SANDIA REPORT
SAND2023-12685

Printed February, 2023

Sandia National Laboratories

UQTk Version 3.1.3 User Manual

Khachik Sargsyan, Cosmin Safta, Caitlin Curry, Luke Boll,
Katherine Johnston, Mohammad Khalil, Kenny Chowdhary, Prashant Rai,
Pieterjan Robbe, Tiernan Casey, Xiaoshu Zeng, Bert Debusschere

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

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

Available to DOE and DOE contractors from

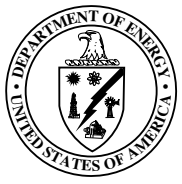
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

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

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



ABSTRACT

The UQ Toolkit (UQTk) is a collection of libraries and tools for the quantification of uncertainty in numerical model predictions. Version 3.1.3 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.

ACKNOWLEDGMENT

This work was supported in large part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through the Advanced Computing (SciDAC) program via the FASTMath Institute.

UQTk has been, and continues to be, the product of collaboration between many people. The key authors of UQTk are (alphabetical by first name):

- Bert Debusschere
- Caitlin Curry
- Cosmin Safta
- Katherine Johnston
- Kenny Chowdhary
- Khachik Sargsyan
- Luke Boll
- Mohammad Khalil
- Pieterjan Robbe
- Prashant Rai
- Tiernan Casey
- Xiaoshu Zeng

Beyond the authors listed above, there is a long and continually growing list of coworkers, students and visitors who have contributed to UQTk over the years. This list includes, but is not limited to (alphabetical by first name):

- Habib Najm
- Helgi Adalsteinsson
- Majid Latif
- Olivier Le Maître
- Omar Knio
- Roger Ghanem

- Sarah Castorena
- Sarah de Bord
- Xun Huan

Further, we are grateful to all the users of UQTk who through their questions and suggestions are continually helping us to improve the software.

CONTENTS

REVISION HISTORY

This manual goes with UQTk version 3.1.3. Previous releases and release dates are listed below along with the report numbers of the corresponding manuals.

- UQTk 3.1.3: 02/20/23, SAND2023-12685
- UQTk 3.1.2: 01/13/22, SAND2022-0377
- UQTk 3.1.1: 03/25/21, SAND2021-3655
- UQTk 3.1.0: 02/28/21, SAND2020-2879
- UQTk 3.0.4: 10/09/17, SAND2017-11051
- UQTk 3.0.3: 05/30/17, SAND2017-5747
- UQTk 3.0.0: 09/16/16, SAND2016-9215
- UQTk 2.1.0: 05/30/14, SAND2014-4968
- UQTk 2.0.0: 10/22/13, SAND2013-9165

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 3.1.3 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, command line apps are provided that allow access to the UQTk functionality from the command line. A comprehensive Python interface is also provided.

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.

2. DOWNLOAD AND INSTALLATION

2.1. 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 has not been tested extensively.

Many of the examples rely on Python, including NumPy, SciPy, and matplotlib packages for postprocessing and graphing. The UQTk Python utilities are compatible with both Python 2.7.x and 3.7.x. However, Python version version 3.7.x with compatible NumPy, SciPy, and matplotlib is 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.

2.2. DOWNLOAD

The most recent version of UQTk, currently 3.1.3, can be cloned from github at:
`https://github.com/sandialabs/UQTk`

2.3. DIRECTORY STRUCTURE

After cloning the git repo, you will find the following directories in the repo:

`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)
`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**
`bare_bcs`	Sparse signal reconstruction for arbitrary basis
`d_spring_series`	Springs in series to demonstrate dimensionality reduction through basis adaptation
`dfi`	Example of Data Free Inference (DFI)
`dfi_app`	Example of Data Free Inference (DFI) with command line app
`heat_transfer_window`	Forward propagation with a heat transfer example
`iuq`	surrogate-enabled inverse UQ workflow
`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
`surrogate_genz`	Jupyter notebooks for various surrogate methods through PyUQTk
`tmcmc_bimodal`	use TMCMC to sample from a 3-dimensional posterior that is aproduct of a Gaussian prior and a bimodal likelihood
`uqpc`	construct Polynomial Chaos surrogates for multiple outputs/functions
`PyUQTk`	**Python scripts and interface to C++ libraries**
`bcs`	interface to Bayesian compressive sensing library
`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`	Python unit tests
`quad`	interface to Quad class
`sens`	Python global sensitivity analysis scripts
`tmcmc`	Interface to tMCMC class
`tools`	interface to UQTk tools
`uqtkarray`	interface to array class
`utils`	interface to UQTk utils

2.4. EXTERNAL SOFTWARE AND LIBRARIES

2.4.1. Required

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. Most of our development happens on either Mac OS X or Linux with the GNU Compiler Suite. For OS X these compilers were installed either using MacPorts, or Homebrew, or directly built from source code. We have also successfully compiled with Intel compilers on Linux.

2. **CMake**. We switched to a CMake-based build/install configuration in version 3.0. The configuration files require a CMake version 3.1 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`

4. **LAPACK and BLAS**. UQTk will use system installed versions of LAPACK and BLAS if possible. If not found, UQTk will use a self contained version.

5. **SUNDIALS**. UQTk requires SUNDIALS version 6.0.0 or higher (older versions will not work.) If SUNDIALS is not yet installed on your system, the UQTk build process will automatically download it from `https://github.com/LLNL/sundials`, configure it, and build it. To use a version of SUNDIALS that is already installed, specify the path to it as indicated in the installation section below.

2.4.2. Optional

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 2.7.x, 3.7.x, and 3.9.x Note that it is important that the Python, NumPy, SciPy, and Matplotlib 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. **Pybind11**. PyUQTk has been tested with Pybind 2.6.2. Instructions for the installation of Pybind11 can be found at: Pybind Installation Instructions. It can be installed via pip, homebrew, or macports.

2.5. 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 2.3.
- *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

The following set of commands, on a high level, generates the build structure, compiles, tests, and installs UQTk:

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

The next sections explain some of the finer details in this process.

2.5.1. 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
- **CMAKE_SUNDIALS_DIR** : Path to install directory for SUNDIALS
- **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
- **PYTHON_EXECUTABLE** : Path to the Python program
- **PYTHON_LIBRARY** : Path to the Python library
- **pybind11_DIR:FILEPATH** : Path to the directory for Pybind11

Several pre-set config files are available in the "*sourcedir*/config" directory. These scripts set the configuration flags mentioned above for some common situations and can be used as a template for your platform. Some of these shell scripts also accept arguments, e.g. `config-options.sh`, to switch between several configurations. Type, for example `config-options.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 running a particular shell script from the command line, *e.g.*

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

In this example, the configuration script is run from the build directory, while it is assumed that the configuration script still sits in the configuration directory 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-options.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.

2.5.2. 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 Pybind 2.6.2.

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 Python interface just set the CMake

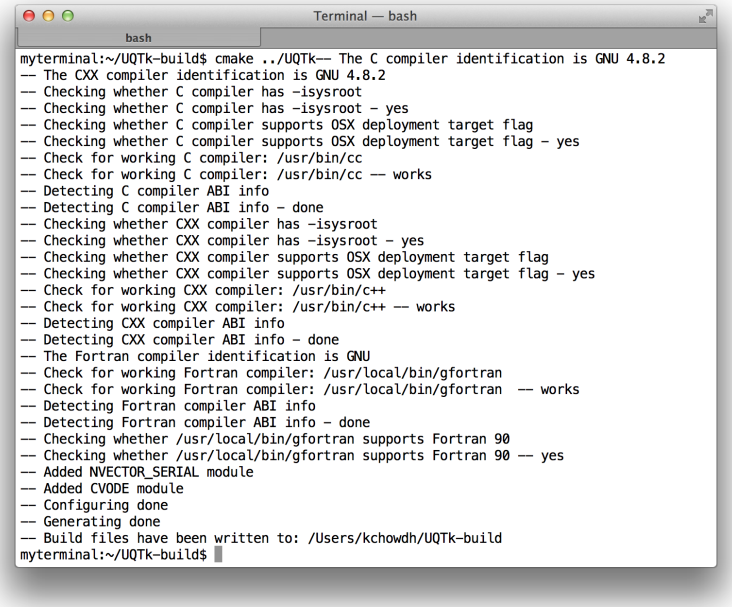


Figure 2-1. CMake configuration without the Python interface.

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
```

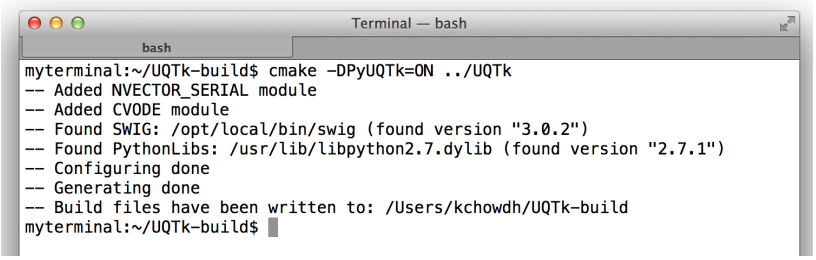


Figure 2-2. CMake configuration with the Python interface.

Figure 2-2 shows the additional output to screen after the Python interface flag is turned on.

If the CMake command has run 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.

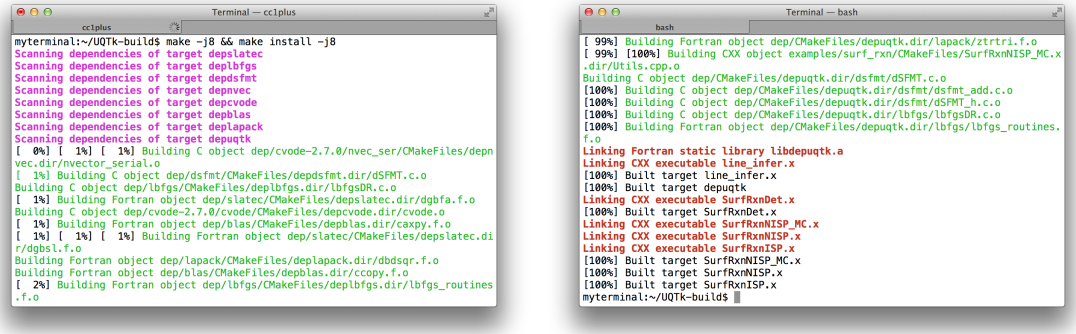


Figure 2-3. Start and end of build without Python interface.

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

```
$ ctest
```

The output should look similar to Figure 2-4.

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

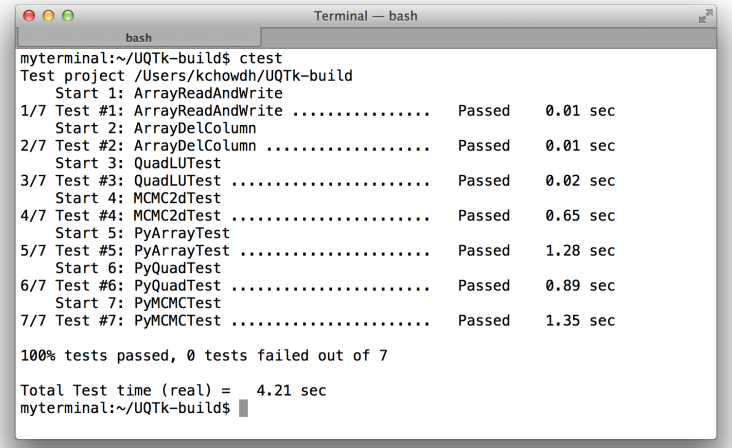


Figure 2-4. Result of ctest after successful build and install. Note that if you do not build PyUQTk, those tests will not be run.

```
$ 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 Pybind 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.

2.5.3. Setting up External Libraries

2.5.3.1. Python

Cmake will very often find the correct python path. However, sometimes cmake cannot identify the correct path and may fail to build or fail the python tests. In this case, you can specify the python library filepath as in the example below.

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

```
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
      -DPYTHON_EXECUTABLE:FILEPATH=/opt/local/bin/python \
      -DPYTHON_LIBRARY:FILEPATH=/opt/local/Library/Frameworks/Python.framework/Versions/3.7/lib/libpython3.7.dylib \
      -DPyUQTk=ON \
      ../UQTk
```

2.5.3.2. SUNDIALS

If you would like to use a version of SUNDIALS that you have already installed on your system (rather than have UQTk download the latest version from github), use the variable `CMAKE_SUNDIALS_DIR` to specify the path to its install folder. For example, your config script may look as follows:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=$PWD/../install \
      -DCMAKE_SUNDIALS_DIR=/Users/myusername/Packages/SUNDIALS/install \
      -DCMAKE_Fortran_COMPILER=gfortran \
      -DCMAKE_C_COMPILER=gcc            \
      -DCMAKE_CXX_COMPILER=g++          \
      -DPyUQTk=ON \
      ../UQTk
```

Note, if your UQTk configuration links to the dynamically linked version of the SUNDIALS library, you will also need to add the location of those libraries to your dynamic library path on your platform (e.g. the `DYLD_LIBRARY_PATH` environment variable on Mac OS X).

2.5.3.3. Pybind11

Pybindıı is a requirement for the usage of the PyUQTk modules. Cmake will often find the correct Pybind path. However, sometimes cmake cannot identify the correct path and may fail to build or fail the python tests. In this case, you can specify the Pybindıı library filepath as in the example below. Additionally, if errors such as, **fatal error:**pybindıı/pybindıı.h: No such file or directory, occur, ensure that Pybindıı is installed in the same directory as the python library being used by CMake. For example, your config script may look as follows:

```
PYTHON_DIR=/opt/local/Library/Frameworks/Python.framework/Versions/3.9/lib
cmake -DCMAKE_INSTALL_PREFIX:PATH=$PWD/../UQTk-install \
      -DCMAKE_Fortran_COMPILER=/opt/local/gcc61/bin/gfortran-6.1.0 \
      -DCMAKE_C_COMPILER=/opt/local/gcc61/bin/gcc-6.1.0 \
      -DCMAKE_CXX_COMPILER=/opt/local/gcc61/bin/g++-6.1.0 \
      -DPYTHON_EXECUTABLE:FILEPATH=/opt/local/bin/python \
      -DPYTHON_LIBRARY:FILEPATH=$PYTHON_DIR/libpython3.9.dylib \
      -Dpybind11_DIR:FILEPATH=$PYTHON_DIR/python3.9/site-packages/pybind11/share/cmake/pybind11 \
      -DPyUQTk=ON \
      ../UQTk
```

Note, some users have found that by installing Pybindıı using pip has made the need for specifying the Pybindıı path unnecessary. Additionally, after a successful build warnings about weak symbols will appear. These can be ignored as they do not prevent the correct implementation of the PyUQTk modules.

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 [11, 34], Karhunen-Loève (KL) expansions [18], numerical quadrature (including sparse quadrature) [26, 8, 20, 54, 16], Bayesian inference [53, 14, 35], Markov Chain Monte Carlo [15, 19, 22, 23], Bayesian compressive sensing [1], and the Rosenblatt transformation [42].

Below, some key aspects and conventions of UQTk Polynomial Chaos expansions, surrogate construction, compressed sensing, and weighted iterative Bayesian Compressed Sensing (wiBCS) are outlined in order to connect the tools in UQTk to the broader theory.

3.1. POLYNOMIAL CHAOS EXPANSIONS

Polynomial chaos expansions (PCEs) can be defined as follows:

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

- X: Random variable represented with multi-D PCE
- c_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 = $\frac{(n+p)!}{n!p!}$

The following are conventions of UQTk PCEs:

- 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

3.2. POLYNOMIAL CHAOS SURROGATE

Consider a *forward* function $Q = f(\boldsymbol{\lambda})$ that maps the input parameter vector $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_d)$ to an output quantity of interest (QoI). Further, assume each input parameter is defined in a range $\lambda_i \in [a_i, b_i]$, for $i = 1, 2, \ldots, d$. By casting each input as a uniform random variable in its range, we write

$$\lambda_i = \frac{a_i + b_i}{2} + \frac{b_i - a_i}{2}\, \xi_i, \tag{3.1}$$

where $\vec{\xi} \in [-1, 1]^d$ is a vector of d independent, identically distributed (*i.i.d.*) uniform random variables. We look for a polynomial expansion for the output Q

$$Q = f(\boldsymbol{\lambda}(\vec{\xi})) \approx \sum_{\boldsymbol{\alpha} \in \mathcal{I}} c_{\boldsymbol{\alpha}} \Psi_{\boldsymbol{\alpha}}(\vec{\xi}) \tag{3.2}$$

with respect to a set of normalized Legendre polynomials $\Psi_{\boldsymbol{\alpha}}(\vec{\xi})$. Each multivariate Legendre polynomial $\Psi_{\boldsymbol{\alpha}}(\vec{\xi})$ corresponds to a multtindex vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$ that defines the polynomial degrees per univariate Legendre polynomial $\psi_k(\xi)$ as $\Psi_{\boldsymbol{\alpha}}(\vec{\xi}) = \psi_{\alpha_1}(\xi_1)\psi_{\alpha_2}(\xi_2)\cdots\psi_{\alpha_d}(\xi_d)$. Also, by convention, the sum of all degrees $||\boldsymbol{\alpha}||_1 = \alpha_1 + \alpha_2 + \cdots + \alpha_d$ is called the *order* of the multivariate polynomial $\Psi_{\boldsymbol{\alpha}}(\vec{\xi})$. Legendre polynomials are orthogonal with respect to the uniform PDF $\pi_{\vec{\xi}}(\vec{\xi}) = 2^{-d}$

$$\langle \Psi_{\boldsymbol{\alpha}}(\vec{\xi})\Psi_{\boldsymbol{\alpha}'}(\vec{\xi})\rangle \equiv \int_{\vec{\xi}} \Psi_{\boldsymbol{\alpha}}(\vec{\xi})\Psi_{\boldsymbol{\alpha}'}(\vec{\xi}) 2^{-d} d\vec{\xi} = 0 \quad \text{if} \quad \boldsymbol{\alpha} \neq \boldsymbol{\alpha}', \tag{3.3}$$

and are normalized such that $||\Psi_{\boldsymbol{\alpha}}||^2 = \int_{\vec{\xi}} \Psi^2_{\boldsymbol{\alpha}}(\vec{\xi}) 2^{-d} d\vec{\xi} = 1$.

The selection of the multiindex set $\mathcal{I}$ and its size $K = |\mathcal{I}|$ is a key modeling step and is typically dictated by the general expected behavior of the forward function $f(\boldsymbol{\lambda})$. Some conventional options for non-adaptive, *i.e.* a priori, *basis selection* are listed in Table 3-1.

	Muliindex set $\mathcal{I}$	Parameters
Total order	$\sum_i \alpha_i \leq p$	Total order p
Tensor product	$\alpha_i \leq p_i$ for all i	Order per dimension p_i
L_q	$\sum_i \alpha_i^q \leq p^q$	Effective order p
Hyperbolic Cross	$\prod_i (1 + \alpha_i) \leq 1 + p$	Hyperbolic order p

Table 3-1. Non-adaptive basis selection options.

Viewing Q as a random variable that is induced by the uniform random vector $\vec{\xi}$, the expansion (3.2) belongs to a general class of Polynomial Chaos (PC) expansions [18, 33]. PC expansions serve as convenient means of representing a large class of random variables, endowed with an efficient machinery for uncertainty propagation through computational models. As such, one can view Eq.(3.2) as a PC representation of the output random variable Q as propagated from uniform random inputs $\boldsymbol{\lambda}$ through the physical model $f(\boldsymbol{\lambda})$.

3.2.1. Construction methods

One can directly employ the basis polynomial orthogonality to compute PC coefficients via *projection* by

$$c_{\boldsymbol{\alpha}}^{Proj} = \langle f(\boldsymbol{\lambda}(\vec{\xi}))\Psi_{\boldsymbol{\alpha}}(\vec{\xi})\rangle = \int_{\vec{\xi}} f(\boldsymbol{\lambda}(\vec{\xi}))\Psi_{\boldsymbol{\alpha}}(\vec{\xi})2^{-d}d\vec{\xi} \text{ for all } \boldsymbol{\alpha} \in \mathcal{I}, \tag{3.4}$$

which minimizes the L_2-distance between the function f and its surrogate

$$\boldsymbol{c}^{Proj} = \arg\min_{\boldsymbol{c}} \int_{\vec{\xi}} \left(f(\boldsymbol{\lambda}(\vec{\xi})) - \sum_{\boldsymbol{\alpha}\in\mathcal{I}} c_{\boldsymbol{\alpha}}\Psi_{\boldsymbol{\alpha}}(\vec{\xi}) \right)^2 d\vec{\xi}. \tag{3.5}$$

Note that in Eq. (3.5), as well as in formulae further in this work, we denote the set of all coefficients as $\boldsymbol{c} = \{c_{\boldsymbol{\alpha}}\}_{\boldsymbol{\alpha}\in\mathcal{I}}$ and call it a PC coefficient vector.

The projection integral in (5.17) is typically computed by quadrature integration

$$c_{\boldsymbol{\alpha}}^{Proj} \approx \sum_{q=1}^{Q} f(\boldsymbol{\lambda}(\vec{\xi}^{(q)}))\Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(q)})w_q, \tag{3.6}$$

where quadrature point-weight pairs $\{(\vec{\xi}^{(q)}, w_q)\}_{q=1}^{Q}$ are chosen such that the integration of the highest-degree polynomial coefficient is sufficiently accurate. If one uses the common, product-grid quadrature, the number of required function-evaluations grows exponentially with dimension, and becomes infeasible even for moderate dimensions of $d = 5$ to 10. For moderate-dimensional problems, one can use sparse quadrature [16, 2, 21, 60] in order to reduce the number of function evaluations for a given level of integration accuracy. However, for high-dimensional problems the projection is generally infeasible, since it requires function evaluations at predefined parameter values $\boldsymbol{\lambda}(\vec{\xi}^{(q)})$ corresponding to quadrature points, and the number of such evaluations grows rapidly with d. As an alternative, regression-based approach is employed here, in which one directly minimizes a distance measure between a set of evaluations of the function $\boldsymbol{f} = \{f(\boldsymbol{\lambda}(\vec{\xi}^{(n)}))\}_{n=1}^{N}$ and the surrogate. The evaluations of the surrogate can be written in a matrix form

$$\boldsymbol{g_c} = \left\{ \sum_{\boldsymbol{\alpha}\in\mathcal{I}} c_{\boldsymbol{\alpha}}\Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(n)}) \right\}_{n=1}^{N} = \vec{G}\boldsymbol{c} \tag{3.7}$$

denoting the *measurement* matrix by $\vec{G}_{nk} = \Psi_k(\vec{\xi}^{(n)})$, where $k = k(\boldsymbol{\alpha})$ is a counting index of the multiindex set $\mathcal{I}$. The minimization problem can then generally be written as

$$\boldsymbol{c}^{Regr} = \arg\min_{\boldsymbol{c}} \ \rho(\boldsymbol{f}, \boldsymbol{g_c}), \tag{3.8}$$

where $\rho(\boldsymbol{u}, \boldsymbol{v})$ is a distance measure between two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. Most commonly, one chooses an ℓ_2 distance $\rho(\boldsymbol{f}, \boldsymbol{g_c}) = ||\boldsymbol{f} - \boldsymbol{g}_c||_2$, leading to a least-squares estimate

$$\boldsymbol{c}^{LSQ} = \arg\min_{\boldsymbol{c}} \ \sum_{n=1}^{N} \left(f(\boldsymbol{\lambda}(\vec{\xi}^{(n)})) - \sum_{\boldsymbol{\alpha}\in\mathcal{I}} c_{\boldsymbol{\alpha}}\Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(n)}) \right)^2 = \arg\min_{\boldsymbol{c}} \ ||\boldsymbol{f} - \vec{G}\boldsymbol{c}||_2 \tag{3.9}$$

that has a closed-form solution

$$\boldsymbol{c}^{LSQ} = (\vec{G}^T\vec{G})^{-1}\vec{G}^T\boldsymbol{f}. \tag{3.10}$$

Both projection and regression fall into the category of *collocation* approaches in which surrogate is constructed using a finite set of evaluations of $f(\boldsymbol{\lambda})$ [60, 61, 35]. In this work, we operate under assumption that model evaluations are expensive, hence one would like to achieve accurate surrogate construction with a limited number of model evaluations. In this regard, the regression approach allows regularization, for example, to build in additional constraints on parameters $\boldsymbol{c}$ via augmenting the distance measure $\rho(\cdot,\cdot)$ with an extra term $r(\boldsymbol{c})$

$$\boldsymbol{c}^{RegulRegr} = \arg\min_{\boldsymbol{c}} \ \left[\rho(\boldsymbol{f}, \boldsymbol{g}_{\boldsymbol{c}}) + r(\boldsymbol{c})\right]. \tag{3.11}$$

Besides, regression allows direct extension to *Bayesian* framework, which allows flexibility and meaningful results even in presence of limited number of evaluations of the expensive forward model $f(\boldsymbol{\lambda})$.

Bayesian regression: Bayesian methods [3, 52, 6] are well-suited to deal with a limited number and potentially noisy function evaluations. They allow constructing an *uncertain* surrogate with any number of samples by describing the uncertainty via posterior probability distribution on PC coefficient vector $\boldsymbol{c}$. Besides, Bayesian techniques are efficient in sequential scenarios where the surrogate is updated *online*, *i.e.* as new evaluations of $f(\boldsymbol{\lambda})$ arrive [48]. While computationally more expensive than the simple minimization (3.8), the Bayesian approach puts the construction of the objective function $\rho(\boldsymbol{f}, \boldsymbol{g}_{\boldsymbol{c}})$ within a formal probabilistic context where the objective function can be interpreted as a Bayesian log-likelihood. For example, the ℓ_2 or least-squares objective function corresponds to an *i.i.d.* Gaussian assumption for the misfit random variable $f(\boldsymbol{\lambda}) - g_{\boldsymbol{c}}(\boldsymbol{\lambda})$. Indeed, Bayes' formula in the regression context reads as

$$\overbrace{p(\boldsymbol{c}|\mathcal{D})}^{\text{Posterior}} = \frac{\overbrace{p(\mathcal{D}|\boldsymbol{c})}^{\text{Likelihood}}\overbrace{p(\boldsymbol{c})}^{\text{Prior}}}{\underbrace{p(\mathcal{D})}_{\text{Evidence}}}, \tag{3.12}$$

relating a prior probability distribution on PC surrogate coefficients $\boldsymbol{c}$ to the posterior distribution, via the likelihood function

$$\mathcal{L}_{\mathcal{D}}(\boldsymbol{c}) = p(\mathcal{D}|\boldsymbol{c}), \tag{3.13}$$

which essentially measures the goodness-of-fit of the model *training* evaluations $\mathcal{D} = \{\boldsymbol{f}\}$ to the surrogate model evaluations $\boldsymbol{g}_{\boldsymbol{c}}$ for a parameter set $\boldsymbol{c}$. As far as the estimation of $\boldsymbol{c}$ is concerned, the evidence $p(\mathcal{D})$ is simply a normalizing factor. The posterior distribution reaches its maximum at the Maximum a Posteriori (MAP) value. Working with logarithms of the prior and posterior distributions as well as the likelihood, the MAP value solves the optimization problem

$$\boldsymbol{c}^{MAP} = \arg\max_{\boldsymbol{c}} \log p(\boldsymbol{c}|\mathcal{D}) = \arg\max_{\boldsymbol{c}} \ \left[\log L_{\mathcal{D}}(\boldsymbol{c}) + \log p(\boldsymbol{c})\right] \tag{3.14}$$

Comparing this formulation with the regularized regression (3.11), we note that (3.14) is equivalent to the deterministic, regularized regression with the negative log-likelihood $-\log L_{\mathcal{D}}(\boldsymbol{c})$ playing the role

of an objective function augmented by the regularization term that is the negative log-prior $-\log p(\boldsymbol{c})$. In principle, Bayesian framework also allows inclusion of nuisance parameters, *e.g.* parameters of the prior or the likelihood, that are inferred together with $\boldsymbol{c}$ and subsequently integrated out to lead to marginal posterior distributions on $\boldsymbol{c}$. In a classical case, assuming a uniform prior $p(\boldsymbol{c})$ and an *i.i.d* Gaussian likelihood with, say, constant variance σ^2,

$$-\log L_{\mathcal{D}}(\boldsymbol{c}) = \frac{N}{2}\log 2\pi + N\log\sigma + \frac{1}{2\sigma^2}||\boldsymbol{f} - \vec{G}\boldsymbol{c}||^2, \tag{3.15}$$

one arrives at a multivariate normal posterior distribution for the coefficient vector

$$\boldsymbol{c} \sim \mathcal{MVN}((\underbrace{(\vec{G}^T\vec{G})^{-1}\vec{G}^T\boldsymbol{f}}_{\boldsymbol{\mu_c}}, \underbrace{\sigma^2(\vec{G}^T\vec{G})^{-1}}_{\boldsymbol{\Sigma_c}}). \tag{3.16}$$

Clearly, the posterior mean value is equal to $\boldsymbol{c}^{MAP}$ and also coincides with the least-squares estimate (3.10). With the probabilistic description of $\boldsymbol{c}$, the PC surrogate is *uncertain*, and is in fact a Gaussian process with analytically computable mean and covariance functions

$$g_{\boldsymbol{c}}(\boldsymbol{\lambda}(\vec{\xi})) \sim \mathcal{GP}\left(\boldsymbol{\Psi}(\vec{\xi})\boldsymbol{\mu_c}, \boldsymbol{\Psi}(\vec{\xi})\boldsymbol{\Sigma_c}\boldsymbol{\Psi}(\vec{\xi}')^T\right), \tag{3.17}$$

where $\boldsymbol{\Psi}(\vec{\xi})$ is the basis measurement vector at parameter value $\vec{\xi}$, *i.e.* its k-th entry is $\boldsymbol{\Psi}(\vec{\xi})_k = \Psi_k(\vec{\xi})$. A key strength of the Bayesian approach is that it leads to a probabilistic surrogate that quantifies the uncertainty due to lack of enough function evaluations.

The curse of dimensionality: High-dimensionality poses major challenges for the PC surrogate construction. First of all, the number of function-evaluations needed to achieve a comparable accuracy grows rapidly with dimensionality, considerably impacting the accuracy standard one wants to achieve, particularly for expensive forward models. In this regard, Bayesian methods are arguably the best option as they provide meaningful surrogates with uncertainty associated with the lack of information, *i.e.* sufficient number of function evaluations. The second major challenge is associated with the rapid growth of the polynomial basis sets. In the present work, the surrogate construction is associated with input parameter vector $\boldsymbol{\lambda}$ of dimensionality $d \approx 50$. The non-adaptive truncation options (as listed in Table 3-1) typically lead to infeasibly large basis sets. For example, the total order truncation with order p leads to $K = (d+p)!/(d!p!)$ basis terms. For $d = 50$, only a second-order expansion already requires $K = 1326$ basis terms. The tensor product truncation would require a much higher number, $K = 3^{50}$, of basis terms. The L_q and hyperbolic cross truncation options require less basis terms, but still grow fast with dimensionality. While Smolyak construction [54, 9], high-dimensional model representation [40] or anisotropic truncations [17] delay the basis growth to an extent and are reasonable options for moderate dimensionalities ($d \approx 10$), they rely on strong assumptions (smoothness, low-rank structure or low effective dimensionality, respectively) of the function $f(\boldsymbol{\lambda})$, and generally are infeasible for $d \approx 50$. The main limitation is that the number of model evaluations N is typically smaller than the degrees of freedom, *i.e.* the number of unknown PC coefficients, in the PC surrogate representation. In such overdetermined cases, the classical least-squares regression is not well-defined, and appropriate regularization techniques need to be applied.

