (for sparse quadrature). Default value: 21.

- *ord*: PCE order. Default value: 20
- *sp*: Quadrature type "full" or "sparse". Default value: "full"
- *npdf*: No. of grid points for Kernel Density Estimate evaluations of output model PDF's. Default value 100
- *npces*: No. of PCE evaluations to estimate output densitie. Default value 10^5

  Note: This example assumes Hermite-Gauss chaos for the model input parameters.

This script uses the following utilities, located in the *bin* directory under the UQTk installation path

- *generate_quad*: Generate quadrature points for full/sparse quadrature and several types of rules.
- *pce_rv*: Generate samples from a random variable defined by a Polynomial Chaos expansion (PCE)
- *pce_eval*: Evaluates PCE for germ samples saved in input file "xdata.dat".
- *pce_resp*: Constructs PCE by Galerkin projection

Sequence of computations:

1. *forUQ_BB_sr.py*
   saves the input parameters' nominal values and standard deviations in a diagonal matrix format in file "pcfile". First it saves the matrix of nominal values, then the matrix of standard deviations. This information is sufficient to define a PCE for a normal random variable in terms of a standard normal germ. For a one parameter problem, this file has two lines.
2. *generate_quad*:
   Generate quadrature points for full/sparse quadrature and several types of rules. The usage with default script arguments `generate_quad -d1 -g'HG' -xfull -p21 > logQuad.dat`
   This generates Hermite-Gauss quadrature points for a 21-point rule in one dimension. Quadrature points locations are saved in "qdpts.dat" and weights in "wghts.dat" and indices of points in the 1D space in "indices.dat". At the end of "generate_quad" execution file "qdpts.dat" is copied over "xdata.dat"
3. *pce_eval*:
   Evaluates PCE of input parameters at quadrature points, saved previously in "xdata.dat". The evaluation is dimension by dimension, and for each dimension the corresponding column from "pcfile" is saved in "pccf.dat". See command-line arguments below.

```
pce_eval -x'PC' -p1 -q1 -f'pccf.dat' -sHG >> logEvalInPC.dat
```

At the end of this computation, file "input.dat" contains a matrix of PCE evaluations. The number of lines is equal to the number of quadrature points and the number of columns to the dimensionality of input parameter space.

4. *Model evaluations*:

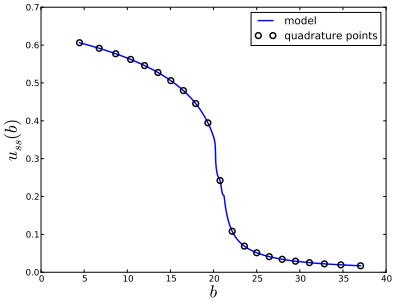
```
funcBB("input.dat","output.dat",xmltpl="surf_rxn.in.xml.tp3",
       xmlin="surf_rxn.in.xml")
```

The Python function *"funcBB"* is defined in file "prob3_utils.py". This evaluates the forward model at sets of input parameters in file "input.dat" and saves the model output in "output.dat". For each model evaluation, specific parameters are inserted in the xml file "surf_rxn.in.xml" which is a copy of the template in "surf_rxn.in.xml.tp3". At the end "output.dat" is copied over "ydata.dat"

5. *pce_resp*:

```
pce_resp -xHG -o20 -d1 -e > logPCEresp.dat
```

Computes a Hermite-Gauss PCE of the model output via Galerkin projection. The model evaluations are taken from "ydata.dat", and the quadrature point locations from "xdata.dat". PCE coefficients are saved in "PCcoeff_quad.dat", the multi-index list in "mindex.dat" and these files are pasted together in "mipc.dat"



(average u as a function of parameter b values. Location of quadrature points is shown with circles.)

6. *pce_rv*:

```
pce_rv -w'PCvar' -xHG -d1 -n100 -p1 -q0 > logPCrv.dat
```

Draw a 100 samples from the germ of the HG PCE. Samples are saved in file "rvar.dat" and also copied to file "xdata.dat"

7. *pce_eval*:

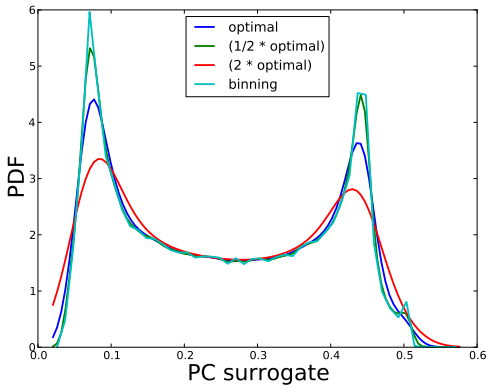
```
pce_eval -x'PC' -p1 -q1 -f'pccf.dat' -sHG >> logEvalInPCrnd.dat
```

See item 3 for details. Results are saved "input_val.dat".

8. Evaluate both the forward model (through the black-box script "funcBB", see item 4) and its PCE surrogate (see item 3) and save results to files "output_val.dat" and

"output_val_pc.dat". Compute L_2 error between the two sets of values using function "compute_err" defined in "utils.py"

9. Sample output PCE and plot the PDF of these samples computed using either a Kernel Density Estimate approach with several kernel bandwidths or by binning:



Numerical Integration

Overview

This example is located in `examples/num_integ`. It contains a collection of Python scripts that can be used to perform numerical integration on six Genz functions: oscillatory, exponential, continuous, Gaussian, corner-peak, and product-peak. Quadrature and Monte Carlo integration methods are both employed in this example.

Theory

In uncertainty quantification, forward propagation of uncertain inputs often involves evaluating integrals that cannot be computed analytically. Such integrals can be approximated numerically using either a random or a deterministic sampling approach. Of the two integration methods implemented in this example, quadrature methods are deterministic while Monte Carlo methods are random.

Quadrature Integration

The general quadrature rule for integrating a function $u(\xi)$ is given by:

$$\int u(\xi)d\xi \approx \sum_{i=1}^{N_q} q^i u(\xi^i) \tag{5.3}$$

where the N_q ξ^i are quadrature points with corresponsing weights q^i.

The accuracy of quadrature integration relies heavily on the choice of the quadrature points. There are countless quadrature rules that can be used to generate quadrature points, such as Gauss-Hermite, Gauss-Legendre, and Clenshaw-Curtis.

When performing quadrature integration, one can use either full tensor product or sparse quadrature methods. While full tensor product quadrature methods are effective for functions of low dimension, they suffer from the curse of dimensionality. Full tensor product quadrature integration methods require N^d quadrature points to integrate a function of dimension d with N quadrature points per dimension. Thus, for functions of high dimension the number of quadrature points required quickly becomes too large for these methods to be practical. Therefore, in higher dimensions sparse quadrature approaches, which require far fewer points, are utilized. When performing sparse quadrature integration, rather than determining the number of quadrature points per dimension, a level is selected. Once a level is selected, the total number of quadrature points can be determined from the dimension of the function. For more information on quadrature integration see `reference here`.

Monte Carlo Integration

One random sampling approach that can be used to evaluate integrals numerically is Monte Carlo integration. To use Monte Carlo integration methods to evaluate the integral of a general function $u(\xi)$ on the d-dimensional $[0,1]^d$ the following equation can be used:

$$\int u(\xi)d\xi \approx \frac{1}{N_s}\sum_{i=1}^{N_s} u(\xi^i) \tag{5.4}$$

The N_s ξ^i are random sampling points chosen from the region of integration according to the distribution of the inputs. In this example, we are assuming the inputs have uniform distribution. One advantage of using Monte Carlo integration is that any number of sampling points can be used, while quadrature integration methods require a certain number of sampling points. One disadvantage of using Monte Carlo integration methods is that there is slow convergence. However, this $O(\frac{1}{\sqrt{N_s}})$ convergence rate is independent of the dimension of the integral.

Genz Functions

The functions being integrated in this example are six Genz functions, and they are integrated over the d-dimensional $[0,1]^d$. These functions, along with their exact integrals, are defined as follows. The Genz parameters w_i represent weight parameters and u_i represent shift parameters. In the current example, the parameters w_i and u_i are set to 1, with one exception. The parameters w_i and u_i are instead set to 0.1 for the Corner-peak function in the `sparse_quad.py` file.

Model	Formula: $f(\lambda)$	Exact Integral: $\int\limits_{[0,1]^d} f(\lambda)d\lambda$
Oscillatory	$\cos(2\pi u_1 + \sum\limits_{i=1}^{d} w_i\lambda_i)$	$\cos\left(2\pi u_1 + \frac{1}{2}\sum\limits_{i=1}^{d} w_i\right) \prod\limits_{i=1}^{d} \frac{2\sin(\frac{w_i}{2})}{w_i}$
Exponential	$\exp(\sum\limits_{i=1}^{d} w_i(\lambda_i - u_i))$	$\prod\limits_{i=1}^{d} \frac{1}{w_i}(\exp(w_i(1-u_i)) - \exp(-w_i u_i))$
Continuous	$\exp(-\sum\limits_{i=1}^{d} w_i\lvert\lambda_i - u_i\rvert)$	$\prod\limits_{i=1}^{d} \frac{1}{w_i}(2 - \exp(-w_i u_i) - \exp(w_i(u_i-1)))$
Gaussian	$\exp(-\sum\limits_{i=1}^{d} w_i^2(\lambda_i - u_i)^2)$	$\prod\limits_{i=1}^{d} \frac{\sqrt{\pi}}{2w_i}(\mathrm{erf}(w_i(1-u_i)) + \mathrm{erf}(w_i u_i))$
Corner-peak	$(1 + \sum\limits_{i=1}^{d} w_i\lambda_i)^{-(d+1)}$	$\frac{1}{d!\prod\limits_{i=1}^{d} w_i} \sum\limits_{r\epsilon\{0,1\}^d} \frac{(-1)^{\lvert\lvert r\rvert\rvert_1}}{1+\sum\limits_{i=1}^{d} w_i r_i}$
Product-peak	$\prod\limits_{i=1}^{d} \frac{w_i^2}{1+w_i^2(\lambda_i-u_i)^2}$	$\prod\limits_{i=1}^{d} w_i(\arctan(w_i(1-u_i)) + \arctan(w_i u_i))$

Implementation

The script set consists of three files:

- `full_quad.py`: a script to compare full quadrature and Monte Carlo integration methods.
- `sparse_quad.py`: a script to compare sparse quadrature and Monte Carlo integration methods.
- `quad_tools.py`: a script containing functions called by `full_quad.py` and `sparse_quad.py`.

full_quad.py

This script will produce a graph comparing full quadrature and Monte Carlo integration methods. Use the command `./full_quad.py` to run this file. Upon running the file, the user will be prompted to select a model from the Genz functions listed.

```
Please enter desired model from choices:
genz_osc
genz_exp
genz_cont
genz_gaus
genz_cpeak
genz_ppeak
```

The six functions listed correspond to the Genz functions defined above. After the user selects the desired model, he/she will be prompted to enter the desired dimension.

```
Please enter desired dimension:
```

The dimension should be entered as an integer without any decimal points. As full quadrature integration is being implemented, this script should not be used for functions of high dimension. If you wish to integrate a function of high dimension, instead use `sparse_quad.py`.

After the user enters the desired dimension, she/he will be prompted to enter the desired maximum number of quadrature points per dimension.

```
Enter the desired maximum number of quadrature points per dimension:
```

Again, this number should be entered as an integer without any decimal points. Several quadrature integrations will be performed, with the first beginning with 1 quadrature point per dimension. For subsequent quadrature integrations, the number of quadrature points will be incremented by one until the maximum number of quadrature points per dimension, as specified by the user, is reached. For example, if the user has requested a maximum of 4 quadrature points per dimension, 4 quadrature integrations will be performed: one with 1 quadrature point per dimension, another with 2 quadrature points per dimension, a third with 3 quadrature points per dimension, and a fourth with 4 quadrature points per dimension.

Next, the script will call the function `generate_qw` from the `quad_tools.py` script to generate quadrature points as well as the corresponding weights.

Then, the exact integral for the chosen function is computed by calling the `integ_exact` function in `quad_tools.py`. This function calculates the exact integral according to the formulas found in the above **Theory** section. The error between the exact integral and the quadrature approximation is then calculated and stored in a list of errors.

Now, for each quadrature integration performed, a Monte Carlo integration is also performed with the same number of sampling points as the total number of quadrature points. To account for the random nature of the Monte Carlo sampling approach, ten Monte Carlo integrations are performed and their errors from the exact integral are averaged. To perform these Monte Carlo integrations and calculate the error in these approximations, the function `find_error` found in `quad_tools.py` is called. Although we are integrating over $[0,1]^d$, the sampling points will be uniformly random points in $[-1,1]^d$. We do this so the same function `func` can be used to evaluate the model at these points and the quadrature points, which are generated in $[-1,1]^d$. The function `func` takes points in $[-1,1]^d$ as input and maps these points to points in $[0,1]^d$ before the function is evaluated at these new points

Finally, the data from both the quadrature and Monte Carlo integrations are plotted. A log-log graph is created that displays the total number of sampling points versus the absolute error in the integral approximation. The graph will be displayed and will be saved as `quad_vs_mc.pdf` as well.

sparse_quad.py

This script is similar to the `full_quad.py` file and will produce a graph comparing sparse quadrature and Monte Carlo integration methods. Sparse quadrature integration rules should be utilized for functions of high dimension, as they do not obey full tensor product rules. Use the command `./sparse_quad.py` to run this script. Upon running the file, the user will be prompted to select a model from the Genz functions listed.

```
Please enter desired model from choices:
genz_osc
genz_exp
genz_cont
genz_gaus
genz_cpeak
genz_ppeak
```

After the user selects the desired model, he/she will be prompted to enter the desired dimension.

```
Please enter desired dimension:
```

The dimension should be entered as an integer without any decimal points. After the user enters the desired dimension, she/he will be prompted to enter the maximum desired level.

```
Enter the maximum desired level:
```

Again, this number should be entered as an integer without any decimal points. Multiple quadrature integrations will be performed, with the first beginning at level 1. For subsequent quadrature integrations, the level will increase by one until the maximum desired level, as specified by the user, is reached.

Next, the script will call the function `generate_qw` from the `quad_tools.py` script to generate quadrature points as well as the corresponding weights.

Then, the exact integral for the chosen function is computed by calling the `integ_exact` function in `quad_tools.py`. The error between the exact integral and the quadrature approximation is then calculated and stored in a list of errors.

Now, for each quadrature integration performed, a Monte Carlo integration is also performed with the same number of sampling points as the total number of quadrature points. This is done in the same manner as in the `full_quad.py` script.

Lastly, the data from both the sparse quadrature and Monte Carlo integration are plotted. A log-log graph is created that displays the total number of sampling points versus the absolute error in the integral approximation. The graph will be displayed and will be saved as `sparse_quad.pdf`.

quad_tools.py

This script contains four functions called by the `full_quad.py` and `sparse_quad.py` files.

- `generate_qw(ndim,param,sp='full',type='LU')`: This function generates quadrature points and corresponding weights. The quadrature points will be generated in the d-dimensional $[-1,1]^d$.
  - *ndim*: The number of dimensions as specified by the user.
  - *param*: Equal to the number of quadrature points per dimension when full quadrature integration is being performed. When sparse quadrature integration is being performed, *param* represents the level.
  - *sp*: The sparsity, which can be set to either full or sparse. The default is set as `sp='full'`, and to change to sparse quadrature one can pass `sp='sparse'` as a parameter to the function.
  - *type*: The quadrature type. The default rule is Legendre-Uniform ('LU'). To change the quadrature type, one can pass a different type to the function. For example, to change to a Gauss-Hermite quadrature rule, pass `type='HG'` to the function. For a complete list of the available quadrature types see the `generate_quad` subsection in the `Applications` section of Chapter 3 of the manual.
- `func(xdata,model,func_params)`: This function evaluates the Genz functions at the selected sampling points.
  - *xdata*: These will either be the quadrature points generated by `generate_qw` or the uniform random points generated in the `find_error` function. The points specified as *xdata* into this function will be in $[-1,1]^d$ and thus will first be mapped to points in $[0,1]^d$ before the function can be evaluated at these new points.
  - *model*: The Genz function specified by the user.
  - *func_params*: The parameters, w_i and u_i, of the Genz function selected. In the `full_quad.py` file, all Genz parameters are set to 1. In the `sparse_quad.py` file, all Genz parameters are set to 1 for all models except `genz_cpeak`. For the `genz_cpeak` model, the Genz parameters are set to 0.1.
- `integ_exact(model,func_params)`: This function computes the exact integral $\int\limits_{[0,1]^d} f(\lambda)d\lambda$ of the selected Genz function, $f(\lambda)$.
  - *model*: The Genz function selected by the user.
  - *func_params*: The parameters, w_i and u_i, of the Genz function selected. In the `full_quad.py` file, all Genz parameters are set to 1. In the `sparse_quad.py` file, all Genz parameters are set to 1 for all models except `genz_cpeak`. For the `genz_cpeak` model, the Genz parameters are set to 0.1.

- `find_error`: This function performs 10 Monte Carlo integrations, and returns their average error from the exact integral. The function takes inputs: *pts, ndim, model, integ_ex, and func_params*.
  - *pts*: The number of uniform random points that will be generated. Equal to the total number of quadrature points used.
  - *ndim*: The number of dimensions as specified by the user.
  - *model*: The Genz function selected by the user.
  - *integ_ex*: The exact integral $\int_{[0,1]^d} f(\lambda) d\lambda$ of the selected Genz function returned by the `integ_exact` function.
  - *func_params*: The parameters, w_i and u_i, of the Genz function selected. In the `full_quad.py` file, all Genz parameters are set to 1. In the `sparse_quad.py` file, all Genz parameters are set to 1 for all models except `genz_cpeak`. For the `genz_cpeak` model, the Genz parameters are set to 0.1.

Sample Results

Try running the `full_quad.py` file with the following input:

```
Please enter desired model from choices:
genz_osc
genz_exp
genz_cont
genz_gaus
genz_cpeak
genz_ppeak

genz_exp
Please enter desired dimension: 5
Enter the desired maximum number of quadrature points per dimension: 10
```

Your graph should look similar to the one in the figure below. Although the Monte Carlo integration curve may vary due to the random nature of the sampling, your quadrature curve should be identical to the one pictured.

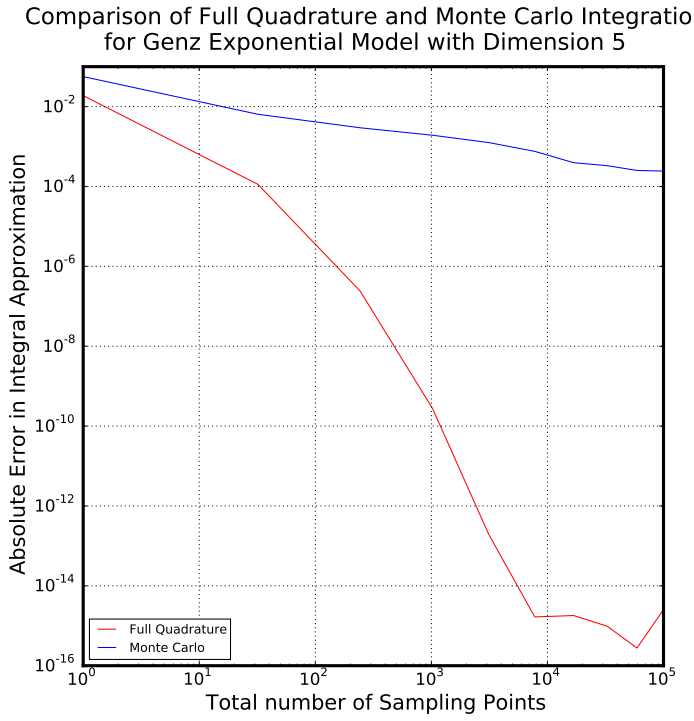


Figure 5.9: Sample results of full_quad.py

Now try running the `sparse_quad.py` file with the following input:

```
Please enter desired model from choices:
genz_osc
genz_exp
genz_cont
genz_gaus
genz_cpeak
genz_ppeak

genz_cont
Please enter desired dimension: 14
Enter the maximum desired level: 4
```

While the quadrature integrations are being performed, the current level will be printed to your screen. Your graph should look similar to the figure below. Again, the Monte Carlo curve may differ but the quadrature curve should be the same as the one pictured.

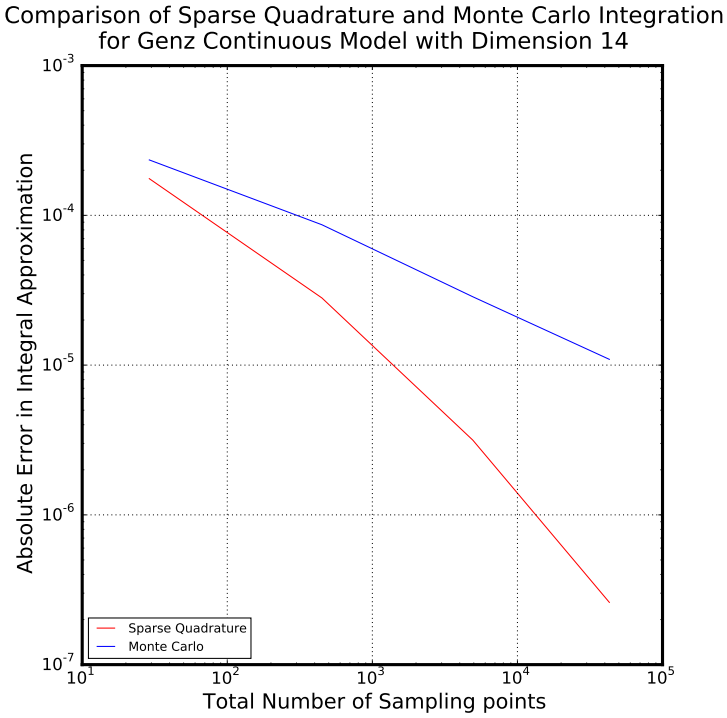


Figure 5.10: Samples results of sparse_quad.py

Next, try running `full_quad.py` with a quadrature rule other than the default Legendre-Uniform. Locate the line in `full_quad.py` that calls the function `generate_quad`. It should read:

```
xpts,wghts=generate_qw(ndim,quad_param)
```

Now, change this line to read:

```
xpts,wghts=generate_qw(ndim,quad_param, type= 'CC')
```

This will change the quadrature rule to Clenshaw-Curtis. Then run the file with input: `genz_gaus, 5, 10`. Sample results can be found in the figure below.

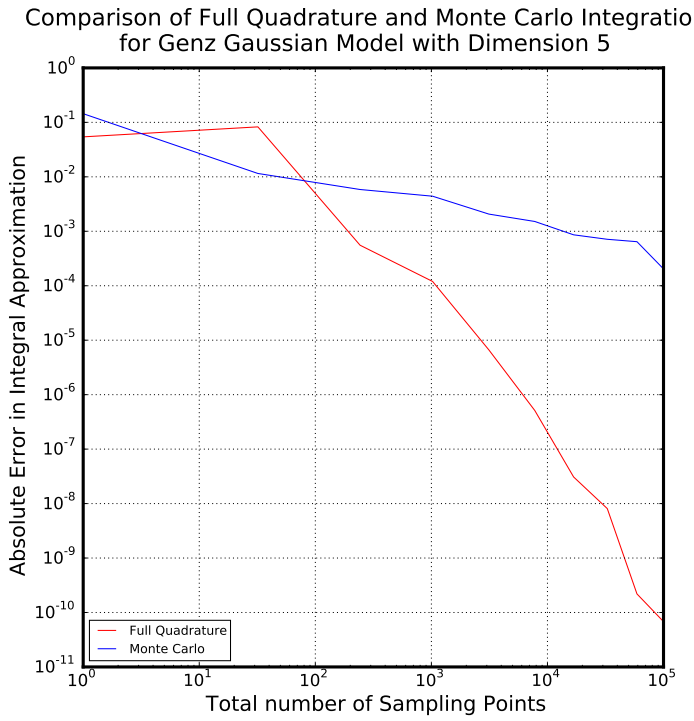


Figure 5.11: Sample results of full_quad.py with Clenshaw-Curtis quadrature rule.