3.2.2. Compressive sensing

Compressive sensing (CS) is a machine learning technique for sparse signal recognition that made a breakthrough in image processing a decade ago [12, 4]. The key premise is that if a sparse signal is present in sufficiently incoherent measurements, one can efficiently recover it with ℓ_1 minimization. In our context, measurements are model evaluations at randomly selected parameter inputs. While the most classical formulation relies on direct ℓ_1 minimization under sufficiently accurate reconstruction constraint, it is generally equivalent to a regularized ℓ_1 minimization problem, which in the PC regression setting reads as

$$\begin{aligned} \boldsymbol{c}^{CS} &= \arg\min_{\boldsymbol{c}} \sum_{n=1}^{N} \left(f(\boldsymbol{\lambda}(\vec{\xi}^{(n)})) - \sum_{\boldsymbol{\alpha}\in\mathcal{I}} c_{\boldsymbol{\alpha}} \Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(n)}) \right)^2 + \gamma \sum_{\boldsymbol{\alpha}\in\mathcal{I}} |\boldsymbol{c}_{\boldsymbol{\alpha}}| = \\ &= \arg\min_{\boldsymbol{c}} \left[||\boldsymbol{f} - \vec{G}\boldsymbol{c}||_2 + \gamma ||\boldsymbol{c}||_1 \right]. \end{aligned} \tag{3.18}$$

In the simplest setting, the regularization parameter $\gamma > 0$ is typically chosen with cross-validation methods[28]. It controls the relative importance of the penalty with respect to the goodness-of-fit. The sparsest solution, *i.e.* the solution with the fewest non-zero PC coefficients, corresponds to the ℓ_0 norm, while the ℓ_1 solution provides the reconstruction, while remaining a computationally tractable convex optimization problem, with high probability *given* sufficiently mild conditions on the sample set $\{\vec{\xi}^{(n)}\}$ and basis functions $\Psi_{\boldsymbol{\alpha}}(\vec{\xi})$[4, 12].

Bayesian compressive sensing: The Bayesian analog of the ℓ_1 regularization is called Bayesian Compressive Sensing (BCS) [30, 1, 49], which uses the same likelihood function as in classical Bayesian regression (3.15) and invokes the sparsity prior in the form of a Laplace distribution,

$$p(\boldsymbol{c}) = \left(\frac{\gamma}{2}\right)^K \exp\left(-\gamma \sum_{\boldsymbol{\alpha}\in\mathcal{I}} |\boldsymbol{c}_{\boldsymbol{\alpha}}| \right), \tag{3.19}$$

where $K = |\mathcal{I}|$. Note that the negative log-prior $-\log p(\boldsymbol{c}) = \text{const} + \quad \gamma \sum_{\boldsymbol{\alpha}\in\mathcal{I}} |\boldsymbol{c}_{\boldsymbol{\alpha}}|$ plays a role of the regularization term in Eq. (3.18). In fact, in order to effectively achieve the prior form (3.19), one can implement a hierarchical Bayesian construction with Gaussian prior distribution on the PC coefficients $\boldsymbol{c}$

$$p(c_{\boldsymbol{\alpha}}|s^2_{\boldsymbol{\alpha}}) = \frac{1}{\sqrt{2\pi s^2_{\boldsymbol{\alpha}}}} \exp\left(-\frac{c^2_{\boldsymbol{\alpha}}}{2s^2_{\boldsymbol{\alpha}}} \right) \tag{3.20}$$

and Gamma prior distribution on the Gaussian widths $s^2_{\boldsymbol{\alpha}}$

$$p(s^2_{\boldsymbol{\alpha}}|\gamma) = \frac{\gamma^2}{2} \exp\left(-\frac{\gamma^2 s^2_{\boldsymbol{\alpha}}}{2} \right), \tag{3.21}$$

which together yield the Laplace prior (3.19) when marginalizing over $s^2_{\boldsymbol{\alpha}}$,

$$p(\boldsymbol{c}|\gamma^2) = \int_0^\infty \prod_{\boldsymbol{\alpha}\in\mathcal{I}} p(c_{\boldsymbol{\alpha}}|s^2_{\boldsymbol{\alpha}}) p(s^2_{\boldsymbol{\alpha}}|\gamma^2) ds^2_{\boldsymbol{\alpha}} = \prod_{\boldsymbol{\alpha}\in\mathcal{I}} \frac{\gamma}{2} e^{-\gamma|c_{\boldsymbol{\alpha}}|}. \tag{3.22}$$

The regression technique with such hierarchical prior construction is a basis for the Bayesian Lasso methodology [39, 24]. However, the solution approach in BCS is very similar to Relevance Vector Machine (RVM) [58]. Unlike BCS, the RVM approach uses the inverse-variances $r_{\boldsymbol{\alpha}} = s_{\boldsymbol{\alpha}}^{-1}$ as a hyperparameter and endows *them* with a Gamma prior instead of Eq. (3.21). Less importantly, RVM is developed in the context of radial basis functions, but the technique is equally applicable with polynomial bases. While the full hierarchical Bayesian solution is generally difficult to evaluate, one resorts to fixed values of the intermediary parameter vector $\boldsymbol{s}^2$, which is a convenient notation we will take for the vector of prior variances $\{s_{\boldsymbol{\alpha}}^2 : \boldsymbol{\alpha} \in \mathcal{I}\}$. With fixed $s_{\boldsymbol{\alpha}}$, σ^2 and γ, the posterior distribution of the coefficient vector $\boldsymbol{c}$ follows a multivariate normal distribution $\boldsymbol{c}|\boldsymbol{s}_{\boldsymbol{\alpha}}^2, \sigma^2, \gamma \sim \mathcal{MVN}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with covariance and mean defined as

$$\boldsymbol{\Sigma} = \sigma^2 \left(\vec{G}^T \vec{G} + \boldsymbol{S}\right)^{-1} \tag{3.23}$$

$$\boldsymbol{\mu} = \sigma^{-2} \boldsymbol{\Sigma} \vec{G}^T \boldsymbol{f} \tag{3.24}$$

where $\boldsymbol{S}$ is the diagonal matrix of prior variance fractions defined as $S_{kk} = \sigma^2 / s_k^2$ for $k = 1, \ldots, K$. Note that we identify each multiindex $\boldsymbol{\alpha}$ with its counting index $k(\boldsymbol{\alpha})$, and then drop $\boldsymbol{\alpha}$ for the simplicity of the notation $s_k = s_{k(\boldsymbol{\alpha})} = s_{\boldsymbol{\alpha}}$.

The values s_k^2 are fixed, and, together with σ^2 are found according to an evidence maximization procedure, effectively assuming very narrow priors for them [58, 59, 1]. The key observation drawn from [58] is that this approximation has relatively mild effect on the prediction, or the quality of the surrogate model approximation. The logarithm of evidence can be expressed analytically directly using its definition,

$$\mathcal{L}_{\boldsymbol{f},\gamma}(\boldsymbol{s}^2, \sigma^2) = \log p(\boldsymbol{f}|\boldsymbol{s}^2, \sigma^2, \gamma) = \log \int_{\boldsymbol{c}} p(\boldsymbol{f}|\boldsymbol{c}, \sigma^2) p(\boldsymbol{c}|\boldsymbol{s}^2) p(\boldsymbol{s}^2|\gamma) d\boldsymbol{c} =$$

$$= -\frac{1}{2}\left(K \log 2\pi + \log|\boldsymbol{Z}| + \boldsymbol{f}^T \boldsymbol{Z}^{-1} \boldsymbol{f}\right) + \underbrace{K \log\left(\frac{\gamma^2}{2}\right) - \frac{\gamma^2}{2} \sum_{i=k}^{K} s_k^2}_{\text{Prior:} \quad \log p(\boldsymbol{s}^2|\gamma)}, \tag{3.25}$$

where $\boldsymbol{Z} = \sigma^2 \left(\boldsymbol{E}_N + \vec{G} \boldsymbol{S}^{-1} \vec{G}^T\right)$, with $\boldsymbol{E}_N$ defined as the $N \times N$ identity matrix. We note two important matrix relations that help rewrite (3.25) before taking derivatives [58]:

$$\begin{aligned}
\log|\boldsymbol{Z}| &= N \log(\sigma^2) + \log|\boldsymbol{E}_N + \vec{G} \boldsymbol{S}^{-1} \vec{G}^T| = \text{[employ Matrix Determinant Lemma]} = \\
&= N \log(\sigma^2) + \log\left(|\boldsymbol{S}|^{-1} |\boldsymbol{S} + \vec{G}^T \vec{G}|\right) = N \log(\sigma^2) - \log|\boldsymbol{S}| + K \log(\sigma^2) - \log|\boldsymbol{\Sigma}| = \\
&= N \log(\sigma^2) + \sum_{k=1}^{K} \log(s_k^2) - \log|\boldsymbol{\Sigma}|
\end{aligned} \tag{3.26}$$

and

$$\begin{aligned}
\boldsymbol{f}^T \boldsymbol{Z}^{-1} \boldsymbol{f} &= \sigma^{-2} \boldsymbol{f}^T \left(\boldsymbol{E}_N + \vec{G} \boldsymbol{S}^{-1} \vec{G}^T\right) \boldsymbol{f} = \text{[employ Woodbury Matrix Identity]} = \\
&= \sigma^{-2} \boldsymbol{f}^T \boldsymbol{f} - \sigma^{-2} \boldsymbol{f}^T \vec{G} \left(\boldsymbol{S} + \vec{G}^T \vec{G}\right)^{-1} \vec{G}^T \boldsymbol{f} = \sigma^{-2} \boldsymbol{f}^T \left(\boldsymbol{f} - \vec{G} \boldsymbol{\mu}\right) = \\
&= \sigma^{-2} ||\boldsymbol{f} - \vec{G} \boldsymbol{\mu}||_2^2 + \sigma^{-2} \boldsymbol{\mu}^T \boldsymbol{S} \boldsymbol{\mu} = \sigma^{-2} ||\boldsymbol{f} - \vec{G} \boldsymbol{\mu}||_2^2 + \boldsymbol{\mu}^T \text{diag}(s_k^{-2}) \boldsymbol{\mu}.
\end{aligned} \tag{3.27}$$

Furthermore, we use the relation

$$\frac{\partial}{\partial X_{ij}} \log |X^{-1}| = -\frac{\partial}{\partial X_{ij}} \log |X| = -\frac{1}{|X|}\frac{\partial |X|}{\partial X_{ij}} = -\frac{(X^*)_{ji}}{|X|} = -(X^{-1})_{ji} \tag{3.28}$$

to derive

$$\frac{\partial \log |\boldsymbol{\Sigma}|}{\partial s_k^2} = \frac{1}{s_k^4}\Sigma_{kk} \qquad \text{and} \qquad \frac{\partial \log |\boldsymbol{\Sigma}|}{\partial \sigma^2} = \frac{K}{\sigma^2} - \frac{1}{\sigma^2}\sum_{k=1}^{K} s_k^{-2}\Sigma_{kk} \tag{3.29}$$

Now, taking derivatives of the evidence (3.25) with respect to s_k^2 leads to

$$\frac{\partial \mathcal{L}_{\boldsymbol{f},\gamma}}{\partial s_k^2} = \frac{1}{2}\left(-\frac{1}{s_k^2} + \frac{\mu_k^2 + \Sigma_{kk}}{s_k^4}\right) - \frac{\gamma^2}{2}. \tag{3.30}$$

We remark that the last term in (3.30) corresponds to the Gamma prior on s_k^2, and has a different form in the RVM formulation [58], in which s_k^2 follows an inverse-gamma distribution. Similarly, one can take derivative with respect to σ^2 to get

$$\frac{\partial \mathcal{L}_{\boldsymbol{f},\gamma}}{\partial \sigma^2} = -\frac{1}{2}\left(\frac{N}{\sigma^2} - \frac{K}{\sigma^2} + \frac{\sum_{k=1}^{K} s_k^{-2}\Sigma_{kk}}{\sigma^2} - \frac{||\boldsymbol{f} - \vec{G}\boldsymbol{\mu}||_2^2}{\sigma^4}\right), \tag{3.31}$$

leading to the best value of σ^2

$$\sigma^2_{best} = \frac{||\boldsymbol{f} - \vec{G}\boldsymbol{\mu}||_2^2}{N - K + \sum_{k=1}^{K} s_k^{-2}\Sigma_{kk}} \tag{3.32}$$

to be used later in the algorithm.

Noting that μ_k and Σ_{kk} depend on s_k and setting the derivatives (3.30) equal to zero can lead to update mechanisms for finding the optimal s_k's. In [59, 1] it has been shown that the optimal s_k's vanish for certain k's essentially suggesting that the corresponding basis should be deleted from the multi-index set. Furthermore, the search for the optimal s_k's admits an efficient basis deletion-addition strategy, in which an iterative procedure leads to a basis set $\mathcal{I}_s \subset \mathcal{I}$ that is taken as the final basis set; see [1, 49], and, in the RVM setting, [58, 59]. In this work, this procedure is generalized to include dimension-specific γ_k, instead of a single γ as described in Section 3.3. After the sparse basis $\mathcal{I}_s$ is found, one can then retain the estimated values for σ^2 and $\boldsymbol{s}^2$ to construct PC coefficient mean and covariance via Eqs (3.24) and (3.23) and, consequently, the uncertain surrogate in a Gaussian Process form (3.17).

3.3. WEIGHTED ITERATIVE CS FOR BASIS SELECTION

In order to achieve more efficient recovery of a sparse set of polynomials, one can generalize the standard ℓ_1 minimization problem to *weighted* regularization, making the parameter γ specific to each PC coefficient,

$$\boldsymbol{c}^{WCS} = \arg\min_{\boldsymbol{c}} \sum_{n=1}^{N}\left(f(\boldsymbol{\lambda}(\vec{\xi}^{(n)})) - \sum_{\boldsymbol{\alpha}\in\mathcal{I}} c_{\boldsymbol{\alpha}}\Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(n)})\right)^2 + \sum_{\boldsymbol{\alpha}\in\mathcal{I}}\gamma_{\boldsymbol{\alpha}}|\boldsymbol{c}_{\boldsymbol{\alpha}}|. \tag{3.33}$$

The a priori selection of weights $\gamma_{\boldsymbol{\alpha}}$ can be quite challenging. The optimal choice $\gamma_{\boldsymbol{\alpha}} = |\boldsymbol{c}_{\boldsymbol{\alpha}}|^{-1}$, which essentially equates the ℓ_1 minimization to the sparsest ℓ_0 minimization problem, is infeasible since the coefficients $\boldsymbol{c}_{\boldsymbol{\alpha}}$ are unknown to begin with, but it suggests an efficient iterative algorithm for sparse coefficient retrieval. This is the basis of iteratively reweighting approaches that start from an initial vector $\gamma_{\boldsymbol{\alpha}}^{(0)}$, then, at each iteration with $\gamma_{\boldsymbol{\alpha}}^{(i)}$, solve the minimization problem (3.33) to achieve new coefficient vector $\boldsymbol{c}_{\boldsymbol{\alpha}}^{(i)}$ and to update the weights for the next iteration $\gamma_{\boldsymbol{\alpha}}^{(i+1)} = \left(|\boldsymbol{c}_{\boldsymbol{\alpha}}^{(i)}| + \epsilon\right)^{-1}$ [13, 5]. The 'nugget' parameter $\epsilon > 0$ is usually selected to be very small in order to simply stabilize the iterative scheme and avoid unrealistically large weights.

In this work, we generalize the Bayesian Compressive Sensing method to allow weighted basis search by making the parameters of the Gamma distribution in (3.21) coefficient-specific:

$$p(s_{\boldsymbol{\alpha}}^2|\gamma_{\boldsymbol{\alpha}}^2) = \frac{\gamma_{\boldsymbol{\alpha}}^2}{2} \exp\left(-\frac{\gamma_{\boldsymbol{\alpha}}^2 s_{\boldsymbol{\alpha}}^2}{2}\right). \tag{3.34}$$

As a consequence, the iterative search procedure for sparse learning [1, 49, 58, 59], which approximates the BCS solution, is generalized to accommodate multiple $\gamma_{\boldsymbol{\alpha}}$ instead of a single γ. Below we describe the algorithm in a nutshell, referring the reader to [1, 58, 59] for technical details in a special case $\gamma_{\boldsymbol{\alpha}} = 0$. For clarity of matrix-vector notations, let us again identify $\gamma_{\boldsymbol{\alpha}} = \gamma_k$ for k-th basis term for some indexing $k(\boldsymbol{\alpha})$. The log-evidence from Eq. (3.25) can be written in a form that isolates the contribution from the k-th basis term as

$$\begin{aligned}
\mathcal{L}_{\boldsymbol{f},\gamma}(\boldsymbol{s}^2, \sigma^2) \;\; = \;\; & \underbrace{-\frac{1}{2}\left(K\log 2\pi + \log|\boldsymbol{Z}_{-k}| + \boldsymbol{f}^T\boldsymbol{Z}_{-k}^{-1}\boldsymbol{f}\right)}_{\mathcal{L}_{\boldsymbol{f},\gamma}(\boldsymbol{s}_{-k}^2,\sigma^2)} - \\
- \;\; & \underbrace{\frac{1}{2}\left[\log\left(1 + r_k s_k^2\right) - \frac{q_k^2 s_k^2}{1 + s_k^2 r_k}\right]}_{l(s_k)} + \\
+ \;\; & \underbrace{\sum_{k=1}^{K}\log\left(\frac{\gamma_k^2}{2}\right) - \frac{1}{2}\sum_{k=1}^{K}\gamma_k^2 s_k^2}_{\text{Prior:}\quad \log p(\boldsymbol{s}^2|\boldsymbol{\gamma})}
\end{aligned} \tag{3.35}$$

where we have isolated the contribution from the k-th basis in $\boldsymbol{Z}_{-k} = \boldsymbol{Z} - s_k^2\phi_k\phi_k^T$, and introduced quantities r_k and q_k as

$$r_k = \phi_k^T\boldsymbol{Z}_{-k}^{-1}\phi_k, \qquad \text{and} \qquad q_k = \phi_k^T\boldsymbol{Z}_{-k}^{-1}\boldsymbol{f}, \tag{3.36}$$

with ϕ_k being the k-th column of $\vec{G}$. Intuitively, the 'sparsity factor' r_k is a measure of 'overlap' the basis vector ϕ_k has with the rest of the bases, while the 'quality factor' is interpreted as a measure of the alignment of ϕ_k with the error of the model with that basis excluded [59]. The derivative of the evidence with respect to s_k^2 is then equal to

$$\frac{\partial\mathcal{L}_{\boldsymbol{f},\gamma}}{\partial s_k^2} = \frac{1}{2}\left(-\frac{r_k}{1 + r_k s_k^2} + \frac{q_k^2}{(1 + r_k s_k^2)^2} - \gamma_k^2\right). \tag{3.37}$$

which is simply another form of (3.30), enabling a convenient basis addition-deletion algorithm as follows. Setting the derivative in (3.37) equal to zero, one arrives at values

$$s_k^2 = \begin{cases} \frac{-r_k(r_k+2\gamma_k)+r_k\sqrt{(r_k+2\gamma_k)^2-4\gamma_k(r_k-q_k^2+\gamma_k)}}{2\gamma_k r_k^2}, & \text{if } q_k^2 - r_k > \gamma_k \\ 0, & \text{if } q_k^2 - r_k \leq \gamma_k \end{cases} \tag{3.38}$$

that maximize the evidence with respect to k-th dimension. This does not guarantee a global maximum, but allows for a sequential, dimension-wise maximization algorithm that is very fast and leads to at least a local maximum of the evidence with respect to the vector $\boldsymbol{s}^2$. The procedure is shown in Algorithm 1. This is a generalization of the fast marginal likelihood maximization algorithm developed in [58, 59] (case $\gamma_k = 0$ for all k), and fast Laplace maximization developed in the original BCS work [1] (case $\gamma_k = \gamma$ for all k). As one intuitively expects, having basis-specific γ_k's allows additional flexibility as higher γ_k indicates that the corresponding basis term is more susceptible to being pruned from the basis set. This is exactly what the weighted regularization (3.33) would suggest as well.

Algorithm 1: Fast iterative algorithm for weighted BCS

Input:

- Model evaluations $\boldsymbol{f}$, regularization weights γ_k's for $k = 1, \ldots, K$
- Initialize $\sigma^2 = 0.01\text{Var}(\boldsymbol{f})$
- Initial basis set $\mathcal{I}$ of size $K = |\mathcal{I}|$
- Initial selected basis set $\mathcal{I}_{WBCS} = \emptyset$
- Set all $s_k^2 = 0$ for all $k = 1, \ldots, K$
- η, stopping criterion is $\mathcal{L}^{(i)} - \mathcal{L}^{(i-1)} < \eta \left(\mathcal{L}^{(i)} - \mathcal{L}^{(0)}\right)$, where $\mathcal{L}^{(i)}$ is log-evidence (3.35) at i-th iteration.
- Iteration counter $i = 0$
- Select a basis index k and add the corresponding basis to $\mathcal{I}_{WBCS}$

  - Typically select k with the highest ratio $||\phi_k^T \boldsymbol{f}||^2/||\phi_k||^2$, see [59].
- Compute $s_k^2 = \frac{||\phi_k^T \boldsymbol{f}||^2/||\phi_k||^2 - \sigma^2}{||\phi_k||^2}$ (special case of (3.36), (3.37) and (3.38) with $\gamma_k = 0$)
- Compute $\boldsymbol{\Sigma}$ and $\boldsymbol{\mu}$ according to (3.23), with a basis set $\mathcal{I}_{WBCS}$ (*i.e.* scalars for now)

while *Stopping criterion not met* **do**

- For all k, compute s_k^2 according to (3.38), with a basis set $\mathcal{I}_{WBCS}$
- Add all bases with $s_k^2 > 0$ to $\mathcal{I}_{WBCS}$
- Update $\boldsymbol{\Sigma}$ and $\boldsymbol{\mu}$ according to (3.23), with a basis set $\mathcal{I}_{WBCS}$
- Update σ^2 according to (3.32)
- Iteration counter $i = i + 1$

end

Result:

- Final basis set $\mathcal{I}_{WBCS}$
- Corresponding coefficients $\boldsymbol{c} \sim \mathcal{MVN}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

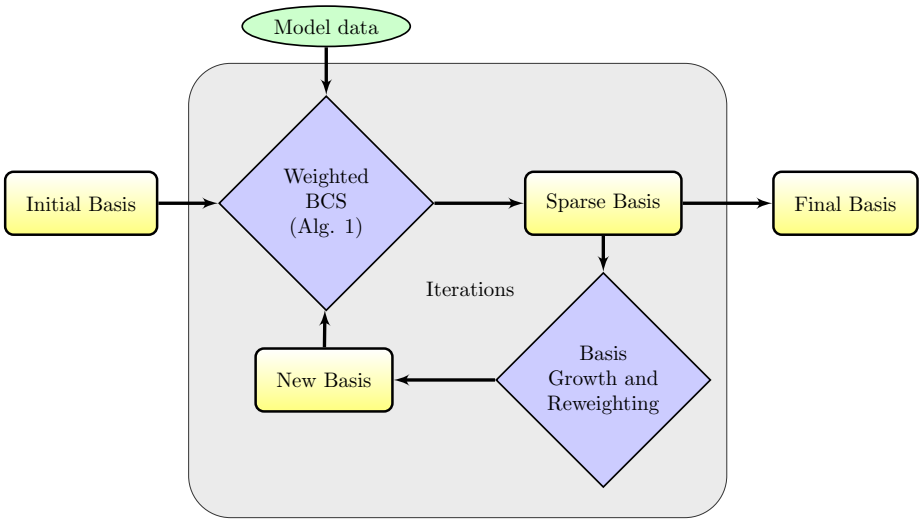


Figure 3-1. Sketch of the overall iterative procedure of basis shrinkage and growth. The steps correspond to Algorithm 2. This procedure is repeated for randomly selected subsets of the training set to arrive at a final basis set that is not overfitting as described in Algorithm 3.

Algorithm 2: Weighted iterative (Bayesian) compressive sensing

Input:

- Model evaluations
- Initial basis set $\mathcal{I}^{(0)}$
- Initial weights $\gamma_{\boldsymbol{\alpha}}^{(0)}$ for $\boldsymbol{\alpha} \in \mathcal{I}^{(0)}$
- ϵ, stopping criterion.
- $i = 0$

while *Stopping criterion not met* **do**

- *Shrink basis set:* Perform weighted BCS described in Algorithm 1 on current basis set $\mathcal{I}^{(i)}$ to arrive at sparse basis set $\mathcal{I}_s^{(i)}$ and corresponding coefficients $c_{\boldsymbol{\alpha}}^{(i)}$ for $\boldsymbol{\alpha} \in \mathcal{I}_s^{(i)}$.
- *Grow basis set:* Enrich the basis set by adding admissible bases to arrive at new basis set $\mathcal{I}^{(i+1)}$.
- Update the weights according to

  [a] $\gamma_{\boldsymbol{\alpha}}^{(i+1)} = \left(\left| c_{\boldsymbol{\alpha}}^{(i)} \right| + \epsilon \right)^{-1}$ if $\boldsymbol{\alpha} \in \mathcal{I}_s^{(i)}$, or

  [b] $\gamma_{\boldsymbol{\alpha}}^{(i+1)} = \epsilon^{-1}$ if $\boldsymbol{\alpha} \in \mathcal{I}^{(i+1)} \backslash \mathcal{I}_s^{(i)}$ (*i.e.* if $\boldsymbol{\alpha}$ is a newly added basis)
- Move the iteration counter $i = i + 1$.

end

Result:

- Final basis set $\mathcal{I}^f = \mathcal{I}^{(i+1)}$
- Corresponding coefficients $c_{\boldsymbol{\alpha}}^f = c_{\boldsymbol{\alpha}}^{(i)}$ for $\boldsymbol{\alpha} \in \mathcal{I}^f$.

The weighted iterative BCS algorithm effectively shrinks the basis sets. However, it assumes an initial basis set to be constructed, which ideally should include the eventual sparse basis set as a subset. Given the high-dimensionality of the parameter space, it is not feasible to start with such a large basis set. For example, a total-order truncation basis of dimensionality $d = 50$ leads to $52!/(50!2!) = 1326$ basis terms for second-order truncation and $53!/(50!3!) = 23426$ basis terms for a third order truncation. This leads to unnecessary computational burden and makes it infeasible to properly interrogate higher-order terms. To this end, we implement a basis growth capability according to admissibility criterion as developed and described in [49, 28]. In this work, we merge the iterative growth procedure with the reweighting strategy described above. The sketch of the algorithm is given in Figure 3-1, while the steps are described in Algorithm 2. As a stopping criterion, we have chosen $i < N_{iter}$, *i.e.* simply pass through the basis shrink/grow procedure a preselected number N_{iter} times. For the purposes of this work $N_{iter} = 4$ has been selected, which allows exploration of up to fifth order basis terms, since the initial basis set $\mathcal{I}^{(0)}$ is taken with a total-order truncation rule with $p = 2$. Our preliminary tests, results not shown, however indicate that some degree of *overfitting* remains present for sufficiently low number of function-evaluations N. In other words, the weighted iterative algorithm of basis selection, while properly selecting the high-coefficient bases, also selects a few spurious basis terms that are given high coefficient for the specific sample set, but do not generalize well to parameter settings away from the training points. This regime is certainly in effect for $d = 68$ and $N = 3000$ case that is described in Section **??**. Our remedy in such situations is a cross-validation study followed by basis intersection to select only the relevant basis terms. The technique is detailed in [49], and is described in Algorithm 3 for

completeness.

Algorithm 3: Cross validation for basis selection

Input:

- Model evaluations
- Total trial number K_{CV}

for *m=1 to* K_{CV} **do**

- Randomly select a subset of size $0.9N$, *i.e.* 90% of the current training simulation set.
- With this subset as a training set, perform weighted iterative BCS described in Algorithm 2 to arrive at sparse basis set $\mathcal{I}^{(m)}$.

end

- Get the intersection of all basis sets $\mathcal{I} = \mathcal{I}^{(1)} \cap \mathcal{I}^{(2)} \cap \cdots \cap \mathcal{I}^{(K_{CV})}$
- Perform a regular Bayesian least-squares fit (3.10) with the basis set $\mathcal{I}$ and the full set of N training simulations.

Result:

- Final basis set $\mathcal{I}$
- Corresponding least-squares coefficients $c_{\boldsymbol{\alpha}}$ for $\boldsymbol{\alpha} \in \mathcal{I}$.

3.4. GLOBAL SENSITIVITY ANALYSIS

In this work, the goal is to perform global sensitivity analysis (GSA) of a model with respect to a large number of parameters. In this regard, Sobol sensitivity indices will be employed [56, 44]. These indices correspond to variance-based decomposition, as they measure fractional contributions of each parameter or group of parameters towards the total output variance. We outline three sensitivity indices:

- *Main effect sensitivities,* also called first-order sensitivities, measure variance contribution due to i-th parameter only, defined as

$$S_i = \frac{\mathrm{V}_{\boldsymbol{\lambda}_i} \mathrm{E}_{\boldsymbol{\lambda}_{-i}}[f(\boldsymbol{\lambda})|\boldsymbol{\lambda}_i]}{\mathrm{V} f(\boldsymbol{\lambda})}, \tag{3.39}$$

  where $\mathrm{V}_{\boldsymbol{\lambda}_i}$ and $\mathrm{E}_{\boldsymbol{\lambda}_{-i}}$ indicate variance with respect to the i-th parameter and expectation with respect to the rest of the parameters, respectively.

- *Total effect sensitivities* measure total variance contribution of the i-th parameter, *i.e.* including interactions with other parameters, and are defined as

$$T_i = \frac{\mathrm{E}_{\boldsymbol{\lambda}_{-i}} \mathrm{V}_{\boldsymbol{\lambda}_i}[f(\boldsymbol{\lambda})|\boldsymbol{\lambda}_{-i}]}{\mathrm{V} f(\boldsymbol{\lambda})} = 1 - \frac{\mathrm{V}_{\boldsymbol{\lambda}_{-i}} \mathrm{E}_{\boldsymbol{\lambda}_i}[f(\boldsymbol{\lambda})|\boldsymbol{\lambda}_{-i}]}{\mathrm{V} f(\boldsymbol{\lambda})}, \tag{3.40}$$

  where $\mathrm{E}_{\boldsymbol{\lambda}_i}$ and $\mathrm{V}_{\boldsymbol{\lambda}_{-i}}$ indicate expectation with respect to the i-th parameter and variance with respect to the rest of the parameters, respectively.

- *Joint sensitivities* measure joint variance contribution due to i-th and j-th parameter and are defined as

$$J_{ij} = \frac{\mathrm{V}_{\boldsymbol{\lambda}_{ij}} \mathrm{E}_{\boldsymbol{\lambda}_{-ij}}[f(\boldsymbol{\lambda})|\boldsymbol{\lambda}_{ij}]}{\mathrm{V}f(\boldsymbol{\lambda})} - S_i - S_j, \tag{3.41}$$

  where $\mathrm{V}_{\boldsymbol{\lambda}_{ij}}$ and $\mathrm{E}_{\boldsymbol{\lambda}_{-ij}}$ indicate variance with respect to the i-th and j-th parameters and expectation with respect to the rest of the parameters, respectively.

While there are random sampling approaches [55, 43, 29, 45] for efficient estimation of the integral quantities in formulae (3.39)-(3.41), they all suffer from the generic deficiency pertinent to all random sampling methods. Namely, in order to get accurate enough estimates, one needs prohibitively large number of model evaluations. In this regard, PC surrogates offer a much more efficient alternative. When the function $f(\boldsymbol{\lambda})$ is approximated by a PC surrogate $g_{\boldsymbol{c}}(\boldsymbol{\lambda})$, one can compute moments and sensitivity indices using the orthogonality of the PC basis functions,

$$\mathrm{E}f(\boldsymbol{\lambda}) \approx c_{\mathbf{0}}, \qquad \mathrm{V}f(\boldsymbol{\lambda}) \approx \sum_{\mathbf{0} \neq \boldsymbol{\alpha} \in \mathcal{I}} c_{\boldsymbol{\alpha}}^2, \text{ with } \mathbf{0} = (0, 0, \ldots, 0) \tag{3.42}$$

$$\begin{aligned} S_i &\approx \frac{1}{\mathrm{V}f(\boldsymbol{\lambda})} \sum_{\boldsymbol{\alpha} \in \mathcal{I}_{S_i}} c_{\boldsymbol{\alpha}}^2, \text{ with } \mathcal{I}_{S_i} = \{\boldsymbol{\alpha} : \alpha_i > 0, \alpha_k = 0 \text{ for } k \neq i\} \\ T_i &\approx \frac{1}{\mathrm{V}f(\boldsymbol{\lambda})} \sum_{\boldsymbol{\alpha} \in \mathcal{I}_{T_i}} c_{\boldsymbol{\alpha}}^2, \text{ with } \mathcal{I}_{T_i} = \{\boldsymbol{\alpha} : \alpha_i > 0\} \\ J_{ij} &\approx \frac{1}{\mathrm{V}f(\boldsymbol{\lambda})} \sum_{\boldsymbol{\alpha} \in \mathcal{I}_{J_{ij}}} c_{\boldsymbol{\alpha}}^2, \text{ with } \mathcal{I}_{J_{ij}} = \{\boldsymbol{\alpha} : \alpha_i > 0, \alpha_j > 0\}, \end{aligned} \tag{3.43}$$

where $\mathcal{I}_{S_i}, \mathcal{I}_{T_i}$ and $\mathcal{I}_{J_{ij}}$ are multiindex subsets that include only the terms of interest for the corresponding sensitivity index. Therefore, having constructed the PC surrogate, one can easily evaluate the sensitivity indices by computing the weighted sum-of-the-squares of appropriately selected PC coefficients [57, 10, 46].

4. SOURCE CODE DESCRIPTION

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

4.1. C++ LIBRARIES

The following libraries are included in UQTk (source code in `cpp/lib`)

- `mcmc` : Markov Chain Monte Carlo Base Class
- `amcmc` : Adaptive Markov Chain Monte Carlo
- `tmcmc` : Transitional Markov Chain Monte Carlo
- `ss` : Single Site Markov Chain Monte Carlo
- `mala` : Metropolis-adjusted Langevin algorithm
- `mmala` : Manifold-variant of MALA

4.1.1. `mcmc`:

This directory features the common functionality between the different flavors of Markov Chain Monte Carlo in UQTk. The functions within this base class are:

- `MCMC(double (*logposterior)(Array1D<double>&, void *), void *postinfo)`: Constructor that takes in a pointer to a log posterior function and an additional pointer to information about the posterior
- `MCMC(LogPosteriorBase& L)` : Constructor that takes in a LogPosteriorBase object
- `MCMC()` : Dummy constructor, used exclusively for TMCMC
- `void setWriteFlag(int I)` : Sets the value of the MCMC object's write flag, which determines if the MCMC will be written to the screen, via an integer. A value of 1 indicates that the MCMC will be written to the screen, all other integers will not be
- `void setFcnAccept(void (*fcnAccept)(void *))` : Sets the accept function given a pointer to the function

- `void setFcnReject(void (*fcnReject)(void *))` : Sets the reject function given a pointer to the function
- `void setChainDim(int chdim)` : Sets the chain dimensionality given an integer
- `void initChainPropCov(Array2D<double>& propcov)` : Sets the proposal covariance matrix given a 2d-array. For AMCMC, this matrix is used only before adaptivity starts
- `void initChainPropCovDiag(Array1D<double>& sig)` : Sets the proposal covariance matrix given a 1d-array. For AMCMC, this matrix is used only before adaptivity starts
- `void setOutputInfo(string outtype, string file,int freq_file, int freq_screen)` : Sets the output information for the MCMC given the type of file, the file name, the frequency that the MCMC should print to the file, and the frequency that the MCMC should print to the screen
- `void namesPrepended()` : Sets the MCMC so the names of the parameters are prepended in the output file
- `void setSeed(int seed)` : Sets the seed for random generation
- `void setLower(double lower, int i)` : Set lower bound of MCMC as a double at the index of i
- `void setUpper(double upper, int i)` : Set upper bound of MCMC as a double at the index of i
- `void setDefaultDomain()` : Set the default unbounded domain for MCMC
- `void setPostInfo(void *postinfo)` : Set the posterior information given a pointer to the posterior information
- `void getChainPropCov(Array2D<double>& propcov)` : By passing a 2d-array into the function it sets it equal to the proposal covariance matrix
- `string getFileName()` : Gets the output file name as a string
- `int getWriteFlag()` : Gets the write flag for the MCMC object as an integer. A value of 1 indicates that the MCMC will be outputted to the screen
- `void getSamples(int burnin, int every,Array2D<double>& samples)` : Gets a selective number of the MCMC samples by passing in an integer for the index after the burn-in phase of MCMC has occurred and an integer for how often the chain's samples are added. The samples are then added to a 2D-array that is passed into the function
- `void getSamples(Array2D<double>& samples)` : Gets the full chain of MCMC samples as a 2D-Array that is passed in
- `void getFcnAccept(void (*fcnAccept)(void *))` : Gets the accept function of the MCMC
- `void getFcnReject(void (*fcnReject)(void *))` : Gets the reject function of the MCMC

- `string getOutputType()` : Get the type of file, either binary or text
- `. int getFileFreq()` : Gets how frequently the MCMC prints its output to the file
- `. int getScreenFreq()` : Gets how frequently the MCMC prints its output to the screen
- `bool getNamesPrepended()` : Gets whether or not the names of the parameters are prepended as a bool
- `int getSeed()` : Gets the seed for random generation as an integer
- `double getLower(int i)` : Gets the lower bound limit of the MCMC chain based on an integer index i