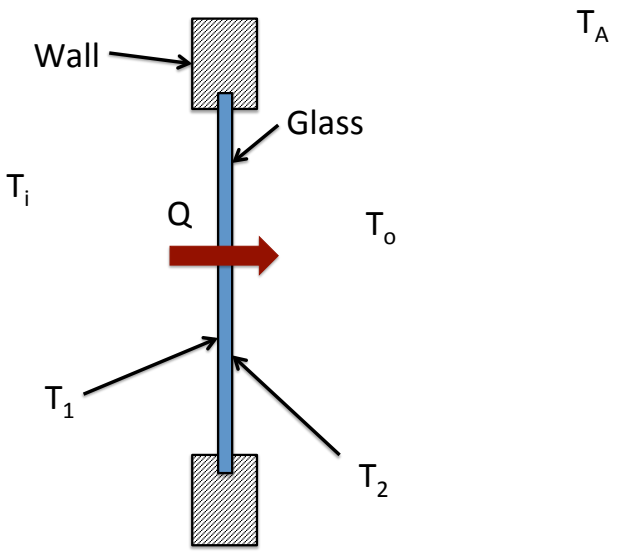
Forward Propagation of Uncertainty with PyUQTk

Overview

This example is located in `examples/fwd_prop`. It contains a pair of Python scripts that propagate uncertainty in input parameters through a heat transfer model using both a Monte Carlo sampling approach and a non-intrusive spectral projection (NISP) via quadrature methods.

Theory

Heat Transfer



In this example, the heat transfer through a window is calculated using samples of seven uncertain Gaussian parameters. These parameters, along with their means and standard deviations are defined below.

Parameter	Mean	Standard deviation (%)
T_i: Room temperature	293 K	0.5
T_o: Outside temperature	273 K	0.5
d_w: Window thickness	0.01 m	1
k_w: Window conductivity	1 W/mK	5
h_i: Inner convective heat transfer coefficient	2 W/m^{2}K	15
h_o: Outer convective heat transfer coefficient	6 W/m^{2}K	15
T_A: Atmospheric temperature	150 K	10

Once we have samples of the 7 parameters, the following forward model is used to calculate heat flux Q:

$$Q = h_i(T_i - T_1) = k_w \frac{T_1 - T_2}{d_w} = h_o(T_2 - T_o) + \epsilon\sigma(T_2^4 - T_A^4)$$

T_1 represents the inner window temperature and T_2 represents the outer window temperature. ϵ is the emissivity of uncoated glass, which we take to be 0.95, and σ is the Stefan-Boltzmann constant.

Polynomial Chaos Expansion

In this example, the forward propagation requires the representation of heat flux Q with a multidimensional Polynomial Chaos Expansion (PCE). This representation can be defined as follows:

$$Q = \sum_{k=0}^{P} Q_k \Psi_k(\xi_1, ..., \xi_n)$$

- Q: Random variable represented with multi-D PCE
- Q_k: PC coefficients
- Ψ_k: Multi-D orthogonal polynomials up to order p
- ξ_i: Gaussian random variable known as the *germ*
- n: Dimensionality = number of uncertain model parameters
- P + 1: Number of PC terms = $\frac{(n+p)!}{n!p!}$

Non-Intrusive Spectral Projection (NISP)

Having specified a PCE form for our heat flux Q, we must determine the PC coefficients. We do so through a non-intrusive Galerkin Projection. The coefficients are determined using the following formula:

$$Q_k = \frac{\langle Q\Psi_k \rangle}{\langle \Psi_k^2 \rangle} = \frac{1}{\langle \Psi_k^2 \rangle} \int Q\Psi_k(\xi) w(\xi) d\xi$$

In this example we use quadrature methods to evaluate this projection integral to determine the PC coefficients.

Kernel Density Estimation

Once we have a large number of heat flux samples, to obtain a probability density function (PDF) curve, a kernel density estimation (KDE) is performed. When performing KDE, the following formula is used to evaluate the PDF at point Q:

$$PDF(Q) = \frac{1}{N_s h} \sum_{i=1}^{N_s} K\left(\frac{Q - Q^i}{h}\right)$$

Q^i are samples of the heat flux, N_s is the number of sample points, K represents the kernel, a non-negative function, and $h > 0$ is the bandwidth. In this example we use the Gaussian kernel, $K(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$. The results rely heavily on the choice of bandwidth, and there are many rules that can be used to calculate the bandwidth. In our example, the built-in SciPy function employed automatically determines the bandwidth using Scott's rule of thumb.

Implementation

The script set consists of two files:

- `rad_heat_transfer_atm_pce.py`: the main script
- `heat_transfer_pce_tools.py`: functions called by `rad_heat_transfer_atm_pce.py`

rad_heat_transfer_atm_pce.py

This script will produce a graph comparing PDFs of heat flux generated using NISP full and sparse quadrature methods and a Monte Carlo sampling method. Use the command `./rad_heat_transfer_atm_pce.py` to run this file

The top section of the script has some flags that are useful to consider:

- main_verbose: set this parameter to 1 for intermediate print statements, otherwise set to 0.
- compute_rad: set to True to include radiation; use False to not use radiation. Note, radiation is used, the nonlinear solver in Python sometimes has a hard time converging, in which case you will see a warning message pop up.
- nord: the order of the PCE
- ndim: the number of dimensions of the PCE
- pc_type: indicates the polynomial type and weighting function. The default is set to *"HG"*, Hermite-Gauss.

- pc_alpha and pc_beta: Free parameters greater than -1. Used with Gamma-Laguerre and Beta-Jacobi PC types.
- param: The parameter used for quadrature point generation. Equal to the number of quadrature points per dimension for full quadrature or the level for sparse quadrature methods. This parameter is generally set to nord + 1 in order to have the right polynomial exactness.

Monte Carlo Sampling Methods
The script begins by assigning the input parameters and their means and standard deviations. Using this information, a large number of random parameter samples (default is 100,000) is generated. With these samples and our forward model, the function `compute_heat_flux` then calculates the heat flux assuming that no radiative heat transfer occurs. This simply requires solving a system of three linear equations. Then, using the values of Q, T_1, and T_2 obtained from `compute_heat_flux` as initial guesses, the function `r_heat_flux` calculates the total heat flux (including radiative heat transfer) using the SciPy nonlinear solver `optimize.fsolve`. Using the heat flux samples, a kernel density estimation is then performed using function `KDE`.

NISP Quadrature Methods
After the Monte Carlo sampling process is complete, forward propagation using projection methods will take place. At the beginning of this section of the script, the following variables are defined:

While running the file, a statement similar to the following will print indicating that PC objects are being instantiated.

```
Instantiating PC Objects

Generating 4^7 = 16384 quadrature points.
Used 4 quadrature points per dimension for initialization.
Generating 4^7 = 16384 quadrature points.
Used 4 quadrature points per dimension for initialization.
Level 0 / 4
Level 1 / 4
Level 2 / 4
Level 3 / 4
Level 4 / 4

Instantiation complete
```

These PC objects contain all of the information needed about the polynomial chaos expansion, such as the number of dimensions, the order of the expansion, and the sparsity (full or sparse) of the quadrature methods to be implemented.

Next, a NumPy array of quadrature points is generated, using the function `get_quadpts`. Then, the quadrature points in ξ are converted to equivalent samples of the input parameters, taking advantage of the fact that the inputs are assumed to be Gaussian. If we let μ represent the mean of an input parameter, σ represent its standard deviation, and qdpts be a vector of quadrature points, the following equation is used to convert these samples in ξ to equivalent samples of the input parameter:

$$parameter\_samples = \mu + \sigma(qdpts)$$

Now that we have samples of all the input parameters, these samples are run through the forward model to obtain values of Q using the function `fwd_model`. Then the actual Galerkin projection is performed on these function evaluations to obtain the PC coefficients, using the function `GalerkinProjection`.

Next, to create a PDF of the output Q from its PCE, germ samples are generated and the PCE is evaluated at these sample points using the function `evaluate_pce`. Lastly, using our PCE evaluations, a kernel density estimation is performed using function `KDE`

This entire process is done twice, once with full tensor product quadrature points and again with sparse quadrature points.

Printing and Graphing
Next, statements indicating the total number of sampling points used for each forward propagation method will be printed.

```
Monte Carlo sampling used 100000 points
Full quadrature method used 16384 points
Sparse quadrature method used 6245 points
```

Finally, a graph is created which displays the three different heat flux PDFs on the same figure. It will be saved under the file name `heat_flux_pce.pdf`.

heat_transfer_pce_tools.py

This script contains several functions called by the `rad_heat_transfer_atm_pce.py` file.

- `compute_heat_flux(Ti, To, dw, kw, hi, ho)`: This function calculates heat flux Q, assuming no radiative heat transfer occurs.
  - *Ti, To, dw, kw, hi, ho*: Sample values (scalars) of the input parameters
- `r_heat_flux(Ti,To,dw,kw,hi,ho,TA,estimates)`: This function calculates heat flux Q, assuming radiative heat transfer occurs. The SciPy non-linear solver `optimize.fsolve` is employed.

  - *Ti, To, dw, kw, hi, ho, TA*: Sample values (scalars) of the input parameters
  - *estimates*: Estimates of Q, T_1, and T_2 required to solve the nonlinear system. For these estimates, we use the output of the function `compute_heat_flux`, which solves the system assuming no radiative heat transfer occurs.

- `get_quadpts(pc_model,ndim)`: This function generates a NumPy array of either full tensor product or sparse quadrature points. Returns number of quadrature points, *totquat*, as well.
  - *pc_model*: PC object with information about the PCE, including the desired sparsity for quadrature point generation.
  - *ndim*: Number of dimensions of the PCE

- `fwd_model(Ti_samples,To_samples,dw_samples,kw_samples,hi_samples, ho_samples,verbose)`:

  This function returns a NumPy array of evaluations of the forward model. This function calls the functions `compute_heat_flux` and `r_heat_flux`.
  - *Ti_samples, To_samples, dw_samples, kw_samples, hi_samples, ho_samples*: 1D NumPy arrays (vectors) of parameter sample values.

- `fwd_model_rad(Ti_samples,To_samples,dw_samples,kw_samples,hi_samples, ho_samples,TA_samples,verbose)`:

  Same as `fwd_model` but with radiation enabled, and an extra argument for the samples of TA.

- `GalerkinProjection(pc_model,f_evaluations)`: This function returns a 1D NumPy array with the PC coefficients.
  - *pc_model*: PC object with information about the PCE.
  - *f_evaluations*: 1D NumPy array (vector) with function to be projected, evaluated at the quadrature points.

- `evaluate_pce(pc_model,pc_coeffs,germ_samples)`: This function evaluates the PCE at a set of samples of the germ.
  - *pc_model*: PC object with information about the PCE.
  - *pc_coeffs*: 1D NumPy array with PC coefficients. This array is the output of the function `GalerkinProjection`
  - *germ_samples*: NumPy array with samples of the PCE germ at which the random variable is to be evaluated.

- `KDE(fcn_evals)`: This function performs a kernel density estimation. It returns a NumPy array of points at which the PDF was estimated, as well as a NumPy array of the corresponding estimated PDF values.
  - *fcn_evals*: A NumPy array of evaluations of the forward model (values of heat flux Q).

Sample Results

Run the file `rad_heat_transfer_atm_pce.py` with the default settings. You should expect to see the following print statement, and your graph should look similar to the one found in the figure below.

```
Monte Carlo sampling used 100000 points
Full quadrature method used 16384 points
Sparse quadrature method used 6245 points
```

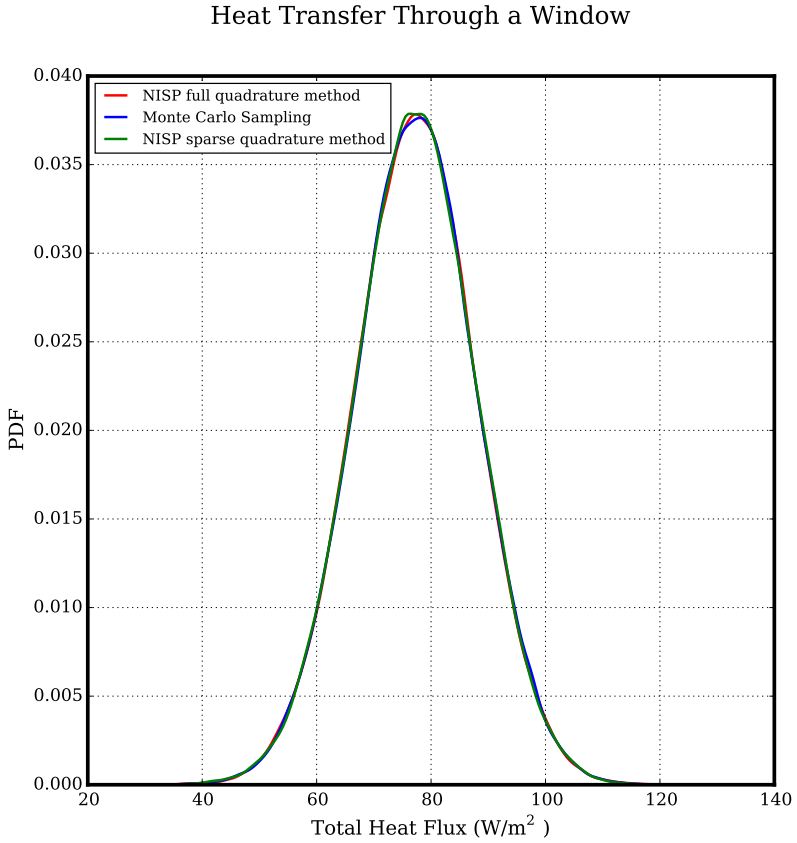


Figure 5.12: Sample results of rad_heat_transfer_atm_pce.py

Now trying changing one of the default settings. Find the line that reads `nord = 3` and change it to read `nord = 2`. This will change the order of the PCE to 2, rather than 3. You should expect to see the following print statement, and a sample graph is found in the figure below.

```
Monte Carlo sampling used 100000 points
Full quadrature method used 2187 points
Sparse quadrature method used 1023 points
```

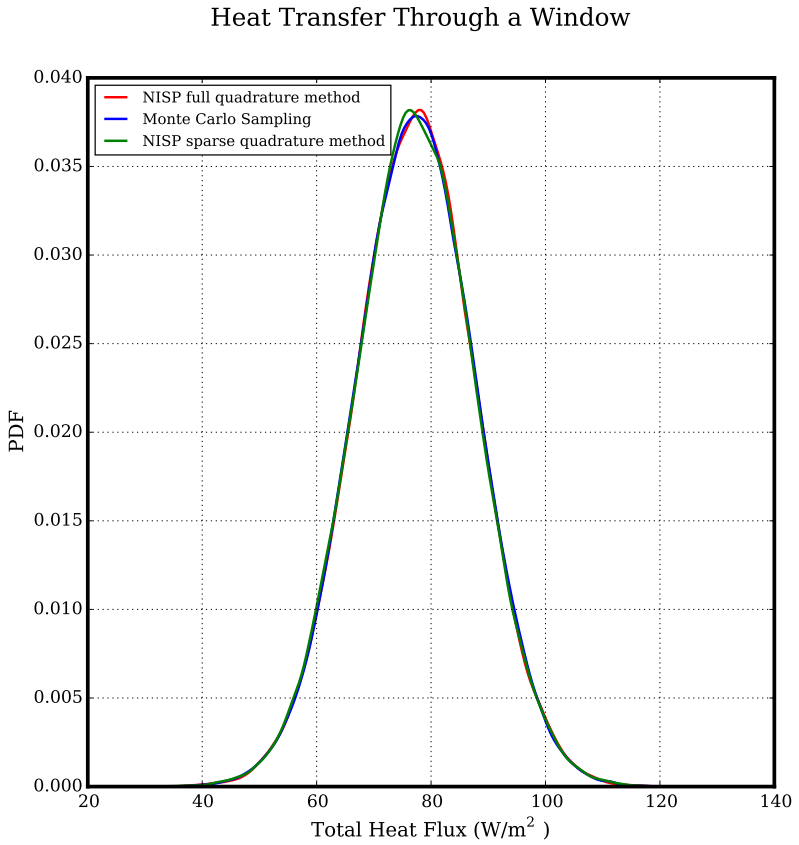


Figure 5.13: Sample results of rad_heat_transfer_atm_pce.py with nord = 2

The user can also change the default `pc_type = "HG"`. For example, to use Legendre-Uniform PC type, change this line to read `pc_type = "LU"`.

Expanded Forward Propagation of Uncertainty - PyUQTk

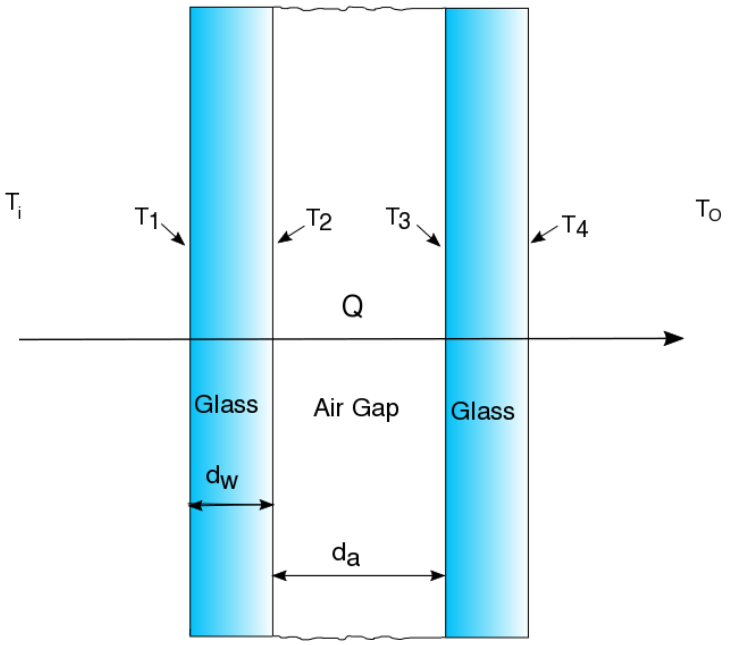
Overview

This example is contains two pairs of python files, each pair consists of a main script and a tools file that holds functions to be called from the main script. They are located in `UQTk/examples/window`. This example is an expanded version of the previous forward propagation problem, it utilizes a more complex model that provides additional parameters. One pair of scripts produces a probability density function of the heat flux for a twelve dimension model, using fourth order PCE's and sparse quadrature. The other pair produces a probability density function of the heat flux for a fifth dimension model, using fourth order PCE's. This script compares the results of using sparse and full quadrature, as well as Monte Carlo sampling. It also produces a separate plot showing the spectral decay of the

PC coefficients.

Theory

Heat Transfer - Dual Pane Window



The heat flux calculations are implemented using random samples from the normal distribution of each of the uncertain Gaussian parameters. The standard deviations assigned are estimates. These parameters are defined in the table below:

The forward model that was developed relies on the same set of steady state heat transfer equations from the first example, with the addition of a combined equation for conduction and convection for the air gap. This is defined as conductance, and was implemented in order to provide an alternative to determining the convection coefficient for this region, which can be challenging[2].

$$Q = h_i(T_i - T_1) = k_w \frac{T_1 - T_2}{d_w} = 0.41 \frac{k_a}{d_a} \left[\left(\frac{g\beta\rho^2 {d_a}^3}{\mu^2} \right) |T_2 - T_3| \right]^{0.16} (T_2 - T_3) = k_w \frac{T_3 - T_4}{d_w}$$
$$= h_o(T_4 - T_o) + \epsilon\sigma(T_4^4 - T_s^4)$$

[1] Leonard, John H. IV, Leinhard, John H. V. *A Heat Transfer Textbook - 4th edition.* pg 579. Phlogiston Press. 2011 [2] Rubin, Michael. *Calculating Heat Transfer Through Windows.* Energy Research. Vol. 6, pg 341-349. 1982.

Parameter	Definition	Mean Value	Standard Deviation %
T_i	Inside temperature	293 K	0.5
T_o	Outside temperature	273 K	0.5
T_s	Sky Temperature[1]	263 K	10
k_w	Conduction constant for glass	1 W/m^2K	5
k_a	Conduction constant for air	0.024 W/m^2K	5
h_i	Inside convection coefficient	2 W/m^2K	15
h_o	Outside convection coefficient	6 W/m^2	15
d_w	Width of the glass	5 mm	1
d_a	Width of the air gap	1 cm	1
μ	Viscosity or air	1.86x10^{-5} kg/m s	5
ρ	Density of air	1.29 kg/m^3	5
β	Thermal expansion coefficient	3.67$x10^{-3}1/K$	5

T_1 represents the glass temperature of the outer facing surface for the first glass layer, and T_2 represents the temperature of the inner surface for the first glass layer, T_3 represents the temperature of the inner surface of the second glass layer, T_4 represents the outer facing surface of the second glass layer. ϵ is the emissivity of uncoated glass, which we take to be 0.95, and σ is the Stefan-Boltzmann constant.

Polynomial Chaos Expansion*

Non-Intrusive Spectral Projection (NISP)*

Kernel Density Estimation*

*Please see previous example.

Implementation

The first set of scripts:

- `5D_fwd_prop.py`: the main script
- `fwd_prop_tools.py`: functions called by `5D_fwd_prop.py`

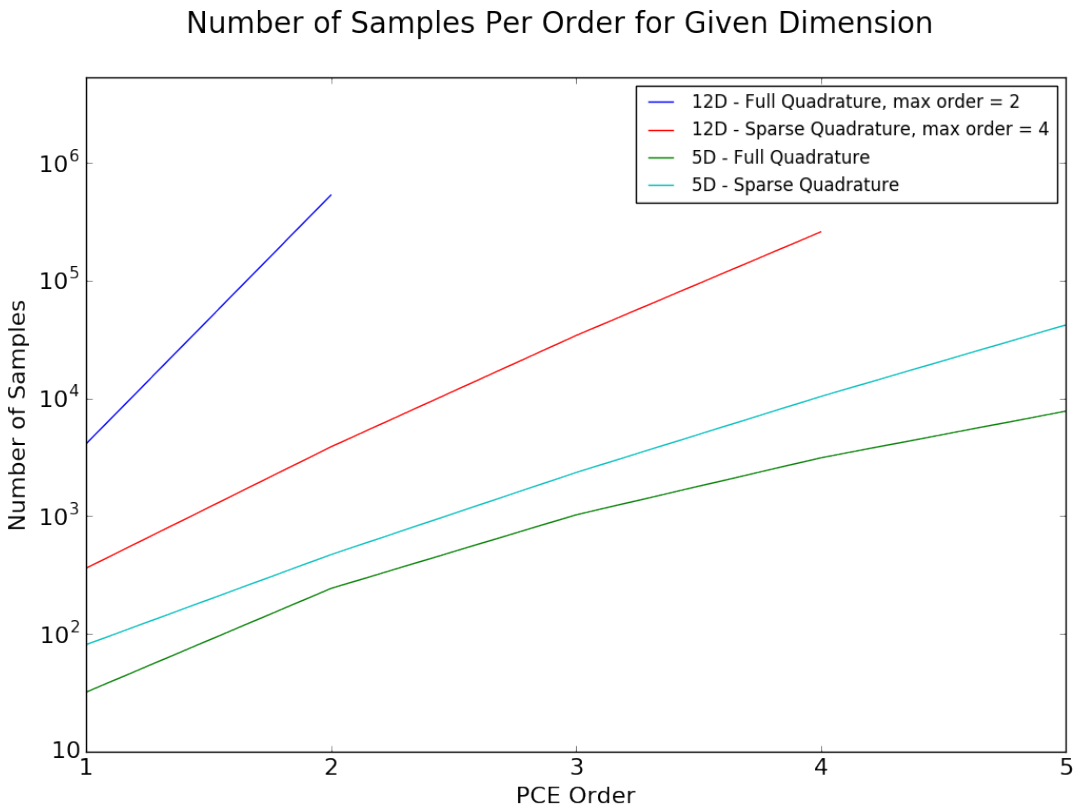


Figure 5.14: Increase in number of samples with change in order.

5D_fwd_prop.py

This script will produce a plot comparing the probability density function of the heat flux using three methods. Non-intrusive spectral projection full and sparse quadrature methods, and Monte Carlo sampling. It also produces a plot showing the spectral decay of the PC coefficient magnitude in relation to the PC order. This plot illustrates the how the projection converges to give a precise representation of the model. Changing the order of this script (`nord`) may alter the results of the second plot, since it has elements that are not dynamically linked. Use the command `python 5D_fwd_prop.py` to run this file from the window directory.

The second set of scripts:

- `highd_sparse.py`: the main script
- `tools_conductance_dp_pce_wrad.py`: functions called by `highd_sparse.py`

highd_sparse.py

This script will produce of the probability density function of the heat flux in twelve dimensions, using only non-intrusive spectral projection with sparse quadrature. Use the command `python highd_sparse.py` to run this file from the window directory. Adjustments may be made to `nord` if desired, though the number of points produced increases dramatically with an increase in order, as illustrated by figure 5.9.

Monte Carlo Sampling Methods
This is very similar to the previous example with the exception of `compute_heat_flux` producing T_1, T_2, T_3, T_4 and Q. This function consists of a solved set of five linear equations, our forward model neglecting convection for the air gap and radiative heat transfer, which are evaluated for the parameter samples. The values obtained are used as initial inputs for `r_heat_flux`, which calculates the total heat flux for all heat transfer modes, using the SciPy nonlinear solver `optimize.fsolve`. Using the heat flux samples, a kernel density estimation is then performed using function `KDE`.

NISP Quadrature Methods
Please see previous example.

Printing and Graphing
Next, statements indicating the total number of sampling points used for each forward propagation method will be printed. The number of Monte Carlo points is fixed, but the number of points produced by quadrature method will vary with the number of uncertain parameters and the order of the PCE.

```
Monte Carlo sampling used 100000 points
Full quadrature method used xxxxxxx points
Sparse quadrature method used xxxxxxx points
```

Finally, a graph is created which displays the three different heat flux PDFs on the same figure. It will be saved under the file name `heat_flux_pce.pdf`.

fwd_prop_tools.py and tools_conductance_dp_pce_wrad.py

This scripts contain several functions called by the `5D_fwd_prop.py` and `highd_sparse.py` files. The five dimension example uses the five parameters with the highest uncertainty, *Ts,hi,ho,kw,ka*.

- `compute_heat_flux(Ti,To,dw,da,kw,ka,hi,ho)`: This function calculates heat flux Q, assuming no radiative heat transfer occurs and no convective heat transfer occurs in the air gap.
  - *Ti, To, dw, da, kw, ka, hi, ho*: Sample values (scalars) of the input parameters
- `r_heat_flux(Ti,To,Ts,dw,da,kw,ka,hi,ho,beta,rho,mu,estimates)`: This function calculates heat flux Q, assuming radiative heat transfer, and convective heat transfer for the air gap occurs. The SciPy non-linear solver `optimize.fsolve` is employed.
  - *Ti, To, Ts, dw, da, kw, ka, hi, ho, beta, rho, mu*: Sample values (scalars) of the input parameters
  - *estimates*: Estimates of Q, T_1, T_2, T_3, and T_4 are required to solve the nonlinear system. For these estimates, we use the output of the function `compute_heat_flux`, which solves the system assuming no radiative or convective heat transfer for the air gap occurs.
- `get_quadpts(pc_model,ndim)`*
- `fwd_model(Ti_samples,To_samples,Ts_samples,dw_samples,da_samples,kw_samples, ka_samples,hi_samples,ho_samples,beta_samples,rho_samples,mu_samples)`:

  This function returns a NumPy array of evaluations of the forward model. This function calls the functions `compute_heat_flux` and `r_heat_flux`.
  - *Ti_samples, To_samples,Ts_samples, dw_samples, da_samples, kw_samples, ka_samples, hi_samples, ho_samples, beta_samples, rho_samples, mu_samples*: 1D NumPy arrays (vectors) of parameter sample values.
- `GalerkinProjection(pc_model,f_evaluations)*`
- `evaluate_pce(pc_model,pc_coeffs,germ_samples)*`:
- `KDE(fcn_evals)*`:
- `get_multi_index(pc_model,ndim)` This function computes the multi indices to be used in the inverse relationship plot.
  - *pc_model*: Contains information about the PC set.
  - *ndim*: Number of model dimensions.
- `plot_mi_dims(pc_model,c_k,ndim)` This function produces the inverse relationship plot using the multi indices produced in the previous function, and the value of the PC coefficients.
  - *pc_model*: Contains information about the PC set.
  - *ndim*: Number of model dimensions.
  - *c_k*: Numpy array of PC coefficients.

*Please see previous example.

Sample Results

Run the file `highd_sparse.py` with the default settings. You should expect to see the following print statement, and your graph should look similar to the one found in the figure below. This sample is for a problem with twelve dimensions, the PCE is fourth order. The following will print to the terminal:

```
Sparse quadrature method used 258681 points
```

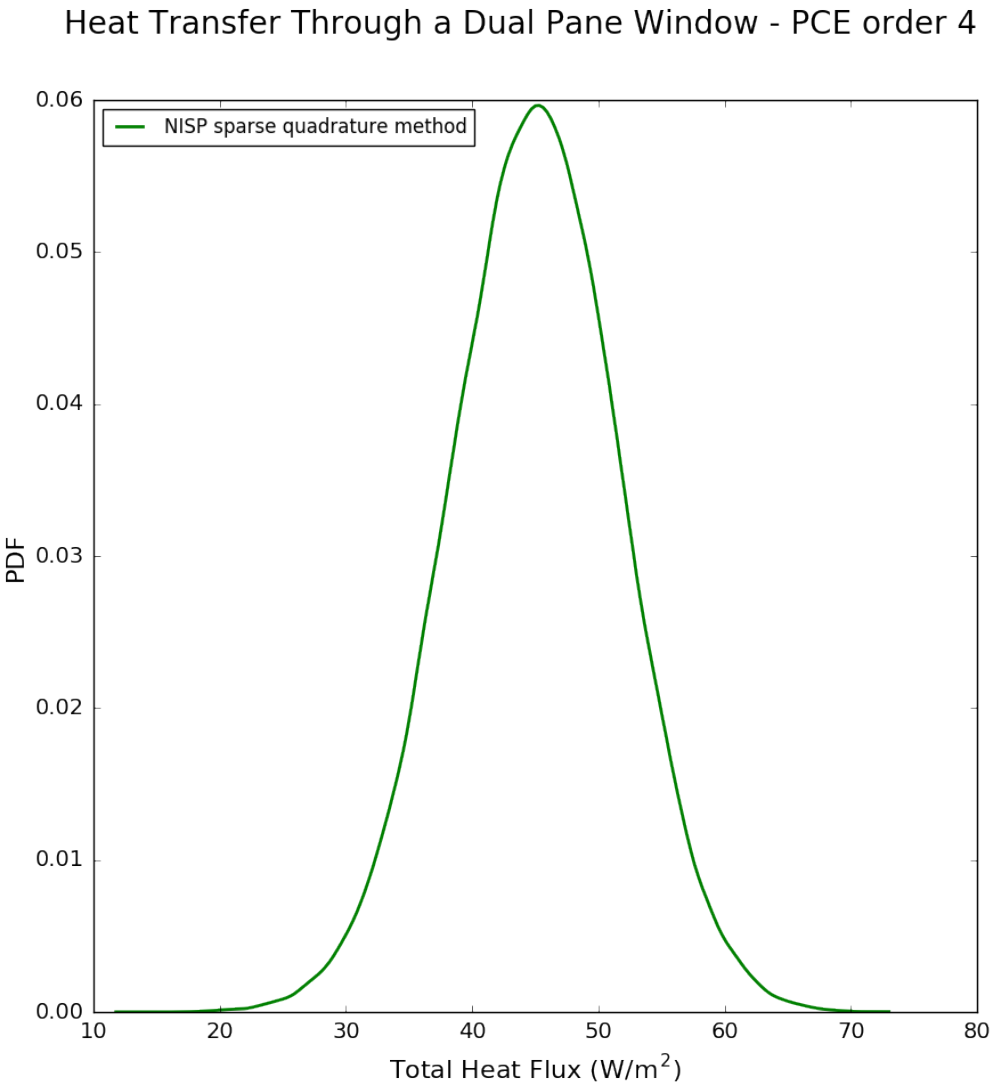


Figure 5.15: Sample results of highd_sparse.py

The next two figures show the two results of the five dimension example with a fourth order PCE. The following will print to the terminal:

```
Monte Carlo sampling used 100000 points
Full quadrature method used 3125 points
Sparse quadrature method used 10363 points
```

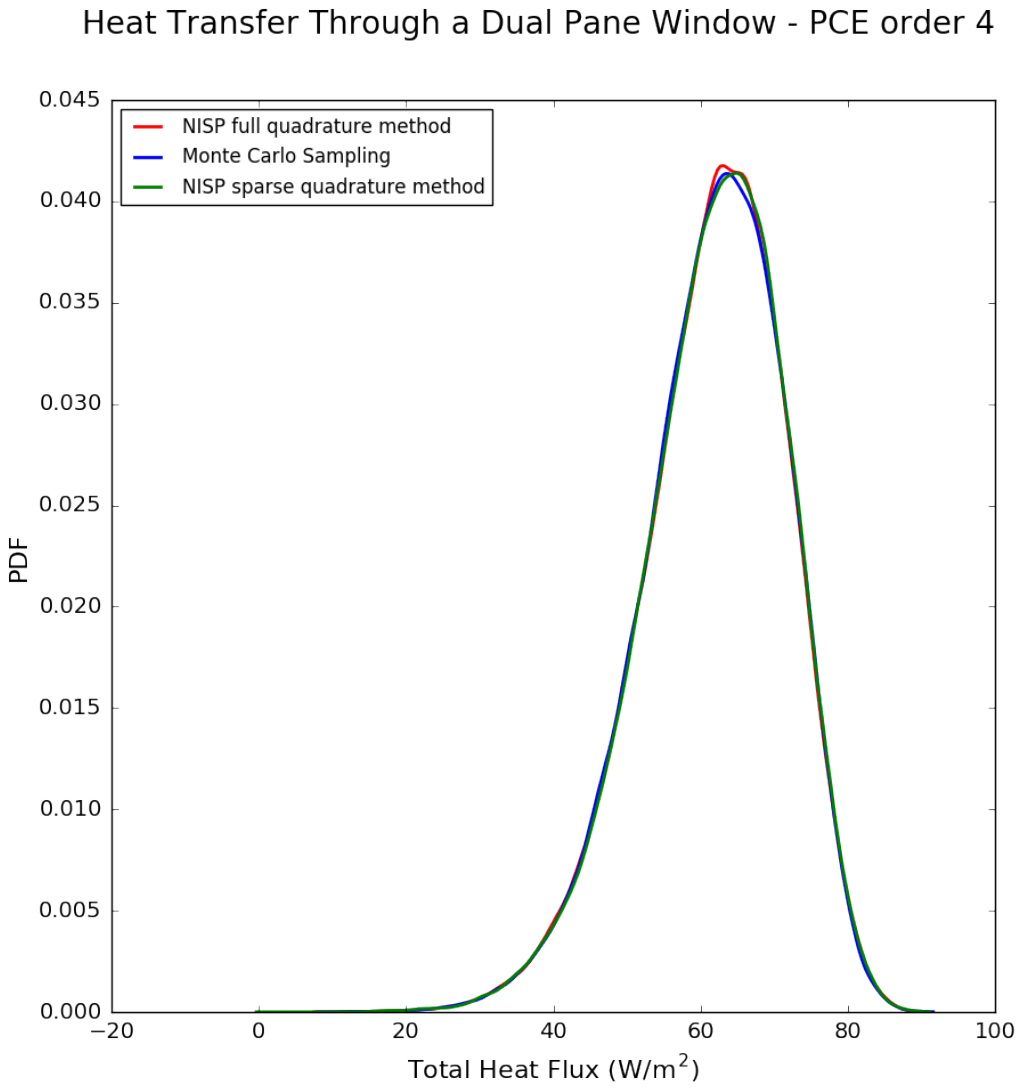


Figure 5.16: Sample results of 5D_fwd_prop.py

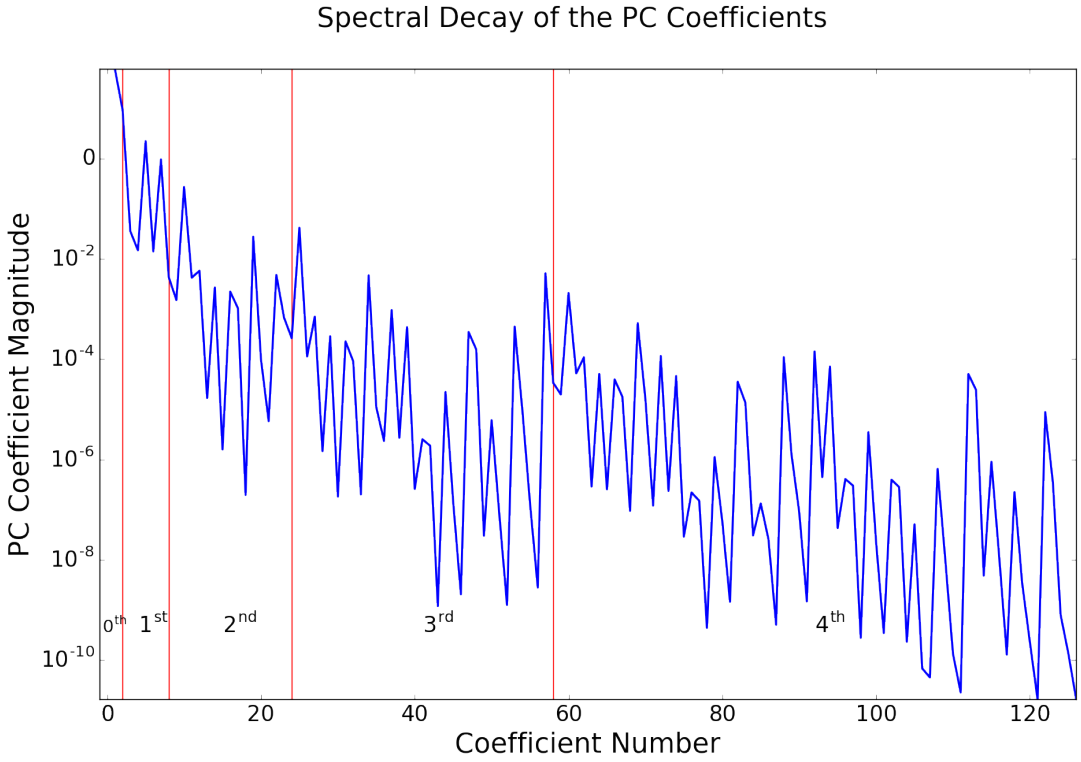


Figure 5.17: Sample results of 5D_fwd_prop.py

Forward Propagation of Uncertainty Using Basis Adaptation

Overview

This example is located in `examples/d_spring_series`. It contains several Python scripts that propagate uncertainty in input parameters through a series springs model using basis adaptation approach, and is compared with Monte Carlo sampling method and non-intrusive spectral projection (NISP) via sparse quadrature method.

Theory

Effective Modulus for d Springs in Series

In this example, the effective modulus for d springs in series is represented as:

$$f(x_1, x_2, ..., x_d) = \frac{d}{1+b} \frac{\prod_{i=1}^{d}(1 + ax_i + bx_i^2)}{\sum_{i=1}^{d} \prod_{\substack{j=1 \\ j \neq i}}^{d}(1 + ax_j + bx_j^2)} \tag{5.5}$$

each spring has modulus $(1 + ax_i + bx^2)$. Where d is the dimension, a and b are coefficients. In our example, we have springs with $\{x_i, i = 1, ..., 7\}$ independent Gaussian distribution, where $x_i \sim \mathcal{N}(5.0, 0.6)$ with $i = 1, ..., 4$ and $x_i \sim \mathcal{N}(4.0, 0.5)$ with $i = 5, ..., 7$. Associated coefficients are $a = 0.5$ and $b = 1.0$.

Basis Adaptation

By emphasizing the mathematical structure on Gaussian Hilbert spaces, a reduced order is obtained, which capture the Gaussian probabilistic information of QoI and maintains its dependence on the original parameter space.

Let $\boldsymbol{A}$ be an isometry on R^d and $\boldsymbol{\eta}$ be:

$$\boldsymbol{\eta} = \boldsymbol{A}\boldsymbol{\xi}, \qquad \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I} \tag{5.6}$$

- $\boldsymbol{\xi} = (\xi_1, ..., \xi_d)$: Gaussian random variable know as the *germ*

Since $\boldsymbol{\eta}$ is another basis just like $\boldsymbol{\xi}$, the orthogonal basis in $\boldsymbol{\eta}$ span the orthogonal basis in $\boldsymbol{\xi}$. Letting $\Psi_k^{\boldsymbol{A}}(\boldsymbol{\eta}) = \Psi_k(\boldsymbol{\xi})$, and we have the equivalent PCEs:

$$Q(\boldsymbol{\xi}) = \sum_{k=0}^{P} Q_k \Psi_k(\boldsymbol{\xi}), \qquad Q^{\boldsymbol{A}}(\boldsymbol{\eta}) = \sum_{l=0}^{P} Q_l \Psi_l^{\boldsymbol{A}}(\boldsymbol{\eta}), \tag{5.7}$$

Letting $Q(\boldsymbol{\xi}) \triangleq Q^{\boldsymbol{A}}(\boldsymbol{\eta})$, yields:

$$Q_l = \sum_{k=0}^{P} Q_k \langle \Psi_k(\boldsymbol{\xi}) \Psi_l^{\boldsymbol{A}}(\boldsymbol{\eta}) \rangle \tag{5.8a}$$

$$Q_k = \sum_{l=0}^{P} Q_l \langle \Psi_l^{\boldsymbol{A}}(\boldsymbol{\eta}) \Psi_k(\boldsymbol{\xi}) \rangle \tag{5.8b}$$

This provides us with a tool to compare coefficients of two PCEs of full dimension.

After the projection of $\boldsymbol{A}$, suppose that important probabilistic information of QoI is concentrated to the first several components of $\boldsymbol{\eta}$, then we can use these components to form a lower dimensional PCE. One of the options would be letting $\boldsymbol{A}$ be such that:

$$\eta_1 = \sum_{i=1}^{d} Q_{\boldsymbol{e}_i} \xi_i \tag{5.9}$$

- $\boldsymbol{e}_i$: subset of multi-indices with 1 at ith location and zeros elsewhere
- $Q_{\boldsymbol{e}_i}$: first order expansion coefficients of d dimension

so that first component of $\boldsymbol{\eta}$ captures the complete Gaussian components of Q. Letting the first row of $\tilde{\boldsymbol{A}}$ be the Gaussian components, the remaining parts of $\tilde{\boldsymbol{A}}$ can be constructed in two approaches. The first one is putting 1 in the diagonal zeros elsewhere, the second one is put the largest Gaussian component in the second row with column position the same as it appears in the first row, and put the second largest in the third row with column position the same as it appears in the first row too, so on so forth. We call the second approach "sort by importance" method. Then $\boldsymbol{A}$ is constructed by the Gram-Schmidt (or other orthogonalization) of matrix $\tilde{\boldsymbol{A}}$.

We first perform the 1 dimensional reduction and obtain associated PC coefficients. Then the 2 dimensional reduction and compare the coefficients with the 1 dimensional PC coefficients, stop if converged or proceed to 3 dimensional reduction if not, so on so forth. To compare coefficients of different dimensional PCEs, say d_i dimensional and d_j dimensional with $d_i < d_j$, we need first project coefficient from $\boldsymbol{C}_{\boldsymbol{\alpha}}$ ($\boldsymbol{\alpha} \in \mathcal{I}_{d_i,p}$) to $\boldsymbol{C}_{\boldsymbol{\beta}}$ ($\boldsymbol{\beta} \in \mathcal{I}_{d_j,p}$), where $\mathcal{I}_{d,p}$ denote the set of all d-dimensional multi-indices of degree less than or equal to p. This is easily done by letting:

$$C_{\tilde{\boldsymbol{\alpha}}} = \begin{cases} C_{\boldsymbol{\alpha}} & \tilde{\boldsymbol{\alpha}}(1,...,d_i) = \boldsymbol{\alpha} \quad \text{and} \quad \tilde{\boldsymbol{\alpha}}(d_i+1,...,d_j) = \mathbf{0} \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

- $\tilde{\boldsymbol{\alpha}}$: multi-indices $\in \mathcal{I}_{d_j,p}$
- $\boldsymbol{C}_{\tilde{\boldsymbol{\alpha}}}$: projected coefficients of $\boldsymbol{C}_{\boldsymbol{\alpha}}$

This provides a convergence criterion.

We can also compare any dimensional PCE in $\boldsymbol{\eta}$ space (rotated space) with PCE in $\boldsymbol{\xi}$ space. Which is done by first projecting coefficients of, say d_0 dimensional, PCE in $\boldsymbol{\eta}$ space to coefficients of d dimensional PCE in $\boldsymbol{\eta}$ space by equation 5.10, and then projecting coefficients in $\boldsymbol{\eta}$ space to $\boldsymbol{\xi}$ space by equation 5.8. Then we can judge the accuracy of reduced PCE with respect to full dimensional PCE by comparing the coefficients, in $\boldsymbol{\xi}$ space.

Implementation

The script set contains three files:

- `run_d_springs.py`: the main script
- `d_springs_tools.py`: function called by `run_d_springs.py`, mainly contains classical PCE needed modules and forward model
- `adaptation_tools.py`: function called by `run_d_springs.py`, contains modules that deal with the basis adaptation. This function is a library files located at "${install} /PyUQTk /PyPCE"

exec_d_springs.py

This scripts will produce two figures, the first figure compare the projected coefficients of 2 dimensional PCE and full dimensional PCE in $\boldsymbol{\xi}$ space, the second figure compares PDFs of effective modulus of the 7 dimensional series springs model generated by 2d Gaussian adaptation method, Monte Carlo sampling method and NISP full dimension sparse quadrature method.

Some of the important input parameters are:

- `nord`: The order of PCE
- `ndim`: The dimension of PCE, set to 7 in our example
- `pc_type`: Polynomial type and weighting function. Hermite-Gauss, "HG", is selected in adaptation method
- `param`: Quadrature level, usually set to `nord`+1 to have the right polynomial exactness
- `method`: Method used to generate $\boldsymbol{A}$ matrix. The default one, $method = 0$, is using Gram-Schmidt of $\tilde{\boldsymbol{A}}$, $method = 1$ is using orthogonal decomposition of $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{A}}^T$, $method = 2$ is using orthogonal decomposition of the Householder matrix, and the last one, $method = 3$, is using "sort by importance" method. The default method is $method = 0$, which is satisfying for most problems, if not, then we recommend to use $method = 3$
- `a`: $a = 0.5$ in our example
- `b`: $b = 1.0$ in our example

There are also other fixed parameters. One is `nord0`, which is equal to 1, denoting the PC order used to compute first order coefficients, while the quadrature level parameter `param0` is equal to 1 too.