- `double getUpper(int i)` : Gets the upper bound limit of the MCMC chain based on an integer index i
- `bool getDimInit()` : Gets if the chain's dimensionality has been set as a bool
- `void getPostInfo(void *post)` : Gets the posterior information given a pointer passed into the function
- `bool getPropCovInit()` : Gets if the proposal covariance matrix has been set as a bool
- `bool getOutputInit()` : Gets if the output information has been set as a bool
- `bool getLastWrite()` : Gets the last index of the MCMC chain written as an integer
- `bool getFcnAcceptInit()` : Gets if the accept function is set as a bool
- `bool getFcnRejectInit()` : Gets if the reject function is set as a bool
- `virtual int getNSubSteps()` : Gets the number of sub steps for the MCMC object. Written as virtual to be redefined for single-site MCMC
- `int getLowerFlag(int i)` : Gets the flag for the lower limit at an integer index of i. A value of 1 indicates that the lower limit has been set, all other values indicate it has not
- `int getUpperFlag(int i)` : Gets the flag for the upper limit at an integer index of i. A value of 1 indicates that the upper limit has been set, all other values indicate it has not
- `void getAcceptRatio(double * accrat)` : Gets the acceptance ratio of the MCMC object by passing in a pointer to a double and setting the value of the object the pointer points to the acceptance ratio
- `double getAcceptRatio()` : Gets the acceptance ratio as a double
- `int GetChainDim() const` : Gets the MCMC chain dimensionality
- `void resetChainState()` : Resets the entire MCMC chain state
- `void resetChainFilename(string filename)` : Resets the MCMC chain state and resets the name of the file that the MCMC will be written to as the string that is passed into the function

- `void parseBinChain(string filename, Array1D<chainstate>& readchain)` : Parses the binary file, passed in as a string, and produces a 1d array of chain-states and writes them to the 1d array passed into the function
- `void writeFullChainTxt(string filename, Array1D<chainstate> fullchain)` : Writes the passed in 1d array of chainstates to the specific text file passed in as a string
- `void getFullChain(Array1D<chainstate>& readchain)` : Gets the full MCMC chain as a passed in 1d array of chainstates
- `void appendMAP()` : Appends the MAP state to the end of the chain
- `double getMode(Array1D<double>& MAPparams)` : Gets the MAP parameters as a double based on the 1d array that is passed into the functions
- `int getFullChainSize()` : Gets the full size of the MCMC chain as an integer
- `void setCurrentStateStep(int i)` : Sets the step of the current state as a given integer
- `void getCurrentStateState(Array1D<double>& state)` : Gets the state of the current state by assigning it to the passed in 1d array
- `double getCurrentStatePost()` : Gets the post of the current state as a double
- `void setCurrentStateState(Array1D<double>& newState)` : Sets the current state's state to the passed in 1d array of doubles
- `void setCurrentStatePost(double newPost)` : Sets the current state's post to the passed in double
- `void setCurrentStateAlfa(double newAlfa)` : Sets the current state's alfa to the passed in double
- `double getModeStatePost()` : Gets the mode state's post as a double
- `void getModeStateState(Array1D<double>& state)` : Gets the mode state's state by assigning it to the 1d array of doubles passed into the function
- `virtual void runOptim(Array1D<double>& start)` : Runs the optimization routine for the MCMC object. Written as a virtual function to be redefined later by derived MCMC classes
- `virtual void runChain(int ncalls, Array1D<double>& chstart)` : Generates the MCMC chain. A pure virtual function that is defined by the derived MCMC classes to reflect their specific MCMC generation variant
- `virtual void runChain(int ncalls)` : Generates the MCMC chain with a trivial initial condition. A pure virtual function that is defined by the derived MCMC classes to reflect their specific MCMC generation variant
- `void runAcceptFcn()` : Runs the accept function for the MCMC object
- `void runRejectFcn()` : Runs the reject function for the MCMC object

- `bool newModeFound()` : Checks to see if a new mode was found during the last call to runChain and returns it as a bool
- `double evalLogPosterior(Array1D<double>& m)` : Evaluates the log-posterior based on the 1d array of doubles passed into the function
- `bool inDomain(Array1D<double>& m)` : Checks if all of the points in the 1d array are in the defined domain of the MCMC and returns the evaluation as a bool
- `void writeChainTxt(string filename)` : Writes the full chain as a text file with the name of the string passed into the function
- `void writeChainBin(string filename)` : Writes the full chain as a binary file with the name of the string passed into the function
- `void setNewMode(bool value)` : Sets the new mode value to the boolean value passed into the function

4.1.2. `amcmc`:

This directory features the functionality and variables for Adaptive Markov Chain Monte Carlo (AMCMC) in UQTk. AMCMC is the most common version of MCMC in UQTk. The functions within this class are:

- `AMCMC(double (*logposterior)(Array1D<double>&, void *), void *postinfo)` : Constructor for AMCMC that takes in a pointer to a log posterior function and an additional pointer to information about the posterior. This constructor delegates to the similar constructor in the MCMC base class
- `AMCMC(LogPosteriorBase& L)` : Constructor that takes in a LogPosteriorBase object. This constructor delegates to the similar constructor in the MCMC base class
- `void initAdaptSteps(int adaptstart,int adaptstep, int adaptend)` : Initializes the adaptivity step parameters for AMCMC. The start of adaptivity, how often the MCMC adapts, and when the adaptivity ends are initialized as integers.
- `void initAMGamma(double gamma_)` : Initializes the scaling factor of gamma for AMCMC as a double
- `void initEpsCov(double eps_cov_)` : Initializes the covariance nugget for AMCMC as a double
- `void getAdaptSteps(Array1D<int> adaptstep_)` : Gets the adaptivity step parameters for AMCMC by setting the passed in 1d array of 3 integers equal to the parameters. The first element is the start of adaptivity. The second element is the step size for adaptivity, or how often the AMCMC adapts. The third element is the end of the adaptivity of the AMCMC.
- `double getGamma()` : Gets the coefficient behind the covariance scaling factor for AMCMC as a double. This is also known as the gamma value

- `double getEpsCov()` : Gets the offset epsilon for Cholesky to be computationally feasible as a double. This is also known as the covariance nugget
- `void printChainSetup()` : Prints the chain information on the screen
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to AMCMC. It is written as a virtual function to allow for any additional derived classes that would be based off of AMCMC
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain based on a trivial initial condition. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to AMCMC. It is written as a virtual function to allow for any additional derived classes that would be based off of AMCMC

4.1.3. `tmcmc`:

This directory features the functionality and variables for Transitional Markov Chain Monte Carlo (TMCMC) in UQTk. The functions within this class are:

- `TMCMC()` : Constructor for TMCMC that takes in no values. This constructor delegates to the similar dummy constructor in the MCMC base class
- `void initDefaults()` : Sets the default values for the TMCMC object
- `void initTMCMCNprocs(int tmcmc_nprocs)` : Initializes the number of processes for TMCMC as an integer
- `void initTMCMCGamma(double tmcmc_gamma)` : Initializes the coefficient behind the covariance scaling factor for TMCMC as a double
- `void initTMCMCCv(double tmcmc_cv)` : Initializes the maximum allowed coefficient of variation for the weights in TMCMC as a double
- `void initTMCMCMFactor(int tmcmc_MFactor)` : Initializes the the multiplicative factor for chain length to encourage mixing in TMCMC as an integer
- `void initTMCMCBasis(bool tmcmc_basis)` : Initializes the choice to resample according to BASIS and CATMIPs in TMCMC as a bool
- `void initTMCMCCATSteps(int tmcmc_CATSteps)` : Initialize the CATMIPs resampling parameter for TMCMC as an integer
- `int getTMCMCNprocs()` : Gets the number of processes for TMCMC as an integer
- `double getTMCMCGamma()` : Gets the coefficient behind the covariance scaling factor for TMCMC as a double
- `double getTMCMCCv()` : Gets the maximum allowed coefficient of variation for the weights in TMCMC as a double

- `int getTMCMCMFactor()` : Gets the multiplicative factor for chain length to encourage mixing in TMCMC as an integer
- `bool getTMCMCBasis()` : Gets the choice to resample according to BASIS and CATMIPs in TMCMC as a bool
- `int getTMCMCCATSteps()` : Gets the CATMIPs resampling parameter for TMCMC as an integer
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to TMCMC. It is written as a virtual function to allow for any additional derived classes that would be based off of AMCMC
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain based on a trivial initial condition. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to TMCMC. It is written as a virtual function to allow for any additional derived classes that would be based off of TMCMC

4.1.4. `ss`:

This directory features the functionality and variables for Single Site Markov Chain Monte Carlo (SS) in UQTk. SS is the most basic and simplest of the types of MCMC in UQTk. The functions within this class are:

- `SS(double (*logposterior)(Array1D<double>&, void *), void *postinfo)` : Constructor for SS that takes in a pointer to a log posterior function and an additional pointer to information about the posterior. This constructor delegates to the similar constructor in the MCMC base class
- `SS(LogPosteriorBase& L)` : Constructor that takes in a LogPosteriorBase object. This constructor delegates to the similar constructor in the MCMC base class
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to SS. It is written as a virtual function to allow for any additional derived classes that would be based off of SS
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain based on a trivial initial condition. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to SS. It is written as a virtual function to allow for any additional derived classes that would be based off of SS
- `int getNSubSteps() override` : Gets the number of sub steps for an SS object. This overrides the virtual function previous defined in the MCMC base class

4.1.5. `mala:`

This directory features the functionality and variables for Metropolis-adjusted Langevin algorithm (MALA) in UQTk. The functions within this class are:

- `MALA(double (*logposterior)(Array1D<double>&, void *), void *postinfo)` : Constructor for MALA that takes in a pointer to a log posterior function and an additional pointer to information about the posterior. This constructor delegates to the similar constructor in the MCMC base class
- `MALA(LogPosteriorBase& L)` : Constructor that takes in a LogPosteriorBase object. This constructor delegates to the similar constructor in the MCMC base class
- `void initEpsMALA(double eps_mala_)` : Initializes the epsilon for MALA as double
- `void setGradient(void (*gradlogPosterior)(Array1D<double>&, Array1D<double>&, void *))` : Sets the gradient function given a pointer to a gradient of a logPosterior function
- `double getEpsMALA()` : Gets the epsilon for MALA as a double
- `void getGradient(void (*gradlogPosterior)(Array1D<double>&, Array1D<double>&, void *))` : Gets gradient function by passing in a pointer
- `bool getGradientFlag()` : Gets if the gradient function is set as a bool
- `void evalGradLogPosterior(Array1D<double>& m, Array1D<double>& grads)` : Evaluates the gradient function based on two 1d arrays of doubles
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to MALA. It is written as a virtual function to allow for any additional derived classes that would be based off of MALA
- `virtual void runChain(int ncalls, Array1D<double>& chstart) override` : Generates the MCMC chain based on a trivial initial condition. This function overrides the pure virtual function in the base class of the MCMC. It generates the MCMC in the manner specific to MALA. It is written as a virtual function to allow for any additional derived classes that would be based off of MALA
- `virtual void runOptim(Array1D<double>& start) override` : Runs the optimization routine for the MCMC object. It generates the MCMC in the manner specific to MALA. It is written as a virtual function to allow for any additional derived classes that would be based off of MALA

4.1.6. `mmala:`

This directory features the functionality and variables for Manifold variant of Metropolis-adjusted Langevin algorithm (MALA) in UQTk. MMALA is a derived class of the MALA class. Thus making the MALA class a base class for MMALA. The functions within this class are:

- `MMALA(double (*logposterior)(Array1D<double>&, void *), void *postinfo)` : Constructor for MMALA that takes in a pointer to a log posterior function and an additional pointer to information about the posterior. This constructor delegates to the similar constructor in the MALA base class
- `MMALA(LogPosteriorBase& L)` : Constructor that takes in a LogPosteriorBase object. This constructor delegates to the similar constructor in the MALA base class
- `void setMetricTensor(void (*metricTensor)(Array1D<double>&, Array2D<double>&, void *))` : Sets the metric tensor function used in MMALA
- `void getMetricTensor(void (*metricTensor)(Array1D<double>&, Array2D<double>&, void *))` : Gets the metric tensor function used in MMALA

4.2. C++ APPLICATIONS

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

- ◎ `dfi` : Data-free inference
- ◎ `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`.

4.2.1. `dfi:`

This app implements a simple data-free inference (DFI) approach to match the pushforward posterior of a model to a given set of summary statistics. In essence, the code performs Bayesian inference with an *Approximate Bayesian Computation* (ABC) log-likelihood given by

$$\log \mathcal{L}_{\boldsymbol{\alpha}}(\boldsymbol{\nu}) \coloneqq -\frac{1}{2}\sum_{d=1}^{D}\alpha_d \sum_{n=1}^{N_d}\left[\log(2\pi\beta_d {s_d^{(n)}}^2) + \frac{1}{K_d\beta_d {s_d^{(n)}}^2}\sum_{k=1}^{K_d}\left(z_d^{(n,k)} - f_d(\boldsymbol{x}_d^{(n)},\boldsymbol{\nu})\right)^2\right]. \quad (4.1)$$

We will outline the different components of (4.1) below.

Consider a set of D data sets $\mathcal{D}_d$, $d = 1, 2, \ldots, D$, with N_d measurement locations each. At each measurement location $\boldsymbol{x}_d^{(n)}$, we have a measurement value $y_d^{(n)}$ as well as an associated uncertainty $s_d^{(n)}$, i.e.,

$$\mathcal{D}_d = \{\boldsymbol{x}_d^{(n)}, y_d^{(n)}, s_d^{(n)}\}_{n=1}^{N_d}, \quad d = 1, 2, \ldots, D, \quad (4.2)$$

and $\mathcal{D} = \{\mathcal{D}_d\}_{d=1}^{D}$. For example, the data could correspond to measurements of a certain physical quantity under different operating conditions. In this case, the measurement locations $\boldsymbol{x}_d^{(n)}$ could be, for example, a set of temperatures, and the data sets $\mathcal{D}_d$, $d = 1, 2, \ldots, D$ could be the measurements under different operating conditions.