The first step of the work flow for adaptation PCE is to compute the Gaussian coefficients (first order coefficients) of the associated QoI. Then, Gaussian coefficients are used to construct rotation matrix $\boldsymbol{A}$. Starting from 1 dimension, the reduced PCEs are then obtained until coefficients of two successive dimensional PCEs are converged.

Printing and Graphing

The statements indicating the total number of sampling points used for each forward propagation method will be printed. The number of Monte Carlo points and number of points produced by sparse quadrature points are fixed, but the number of total quadrature points produced in the adaptation method depends on when the convergence is reached.

```
Monte Carlo sampling used 100000 points
```

```
Sparse quadrature method used 6245 points
Adaptation method used 244 points
```

Note that the points used in the adaptation method include points in calculating Gaussian coefficients, 1d adaptation of PCE, and 2d adaptation of PCE (used to ensure the convergence of 1d adaptation). So actually, only 1d adaptation is enough to get a good result.

Then two graphs are generated. The first figure is a verification of 2d Gaussian adaptation with full dimension PCE by comparing the coefficients, coefficients of 2d Gaussian adaptation are projected to full dimensional PCE space. The second figure gives the PDFs of effective modulus generated by different methods.

d_springs_tools.py

This script contains several functions called by `run_d_springs.py` file.

- `fwd_model(xx, a, b)`: This function compute the effective modulus of the d series springs, and the output is a NumPy array with dimension the size of samples.
  - xx: $N_{samples} \times d$ NumPy array, where $N_{samples}$ is the size of samples
  - a, b: Input parameters in the d series springs model.
- `KDE(fcn_evals)`*
- `EvaluatePCE(pc_model,pc_coeffs,germ_samples)`: This function evaluate QoI using the PCE model and coefficients at customized samples.
  - *pc_model*: Known PCE model
  - *pc_coefficients*: Feed in PC coefficients
  - *germ_samples*: Germ samples used to evaluate

*Please see previous examples.

adaptation_tools.py

This script contains functions related to Gaussian adaptation method.

- `gauss_adaptation(c_k, ndim, method = 0)`: Function to obtain rotation matrix $\boldsymbol{A}$ from first order PC coefficients.
  - c_k: First order PC coefficients with size equal the dimension of the problem
  - *ndim*: Same as before, the dimension of the problem

- *method*: Methods used to construct matrix $\boldsymbol{A}$, default $method = 0$ refers to Gram-Schmidt procedure on matrix $\tilde{\boldsymbol{A}}$ with Gaussian coeffs (normalized) at its first row, and ones along diagonal zeros elsewhere for other rows. And $method = 1$ refers to orthogonal decomposition of $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{A}}^T$, $method = 2$ refers to orthogonal decomposition of Householder matrix $\boldsymbol{H} = \boldsymbol{I} - \frac{2\tilde{\boldsymbol{A}}\tilde{\boldsymbol{A}}^T}{\|\tilde{\boldsymbol{A}}\|^2}$, and $method = 3$ refers to "sort by importance" method.

- `eta_to_xi_mapping(eta, A, zeta = None)`: This function maps lower dimensional $\boldsymbol{\eta}$ space to full dimensional $\boldsymbol{\xi}$ space.

  - *eta*: $\boldsymbol{\eta}$ array with size $N_{samples} \times d$
  - *A*: Rotation matrix
  - *zeta*: Provides an option to specify augment matrix of $\boldsymbol{\eta}$ to match the size of $\boldsymbol{\xi}$. Augment matrix is $\mathbf{0}$ if not specified

- `mi_terms_loc(d1, d2, nord, pc_type, param, sf, pc_alpha=0.0, pc_beta=1.0)`: Find multi-indices "locations" of $d1$ dimensional PCE in $d2$ dimensional PCE. Where the "locations" refers to locations of multi-indices in $d2$ dimensional PCE, where the first $d1$ terms of which equal to multi-indices of $d1$ dimensional PCE and the remaining terms equal to 0, as described in equation 5.10. This function is called by `l2_error_eta(.)` function in file `adaptation_tools.py`.

  - *d1, d2*: Dimensions of PCEs with $d1 < d2$
  - *nord, pc_type, param, sf*: Parameters of the polynomial basis and quadrature method, where *nord* refers to order, *pc_type* refers to polynomial type, *param* refers to quadrature level, and sf refers to choice of "$sparse$" or "$full$" quadrature
  - *pc_alpha, pc_beta**

- `l2_error_eta(c_1, c_2, d1, d2, nord, pc_type, param, sf, pc_alpha=0.0, pc_beta=1.0)` Function to compute the $l2$ error of coefficients of $d1$ dimensional PCE and $d2$ dimensional PCE, where coefficients of $d1$ dimensional PCE are projected to $d1$ dimensional PCE. The projected coefficients of $d1$ dimensional PCE are also returned.

  - *c_1, c_2*: Coefficients of two different dimensional PCEs
  - *d1, d2*: Dimensions of PCEs
  - *nord*: Order of PCEs
  - *pc_type, param, sf, pc_alpha, pc_beta* *

- `transf_coeffs_xi(coeffs, nord, ndim, eta_dim, pc_type, param, R, sf="sparse", pc_al` Transfer coefficients from $\boldsymbol{\eta}$ space to $\boldsymbol{\xi}$ space. Only make sense when $eta_dim = ndim$.

  - *coeffs*: Coefficients in $\boldsymbol{\eta}$ space
  - *eta_dim*: Dimension of $\boldsymbol{\eta}$
  - *R*: Rotation matrix

– *nord, ndim, pc_type, param, sf, pc_alpha, pc_beta* *

*Same as mentioned before in this example.

Sample Results

Run the file `run_d_springs.py` with the default settings. One should obtain the two figures as below:

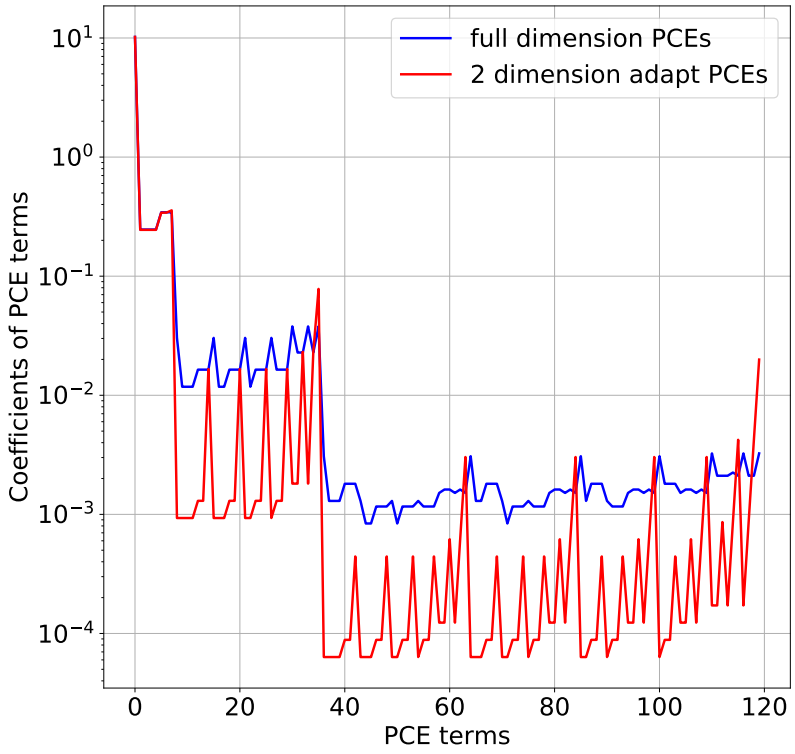


Figure 5.18: Coefficients comparison of adaptation method and full dimension PCE

Note that y axis of Figure 5.18 is plot in *log* scale, so the dominant coefficients of these two are very close. The PDF showed in Figure 5.19 proves that the basis adaptation method can achieve high accuracy.

Here we use 2 dimension adaptation to make a comparison, but 1 dimension adaptation is already very accurate (PC coefficients of which are converged to the 2 dimension values).

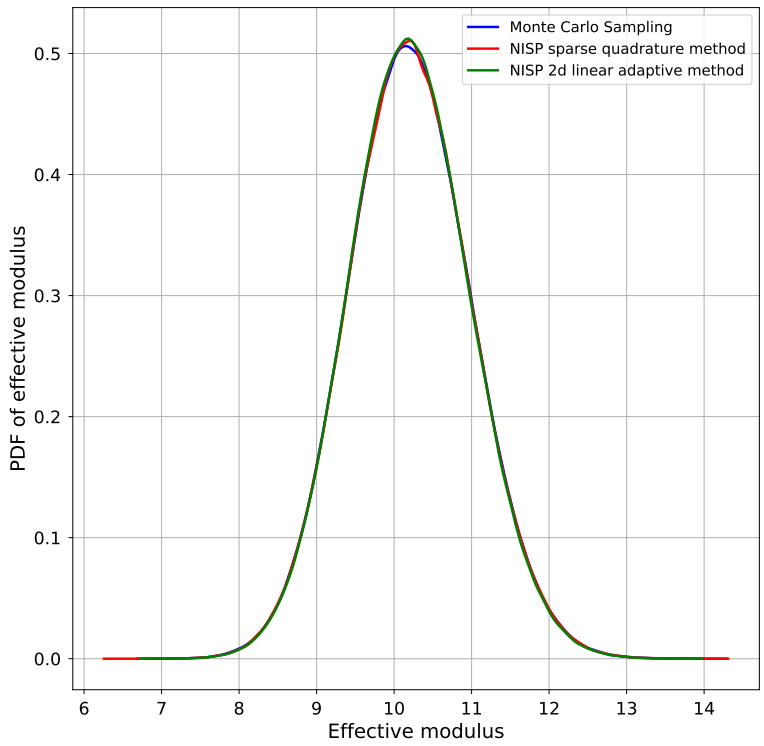


Figure 5.19: PDFs of effective modulus generated with different methods

Bayesian Inference of a Line

Overview

This example is located in `examples/line_infer` It infers the slope and intercept of a line from noisy data using Bayes' rule. The C++ libraries are called directly from the driver program. By changing the likelihood function and the input data, this program can be tailored to other inference problems.

To run an example, type `./line_infer.py` directly. This file contains quite a bit of inline documentation about the various settings and methods used. To get a listing of all command line options, type `./line_infer.py -h"`. A typical run, with parameters changed from command-line, is as follows:

```
./line_infer.py --nd 5 --stats
```

This will run the inference problem with 5 data points, generate plots of the posterior

distributions, and generate statistics of the MCMC samples. If no plots are desired, also give the `--noplots` argument.

More details

After setting a number of default values for the example problem overall, the `line_infer.py` script sets the proper inference inputs in the file `line_infer.xml`, starting from a set of defaults in `line_infer.xml.templ`. The file `line_infer.xml` is read in by the C++ code `line_infer.x`, which does the actual Bayesian inference. After that, synthetic data is generated, either from a linear, or cosine model, with added noise.

Then, the script calls the C++ line inference code `line_infer.x` to infer the two parameters (slope and intercept) of a line that best fits the artificial data. (Note, one can also run the inference code directly by manually editing the file `line_infer.xml` and typing the command `./line_infer.x`)

The script then reads in the MCMC posterior samples file, and performs some postprocessing. Unless the flag `--noplots` is specified, the script computes and plots the following:

- The pushed-forward and posterior predictive error bars
  - Generate a dense grid of x-values
  - Evaluate the linear model $y = a + bx$ for all posterior samples (a, b) after the burn-in
  - Pushed-forward distribution: compute the sample mean and standard deviation of using the sampled models
  - Posterior predictive distribution: combine pushed-forward distribution with the noise model
- The MCMC chain for each variable, as well as a scatter plot for each pair of variables
- The marginal posterior distribution for each variable, as well as the marginal joint distribution for each pair of variables

If the flag `--stats` is specified, the following statistics are also computed:

- The mean, MAP (Maximum A Posteriori), and standard deviations of all parameters
- The covariance matrix
- The average acceptance probability of the chain
- The effective sample sizes for each variable in the chain

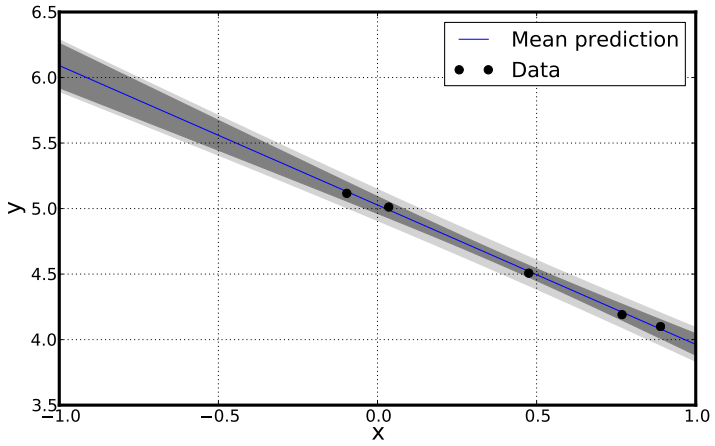


Figure 5.20: The pushed forward posterior distribution (dark grey) and posterior predictive distribution (light grey).

Sample Results

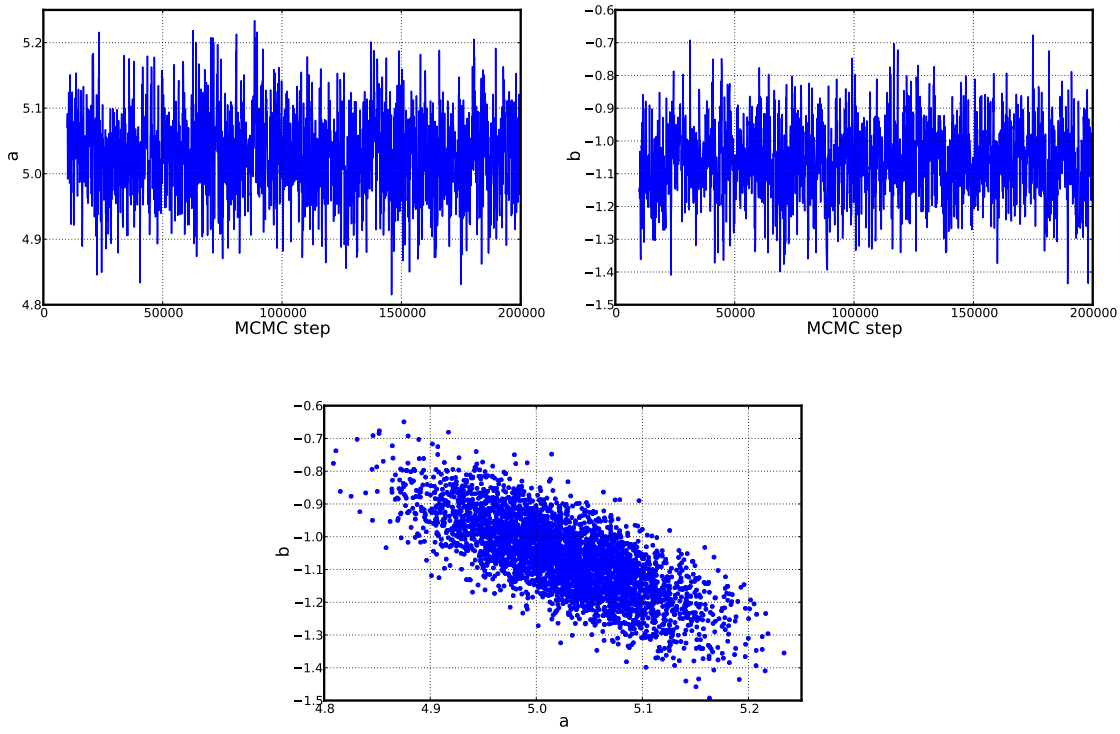


Figure 5.21: MCMC chains for parameters a and b, as well as a scatter plot for a and b

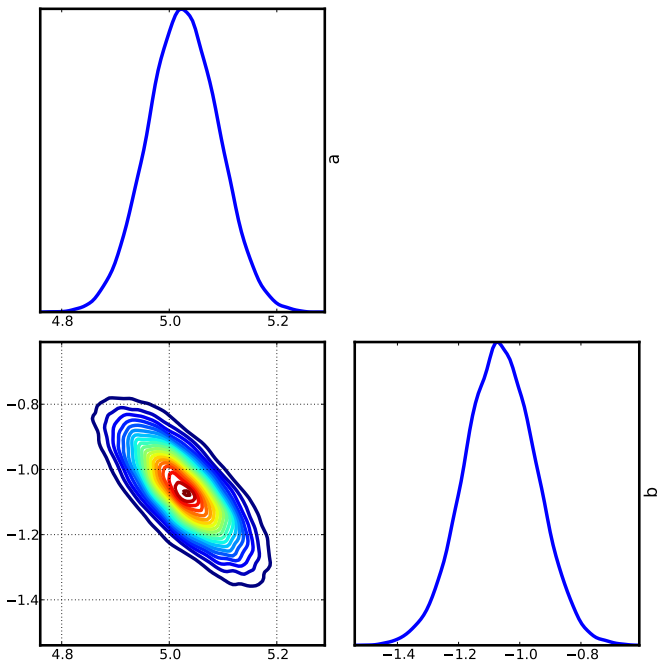


Figure 5.22: Marginal posterior distributions for all variables, as well as marginal joint posteriors

Sampling of Multimodal Posterior PDFs using TMCMC

Overview

This example is located in `examples/tmcmc_bimodal`. It generates samples distributed according to an underlying 3-dimensional bimodal posterior PDF, being a product of a Normal prior PDF and a bimodal likelihood PDF. It utilizes the Transitional Markov chain Monte Carlo (TMCMC) method [2], a variant of a class of MCMC algorithms known as tempering methods, which also provides an estimate of the model evidence at no extra computational cost (i.e. no further evaluations of likelihood and/or prior PDFS). The C++ libraries are called directly from the driver program. By changing the likelihood function and prior PDF (in `bimodal.cpp`), along with providing consistent samples from the prior PDF in `tmcmc_prior_samples.dat`, this program can be tailored to other problems. It utilizes shell scripts to spawn multiple processes for parallel evaluation of likelihood and prior PDFs.

To run an example, type `./tmcmc_bimodal.py` directly. To get a listing of all command line options, type `./tmcmc_bimodal.py -h`. A typical run is as follows:

```
./tmcmc_bimodal.py
```

This will run the TMCMC sampler, starting with 5000 samples from the prior PDF, generate plots of the posterior distributions along with intermediate samples (artifacts of TMCMC). If no plots are desired, also give the `--noplots` argument.

More details

TMCMC combines aspects of simulated annealing optimization with Markov chain Monte Carlo, creating an algorithm that has strong capacity for parallelism, and provides an estimate of model evidence, a component of Bayesian model selection. It starts with samples from the prior distribution $\Pr(\theta)$, and utilizes importance sampling to provide samples from intermediate PDFs given by $\Pr(D|\theta)^{\beta}\Pr(\theta)$ while introducing diversity through MCMC steps. $\Pr(D|\theta)$ is the likelihood function and β is the temperature parameter that monotonically increases from 0 to 1, with step sizes chosen adaptively (i.e. varying from one step to the next) such that the coefficient of variation of the importance sampling weights does not exceed a threshold (see [34] for a relevant discussion).

This example involves a driver python script, `tmcmc_bimodal.py`, that invokes the executable (based on provided C++ code) `tmcmc_bimodal.x`. This executable sets up the MCMC class object, specifying the dimensionality of the problem, number of samples required, and number of processes for parallel evaluation of likelihood and prior, along with other algorithmic choices. The TMCMC algorithm proceeds with loading the user-provided prior PDF samples from `tmcmc_prior_samples.dat`, and iterating through the cooling steps.

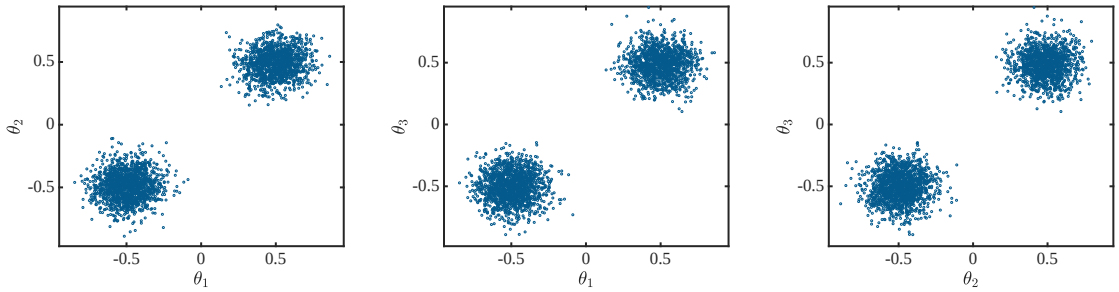


Figure 5.23: 2-dimensional scatter plots of posterior samples

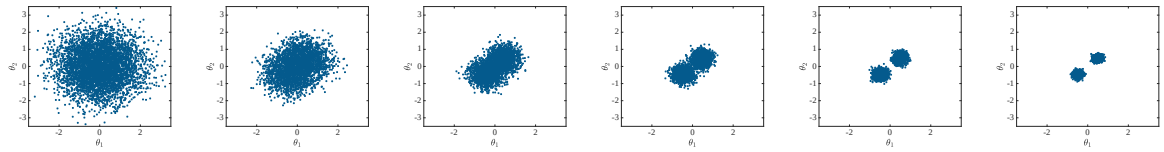


Figure 5.24: 2-dimensional scatter plots of intermediate TMCMC samples, from prior to posterior

In each step, two shell scripts are invoked to spawn multiple processes for parallel evaluation of likelihood and prior PDFs, namely `tmcmc_getLL.sh` and `tmcmc_getLP.sh`, respectively. In turn, each process involves the execution of `bimodal.x` (with corresponding C++ source `bimodal.cpp`) which evaluates the prior and/or likelihood for an ensemble of samples at one particular TMCMC step.

The script then reads in the MCMC posterior samples file, and performs some postprocessing. Unless the flag `--noplots` is specified, the script computes and plots the following:

- 2-dimensional scatter plots of posterior samples
- 2-dimensional scatter plots of intermediate TMCMC samples (for intermediate β values)
- The marginal posterior distribution for each variable, as well as the marginal joint distribution for each pair of variables

Sample Results

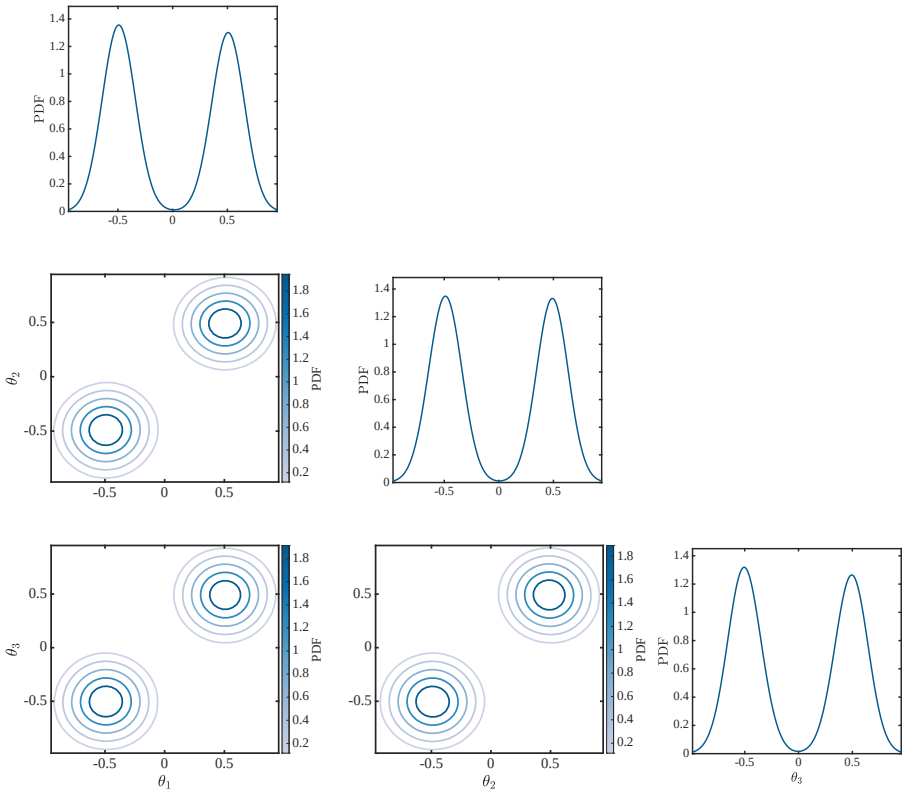


Figure 5.25: Marginal posterior distributions for all variables, as well as marginal joint posteriors

Forward Propagation of Uncertainties, Surrogate Construction and Global Sensitivity Analysis

Overview

- Located in `examples/uqpc`

- A collection of scripts that propagate input parameter uncertainties to output via PC expansions. As a special, and most commonly used, case the scripts can construct a PC surrogate for a multi-output computational model. The latter is as a black box simulation code. The workflow also provides tools for global sensitivity analysis of the outputs of this black box model with respect to its input parameters or input PC germs.

Theory

Consider a function $f(\lambda; x)$ where $\lambda = (\lambda_1, \ldots, \lambda_d)$ are the model *input* parameters of interest, while $x \in \mathbb{R}^m$ are *design* parameters with controllable values. For example, x can denote a spatial coordinate or a time snapshot, or it can simply enumerate multiple quantities of interest. Furthermore, assume the input parameters are given by a (generally, joint) Polynomial Chaos expansions as

$$\lambda_i = \sum_{k=0}^{K_{in}-1} a_{ik}\Psi_k(\xi), \text{ for } i = 1, \ldots, d, \tag{5.11}$$

where $\Psi_k(\xi) = \Psi_k(\xi_1, \ldots, \xi_{\tilde{d}})$ are standard multivariate polynomials, defined as products of univariate polynomials $\psi_{k_i}(\xi_i)$ as follows:

$$\Psi_k(\xi) = \psi_{k_1}(\xi_1) \ldots \psi_{k_d}(\xi_{\tilde{d}}). \tag{5.12}$$

Note that the *stochastic* input $\xi = (\xi_1, \ldots, \xi_{\tilde{d}})$ does not need to have the same dimensionality as the parameter vector $\lambda = (\lambda_1, \ldots, \lambda_d)$, i.e. $d \neq \tilde{d}$ in general. However, most commonly, it is. For example, if parameters are given by their ranges only,

$$\lambda_i \in [a_i, b_i] \qquad \text{for} \quad i = 1, \ldots .d, \tag{5.13}$$

one can think of it as first-order Legendre-Uniform PC by the linear transformation

$$\lambda_i = \frac{b_i + a_i}{2} + \frac{b_i - a_i}{2}\xi_i, \quad \text{for } i = 1, \ldots, d. \tag{5.14}$$

The goal is to build a PC representation for each value of design parameter x, i.e. for $l = 1, \ldots, L$,

$$f(\lambda; x_l) \approx g_c(\lambda; x_l) = \sum_{k=0}^{K-1} c_{kl}\Psi_k(\xi). \tag{5.15}$$

Note that if inputs are given independently on their respective ranges, $\lambda \in [a_i, b_i]$, the PC expansion (5.15) is simply a polynomial surrogate with respect to scaled inputs

$$\xi_i = \frac{\lambda_i - \frac{b_i + a_i}{2}}{\frac{b_i - a_i}{2}} \in [-1, 1] \qquad \text{for} \quad i = 1, \ldots .d. \tag{5.16}$$

A typical truncation rule in (5.15) is defined according to the total order of the basis terms, i.e. only polynomials with the total order $\leq p$ are retained for some positive integer order p, implying $|k_1| + \cdots + |k_d| \leq p$, and $K = (d + p)!/(d!p!)$. The scalar index k is simply counting the *multi-indices* $(k_1, \ldots, k_d)$.

The three generic methods of finding the PC coefficients c_{kl} are detailed below.

Projection: The basis orthogonality enables the projection formulae

$$c_{kl} = \int_{\Omega} f(\lambda(\xi); x_l)\Psi_k(\xi)\pi(\xi)d\xi \tag{5.17}$$

where $\lambda(\xi)$ simply denotes the PC form (5.13) or the linear scaling relation in (5.14), and $\pi(\xi)$ is the PDF of ξ. Note that $\pi(\xi) = 2^{-d}$ for the linear, Legendre-Uniform PC case.

The projection integral is taken by quadrature integration

$$c_{kl} \approx \sum_{q=1}^{N} w_q f(\lambda(\xi^{(q)}); x_l)\Psi_k(\xi^{(q)}), \tag{5.18}$$

where $\xi^{(q)}$ are Gaussian quadrature points, and w_q are the associated weights. See the description of the app `pce_resp` as well.

Bayesian Least-Squares Regression: In cases when model outputs are noisy, or highly non-linear, or when one can not afford model evaluations at a *predefined* quadrature locations, it is convenient to reformulate the coefficient finding as a regression problem. More specifically, consider the least-squares problem that attempts to solve, for each design condition $l = 1, \ldots, L$,

$$\arg\min_{c} \sum_{s=1}^{N} \left(f(\lambda(\xi^{(s)}); x_l) - \sum_{k=0}^{K-1} c_{kl}\Psi_k(\xi^{(s)}) \right)^2. \tag{5.19}$$

Due to linearity of the polynomial form with respect to coefficients c_{kl}, the exact solution of this minimization problem is available via matrix manipulations, see, e.g. [26]. In the description of the app `regression`, the Bayesian generalization of this least-squares fit is described.

Bayesian Compressive Sensing (BCS): For high-dimensional problems, i.e. when d is sufficiently large, the number of terms K for a reasonable truncation order in the output PC (5.15) is large. In such cases, one typically has fewer model evaluations available than the number of basis terms, i.e. the problem is *underdetermined*. In such situations, one can employ ℓ_1 regularization techniques, building on the compressive sensing work from image processing community. Here, we have implemented the Bayesian reformulation of such an algorithm, with approximate and fast procedure of pruning the unnecessary terms in the PC expansion. See [1, 28] for more details on BCS.

After computing the PC coefficients c_{kl}, one can extract the global sensitivity information, also called Sobol indices or variance-based decomposition. For example, the *main sensitivity* index with respect to the dimension i (or variable ξ_i) is

$$S_i(x_l) = \frac{\sum_{k\in\mathbb{I}_i} c_{kl}^2 ||\Psi_k||^2}{\sum_{k=1}^{K-1} c_{kl}^2 ||\Psi_k||^2}, \tag{5.20}$$

where $\mathbb{I}_i$ is the indices of basis terms that involve only the variable ξ_i, i.e. the one-dimensional monomials $\psi_1(\xi_i), \psi_2(\xi_i), ...$ In other words, these are basis terms corresponding to multi-indices with the only non-zero entry at the i-th location. For further details regarding global sensitivity analysis (GSA), see the theory side of the description of the ''GSA via Sampling' workflow, and the description of the app `pce_sens` in Section 4.

Implementation

The script set consists of the following files:

- `uq_pc.py` : the main script, see Table 5.1. Also one can run `uq_pc.py -h` for help in the terminal.
- `model.py` : black-box example model. See Figure 5.26 for visual explanation of the expected input-output structure. Try `model.py -h` for help in the terminal. The syntax of this script is

  ```
  model.py -i <input_file>  -o <output_file> -m <model_name>
  ```

  The list of arguments:

  `-i <input_file>` : $N \times d$ file that stores the input parameter ensemble of N samples of d-dimensional input.

  `-o <output_file>` : $N \times L$ file where output $f(\lambda^{(i)}, x_l)$ is stored, with N rows (number of input parameter samples) and L columns (number of outputs, or number of design parameter values).

  `-m <model_name>` : Name of the model. Options are `example` (default) and `genz`.

  * `example` : an example function $f(\lambda; x) = \left(\sum_{i=1}^{d} \lambda_i\right)\left(\sum_{i=1}^{d} \frac{\lambda_i+\lambda_i^2}{i^x}\right)$ is implemented that also produces the file `designPar.dat` for design parameters $x_j = j$ for $j = 1, \ldots, L$, with $L = 7$. The function has d inputs and $L = 7$ outputs.
  * `genz` : this function has two outputs ($L = 2$): Gaussian and Osccillatory Genz functions.

  User can create a black-box `model.py` with similar I/O structure, or augment `model.py` with their own function.

- `plot_prep.py` : plotting before surrogate construction. The syntax of the script is `plot_prep.py <plot_type> <...>`.
  Try `plot_prep.py -h` or `plot_prep.py <plot_type> -h`, where `plot_type` is

  `pcoord` : Plots the inputs in parallel coordinates.

`xx` : Plots one input parameter versus another.

`xy` : Plots one of the outputs versus one of the inputs.

`xxy` : Surface-plot of one of the outputs versus two inputs.

- `plot.py` : plotting after surrogate construction, reading the pickle file `results.pk` produced by `uq_pc.py`. The syntax of the script is `plot.py <plot_type> <...>`. Try `plot.py -h` or `plot.py <plot_type> -h`, where `plot_type` is

  `sens` : Plots the sensitivity information in a bar-plot. This command also produces `allsens_main.dat` or `allsens_total.dat`, the sensitivity indices in a format $r \times d$, where each row corresponds to a single value for the design parameter, and each column corresponds to the sensitivity index of a parameter.

  `senscirc` : Plots sensitivity circular plots for all outputs, and averaged as well.

  `sensmat` : Plots sensitivity matrix for all outputs and for the most important inputs.

  `dm` : Plots model-vs-data for all values of the design parameter (i.e. for all outputs).

  `idm` : Plots model and data values on the same axis, for all the values of the design parameter.

  `1d` : Plots 1d surrogate (the rest of parameters, if any, at nominal) versus data, for all outputs.

  `2d` : Plots 2d surrogate (the rest of parameters, if any, at nominal) versus data, for all outputs.

  `mindex` : Visualizes the multiindex for all outputs.

  `micf` : Plots the multiindex for all outputs in a different way, meaningful only for 2d and 3d.

  `pdf` : Plots the PDF of the output. Sampling size parameter is hardwired.

  `senserb1` : Computes sensitivities with errorbars. Not tested enough. Some hardwired parameters. Requires `uq_pc.py` method (`-m`) `lsq` or `bcs` and prediction mode (`-i`) `msc`. Relies on script `model_sens.x` as a black-box model-sensitivity evaluator for each fixed sample pf PC coefficients.

  `senserb2` : Plots the sensitivities with errorbars. Not tested enough. Needs to be run only after `plot.py senserb1`.

  The user is encouraged to enhance or change the visualization scripts on their own, taking `plot.py` as an example of unrolling the surrogate construction output pickle file `results.pk`.

  Both `plot_prep.py` and `plot.py` would accept (but not require!) parameter name file `pnames.txt` (d rows) and output names file `outnames.txt` (r rows) if one wants to have informative plot labels.

Other auxiliary or example scripts are listed below:

- `prepare_inpc.py` : Prepares PC coefficient file given marginal PCs or samples. The output, `param_pcf.txt` file can be used with flag `-c` in `uq_pc.py`.
- `generate_inputsamples.py` : Auxiliary script to generate example jointly distributed random samples.
- `join_results.py` : Auxiliary script as an example of joining a set of surrogate construction pickle files into a single pickle file `results.pk`.
- `model_sens.x` : Auxiliary script as a sensitivity evaluation black-box for given PC coefficients.
- `transpose_file.x` : Transpose a given matrix file.
  Syntax: `transpose_file.x <file_in> > <file_out>`
- `scale.x` : Scale given matrix file to or from a given hypercube to $[-1, 1]^d$. Syntax: `scale.x <input> <to or from> <domain> <output>`
- `getrange.x` : Get parameter ranes of a given set of samples. Syntax: `getrange.x <samples.dat> [cushion_fraction] > <ranges.dat>`
- `example_0.x` : Minimal example workflow. Assumes `input.dat` ($N \times d$) and `output.dat` ($N \times L$) are given.
- `example_1.x` : Surrogate construction example workflow.
- `example_2.x` : Uncertainty propagation example workflow.
- `example_3.x` : Surrogate-for-time-series (i.e. each output is a snapshot) example workflow.

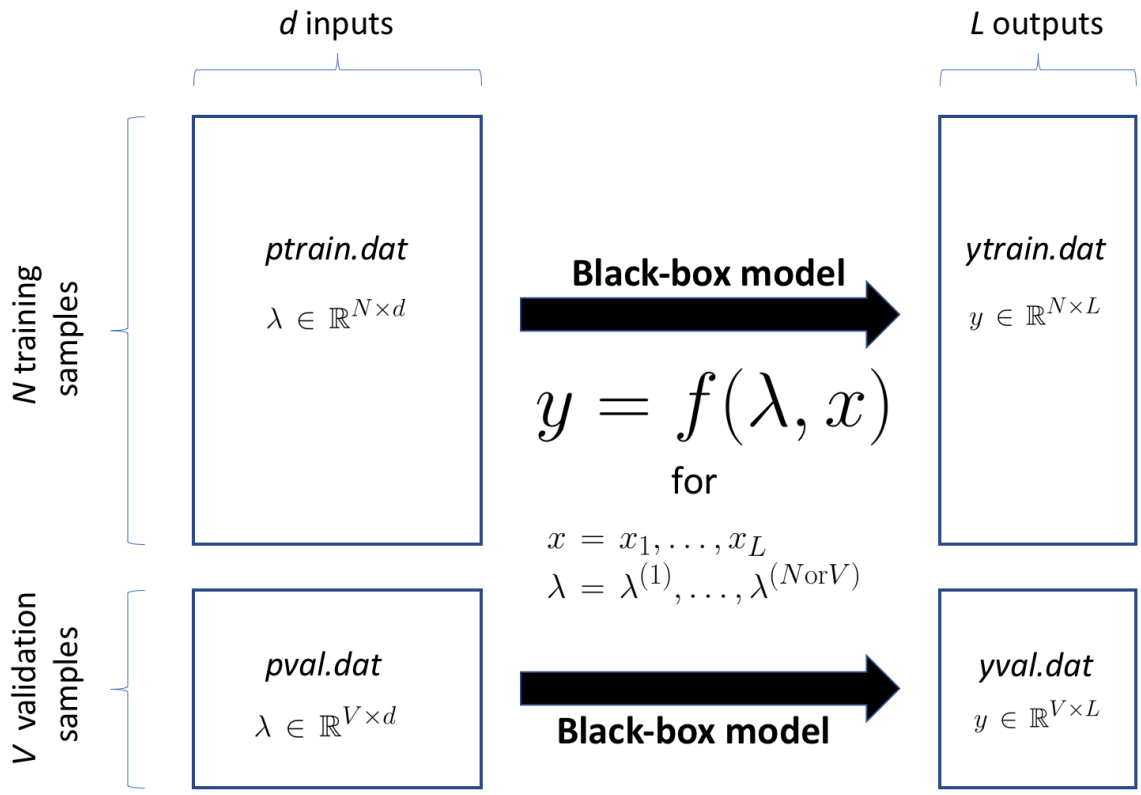


Figure 5.26: Sketch of the expected input-output structure of the black-box model.

Argument	Options	Description
`-r <run_regime>`		The regime in which the workflow is employed.
	`online_example`	A black-box model `model(...)`, defined in `model.py`, is run directly as parameter ensemble becomes available. User can provide their own `model(...)` with minimal surgery.
	`online_bb`	A black-box model script `model.x <input_file> <output_file>` is executed. The intention is that the user provides the `model.x` script with the appropriate I/O.
	`offline_prep`	Prepare the input parameter ensemble and store in `ytrain.dat` and, if validation is requested, `yval.dat`. The user then should run the model (`model.py ptrain.dat ytrain.dat` and perhaps `model.py pval.dat yval.dat`) in order to provide ensemble output for the `offline_post` stage.
	`offline_post`	Postprocess the output ensemble, assuming the model is run offline with input ensemble provided in the `offline_prep` stage producing `ytrain.dat` and, if validation is requested, `yval.dat`. The rest of the arguments should remain the same as in `offline_prep`.
`-p <domain_file>`		A file with d rows and 2 columns, where d is the number of parameters and each row consists of the lower and upper bound of the corresponding parameter.
`-c <inpc_file>`		Input PC coefficient file.
`-d <in_pcdim>`		Input PC stochastic dimension.
`-x <pctype>`	`HG,LU,LU_N, GLG,JB,SW`	PC type.
`-o <in_pcord>`		Input PC order.
`-m <fit_method>`		The method of finding the PC surrogate coefficients.
	`proj`	Projection method outlined in (5.17) and (5.18)
	`lsq`	Bayesian least-squares.
	`bcs`	Bayesian compressive sensing.
`-s <sam_method>`		The input parameter sampling method.
	`rand`	Uniformly random points. To be implemented.
	`quad`	Quadrature points. This sampling scheme works with the projection method only, described in (5.18)
`-n <nqd>`		Number of samples requested if `sam_method=rand`, or the number of quadrature points per dimension, if `sam_method=quad` and `sparsity=full`, or the level of quadrature if `sam_method=quad` and `sparsity=sparse`.
`-v <nval>`		Number of uniformly random samples generated for PC surrogate validation, can be equal to 0 to skip validation.
`-f <sparsity>`	`full, sparse`	Sparsity, if `sam_method=quad`.
`-t <out_pcord>`		Output PC order.
`-i <pred_mode>`	`m, ms, msc`	Prediction mode to compute the mean only (`m`), mean and standard deviation (`ms`), mean and full covariance with respect to x (`msc`).
`-e <tolerance>`		Tolerance parameter (currently for `fit_method=bcs` only).
`-z <cleanup>`		Flag to cleanup after (be careful: removes *log and *dat files).

Hardwired inputs	(also see Figure 5.26)
`ptrain.dat`	$N \times d$ matrix, each row is a d-variate parameter sample
`qtrain.dat`	the same scaled to [-1,1]
`wtrain.dat`	quadrature weights only if sampling method is quadrature
`ytrain.dat`	$N \times L$ vector of outputs
`pval.dat`	$V \times d$ matrix, each row is a d-variate parameter sample
`qval.dat`	the same scaled to [-1,1]
`yval.dat`	$V \times L$ vector of outputs
Output file	
`results.pk`	Python pickle file containing a dictionary with all the results. The visualization `plot.py` serves as an example of how to unroll it.

Table 5.1: Arguments of the main script `uq_pc.py`.

Global Sensitivity Analysis via Sampling

Overview

- Located in `PyUQTk/sens`

- A collection of Python functions that generate input samples for black-box models, followed by functions that post-process model outputs to generate total, first-order, and joint effect Sobol indices

Theory

Let $X = (X_1, \cdots, X_n) : \Omega \rightarrow \mathcal{X} \subset \mathbb{R}^n$ be an $n-$dimensional Random Variable in $L^2(\Omega, \mathcal{S}, P)$ with probability density $X \sim p_X(x)$. Let $x = (x_1, \cdots, x_n) \in \mathcal{X}$ be a sample drawn from this density, with $\mathcal{X} = \mathcal{X}_1 \otimes \mathcal{X}_2 \otimes \cdots \otimes \mathcal{X}_n$, and $\mathcal{X}_i \subset \mathbb{R}$ is the range of X_i.

Let $X_{-i} = (X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_n) : \Omega \rightarrow \mathcal{X}_{-i} \subset \mathbb{R}^{n-1}$, where $X_{-i} \sim p_{X_{-i}|X_i}(x_{-i}|x_i) = p_X(x)/p_{X_i}(x_i)$, $p_{X_i}(x_i)$ is the marginal density of X_i, $x_{-i} = (x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$, and $\mathcal{X}_{-i} = \mathcal{X}_1 \otimes \cdots \otimes \mathcal{X}_{i-1} \otimes \mathcal{X}_{i+1} \otimes \cdots \otimes \mathcal{X}_n$.

Consider a function $Y = f(X) : \Omega \rightarrow \mathbb{R}$, with $Y \in L^2(\Omega, \mathcal{S}, P)$. Further, let $Y \sim p_Y(y)$, with $y = f(x)$. Given the variance of f is finite , one can employ the law of total variance[1,2] to decompose the variance of f as

$$V[f] = V_{x_i}[E[f|x_i]] + E_{x_i}[V[f|x_i]] \tag{5.21}$$

The conditional mean, $E[f|x_i] \equiv E[f(X)|X_i = x_i]$, and conditional variance, $V[f|x_i] = V[f(X)|X_i = x_i]$, are defined as

$$\langle f \rangle_{-i} \equiv E[f|x_i] = \int_{\mathcal{X}_{-i}} f(x) p_{X_{-i}|X_i}(x_{-i}|x_i) dx_{-i} \tag{5.22}$$

$$\begin{aligned} V[f|x_i] &= E[(f - \langle f \rangle_{-i})^2 | x_i] \\ &= E[(f^2 - 2f\langle f \rangle_{-i} + \langle f \rangle_{-i}^2) | x_i] \\ &= E[f^2|x_i] - 2\langle f \rangle_{-i}\langle f \rangle_{-i} + \langle f \rangle_{-i}^2 \\ &= \int_{\mathcal{X}_{-i}} f(x)^2 p_{X_{-i}|X_i}(x_{-i}|x_i) dx_{-i} - \langle f \rangle_{-i}^2 \end{aligned} \tag{5.23}$$

[1] `en.wikipedia.org/wiki/Law_of_total_variance`

[2] `en.wikipedia.org/wiki/Law_of_total_expectation`

The terms in the *rhs* of Eq. (5.21) can be written as

$$\begin{aligned} V_{x_i}[E[f|x_i]] &= E_{x_i}[\ (E[f|x_i] - E_{x_i}[E[f|x_i]])^2\ ] \\ &= E_{x_i}[\ (E[f|x_i] - f_0)^2\ ] \\ &= E_{x_i}[\ (E[f|x_i])^2\ ] - f_0^2 \\ &= \int_{\mathcal{X}_i} E[f|x_i]^2 p_{X_i}(x_i) dx_i - f_0^2 \end{aligned} \tag{5.24}$$

where $f_0 = E[f] = E_{x_i}[E[f|x_i]]$ is the expectation of f, and

$$E_{x_i}[V[f|x_i]] = \int_{\mathcal{X}_i} V[f|x_i] p_{X_i}(x_i) dx_i \tag{5.25}$$

The ratio

$$S_i = \frac{V_{x_i}[E[f|x_i]]}{V[f]} \tag{5.26}$$

is called the first-order Sobol index [33] and

$$S_{-i}^T = \frac{E_{x_i}[V[f|x_i]]}{V[f]} \tag{5.27}$$

is the total effect Sobol index for x_{-i}. Using Eq. (5.21), the sum of the two indices defined above is

$$S_i + S_{-i}^T = S_{-i} + S_i^T = 1 \tag{5.28}$$

Joint Sobol indices S_{ij} are defined as

$$S_{ij} = \frac{V_{x_i,x_j}[E[f|x_i, x_j]]}{V[f]} - S_i - S_j \tag{5.29}$$

for $i, j = 1, 2 \ldots, n$ and $i \neq j$.