Our goal is to calibrate an assumed model $f_d(\boldsymbol{x}, \boldsymbol{\nu})$, $d = 1, 2, \ldots, D$, with respect to the model parameters $\boldsymbol{\nu}$. We assume that we only have summary statistics of the data available, a setting which is different from the classic Bayesian calibration setting with noisy measurements. In particular, we assume the reported error bars reflect uncertainty in the model parameters, and our goal is to match the model predictions with *both* the reported measurement value and the measurement error.

A typical DFI procedure would then compute a joint posterior density on both the data and model parameters simultaneously, where consistency between the reported summary statistics and the statistics of the posterior is enforced using a *maximum entropy* principle. The algorithm entails a nested sampling procedure, with an outer loop evolving over the data space, and an inner loop evolving in the parameter space. At each iteration of the outer loop, a new data set is proposed, after which the inner loop is executed in order to compute the proposed parameter posterior. Samples of the proposed posterior are then used to check for consistency of the proposed data set with the reported summary statistics. Each accepted data set provides a consistent posterior on the model parameters. The final *pooled posterior* is ultimately obtained by combining the consistent data sets together.

However, the classic DFI algorithm can be computationally demanding. We use a more flexible approach, in which a consistent data set is defined as a data set for which the statistics, computed from the data is, up to a tolerance, equal to the reported summary statistics.

To this end, we define a collection of K_d synthetic data sets $\mathcal{Z}_d^{(k)} \coloneqq \{z_d^{(n,k)}\}_{n=1}^{N_d}$, $k = 1, 2, \ldots, K_d$, for each experimental data set $\mathcal{D}_d$, where

$$z_d^{(n,k)} \sim \mathcal{N}\left(y_d^{(n)}, \beta_d {s_d^{(n)}}^2\right), \tag{4.3}$$

with $y_d^{(n)}$ and $s_d^{(n)}$ the measurement value and measurement error respectively, see (4.2), and where $\beta_d > 0$ is a scale factor for the variance of the synthetic data set. Hence, the data set $\mathcal{Z}_d^{(k)}$ contains synthetic observations that are sampled from a multivariate Gaussian distribution centered at the measurement mean $y_d^{(n)}$ for each $n = 1, 2, \ldots, N_d$, and with a variance that can be tuned by choosing appropriate values for β_d.

Each synthetic data set $\mathcal{Z}_d^{(k)}$, $k = 1, 2, \ldots, K_d$, represents an opinion about the posterior, denoted by $p(\boldsymbol{\nu}|\mathcal{Z}_d^{(k)})$. This posterior is obtained by setting up an inference problem with log likelihood

$$\log \mathcal{L}_d^{(k)}(\boldsymbol{\nu}) \coloneqq -\frac{1}{2}\sum_{n=1}^{N_d}\left[\log(2\pi\beta_d {s_d^{(n)}}^2) + \frac{\left(z_d^{(n,k)} - f_d(\boldsymbol{x}_d^{(n)}, \boldsymbol{\nu})\right)^2}{\beta_d {s_d^{(n)}}^2}\right], \tag{4.4}$$

where we used the assumption that the synthetic data is generated from a multivariate Gaussian distribution.

The K_d different opinions can be combined using *logarithmic pooling*. This can be accomplished by gathering all synthetic data set $\mathcal{Z}_d^{(k)}$ into a single, large data set $\mathcal{Z}_d \coloneqq \{\mathcal{Z}_d^{(k)}\}_{k=1}^{K_d}$, and setting up an inference problem that uses a *log-pooled* log likelihood

$$\log \mathcal{L}_d(\boldsymbol{\nu}) \coloneqq -\frac{1}{K_d}\sum_{k=1}^{K_d} \log \mathcal{L}_d^{(k)}(\boldsymbol{\nu}). \tag{4.5}$$

Once we obtain the posterior density $p(\boldsymbol{\nu}|\mathcal{Z}_d)$ on the model parameters, samples from the posterior can be propagated through the assumed forward model $f_d(\boldsymbol{x}, \boldsymbol{\nu})$, in order to obtain samples from the *pushforward posterior*. From the pushforward posterior density, we can extract a set of statistics $\tilde{\boldsymbol{s}}_d \coloneqq \{\tilde{s}_d^{(n)}\}_{n=1}^{N_d}$, such as the standard deviation at each measurement location. These computed statistics can then be compared to the reported summary statistics $\boldsymbol{s}_d \coloneqq \{s_d^{(n)}\}_{n=1}^{N_d}$ to decide whether the proposed data set $\mathcal{Z}_d$ is consistent. For example, we may define a consistent data set $\mathcal{Z}_d$ to be a data set that satisfies

$$\|\boldsymbol{s}_d - \tilde{\boldsymbol{s}}_d\|_p \leq \varepsilon \tag{4.6}$$

for a given p-norm and given tolerance $\varepsilon > 0$. Equation (4.6) may be satisfied by choosing an appropriate value for the scale factor β_d in (4.3).

Once we have obtained consistent synthetic data sets $\mathcal{Z}_d$ for each experimental data set $d = 1, 2, \ldots, D$, we combine the different consistent data sets in a single data set $\mathcal{Z} = \{\mathcal{Z}_d\}_{d=1}^{D}$ and set up a final inference problem with log likelihood

$$\log \mathcal{L}(\boldsymbol{\nu}) \coloneqq \sum_{d=1}^{D} \alpha_d \log \mathcal{L}_d(\boldsymbol{\nu}). \tag{4.7}$$

In this expression, the weights $\{\alpha_d\}_{d=1}^D$ can be used to express our confidence in the respective experimental data sets. We may use these weights, for example, to account for the different number of measurement locations N_d in each data set. In that case, the weights could be chosen as

$$\alpha_d := \frac{DN_d^{-1}}{\sum_{d=1}^D N_d^{-1}}. \tag{4.8}$$

Combining equations (4.4), (4.5) and (4.7), we obtain the likelihood shown in (4.1).

Below, we detail the necessary input files, as well as the most important options of the program. See also the "help" message (type `dfi -h`) for additional details.

Input files:

- `-d <data_file>` size $N_d \times 2$ default: `data.{d}.dat`
  Name of the file that contains the experimental data. This file should contain two columns, a first column with the measurement values, and a second column with the associated uncertainty. Multiple numbered data files can be provided by using "`{d}`" in the filename. UQTk will expand these tokens by consecutive data set numbers starting from 1.

- `-c <pccf_file>` size $P_d^{(n)} \times 1$ default: `pccf.{d}.{n}.dat`
  Name of the files that contain the set of PCE coefficients. Specify multiple PCEs at different measurement stations by using "`{n}`" in the filename. The code will expand these tokens by consecutive PCE numbers starting from 1. Multiple sets of PCEs for the quantities predicted in each data set can be provided by using "`{d}`" in the filename.

- `-i <mindex_file>` size $P_d^{(n)} \times s$ default: `mindex.{d}.{n}.dat`
  Name of the files that contain the set of PCE multi-indices. Every row contains the (integer) order of the basis polynomial in every dimension. Specify multiple PCEs at different measurement stations by using "`{n}`" in the filename. Multiple sets of PCEs for the quantities predicted in each data set can be provided by using "`{d}`" in the filename.
  Currently, only Legendre–uniform PCEs (`LU`) are supported.

- `-p <prior_file>` size $s \times 3$ default: `prior.dat`
  Name of the file that contains the prior specifications. Every line of this file corresponds to the prior specification of a single parameter. Uniform and Gaussian priors are supported. Specify uniform priors as "`uniform a b`" where "`a`" is the lower bound and "`b`" is the upper bound of the distribution for this parameter. Specify Gaussian priors as "`gaussian mu sigma`" where "`mu`" is the mean and "`sigma`" is the standard deviation of the distribution for this parameter.

- `-n <nb_of_mcmc>` integer default: 100,000
  Number of MCMC iterations.

- `-k <nb_of_synth>` integer or list default: 100
  Number of synthetic data sets. Specify a different number of synthetic data sets for each experimental data set using a comma-separated list.

- `-b <scaling_factor>` integer or list default: 1
  Scaling factor for the standard deviation used to generate the synthetic data sets. Specify a different scaling factor for each experimental data set using a comma-separated list.

- `-j <weights>` list or '`data_size`' default: 1
  Weights used in the likelihood formulation. When this key is not specified, we assume the weights are equal to 1 for each data set. With the special option '`data_size`', we use weights that compensate for the number of measurements in each data set.

There are other options to set a random seed (`-r`), change the MCMC proposal jump size (`-g`), prepend the MCMC with optimization to compute good initial conditions (`-o`) or specify a custom initial condition (`-t`), set the output frequency of the chain (`-u`) and use custom synthetic data sets (`-s`).

For convenience, we also provide the following options:

- `-z`
  Compute the standard deviation of the pushforward posteriors for each data set and each measurement station on the fly when this key is specified. These values can be used as a possible statistic to determine consistency of the synthetic data sets with the reported measurement errors. Use the optional keys `-m` and `-e` to specify the burnin and subsampling rate of the samples used to evaluate the standard deviations.

- `-v <pccf_file>` size $R_d^{(n)} \times 1$ default: `pushforward.pccf.{d}.{n}.dat`
  `-w <mindex_file>` size $R_d^{(n)} \times s$ default: `pushforward.mindex.{d}.{n}.dat`
  Specify a set of PCEs where the posterior samples should be evaluated. These PCEs are potentially different from the PCEs at each measurement station where data is available. Use the optional keys `-m` and `-e` to specify the burnin and subsampling rate of the samples used to evaluate the pushforward.

- `-q`
  Compute the expectation of the Fisher information matrix over the posterior when specified. This is a metric useful when constructing a likelihood-informed subspace. Use the optional keys `-m` and `-e` to specify the burnin and subsampling rate of the samples used to evaluate the metric. **Warning**: this option makes the MCMC sampling very slow!

A set of example notebooks illustrating the capabilities of `dfi` using a simple quadratic model problem is located in `examples/dfi_app/`.

4.2.2. `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.

4.2.3. `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 $N_{p_0,\ldots,p_{k_{max}}}^{HDMR} = \sum_{k=0}^{k_{max}}(p_k + k)!/(p_k!k!)$.

4.2.4. `gp_regr:`

This utility performs Gaussian process regression [41], in particular using the Bayesian perspective of constructing GP emulators, see e.g. [25, 37]. 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

$$\begin{aligned} \hat{\boldsymbol{c}} &= V^*(V^{-1}\boldsymbol{c}_0 + H^T A^{-1} \boldsymbol{y}) & \hat{\sigma}^2 &= \frac{2\beta + \boldsymbol{c}_0^T V^{-1} \boldsymbol{c}_0 + \boldsymbol{y}^T A^{-1} \boldsymbol{y} - \hat{\boldsymbol{c}}^T (V^*)^{-1} \hat{\boldsymbol{c}}}{N + 2\alpha - K - 2} \\ \boldsymbol{t}(x)^T &= (c(x, x^{(1)}), \ldots, c(x, x^{(N)})) & V^* &= (V^{-1} + H^T A^{-1} H)^{-1} \\ H &= (\boldsymbol{h}(x^{(1)})^T, \ldots, \boldsymbol{h}(x^{(N)})^T) & A_{mn} &= c(x^{(m)}, x^{(n)}) \end{aligned} \tag{4.9}$$

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 $c_0 = \boldsymbol{0}$, $V^{-1} = 0_{K \times K}$. 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 [37]. 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.

4.2.5. `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

$$u(x) \approx \tilde{u}(x) = \sum_{j=0}^{p} v_j \phi_j(x), \tag{4.10}$$

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

$$u(\boldsymbol{x}) \approx \tilde{u}(\boldsymbol{x}) = \sum_{j_1=0}^{p_1} \cdots \sum_{j_m=0}^{p_m} v_{j_1,\ldots,j_m} \phi_{j_1}^{(1)}(x_1) \cdots \phi_{j_m}^{(m)}(x_m), \tag{4.11}$$

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

$$u(\boldsymbol{x}) \approx \tilde{u}(\boldsymbol{x}) = \sum_{k=1}^{r} \prod_{i=1}^{m} w_k^{(i)}(x_i), \tag{4.12}$$

with each univariate function $w_k^{(i)}(x_i)$ being represented, in analogy to Eq. (4.10), as

$$w_k^{(i)}(x_i) = \sum_{j_i=0}^{p_i} w_{k,j_i}^{(i)} \phi_{j_i}^{(i)}(x_i). \tag{4.13}$$

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

$$u(\boldsymbol{x}) \approx \sum_{k=1}^{r} \prod_{i=1}^{m} \left\{ \sum_{j_i=0}^{p_i} w_{k,j_i}^{(i)} \phi_{j_i}^{(i)}(x_i) \right\}. \tag{4.14}$$

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. (4.10). 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

$$\boldsymbol{\Phi} = \begin{bmatrix} \phi_0(\mathrm{x}^1) & \ldots & \phi_p(\mathrm{x}^1) \\ \vdots & \ddots & \vdots \\ \phi_0(\mathrm{x}^S) & \ldots & \phi_p(\mathrm{x}^S) \end{bmatrix}, \tag{4.15}$$

we can express Eq.(4.10) on the sample points as

$$\mathbf{u} \approx \mathbf{\Phi}\mathbf{v}, \tag{4.16}$$

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,

$$\|\mathbf{u} - \mathbf{\Phi}\mathbf{v}\|_2^2, \tag{4.17}$$

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

$$\min_{\mathbf{v}} \|\mathbf{u} - \mathbf{\Phi}\mathbf{v}\|_2^2, \tag{4.18}$$

which has a closed-form solution,

$$\mathbf{v} = \left(\mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T\mathbf{u}, \tag{4.19}$$

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,

$$\min_{\{w\}} \|u - \tilde{u}\|_2^2, \tag{4.20}$$

where $\tilde{u}$ is written as Eq. (4.14). 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

$$\mathbf{z}^{(i)} = \begin{bmatrix} \mathbf{w}_1^{(i)} \\ \vdots \\ \mathbf{w}_r^{(i)} \end{bmatrix}, \tag{4.21}$$

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

$$\mathbf{\Phi}^{(i)} = \left[\mathbf{\Phi}_1^{(i)} \cdots \mathbf{\Phi}_r^{(i)}\right], \tag{4.22}$$

with

$$\mathbf{\Phi}_k^{(i)} = \begin{bmatrix} c_{k,1}^{(i)}\phi_0^{(i)}(\mathrm{x}_i^1) & \cdots & c_{k,1}^{(i)}\phi_{p_i}^{(i)}(\mathrm{x}_i^1) \\ \vdots & \ddots & \vdots \\ c_{k,S}^{(i)}\phi_0^{(i)}(\mathrm{x}_i^S) & \cdots & c_{k,S}^{(i)}\phi_{p_i}^{(i)}(\mathrm{x}_i^S) \end{bmatrix}, \tag{4.23}$$

where

$$c_{k,s}^{(i)} = \prod_{l=1,l\neq i}^{m} w_k^{(l)}(\mathrm{x}_l^s) \tag{4.24}$$

$$= \prod_{l=1,l\neq i}^{m} \left[\phi_0^{(l)}(\mathrm{x}_l^s)\ldots\phi_{p_l}^{(l)}(\mathrm{x}_l^s)\right]\cdot\mathbf{w}_k^{(l)}, \tag{4.25}$$

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

According to Eq. (4.19), we find

$$\mathbf{z}^{(i)} = \left(\mathbf{\Phi}^{(i)^T}\mathbf{\Phi}^{(i)}\right)^{-1}\mathbf{\Phi}^{(i)^T}\mathbf{u}. \tag{4.26}$$

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. (4.12))
- `-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. (4.13)). 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 run time. Do not use it if you want only the bare minimum.

4.2.6. `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:

$$f(\boldsymbol{\lambda}; \boldsymbol{x}) \approx g(\boldsymbol{x}). \tag{4.27}$$

The full error budget takes the following form

$$y^{(i)} = f(\boldsymbol{\lambda}; x^{(i)}) + \delta(x^{(i)}) + \epsilon_i, \tag{4.28}$$

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

$$\epsilon_i \sim N(0, \sigma^2), \text{ for all } i = 1, \ldots, L. \tag{4.29}$$

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. (4.37).
- 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 [31], 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 [47]. 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

$$\boldsymbol{\lambda} \to \boldsymbol{\Lambda} = \boldsymbol{\lambda} + A(\boldsymbol{\alpha})\vec{\xi}, \tag{4.30}$$

where

$$A(\boldsymbol{\alpha}) = \begin{bmatrix} \alpha_{11} & 0 & 0 & \dots & 0 \\ \alpha_{21} & \alpha_{22} & 0 & \dots & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{M1} & \alpha_{M2} & \alpha_{M3} & \dots & \alpha_{MM} \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}_{D\times M}, \text{ and } \vec{\xi} = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_M \end{bmatrix} \tag{4.31}$$

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

$$y^{(i)} = f(\boldsymbol{\lambda} + A(\boldsymbol{\alpha})\vec{\xi}; x^{(i)}) + \epsilon_i \tag{4.32}$$