S_i can be interpreted as the fraction of the variance in model f that can be attributed to the i-th input parameter only, while S_{ij} is the variance fraction that is due to the joint contribution of i-th and j-th input parameters. S_i^T measures the fractional contribution to the total variance due to parameter x_i and its interactions with all other model parameters.

The Sobol indices are numerically estimated using Monte Carlo (MC) algorithms proposed by Saltelli [25] and Kucherenko *et al* [17]. Let $x^k = (x_1, \cdots, x_n)^k$ be a sample of X drawn from p_X. Let x'^k_{-i} be a sample from the conditional distribution $p_{X_{-i}|X_i}(x'_{-i}|x_i^k)$, and x''^k_i a sample from the conditional distribution $p_{X_i|X_{-i}}(x''_i|x^k_{-i})$.

The expectation $f_0 = E[f]$ and variance $V = V[f]$ are estimated using the x^k samples as

$$f_0 \approx \frac{1}{N} \sum_{k=1}^{N} f(x^k), \quad V \approx \frac{1}{N} \sum_{k=1}^{N} f(x^k)^2 - f_0^2 \tag{5.30}$$

where N is the total number of samples. The first-order Sobol indices S_i are estimated as

$$S_i \approx \frac{1}{V}\left(\frac{1}{N}\sum_{k=1}^{N} f(x^k) f(x'^k_{-i} \cup x^k_i) - f_0^2\right) \tag{5.31}$$

The joint Sobol indices are estimated as

$$S_{ij} \approx \frac{1}{V}\left(\frac{1}{N}\sum_{k=1}^{N} f(x^k) f(x'^k_{-(i,j)} \cup x^k_{i,j}) - f_0^2\right) - S_i - S_j \tag{5.32}$$

For S_i^T, UQTk offers two alternative MC estimators. In the first approach, S_i^T is estimated as

$$S_i^T = 1 - S_{-i} \approx 1 - \frac{1}{V}\left(\frac{1}{N}\sum_{k=1}^{N} f(x^k) f(x''^k_i \cup x^k_{-i}) - f_0^2\right) \tag{5.33}$$

In the second approach, S_i^T is estimated as

$$S_i^T \approx \frac{1}{2V}\left(\frac{1}{N}\sum_{k=1}^{N} \left(f(x^k) - f(x^k_{-i} \cup x''^k_i)\right)^2\right) \tag{5.34}$$

Implementation

Directory `pyUQTk/sensitivity` contains two Python files

- `gsalib.py` : set of Python functions implementing the MC sampling and estimators for Sobol indices
- `gsatest.py` : workflow illustrating the computation of Sobol indices for a toy problem

`gsalib.py` implements the following functions

- `genSpl_Si(nspl,ndim,abrng,**kwargs)` : generates samples for Eq. (5.31). The input parameters are as follows

  `nspl`: number of samples N,

  `ndim`: dimensionality n of the input parameter space ,

  `abrng`: a 2-dimensional array $n \times 2$, containing the range for each component x_i.

  The following optional parameters can also be specified

  `splout`: name of ascii output file for MC samples

  `matfile`: name of binary output file for select MC samples. These samples are used in subsequent calculations of joint Sobol indices

  `verb`: verbosity level

  `nd`: number of significant digits for ascii output

  The default values for optional parameters are listed in `gsalib.py`

- `genSens_Si(modeval,ndim,**kwargs)` : computes first-order Sobol indices using Eq. (5.31). The input parameters are as follows

  `modeval`: name of ascii file with model evaluations,

  `ndim`: dimensionality n of the input parameter space

  The following optional parameter can also be specified

  `verb`: verbosity level

  The default value for the optional parameter is listed in `gsalib.py`

- `genSpl_SiT(nspl,ndim,abrng,**kwargs)` : generates samples for Eqs. (5.33-5.34). The input parameters are as follows

  `nspl`: number of samples N,

  `ndim`: dimensionality n of the input parameter space ,

  `abrng`: an 2-dimensional array $n \times 2$, containing the range for each component x_i.

  The following optional parameters can also be specified

  `splout`: name of ascii output file for MC samples

  `matfile`: name of binary output file for select MC samples. These samples are used in subsequent calculations of Sobol indices

  `verb`: verbosity level

  `nd`: number of significant digits for ascii output

  The default values for optional parameters are listed in `gsalib.py`

- `genSens_SiT(modeval,ndim,**kwargs)` : computes total Sobol indices using either Eq. (5.33) or Eq. (5.34). The input parameters are as follows

  `modeval`: name of ascii file with model evaluations,

  `ndim`: dimensionality n of the input parameter space

  The following optional parameter can also be specified

  `type`: specifies wether to use Eq. (5.33) for type = "type1" or Eq. (5.34) for type $\neq$ "type1"

  `verb`: verbosity level

  The default value for the optional parameter is listed in `gsalib.py`

- `genSpl_Sij(ndim,**kwargs)` : generates samples for Eq. (5.32). The input parameters are as follows

  `ndim`: dimensionality n of the input parameter space ,

  The following optional parameters can also be specified

  `splout`: name of ascii output file for MC samples

  `matfile`: name of binary output file for select MC samples saved by `genSpl_Si`.

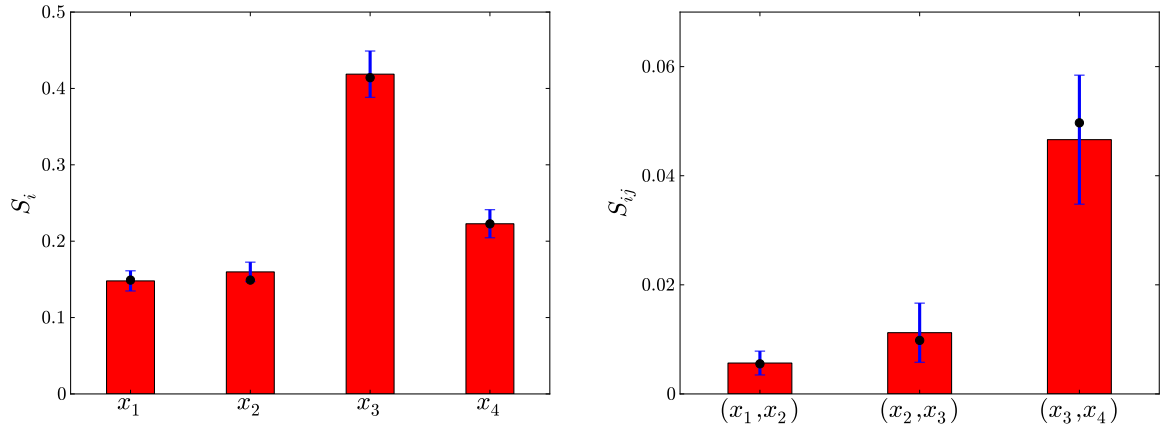


Figure 5.27: First-order (left frame) and joint (right frame) Sobol indices for the model given in Eq. (5.35). The black circles show the theorerical values, computed analytically, and the error bars correspond to $\pm\sigma$ computed based on an ensemble of 10 runs.

`verb`: verbosity level

`nd`: number of significant digits for ascii output

The default values for optional parameters are listed in `gsalib.py`

- `genSens_Sij(sobolSi,modeval,**kwargs)` : computes joint Sobol indices using Eq. (5.32). The input parameters are as follows

  `sobolSi`: array with values for first-order Sobol indices S_i

  `modeval`: name of ascii file with model evaluations.

  The following optional parameter can also be specified

  `verb`: verbosity level

  The default value for the optional parameter is listed in `gsalib.py`

`gsatest.py` provides the workflow for the estimation of Sobol indices for a simple model given by

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n-1} i^2 x_i x_{i+1} \tag{5.35}$$

In the example provided in this file, n (`ndim` in the file) is set equal to 4, and the number of samples N (`nspl` in the file) to 10^4. Figures 5.27 and 5.28 show results based on an ensemble of 10 runs. To generate these results run the example workflow:

```
python gsatest.py
```

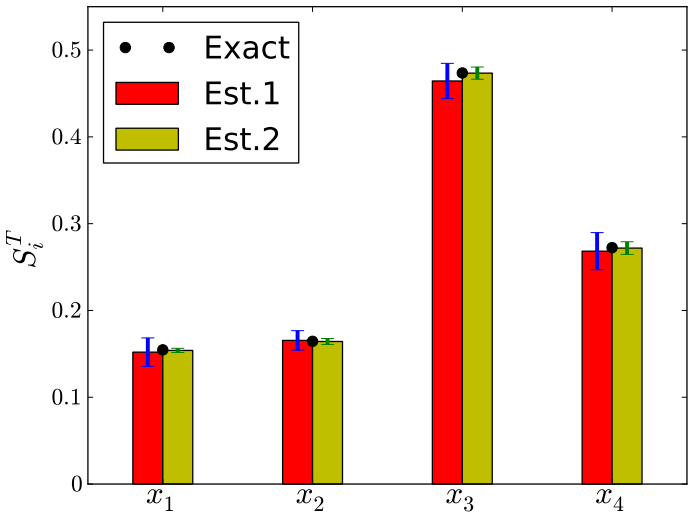


Figure 5.28: Total-order Sobol indices for the model given in Eq. (5.35). The red bars shows results based on Eq. (5.33) while the yellow bars are based on Eq. (5.34). The black circles show the theorerical values, computed analytically, and the error bars correspond to $\pm\sigma$ computed based on an ensemble of 10 runs. For this model, Eq. (5.34) provides more accurate estimates for S_i^T compared to results based on Eq. (5.33).

Karhunen-Loève Expansion of a Stochastic Process

- Located in `examples/kle_ex1`
- Some examples of the construction of 1D and 2D Karhunen-Loève (KL) expansions of a Gaussian stochastic process, based on sample realizations of this stochastic process.

Theory

Assume stochastic process $F(x,\omega) : D \times \Omega \to \mathbb{R}$ is L^2 random field on D, with covariance function $C(x,y)$. Then F can be written as

$$F(x,\omega) = \langle F(x,\omega) \rangle_\omega + \sum_{k=1}^{\infty} \sqrt{\lambda_k} f_k(x) \xi_k \tag{5.36}$$

where $f_k(x)$ are eigenfunctions of $C(x,y)$ and λ_k are corresponding eigenvalues (all positive). Random variables ξ_k are uncorrelated with unit variance. Projecting realizations of F onto f_k leads to samples of ξ_k. These samples are generally not independent. In the special case when F is a Gaussian random process, ξ_k are i.i.d. normal random variables.

The KL expansion is *optimal*, i.e. of all possible orthonormal bases for $L^2(D \times \Omega)$ the above $\{f_k(x) | k = 1, 2, \ldots\}$ minimize the mean-square error in a finite linear representation

of $F(\cdot)$. If known, the covariance matrix can be specified analytically, e.g. the square-exponential form

$$C(x,y) = \exp\left(-\frac{|x-y|^2}{c_l^2}\right) \tag{5.37}$$

where $|x-y|$ is the distance between x and y and c_l is the correlation length. The covariance matrix can also be estimated from realizations, e.g.

$$C(x,y) = \frac{1}{N_\omega}\sum_{\omega}(F(x,\omega) - \langle F(x,\omega)\rangle_\omega)(F(y,\omega) - \langle F(y,\omega)\rangle_\omega) \tag{5.38}$$

where N_ω is the number of samples, and $\langle F(x,\omega)\rangle_\omega$ is the mean over the random field realizations at x.

The eigenvalues and eigenvectors in Eq. (5.36) are solutions of the Fredholm equation of second kind:

$$\int C(x,y)f(y)dy = \lambda f(x) \tag{5.39}$$

One can employ the Nystrom algorithm [20] to discretize of the integral in the left-hand side of Eq. (5.39)

$$\sum_{i=1}^{N_p} w_i C(x,y_i)f(y_i) = \lambda f(x) \tag{5.40}$$

Here w_i are the weights for the quadrature rule that uses N_p points y_i where realizations are provided. In a 1D configuration, one can employ the weights corresponding to the trapezoidal rule:

$$w_i = \begin{cases} \frac{y_2-y_1}{2} & \text{if } i=1, \\ \frac{y_{i+1}-y_{i-1}}{2} & \text{if } 2 \le i < N_p, \\ \frac{y_{N_p}-y_{N_p-1}}{2} & \text{if } i = N_p, \end{cases} \tag{5.41}$$

After further manipulation, Eq. (5.40) is written as

$$Ag = \lambda g$$

where $A = WKW$ and $g = Wf$, with W being the diagonal matrix $W_{ii} = \sqrt{w_i}$ and $K_{ij} = C(x_i, y_j)$. Since matrix A is symmetric, one can employ efficient algorithms to compute its eigenvalues λ_k and eigenvectors g_k. Currently **UQTk** relies on the *dsyevx* function provided by the LAPACK library.

The KL eigenvectors are computed as $f_k = W^{-1}g_k$ and samples of random variables ξ_k are obtained by projecting realizations of the random process F on the eigenmodes f_k

$$\xi_k|_{\omega_l} = \langle F(x,\omega_l) - \langle F(x,\omega)\rangle_\omega, f_k(x)\rangle_x / \sqrt{\lambda_k}$$

Numerically, these projections can be estimated via quadrature

$$\xi_k|_{\omega_l} = \sum_{i=1}^{N_p} w_i \left(F(x_i,\omega_l) - \langle F(x_i,\omega)\rangle_\omega\right) f_k(x_i)/\sqrt{\lambda_k} \tag{5.42}$$

If F is a Gaussian process, ξ_k are *i.i.d.* normal RVs, i.e. automatically have first order Wiener-Hermite Polynomial Chaos Expansions (PCE). In general however, the KL RVs can be converted to PCEs (not shown in the current example).

1D Examples

In this section we are presenting 1D RFs generated with **kl_1D.x**. The RFs are generated on a non-uniform 1D grid, with smaller grid spacing near $x = 0$ and larger grid spacing towards $x = 1$. This grid is computed using an algebraic expression [15]

$$x_i = L\frac{\beta + 1 - (\beta - 1)r_i}{r_i + 1}, \;\; r_i = \left(\frac{\beta + 1}{\beta - 1}\right)^{1-\eta_i}, \;\; \eta_i = \frac{i-1}{N_p - 1}, \;\; i = 1, 2, \ldots, N_p \tag{5.43}$$

The $\beta > 1$ factor in the above expression controls the compression near $x = 0$. It results in higher compression as β gets closer to 1. The examples shown in this section are based on default values for the parameters that control the grid definition in **kl_1D**:

$$\beta = 1.1, \;\; L = 1, \;\; N_p = 129$$

Figure 5.29 shows sample realizations for 1D random fields (RF) generated with a square-exponential covariance matrix employing several correlation lenghts c_l. These figures were generated with

```
./mkplots.py samples 0.05
./mkplots.py samples 0.10
./mkplots.py samples 0.20
```

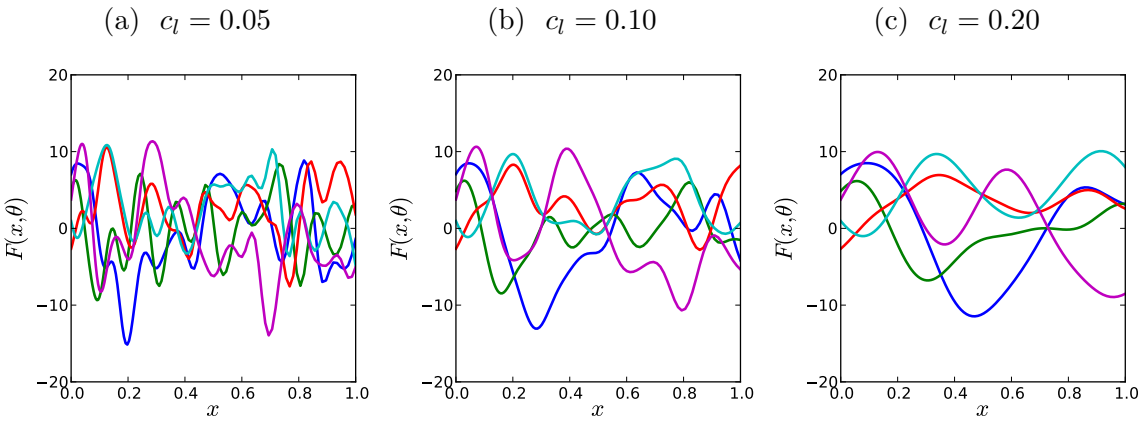


Figure 5.29: Sample 1D random field realizations for several correlation lengths c_l.

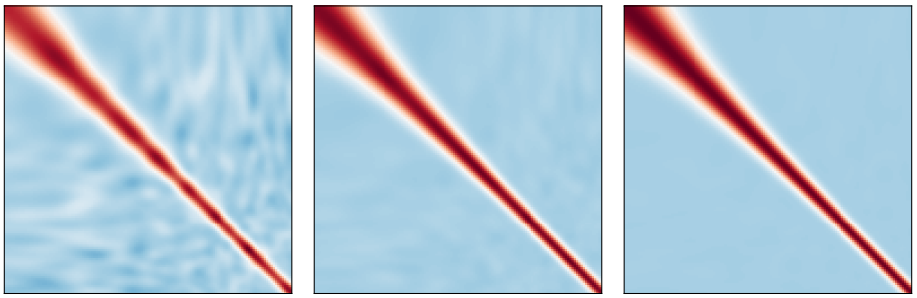
Once the RF realizations are generated the covariance matrix is discarded and a "numerical" covariance matrix is estimated based on the available realizations. Figure 5.30 shows shaded illustration of covariance matrices computed using several sets of 1D RF samples.

These figures were generated with

```
./mkplots.py numcov 0.05 512      ./mkplots.py numcov 0.20 512
./mkplots.py numcov 0.05 8192     ./mkplots.py numcov 0.20 8192
./mkplots.py numcov 0.05 131072   ./mkplots.py numcov 0.20 131071
```

These matrices employ RF samples generated on a non-uniform grid with higher densitiy of points near the left boundary. Hence, the matrix entries near the diagonal in the upper right corner show larger values. Grids grow further apart away from the left boundary hence the region near the diagonal grows thinner for these grid points.

(a) $c_l = 0.05, N_\omega = 2^9$ (b) $c_l = 0.05, N_\omega = 2^{13}$ (c) $c_l = 0.05, N_\omega = 2^{17}$



(d) $c_l = 0.20, N_\omega = 2^9$ (e) $c_l = 0.20, N_\omega = 2^{13}$ (f) $c_l = 0.20, N_\omega = 2^{17}$

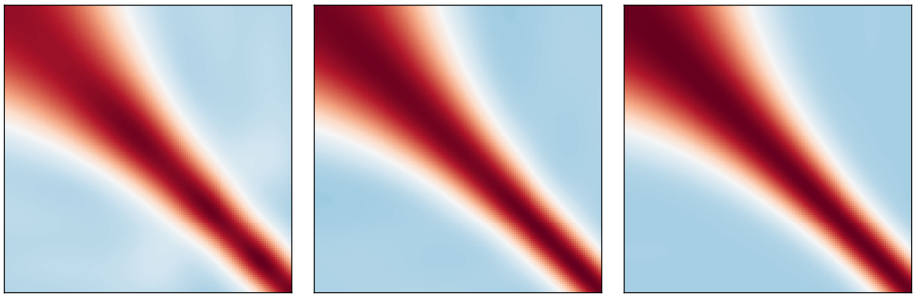


Figure 5.30: Illustration of covariance matrices computed from 1D RF realizations. Red corresponds to large values, close to 1, while blue corresponds to small values, close to 0.

Figure 5.31 shows the eigenvalue solution of Fredholm equation (5.39) in its discretized form given by Eq. (5.40). This figure was generated with

```
./mkplots.py pltKLeig1D 512 131072
```

For this 1D example problem, $2^9 = 512$ RF realizations are sufficient to estimate the KLE eigenvalue spectrum. As the correlation length decreases the eigenvalues decrease more slowly suggesting that more terms are needed to represent RF fluctuations.

Figure 5.32 shows first four KL eigenvectors corresponding to $c_l = 0.05$, scaled by the square rood of the corresponding eigenvalue. These plots were generated with

```
./mkplots.py numKLevec 0.05 512 on
./mkplots.py numKLevec 0.05 8192 off
```

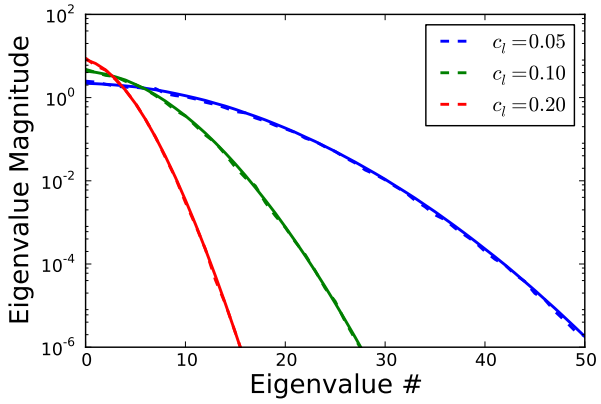


Figure 5.31: KL eigenvalues estimated with two sets of RF realizations: $2^9 = 512$ (dashed lines) and $2^{17} = 131072$ (solid lines).

```
./mkplots.py numKLevec 0.05 131072 off
```

Unlike the eigenvalue spectrum, the eigenvectors are very sensitive to the covariance matrix entries. For $c_l = 0.05$, a large number of RF realizations, e.g. $N_\omega = 2^{17}$ in Fig. 5.32c, are required for computing a covariance matrix with KL modes that are close to the ones based on analytical covariance matrix (analytical modes not shown).

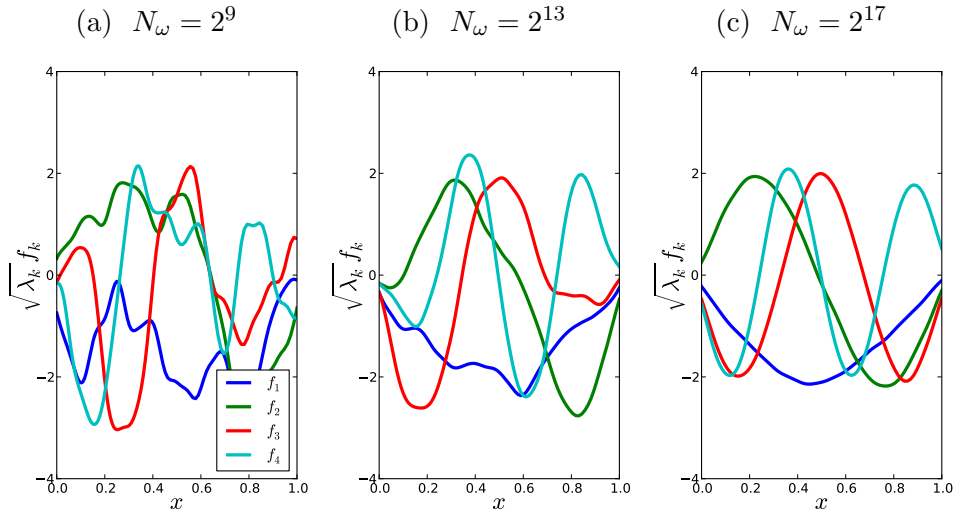


Figure 5.32: Illustration of first 4 KL modes, computed based on a numerical covariance matrices using three sets of RF realizations with $c_l = 0.05$

Figure 5.33 shows first four KL eigenvectors corresponding to $c_l = 0.20$, scaled by the square rood of the corresponding eigenvalue. These plots were generated with

```
./mkplots.py numKLevec 0.20 512 on
./mkplots.py numKLevec 0.20 8192 off
./mkplots.py numKLevec 0.20 131072 off
```

For larger correlation lengths, a smaller number of samples is sufficient to estimate a covariance matrix and subsequently the KL modes. The results based on $N_\omega = 2^{13} = 8192$ RF realizations, in Fig. 5.33b, are close to the ones based on a much larger number of realizations, $N_\omega = 2^{17} = 131072$ in Fig. 5.33c.