or

$$y^{(i)} = f_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}; \vec{\xi}) + \sigma^2 \xi_{M+i}, \tag{4.33}$$

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,

$$y^{(i)} = \sum_{k=0}^{K-1} f_{ik}(\boldsymbol{\lambda}, \boldsymbol{\alpha})\Psi_k(\vec{\xi}) + \sigma^2 \xi_{M+i}, \tag{4.34}$$

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

$$\underbrace{p(\hat{\boldsymbol{\lambda}}|\mathcal{D})}_{\text{posterior}} \propto \underbrace{p(\mathcal{D}|\hat{\boldsymbol{\lambda}})}_{\text{likelihood}} \underbrace{p(\hat{\boldsymbol{\lambda}})}_{\text{prior}}, \tag{4.35}$$

where the key function is the likelihood function

$$\mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = p(\mathcal{D}|\hat{\boldsymbol{\lambda}}) \tag{4.36}$$

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 [47]. To

alleviate the invariance with respect to sign-flips, we use a prior that enforces $\alpha_{Mi} > 0$ for $i = 1, \ldots, M$. Also, one can either fix σ^2 or infer it together with $\hat{\boldsymbol{\lambda}}$.

Exact computation of the potentially high-dimensional posterior (4.35) 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' run 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

$$\log \mathcal{L}_{\mathcal{D}}(\boldsymbol{\lambda}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - f(\boldsymbol{\lambda}; \boldsymbol{x}^{(i)}))^2}{2\sigma^2} - \frac{L}{2} \log (2\pi\sigma^2), \tag{4.37}$$

- `koh` : Kennedy-O'Hagan likelihood with explicit additive representation of model discrepancy [31].

- `full` : This is the exact likelihood

$$\mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = \pi_{\boldsymbol{h}_{\hat{\boldsymbol{\lambda}}}}(y^{(1)}, \ldots, y^{(L)}), \tag{4.38}$$

  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 [47]. 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

$$\mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = \prod_{i=1}^{L} \pi_{\boldsymbol{h}_{\hat{\boldsymbol{\lambda}},i}}(y^{(i)}), \tag{4.39}$$

  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

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu}_{\hat{\boldsymbol{\lambda}}})^T \Sigma_{\hat{\boldsymbol{\lambda}}}^{-1} (\boldsymbol{y} - \boldsymbol{\mu}_{\hat{\boldsymbol{\lambda}}}) - \frac{L}{2} \log (2\pi) - \frac{1}{2} \log (\det \Sigma_{\hat{\boldsymbol{\lambda}}}), \tag{4.40}$$

  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

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))^2}{2\left(\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}) + \sigma^2\right)} - \frac{1}{2} \sum_{i=1}^{L} \log 2\pi \left(\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}) + \sigma^2\right). \tag{4.41}$$

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

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))^2}{2\epsilon^2} - \frac{1}{2}\log(2\pi\epsilon^2), \tag{4.42}$$

- `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

$$\log \mathcal{L}_{\mathcal{D}}(\hat{\boldsymbol{\lambda}}) = -\sum_{i=1}^{L} \frac{(y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}))^2 + \left(\gamma|y^{(i)} - \mu_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)})| - \sqrt{\sigma^2_{\hat{\boldsymbol{\lambda}}}(\boldsymbol{x}^{(i)}) + \sigma^2}\right)^2}{2\epsilon^2} - \frac{1}{2}\log(2\pi\epsilon^2), \tag{4.43}$$

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

4.2.7. `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$\rightarrow$LU is a map from standard normal random variable to a uniform random variable in $[-1, 1]$.

4.2.8. `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

$$\vec{X} \simeq \sum_{k=0}^{K} \boldsymbol{c}_k \Psi_k(\vec{\xi}), \tag{4.44}$$

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

$$\boldsymbol{c}_k = \frac{\langle \vec{X} \Psi_k(\vec{\xi}) \rangle}{\langle \Psi_k^2(\vec{\xi}) \rangle} = \frac{\langle \vec{G}(\vec{\xi}) \Psi_k(\vec{\xi}) \rangle}{\langle \Psi_k^2(\vec{\xi}) \rangle}. \tag{4.45}$$

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),

$$\begin{aligned} \eta_1 &= 2F_1(X_1) - 1 \\ \eta_2 &= 2F_{2|1}(X_2|X_1) - 1 \\ \eta_3 &= 2F_{3|2,1}(X_3|X_2, X_1) - 1 \\ &\vdots \\ \eta_d &= 2F_{d|d-1,\ldots,1}(X_d|X_{d-1}, \ldots, X_1) - 1. \end{aligned} \tag{4.46}$$

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

$$\vec{R}(\vec{X}) = \vec{H}(\vec{\xi}), \qquad \text{or} \qquad \vec{X} = \underbrace{\vec{R}^{-1} \circ \vec{H}}_{\vec{G}}(\vec{\xi}) \tag{4.47}$$

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 $\boldsymbol{x}^{(i)}$:

$$p_{\vec{X}}(\boldsymbol{x}) = \frac{1}{N\sigma^d(2\pi)^{d/2}} \sum_{i=1}^{N} \exp\left(-\frac{(\boldsymbol{x} - \boldsymbol{x}^{(i)})^T(\boldsymbol{x} - \boldsymbol{x}^{(i)})}{2\sigma^2}\right) \tag{4.48}$$

or

$$p_{\vec{X}_1,\ldots,\vec{X}_d}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d) = \frac{1}{N\sigma^d(2\pi)^{d/2}} \sum_{i=1}^{N} \exp\left(-\frac{(\boldsymbol{x}_1 - \boldsymbol{x}_1^{(i)})^2 + \cdots + (\boldsymbol{x}_d - \boldsymbol{x}_d^{(i)})^2}{2\sigma^2}\right), \tag{4.49}$$

where the *bandwidth* σ should be chosen to balance smoothness and accuracy, see [50, 51] 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

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

$$\begin{aligned} &= \int_{-\infty}^{\boldsymbol{x}_k} \frac{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)\times\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(\boldsymbol{x}_k^{'}-\boldsymbol{x}_k^{(i)})^2}{2\sigma^2}\right)}{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)}d\boldsymbol{x}_k^{'} \\ &= \frac{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)\times\Phi\left(\frac{\boldsymbol{x}_k-\boldsymbol{x}_k^{(i)}}{\sigma}\right)}{\sum_{i=1}^{N}\exp\left(-\frac{(\boldsymbol{x}_1-\boldsymbol{x}_1^{(i)})^2+\cdots+(\boldsymbol{x}_{k-1}-\boldsymbol{x}_{k-1}^{(i)})^2}{2\sigma^2}\right)}, \end{aligned} \tag{4.50}$$

where $\Phi(z)$ is the CDF of a standard normal random variable. Note that the numerator in (4.50) 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 (4.47) 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 (4.47) available, we can proceed to evaluation the numerator of the orthogonal projection (4.45)

$$\langle\vec{G}(\vec{\xi})\Psi_k(\vec{\xi})\rangle = \int_{\vec{\xi}}\vec{G}(\boldsymbol{x})\Psi_k(\boldsymbol{x})\pi_{\vec{\xi}}(\vec{\xi})d\vec{\xi}, \tag{4.51}$$

where $\pi_{\vec{\xi}}(\vec{\xi})$ is the PDF of $\vec{\xi}$. The projection integral (4.51) is computed via quadrature integration

$$\int_{\vec{\xi}}\vec{G}(\vec{\xi})\Psi_k(\vec{\xi})\pi_{\vec{\xi}}(\vec{\xi})d\vec{\xi} \approx \sum_{q=1}^{Q}\vec{G}(\vec{\xi}_q)\Psi_k(\vec{\xi}_q)w_q = \sum_{q=1}^{Q}\vec{R}^{-1}(\vec{H}(\vec{\xi}_q))\Psi_k(\vec{\xi}_q)w_q, \tag{4.52}$$

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

4.2.9. `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

$$f(\vec{\xi}) \approx \sum_{||\boldsymbol{\alpha}||_1 \leq p} c_{\boldsymbol{\alpha}} \Psi_{\boldsymbol{\alpha}}(\vec{\xi}) \equiv g(\vec{\xi}). \tag{4.53}$$

The orthogonal projection computed by this utility is

$$c_{\boldsymbol{\alpha}} = \frac{1}{\langle \Psi^2_{\boldsymbol{\alpha}} \rangle} \int_{\vec{\xi}} f(\vec{\xi}) \Psi_{\boldsymbol{\alpha}}(\vec{\xi}) \pi_{\vec{\xi}}(\vec{\xi}) d\vec{\xi} \approx \frac{1}{\langle \Psi^2_{\boldsymbol{\alpha}} \rangle} \sum_{q=1}^{Q} w_q f(\vec{\xi}^{(q)}) \Psi_{\boldsymbol{\alpha}}(\vec{\xi}^{(q)}). \tag{4.54}$$

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

$$\frac{||f-g||_2}{||f||_2} \approx \sqrt{\frac{\sum_{q=1}^{Q} w_q (f(\vec{\xi}^{(q)}) - g(\vec{\xi}^{(q)}))^2}{\sum_{q=1}^{Q} w_q f(\vec{\xi}^{(q)})^2}}. \tag{4.55}$$

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.

4.2.10. `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.

4.2.11. `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.

4.2.12. `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.

4.2.13. `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:

$$y^{(i)} \approx \sum_{k=1}^{K} c_k P_k(x^{(i)}), \tag{4.56}$$

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. [38]

Regression methods: `-f <meth>`

- `lsq` : Bayesian least-squares, see [46] and more details below.
- `wbcs` : Weighted Bayesian compressive sensing, see [49].

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

$$\arg\min_{\boldsymbol{c}} \underbrace{||\boldsymbol{y} - \boldsymbol{P}\boldsymbol{c}||^2 + ||\boldsymbol{\Lambda}\boldsymbol{c}||^2}_{R(\boldsymbol{c})} \tag{4.57}$$

with a closed form solution

$$\hat{\boldsymbol{c}} = \underbrace{(\boldsymbol{P}^T\boldsymbol{P} + \boldsymbol{\Lambda})^{-1}}_{\boldsymbol{\Sigma}}\boldsymbol{P}^T\boldsymbol{y} \tag{4.58}$$

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

The Bayesian analog of this, detailed in [46], infers coefficient vector $\boldsymbol{c}$ and data noise variance σ^2, given data $\boldsymbol{y}$, employing Bayes' formula

$$\overbrace{p(\boldsymbol{c}, \sigma^2|\boldsymbol{y})}^{\text{Posterior}} \propto \overbrace{p(\boldsymbol{y}|\boldsymbol{c}, \sigma^2)}^{\text{Likelihood}}\overbrace{p(\boldsymbol{c}, \sigma^2)}^{\text{Prior}} \tag{4.59}$$

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,

$$p(\boldsymbol{y}|\boldsymbol{c}, \sigma^2) \equiv L_{\boldsymbol{c},\sigma^2}(\boldsymbol{y}) = (2\pi\sigma^2)^{-\frac{N}{2}} \exp\left(-\frac{1}{2\sigma^2}||\boldsymbol{y} - \boldsymbol{P}\boldsymbol{c}||^2\right) \tag{4.60}$$

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:

$$p(\boldsymbol{c}, \sigma^2) = \underbrace{\left(\prod_{k=1}^{K}\frac{\lambda_k}{2\pi}\right)^{\frac{1}{2}} \exp\left(-\frac{1}{2}||\boldsymbol{\Lambda}\boldsymbol{c}||^2\right)}_{p(\boldsymbol{c})} \underbrace{(\sigma^2)^{-\alpha-1}\exp\left(-\frac{\beta}{\sigma^2}\right)}_{p(\sigma^2)} \tag{4.61}$$

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

$$p(\boldsymbol{c}|\sigma^2, \boldsymbol{y}) \sim MVN(\hat{\boldsymbol{c}}, \sigma^2\boldsymbol{\Sigma}), \tag{4.62}$$

$$p(\sigma^2|\boldsymbol{y}) \sim IG\left(\underbrace{\alpha + \frac{N-K}{2}}_{\alpha^*}, \underbrace{\beta + \frac{R(\hat{\boldsymbol{c}})}{2}}_{\beta^*}\right) \tag{4.63}$$

where $\hat{\boldsymbol{c}}$ and $\boldsymbol{\Sigma}$, as well as the residual $R(\cdot)$ are defined via the classical least-squares problem (4.57) and (4.58). 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 $R(\hat{\boldsymbol{c}}) = \boldsymbol{y}^T \left(\boldsymbol{I}_N - \boldsymbol{P}\left(\boldsymbol{P}^T\boldsymbol{P} + \boldsymbol{\Lambda}\right)^{-1}\boldsymbol{P}^T\right)\boldsymbol{y}$. One can integrate out σ^2 from (4.61) to arrive at a multivariate t-distribution

$$p(\boldsymbol{c}|\boldsymbol{y}) \sim MVT\left(\hat{\boldsymbol{c}}, \frac{\beta^*}{\alpha^*}\boldsymbol{\Sigma}, 2\alpha^*\right) \tag{4.64}$$

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

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 $C(x, x') = \frac{\beta^*}{\alpha^*}\boldsymbol{P}(x)\boldsymbol{\Sigma}\boldsymbol{P}(x')$ 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.

4.2.14. `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 5.12. 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

4.3. PYTHON MODULES

4.3.1. Polynomial Chaos Expansion Tools

Python tools for polynomial chaos expansions are located in `PyUQTk/PyPCE/pce_tools.py`. This is a file of Python wrappers for many C++ operations. These capabilities are exemplified in `examples/heat_transfer_window/` and `examples/surrogate_genz/`.

4.3.2. 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

$$p(\lambda|y,\mathcal{M}) = \frac{p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})}{p(y|\mathcal{M})}, \tag{4.65}$$

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

$$p(\mathcal{M}_k|y) = \frac{p(y|\mathcal{M}_k)p(\mathcal{M}_k)}{p(y)}. \tag{4.66}$$

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

$$\frac{p(\mathcal{M}_1|y)}{p(\mathcal{M}_2|y)} = \frac{p(y|\mathcal{M}_1)p(\mathcal{M}_1)}{p(y|\mathcal{M}_2)p(\mathcal{M}_2)}. \tag{4.67}$$

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

$$\frac{p(\mathcal{M}_1|y)}{p(\mathcal{M}_2|y)} = \frac{p(y|\mathcal{M}_1)}{p(y|\mathcal{M}_2)}. \tag{4.68}$$

The RHS of (4.68), being the ratio of model likelihoods (which is also the ratio of evidence terms as defined in (4.65)) 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.

4.3.2.1. `LikelihoodMC_PriorSamples:`

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

$$p(y|\mathcal{M}) = \int_\lambda p(y|\lambda, \mathcal{M}) p(\lambda|\mathcal{M})\, d\lambda \approx \frac{1}{N} \sum_{i=1}^{N} p\left(y|\lambda^{(i)}, \mathcal{M}\right). \tag{4.69}$$

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

4.3.2.2. `ImportanceLikelihoodMC_PosteriorSamples:`

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

$$p(y|\mathcal{M}) = \int_\lambda p(y|\lambda, \mathcal{M}) \frac{p(\lambda|\mathcal{M})}{p_b(\lambda|\mathcal{M})} p_b(\lambda|\mathcal{M})\, d\lambda \approx \frac{1}{N} \sum_{i=1}^{N} p\left(y|\lambda^{(i)}, \mathcal{M}\right) \frac{p\left(\lambda^{(i)}|\mathcal{M}\right)}{p_b\left(\lambda^{(i)}|\mathcal{M}\right)}. \tag{4.70}$$

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

4.3.2.3. *PosteriorGaussian_PosteriorSamples:*

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

$$p(y|\mathcal{M}) = \frac{p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})}{p(\lambda|y,\mathcal{M})} \approx \frac{p(y|\lambda,\mathcal{M})p(\lambda|\mathcal{M})}{\tilde{p}(\lambda|y,\mathcal{M})}. \tag{4.71}$$

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 4.71 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

4.3.2.4. *`Harmonic_PosteriorSamples:`*

This function estimates the evidence via the Harmonic approximation formula:

$$p(y|\mathcal{M}) \approx \left\{ \frac{1}{N} \sum_{i=1}^{N} \frac{1}{p\left(y|\lambda^{(i)}, \mathcal{M}\right)} \right\}^{-1}. \tag{4.72}$$

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

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` to access the Python scripts.

5.1. 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.* [11].

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.

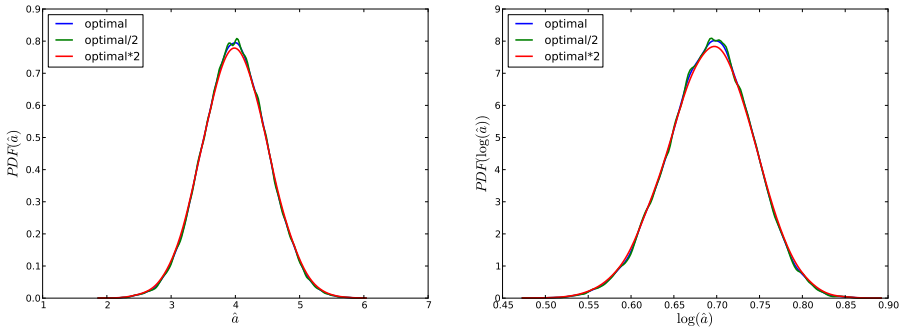


Figure 5-1. Probability densities for $\hat{a}$ and $\log(\hat{a})$ computed via KDE. Results generated using several KDE bandwidths. This feature is available in the Python's SciPy package starting with version 0.11

5.2. 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 noise.
    - 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 sampling.
        - `*_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:

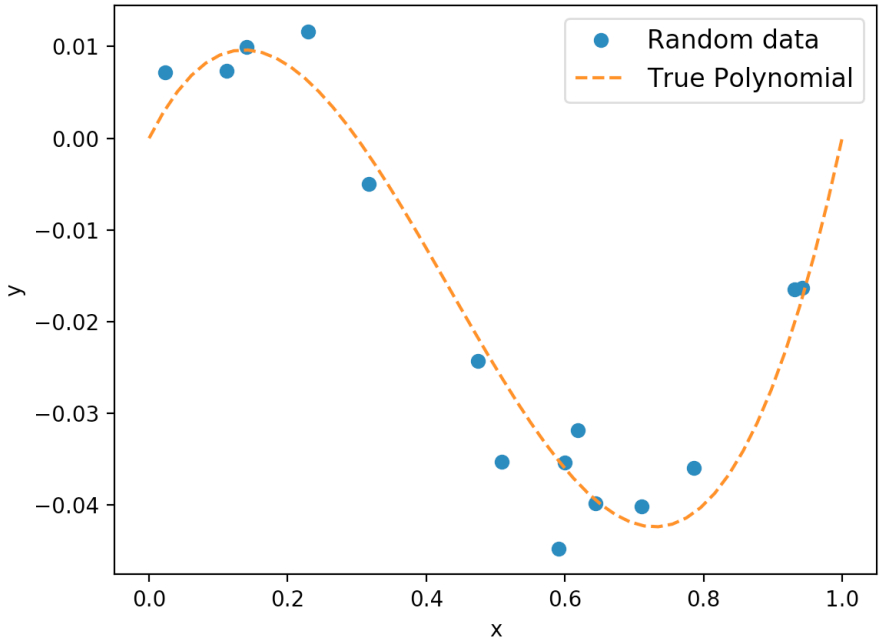


Figure 5-2. Example output of get_data.py -g -e

- `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`

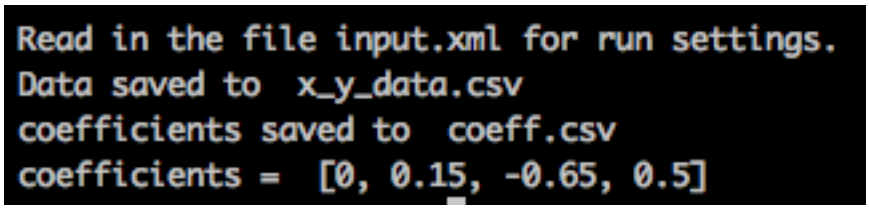


Figure 5-3. Command line output of get_data.py

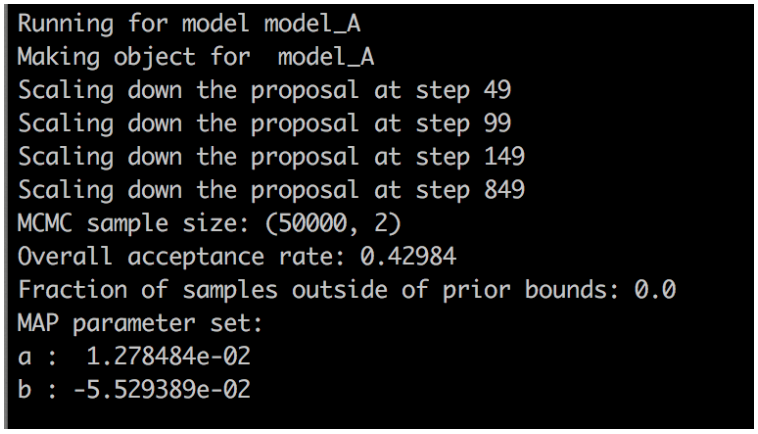


Figure 5-4. Command line output of fit.py

- `../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.

- `../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:

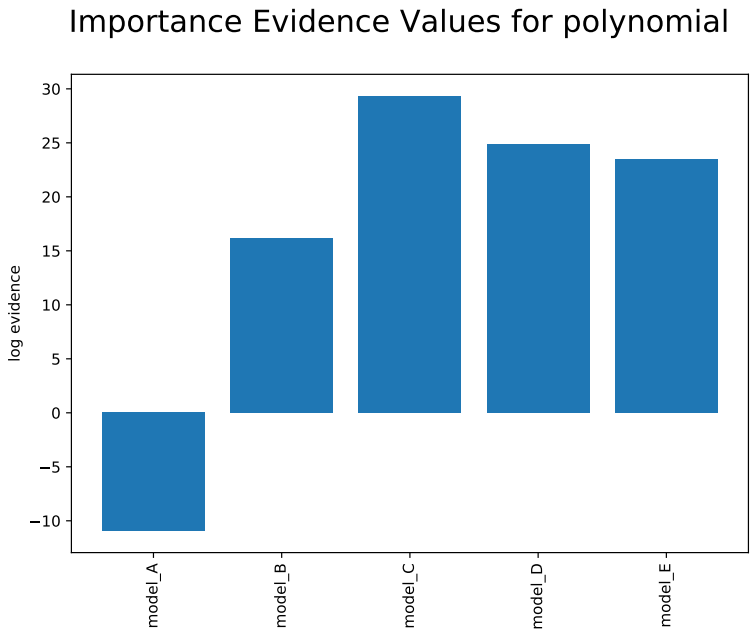


Figure 5-5. Importance Evidence Values for the Polynomial Model. As you can see, model C has the highest evidence value, implying the best fit. This is good because our true solution is of order 3.

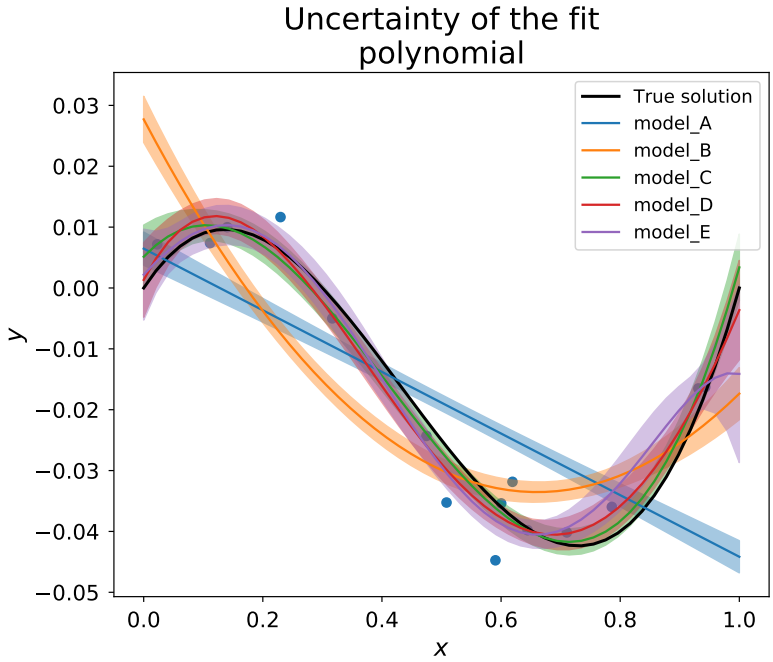


Figure 5-6. Here is the fitted models with uncertainties for all models. The shaded regions show 1 standard deviation. You can also see the true solution and the data used.

  - 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
- 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.

5.3. 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>]
```