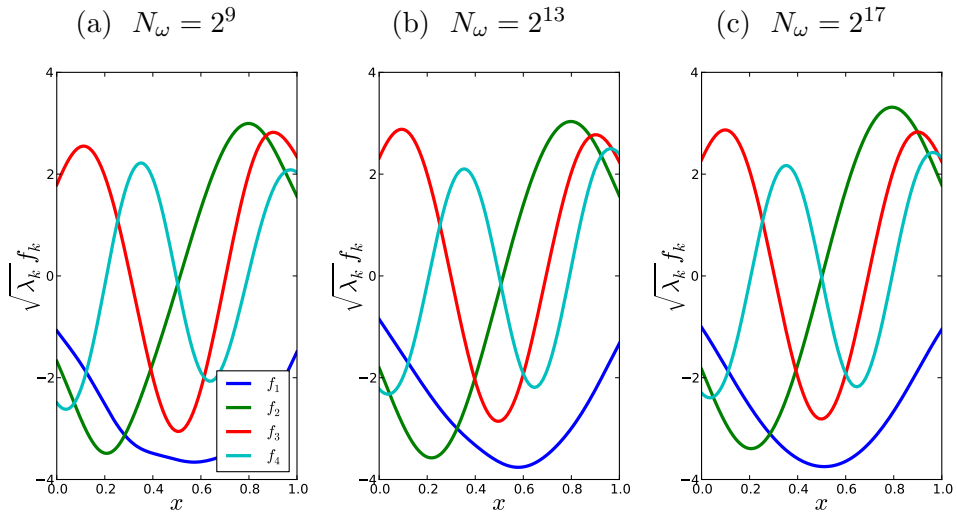


Figure 5.33: Illustration of first 4 KL modes, computed based on a numerical covariance matrices using three sets of RF realizations with $c_l = 0.20$

One can explore the orthogonality of the KLE modes to compute samples of germ ξ_k, introduced in Eq. (5.36). These samples are computed via Eq. (eq:xirealiz) and are saved in files *xidata** in the corresponding run directories. Using the ξ samples, one can estimate their density via Kernel Density Estimate (KDE). Figures 5.34 and 5.35. These figures were generated with

```
./mkplots.py xidata 0.05 512      ./mkplots.py xidata 0.20 512
./mkplots.py xidata 0.05 131072   ./mkplots.py xidata 0.20 131072
```

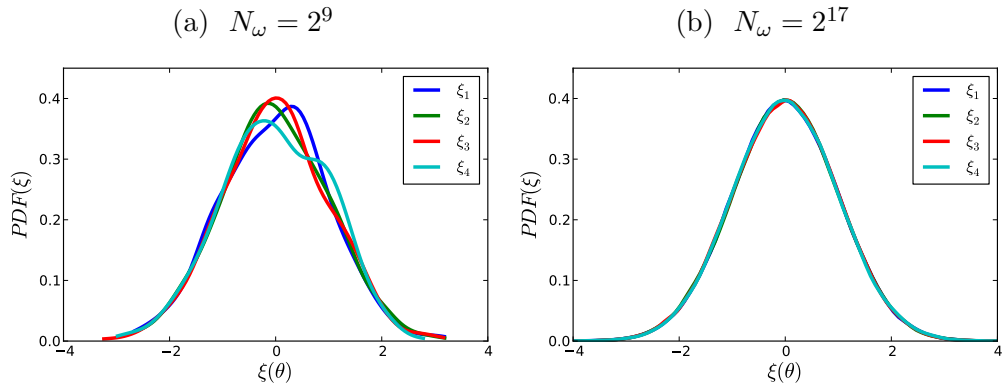


Figure 5.34: Probability densities for ξ_k obtained via KDE using samples obtained by projecting RF realizations onto KL modes. Results correspond to $c_l = 0.05$.

Independent of the correlation length, a relatively large number of samples is required for "converged" estimates for the density of ξ.

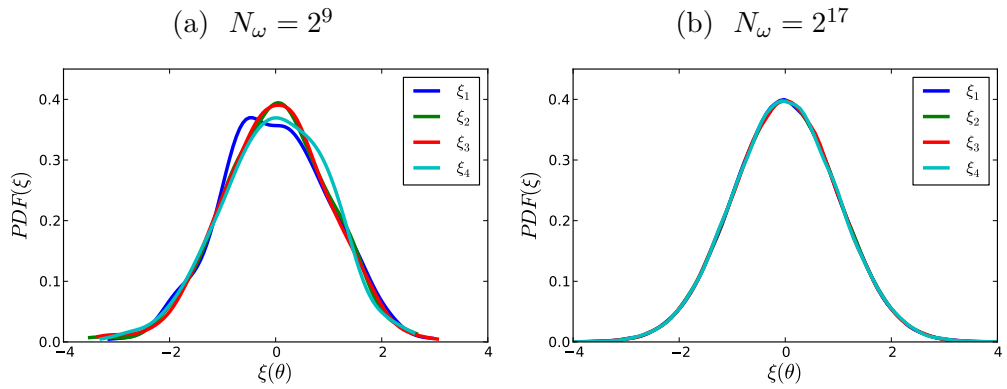


Figure 5.35: Probability densities for ξ_k obtained via KDE using samples obtained by projecting RF realizations onto KL modes. Results correspond to $c_l = 0.20$.

Figures 5.36 and 5.37 show reconstructions of select RF realizations. As observed in the figure showing the decay in the magnitude of the KL eigenvalues, more terms are needed to represents small scale features occuring for smaller correlation lengths, in Fig. 5.36, compared to RF with larger correlation lengths, e.g. the example shown in Fig. 5.37. The plots shown in Figs. 5.36 and 5.37 were generated with

```
./mkplots.py pltKLrecon1D 0.05 21 51 10
./mkplots.py pltKLrecon1D 0.10 63 21 4
```

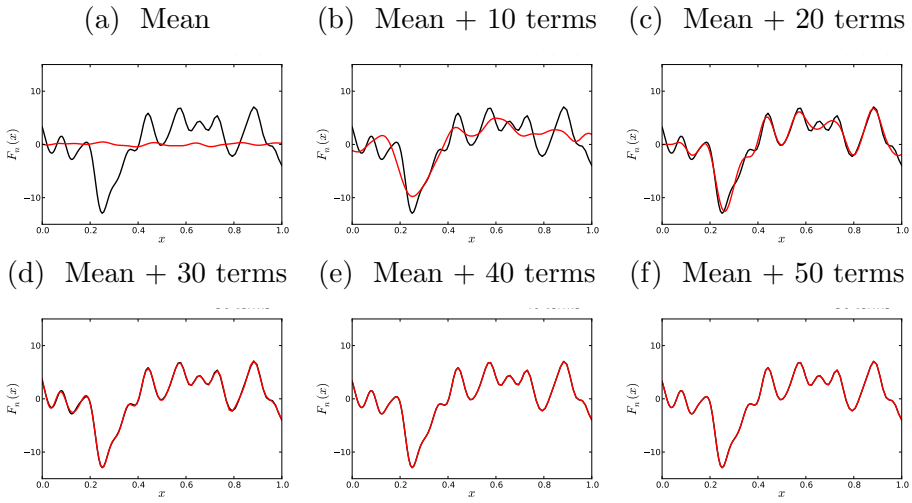


Figure 5.36: Reconstructing realizations with an increasing number of KL expansion terms for $c_l = 0.05$

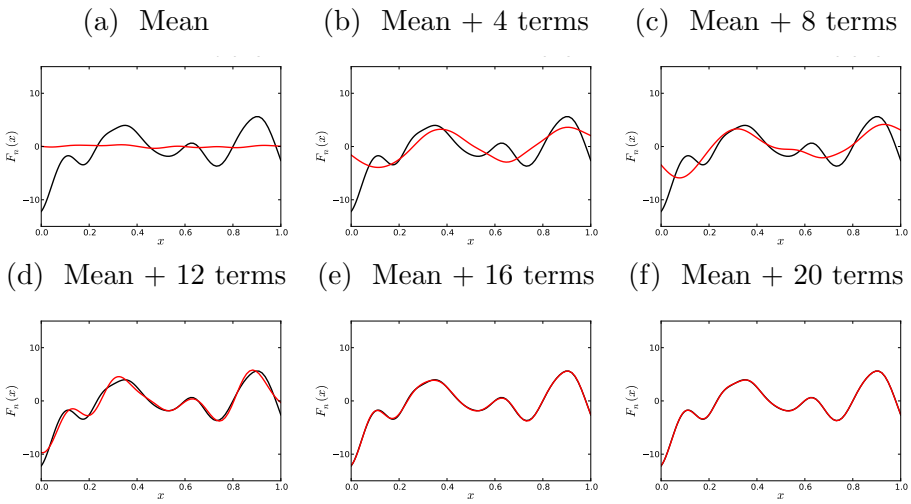


Figure 5.37: Reconstructing realizations with an increasing number of KL expansion terms for $c_l = 0.10$

2D Examples on Structured Grids

In this section we are presenting 2D RFs generated with **kl_2D.x**. The RFs are generated on a non-uniform structured 2D grid $[0, L_x] \times [0, L_y]$, with smaller grid spacing near the boundaries and larger grid spacing towards the center of the domain. This grid is computed using an algebraic expression [15]. The first coordinate is computed via

$$x_i = L_x \frac{(2\alpha+\beta)r_i + 2\alpha - \beta}{(2\alpha+1)(1+r_i)}, \quad r_i = \frac{\beta+1}{\beta-1}^{\frac{\eta_i-\alpha}{1-\alpha}}, \quad \eta_i = \frac{i-1}{N_p-1}, \; i = 1, 2, \ldots, N_x \tag{5.44}$$

The $\beta > 1$ factor in the above expression controls the compression near $x = 0$ and $x = L_x$, while $\alpha \in [0, 1]$ determines where the clustering occurs. The examples shown in this section are based on default values for the parameters that control the grid definition in **kl_2D.x**:

$$\alpha = 1/2, \;\; \beta = 1.1, \;\; L_{x_1} = L_{x_2} = L = 1, \;\; N_{x_1} = N_{x_2} = 65$$

Figure 5.38 shows the 2D computational grid created with these parameters. This figure was generated with the Python script "pl2Dsgrid.py"

```
./pl2Dsgrid.py cvspl2D_0.1_4096
```

Figure 5.39 shows 2D RF realizations with correlation lengths $c_l = 0.1$ and $c_l = 0.2$. As the correlation length increases the realizations become smoother. These figure were generated with

```
./mkplots.py samples2D 0.1 4096 2 (Figs. 5.39a,5.39b)
./mkplots.py samples2D 0.2 4096 2 (Figs. 5.39c,5.39d)
```

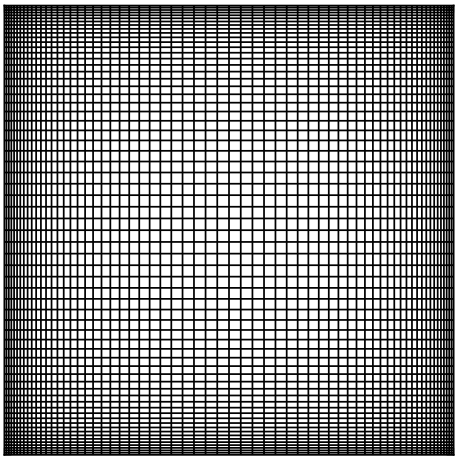


Figure 5.38: Structured grid employed for 2D RF examples.

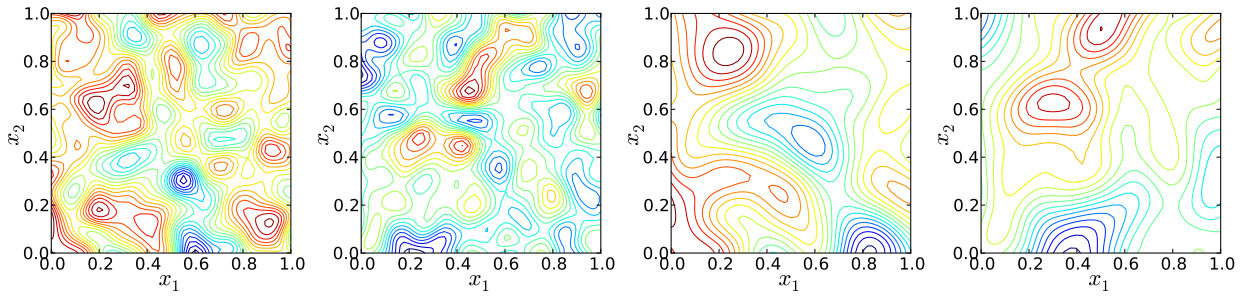


Figure 5.39: Sample 2D random field realizations for $c_l = 0.1$ and $c_l = 0.2$.

In a 2D configuration the rhs of Eq. (eq:fredint) is discretized using a 2D finite volume approach:

$$\int Cov(x,y)f(y)dy \approx \sum_{i=1}^{N_{x_1}-1} \sum_{j=1}^{N_{x_2}-1} \left(Cov(x,y)f(y)\right)|_{ij} A_{ij} \tag{5.45}$$

Here, A_{ij} is the area of rectangle (ij) with lower left corner (i,j) and upper right corner $(i+1, j+1)$, and $\left(Cov(x,y)f(y)\right)|_{ij}$ is the average over rectangle (ij) computed as the arithmetic average of values at its four vertices. Eq. (5.45) can be further cast as

$$\int Cov(x,y)f(y)dy \approx \sum_{i=1}^{N_{x_1}} \sum_{j=1}^{N_{x_2}} \left(Cov(x,y)f(y)\right)_{i,j} w_{i,j}, \tag{5.46}$$

where $w_{i,j}$ is a quarter of the area of all rectangles that surround vertex (i,j).

Figures 5.40 and 5.41 shows first 8 KL modes computed based on covariance matrices that where estimated from $2^{12} = 4096$ and $2^{16} = 65536$ number of RF samples, respectively, and correlation length $c_l = 0.1$ for both sets. The results in Fig. 5.41 are close to the KL modes

corresponding to the analytical covariance matrix (results not shown), while the results in Fig. 5.40 indicate that 2^{12} RF realizations is not sufficient to generate converged KL modes. These figures were generated with

```
./mkplots.py numKLevec2D 0.1 4096  (Fig. 5.40)
./mkplots.py numKLevec2D 0.1 65536 (Fig. 5.41)
```

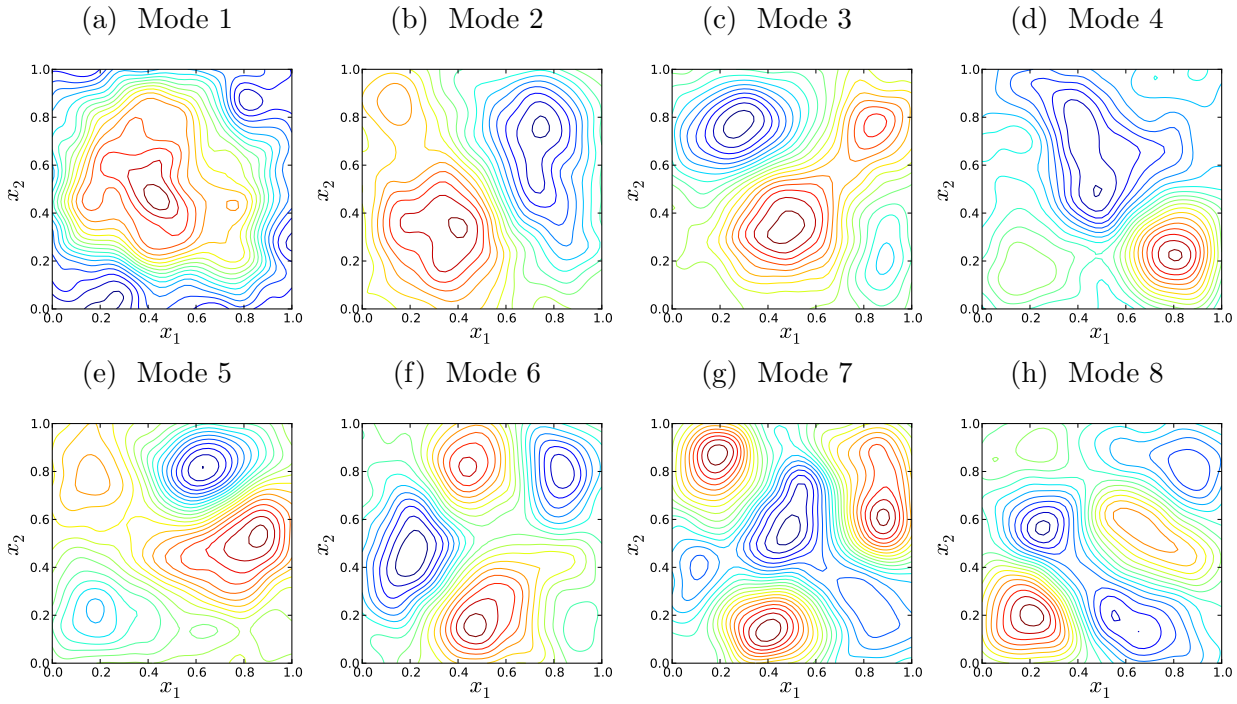


Figure 5.40: Illustration of first 8 KL modes, computed based on a numerical covariance matrix estimated using 2^{12} 2D RF realizations with $c_l = 0.1$

Figure 5.42 shows first 8 KL modes computed based on a covariance matrix that was estimated from $2^{12} = 4096$ number of RF samples. For these results, with correlation length $c_l = 0.5$, 2^{12} samples are sufficient to estimate the covariance matrix and subsequently KL modes that are close to analytical results (results not shown). The plots in Fig. 5.42 were generated with

```
./mkplots.py numKLevec2D 0.5 4096
```

Figures 5.43 and 5.44 show reconstructions of select 2D RF realizations. As observed in the previous section for 1D RFs, more terms are needed to represents small scale features occuring for smaller correlation lengths, in Fig. 5.43, compared to RF with larger correlation lengths, e.g. the example shown in Fig. 5.44. The plots shown in Figs. 5.43 and 5.44 were generated with

```
./mkplots.py pltKLrecon2D 0.2 3  85 12 (Fig. 5.43)
./mkplots.py pltKLrecon2D 0.5 37 36 5  (Fig. 5.44)
```

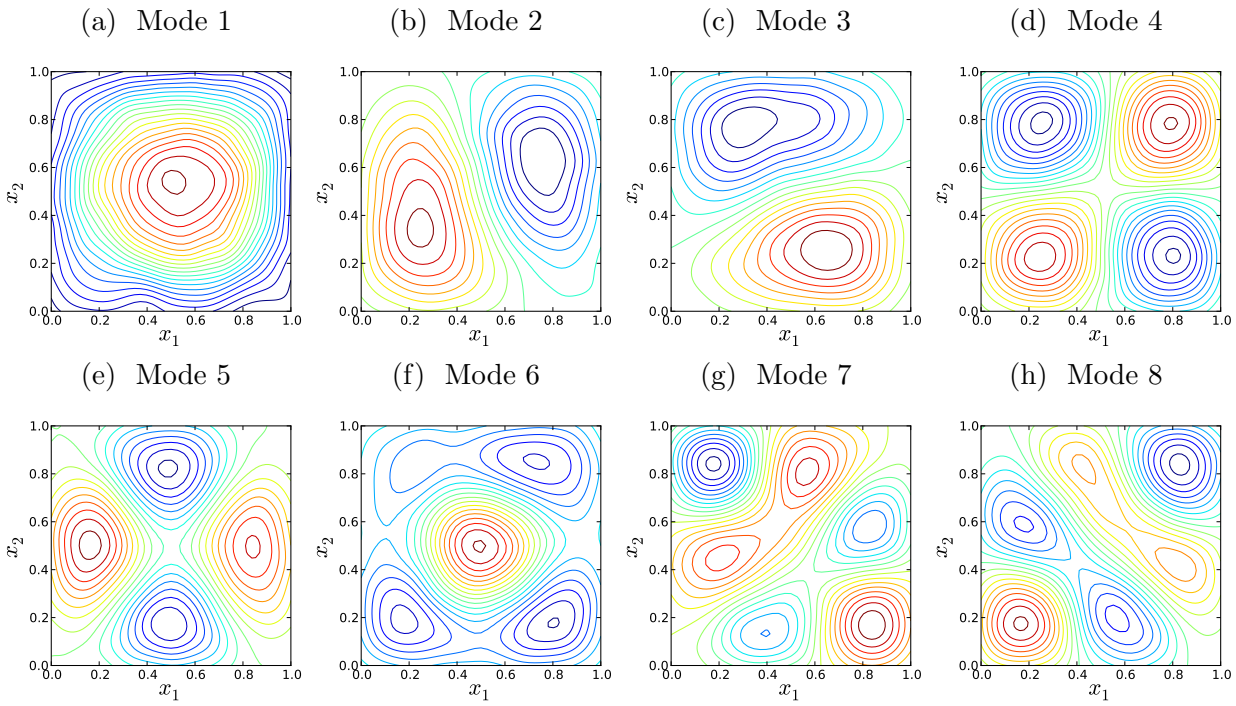


Figure 5.41: Illustration of first 8 KL modes, computed based on a numerical covariance matrix estimated using 2^{16} 2D RF realizations with $c_l = 0.1$

2D Examples on Unstructured Grids

For this example we choose a computational domain that resembles the shape of California. A number of $2^{12} = 4096$ points were randomly distributed inside this computational domain, and a triangular grid with 8063 triangles was generated via Delaunay triangulation. The 2D grid point locations are provided in "data/cali_grid.dat" and the grid point connectivities are provided in "data/cali_tria.dat". Figure 5.45 shows the placement of these grid points, including an inset plot with the triangular grid connectivities. This figure shows the grids on a uniform scale in terms of latitude and longitude degrees and was generated with

```
./pl2Dugrid.py
```

Figure 5.46 shows 2D RF realizations with correlation lengths $c_l = 0.5°$ and $c_l = 2°$. These figure were generated with

```
./mkplots.py samples2Du 0.5 4096 2 (Figs. 5.46a,5.46b)
./mkplots.py samples2Du 2.0 4096 2 (Figs. 5.46c,5.46d)
```

Figure 5.47 shows first 16 KL modes computed based on a covariance matrix that was estimated from $2^{16} = 65536$ number of RF samples, with correlation length $c_l = 0.5°$. The KL modes corresponding to an analytically estimated covariance matrix with the same correlation length are shown in Fig. 5.48. For this example, it seems that 2^{16} samples are sufficient to estimate the first 12 to 13 modes accurately. Please note that some of the

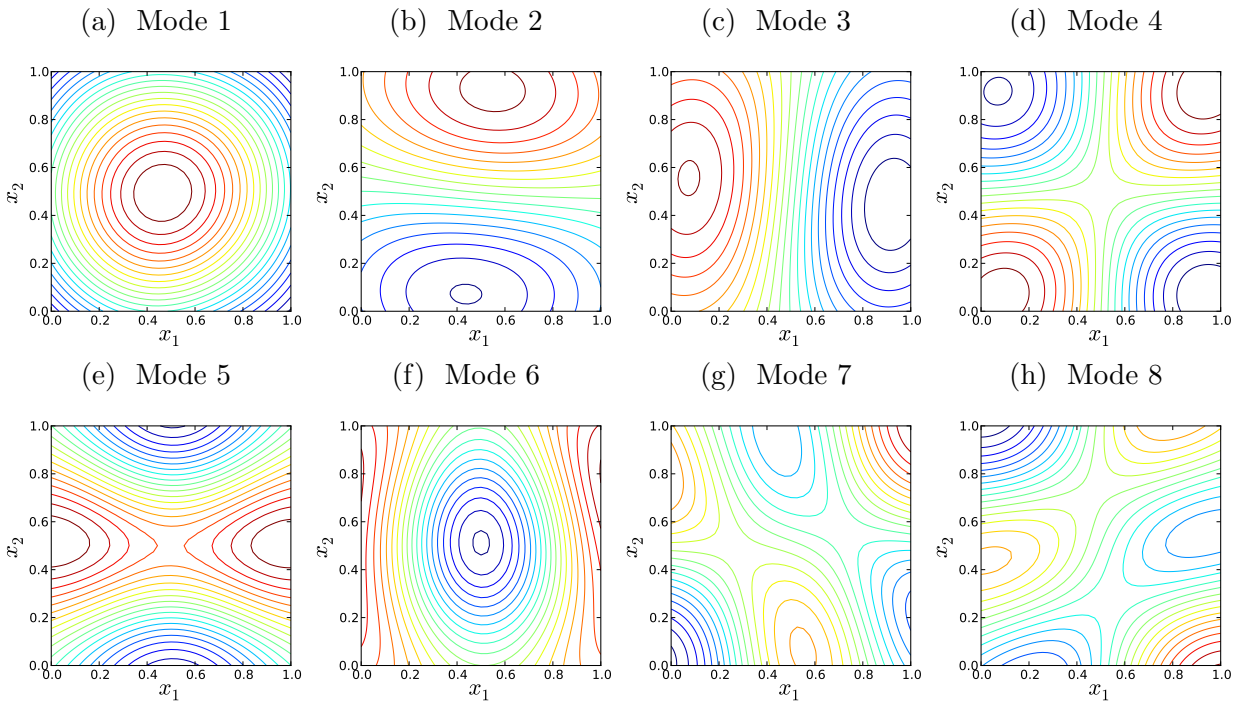


Figure 5.42: Illustration of first 8 KL modes, computed based on a numerical covariance matrix estimated using 2^{12} 2D RF realizations with $c_l = 0.5$

modes can differ up to a multiplicative factor of -1, hence the colorscheme will be reversed. Higher order modes start diverging from analytical estimates, e.g. modes 14 through 16 in this example. Figure 5.49 shows KL modes corresponding to a covariance matrix estimated from RF realizations with $c_l = 2°$. For this correlation length, 2^{16} samples are sufficient to generate KL modes that are very close to analytical results (not shown). These figures were generated with

`./mkplots.py numKLevec2Du 0.5 65536` (Fig. 5.47)
`./mkplots.py anlKLevec2Du SqExp 0.5` (Fig. 5.48)
`./mkplots.py numKLevec2Du 2.0 65536` (Fig. 5.49)

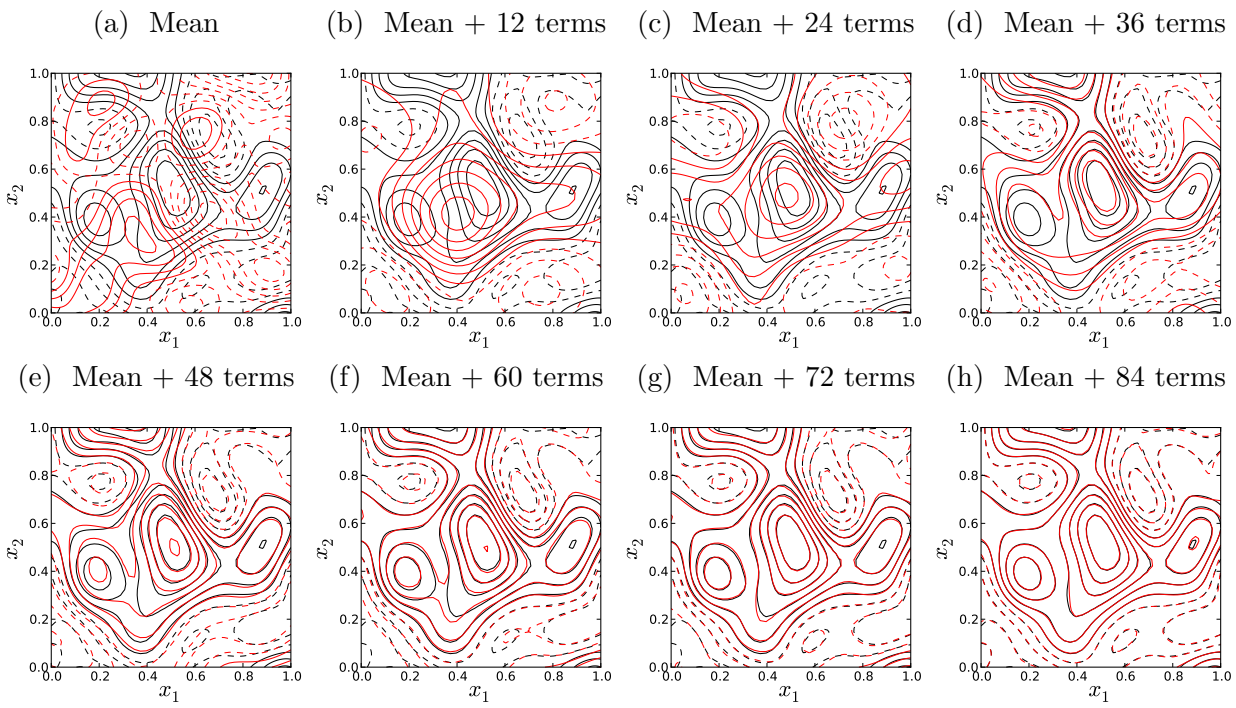


Figure 5.43: Reconstructing 2D realizations with an increasing number of KL expansion terms for $c_l = 0.2$

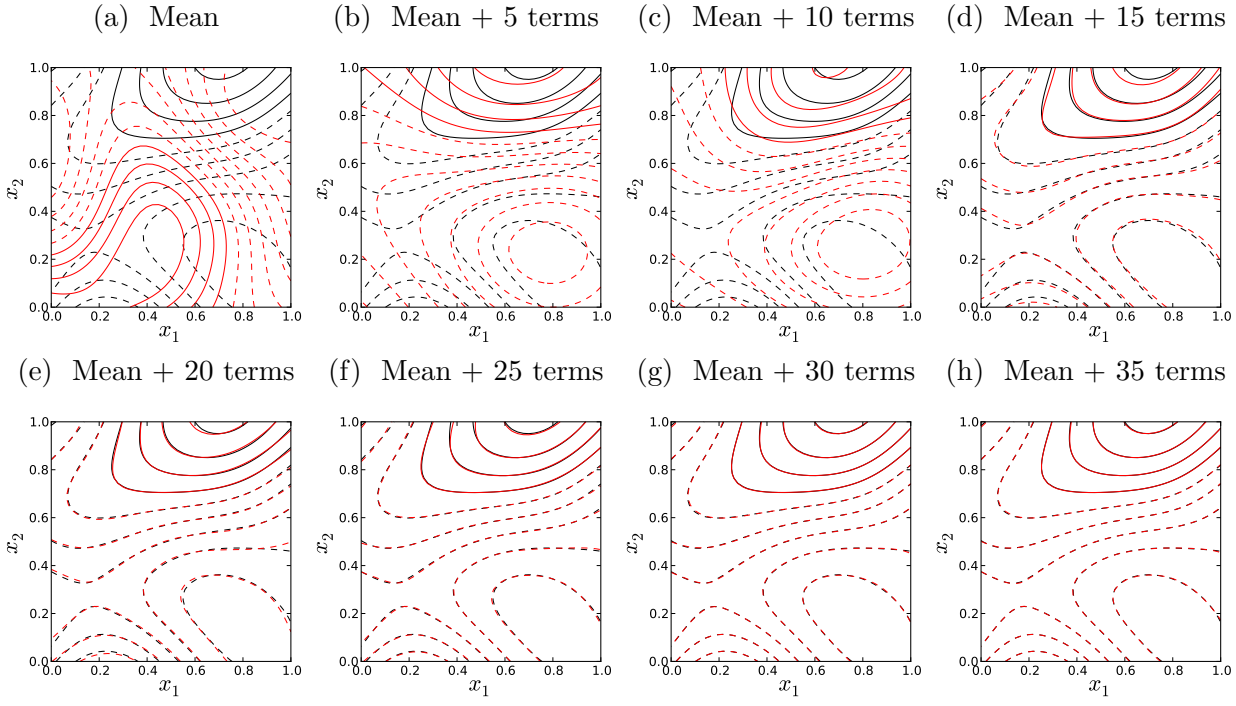


Figure 5.44: Reconstructing 2D realizations with an increasing number of KL expansion terms for $c_l = 0.5$

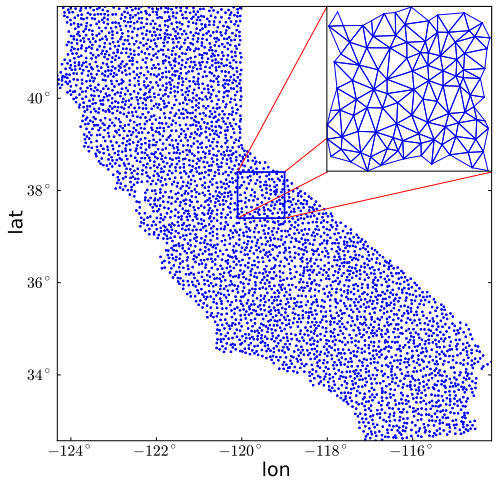


Figure 5.45: Unstructured grid generated via Delaunay traingulation overlaid over California.

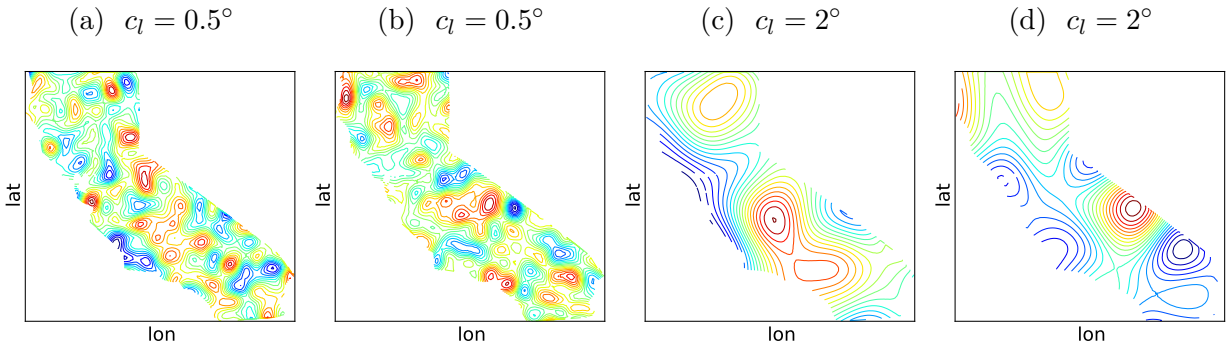


Figure 5.46: Sample 2D random field realizations on an unstructured grid for $c_l = 0.5°$ and $c_l = 2°$.

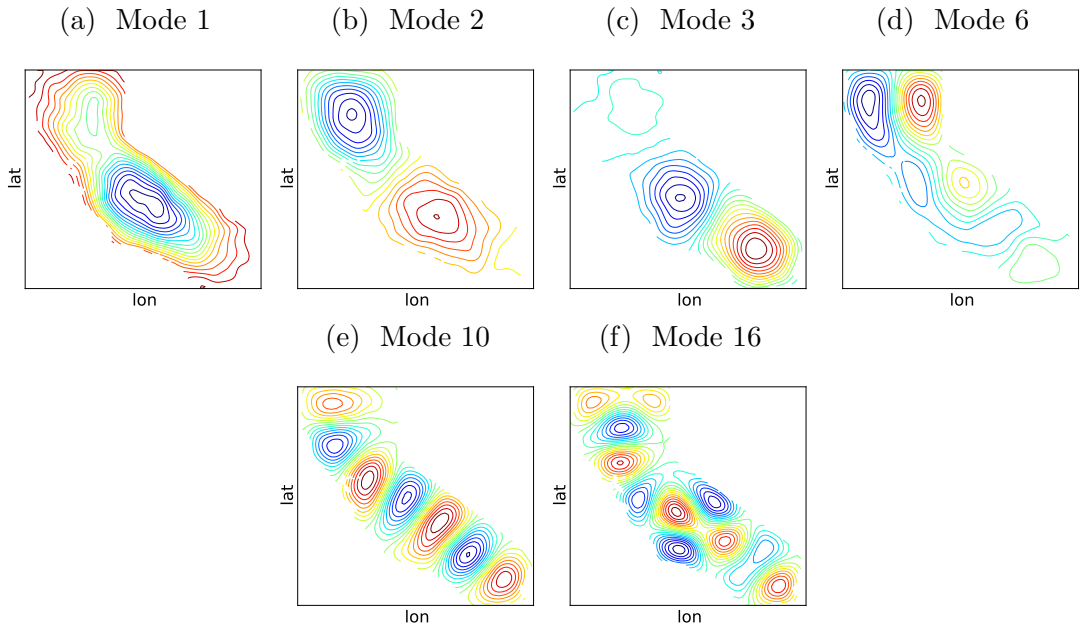


Figure 5.47: Illustration of seclect KL modes, computed based on a numerical covariance matrix estimated using 2^{16} 2D RF realizations on an unstructured grid with $c_l = 0.5°$.

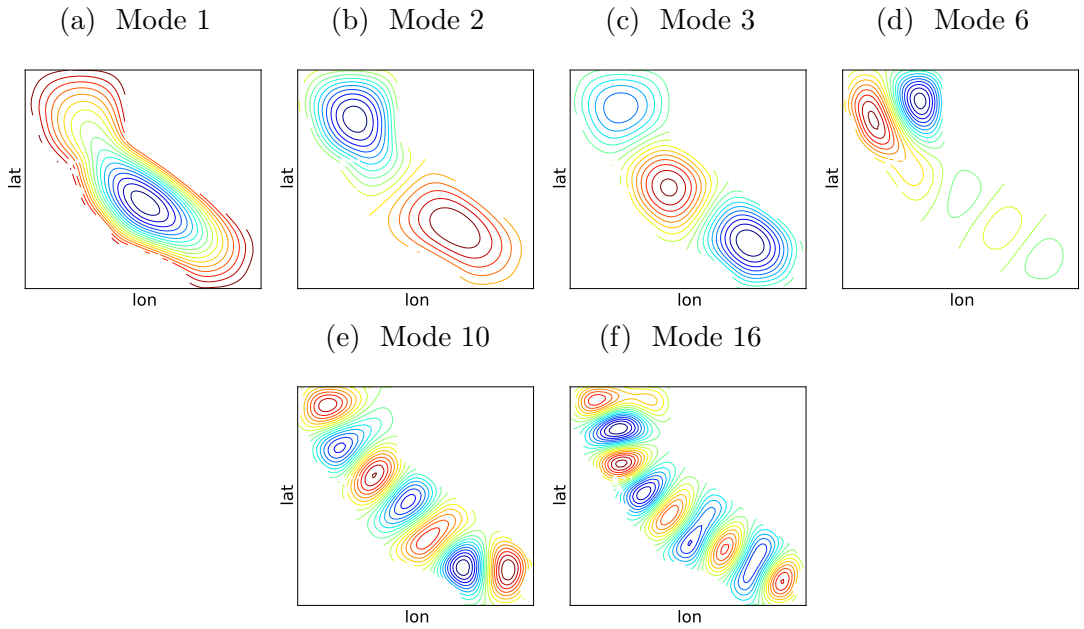


Figure 5.48: Illustration of select KL modes, computed based on an analytical covariance matrix for 2D RF realizations on an unstructured grid with $c_l = 0.5°$ and a square-exponential form.

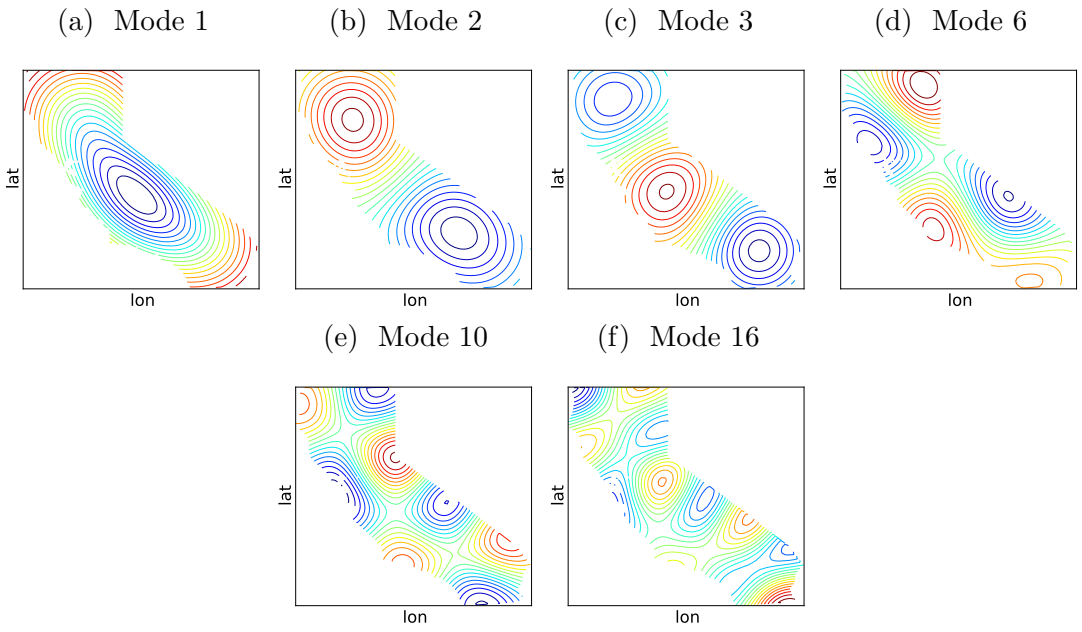


Figure 5.49: Illustration of select KL modes, computed based on a numerical covariance matrix estimated using 2^{16} 2D RF realizations on an unstructured grid with $c_l = 2°$.

Chapter 6

Support

UQTk is the subject of continual development and improvement. If you have questions about or suggestions for UQTk, feel free to e-mail the UQTk Users list, at `uqtk-users@software.sandia.gov`. You can sign up for this mailing list at `https://software.sandia.gov/mailman/listinfo/uqtk-users`.

References

[1] S. Babacan, R. Molina, and A. Katsaggelos. Bayesian compressive sensing using Laplace priors. *IEEE Transactions on Image Processing*, 19(1):53–63, 2010.

[2] J. Ching and Y.-C. Chen. Transitional markov chain monte carlo method for bayesian model updating, model class selection, and model averaging. *Journal of Engineering Mechanics*, 133(7):816–832, 2007.

[3] C. W. Clenshaw and A. R. Curtis. A method for numerical integration on an automatic computer. *Numerische Mathematik*, 2:197–205, 1960.

[4] B.J. Debusschere, H.N. Najm, P.P. Pébay, O.M. Knio, R.G. Ghanem, and O.P. Le Maître. Numerical challenges in the use of polynomial chaos representations for stochastic processes. *SIAM Journal on Scientific Computing*, 26(2):698–719, 2004.

[5] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall CRC, 2 edition, 2003.

[6] Stuart Geman and D. Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-6(6):721–741, 1984.

[7] Thomas Gerstner and Michael Griebel. Numerical integration using sparse grids. *Numerical Algorithms*, 18(3-4):209–232, 1998. (also as SFB 256 preprint 553, Univ. Bonn, 1998).

[8] R.G. Ghanem and P.D. Spanos. *Stochastic Finite Elements: A Spectral Approach*. Springer Verlag, New York, 1991.

[9] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall, London, 1996.

[10] G. H. Golub and J. H. Welsch. Calculation of Gauss quadrature rules. *Math. Comp.*, 23:221–230, 1969.

[11] H. Haario, E. Saksman, and J. Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, 7:223–242, 2001.

[12] Heikki Haario, Marko Laine, Antonietta Mira, and Eero Saksman. Dram: Efficient adaptive mcmc. *Statistics and Computing*, 16(4):339–354, 2006.

[13] R.G. Haylock and A. O'Hagan. On inference for outputs of computationally expensive algorithms with uncertainty on the inputs. *Bayesian statistics*, 5:629–637, 1996.

[14] F.B. Hildebrand. *Introduction to Numerical Analysis*. Dover, 1987.

[15] K.A Hoffmann and S.T. Chiang. *Computational Fluid Dynamics*, volume 1, chapter 9, pages 358–426. EES, 2000.

[16] M. C. Kennedy and A. O'Hagan. Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B*, 63(3):425–464, 2001.

[17] S. Kucherenko, S. Tarantola, and P. Annoni. Estimation of global sensitivity indices for models with dependent variables. *Computer Physics Communications*, 183:937–946, 2012.

[18] O.P. Le Maître and O.M. Knio. *Spectral Methods for Uncertainty Quantification: With Applications to Computational Fluid Dynamics (Scientific Computation)*. Springer, 1st edition. edition, April 2010.

[19] Y. M. Marzouk and H. N. Najm. Dimensionality reduction and polynomial chaos acceleration of Bayesian inference in inverse problems. *Journal of Computational Physics*, 228(6):1862–1902, 2009.

[20] E.J. Nyström. Über die praktische auflösung von integralgleichungen mit anwendungen auf randwertaufgaben. *Acta Mathematica*, 54(1):185–204, 1930.

[21] J. Oakley and A. O'Hagan. Bayesian inference for the uncertainty distribution of computer model outputs. *Biometrika*, 89(4):769–784, 2002.

[22] Mark Orr. Introduction to radial basis function networks. *Technical Report, Center for Cognitive Science, University of Edinburgh*, 1996.

[23] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[24] M. Rosenblatt. Remarks on a multivariate transformation. *Annals of Mathematical Statistics*, 23(3):470 – 472, 1952.

[25] A. Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145:280–297, 2002.

[26] K. Sargsyan. Surrogate models for uncertainty propagation and sensitivity analysis. In R. Ghanem, D. Higdon, and H. Owhadi, editors, *Handbook of Uncertainty Quantification*. Springer, 2017.

[27] K. Sargsyan, H.N. Najm, and R. Ghanem. On the Statistical Calibration of Physical Models. *International Journal of Chemical Kinetics*, 47(4):246–276, 2015.

[28] K. Sargsyan, C. Safta, H. Najm, B. Debusschere, D. Ricciuto, and P. Thornton. Dimensionality reduction for complex models via Bayesian compressive sensing. *International Journal of Uncertainty Quantification*, 4(1):63–93, 2014.

[29] D.W. Scott. *Multivariate Density Estimation. Theory, Practice and Visualization*. Wiley, New York, 1992.

[30] B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986.

[31] D.S. Sivia. *Data Analysis: A Bayesian Tutorial*. Oxford Science, 1996.

[32] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics Dokl.*, 4:240–243, 1963.

[33] I. M. Sobol. Sensitivity estimates for nonlinear mathematical models. *Math. Modeling and Comput. Exper.*, 1:407–414, 1993.

[34] K. M. Zuev and J. L. Beck. Asymptotically independent markov sampling: A new mcmc scheme for Bayesian inference. In *Vulnerability, Uncertainty, and Risk : Quantification, Mitigation, and Management - CDRM 9*, pages 2022–2031. 2014.

DISTRIBUTION:

1 MS 0899 Technical Library, 8944 (electronic copy)


Sandia National Laboratories