For example

```
./forUQ_sr.py HG 3 ISP NISP
```

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 (More than one can be specified)
- *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
    - $u \rightarrow \{u_k\}, v \rightarrow \{v_k\}, w \rightarrow \{w_k\}, z \rightarrow \{z_k\},$
    - $a \rightarrow \{a_k\}, b \rightarrow \{b_k\}, c \rightarrow \{c_k\}, d \rightarrow \{d_k\}, e \rightarrow \{e_k\}, f \rightarrow \{f_k\}.$

4. Step forward in time according to PC arithmetics, *e.g.*
   $a \cdot u \rightarrow \{(a \cdot u)_k\}$ 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. Note, these scripts assume that the model has first been run with the methods requested so that the corresponding data files are available. Sample results are shown in Fig. 5-7.

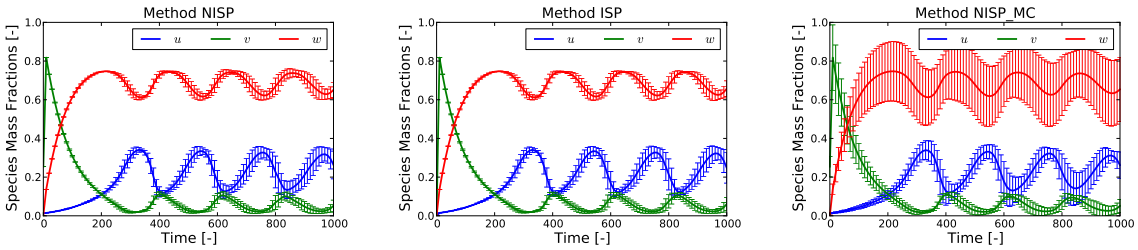


Figure 5-7. Time series of mean and standard deviations for u, v, and w with NISP, ISP, and NISP_MC, respectively.

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.

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
```

Note that all arguments have a default value, to the script can be run without specifying any arguments. If desired, the following parameters can be controlled by the user through the argument list.

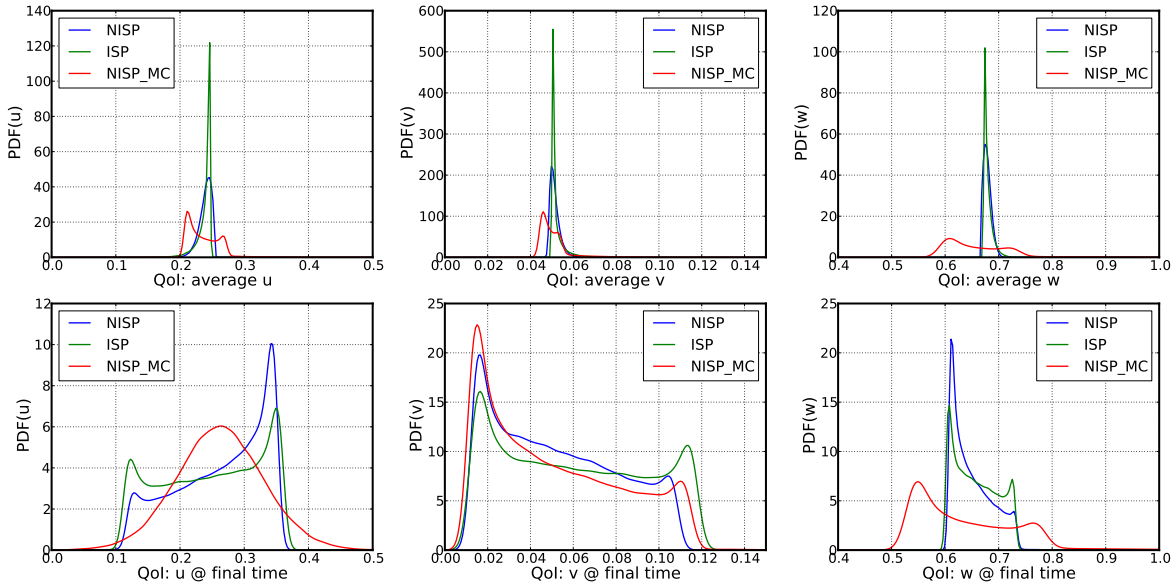


Figure 5-8. PDFs for u, v, and w; Top row shows results for average u, v, and w; Bottom row shows results corresponding to values at the last integration step (final time).

- *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 density. 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" the run, 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
   ```

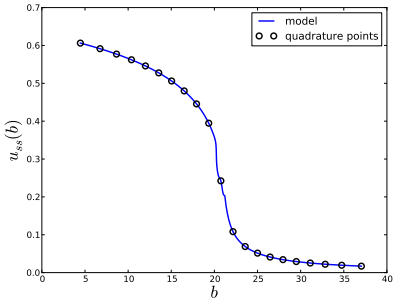
   At the end of this computation, file "input.dat" contains a matrix of PCE evaluations. The number of lines is equal to the number of quadrature points and the number of columns to the dimensionality of input parameter space.

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`

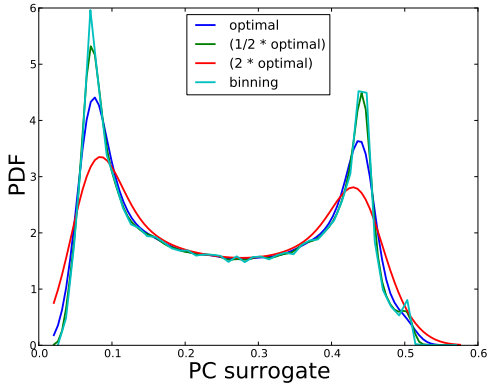
   Draw a 100 samples from the germ of the HG PCE. Samples are saved in file "rvar.dat" and also copied to file "xdata.dat"

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:



5.4. 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 corresponding 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: $\int_{[0,1]^d} f(\lambda)d\lambda$
Oscillatory	$\cos(2\pi u_1 + \sum_{i=1}^{d} w_i\lambda_i)$	$\cos\left(2\pi u_1 + \frac{1}{2}\sum_{i=1}^{d} w_i\right)\prod_{i=1}^{d}\frac{2\sin(\frac{w_i}{2})}{w_i}$
Exponential	$\exp(\sum_{i=1}^{d} w_i(\lambda_i - u_i))$	$\prod_{i=1}^{d}\frac{1}{w_i}(\exp(w_i(1-u_i)) - \exp(-w_iu_i))$
Continuous	$\exp(-\sum_{i=1}^{d} w_i\lvert\lambda_i - u_i\rvert)$	$\prod_{i=1}^{d}\frac{1}{w_i}(2 - \exp(-w_iu_i) - \exp(w_i(u_i-1)))$
Gaussian	$\exp(-\sum_{i=1}^{d} w_i^2(\lambda_i - u_i)^2)$	$\prod_{i=1}^{d}\frac{\sqrt{\pi}}{2w_i}(\text{erf}(w_i(1-u_i)) + \text{erf}(w_iu_i))$
Corner-peak	$(1 + \sum_{i=1}^{d} w_i\lambda_i)^{-(d+1)}$	$\frac{1}{d!\prod_{i=1}^{d} w_i}\sum_{r\epsilon\{0,1\}^d}\frac{(-1)^{\lvert\lvert r\rvert\rvert_1}}{1+\sum_{i=1}^{d} w_ir_i}$
Product-peak	$\prod_{i=1}^{d}\frac{w_i^2}{1+w_i^2(\lambda_i-u_i)^2}$	$\prod_{i=1}^{d} w_i(\arctan(w_i(1-u_i)) + \arctan(w_iu_i))$

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 $\int_{[0,1]^d} f(\lambda)d\lambda$ 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.

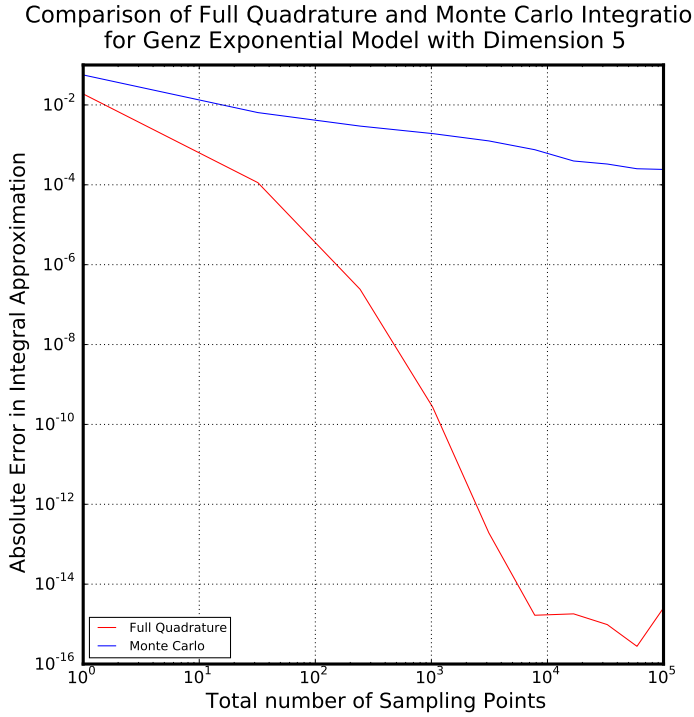


Figure 5-9. Sample results of full_quad.py

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.

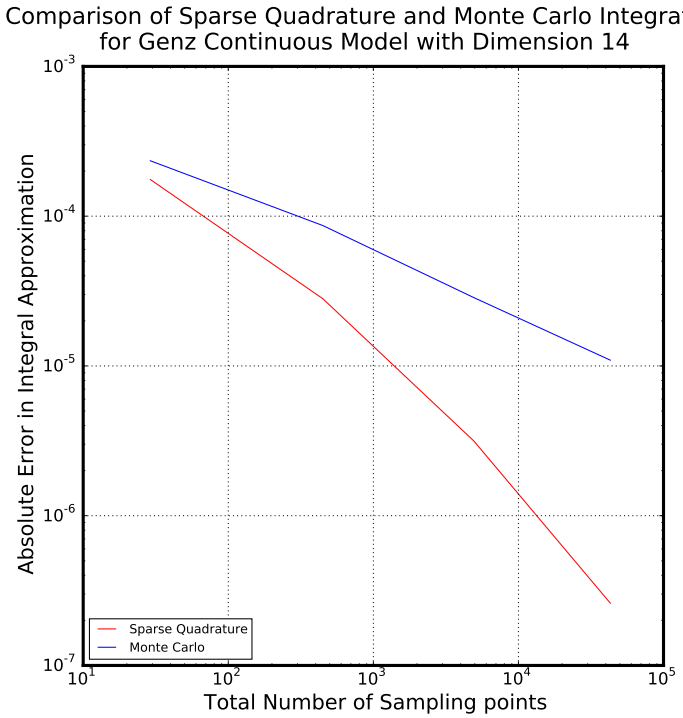


Figure 5-10. Samples results of sparse_quad.py

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.

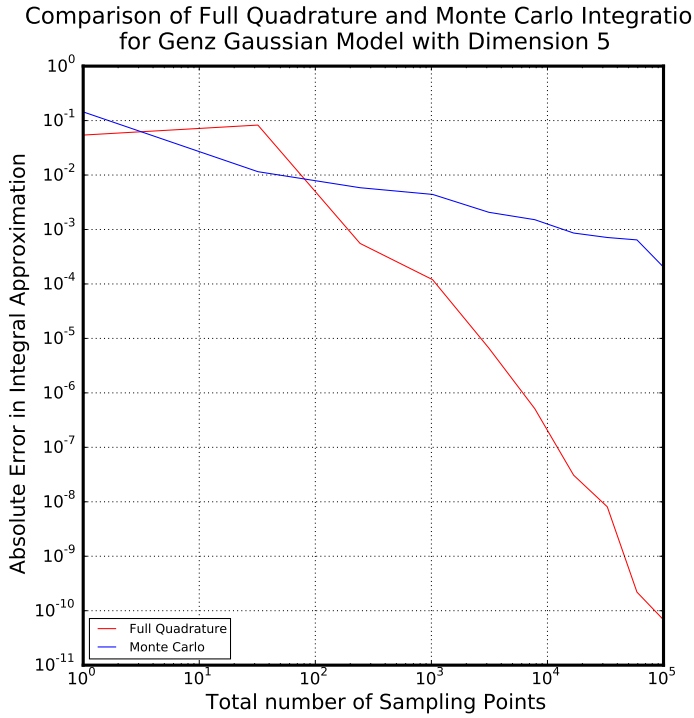


Figure 5-11. Sample results of full_quad.py with Clenshaw-Curtis quadrature rule.

5.5. FORWARD PROPAGATION OF UNCERTAINTY WITH PYUQTK

Overview

This example is located in `examples/heat_transfer_window/`. It contains a Jupyter notebook and a file of Python methods used in the notebook. The notebook demonstrates the propagation of uncertainty in input parameters through three heat transfer models using both a Monte Carlo sampling approach and non-intrusive spectral projection (NISP) via quadrature methods.

In this example, the forward propagation requires the representation of heat flux Q with a multidimensional Polynomial Chaos Expansion (PCE). The theory for this can be found in section 3.1.

Implementation

The script set consists of two files:

- `heat_transfer_window.ipynb`: Jupyter notebook that produces a graph comparing PDFs of heat flux generated using NISP full and sparse quadrature methods and Monte Carlo sampling methods for three window models
- `window_tools.py`: problem-specific functions called by `heat_transfer_window.ipynb`

5.6. SURROGATE CONSTRUCTION FOR GENZ FUNCTIONS WITH PYUQTK

Overview

This example is located in `examples/surrogate_genz/`. It contains four Jupyter notebooks that construct PC surrogates for Genz functions using Galerkin projection, regression, or Bayesian compressive sensing. The normalized root mean square error between the surrogate and the actual function is also calculated.

For more information on PC representations, see section 3.1.

Implementation

The four Jupyter notebooks are

- `surrogate_genz-Galerkin.ipynb`: surrogate construction via Galerkin projection
- `surrogate_genz-Regression.ipynb`: surrogate construction via regression
- `surrogate_genz-BCS.ipynb`: surrogate construction via Bayesian compressive sensing
- `surrogate_genz.ipynb`: surrogate construction (via Galerkin projection, regression, and Bayesian compressive sensing) and error comparison

5.7. SPARSE BASIS SELECTION WITH PYUQTK

Overview

This example is located in `examples/bare_bcs/`. It contains a Jupyter notebook that compares sparse basis selection with the PyUQTk Bayesian Compressed Sensing (BCS) approach to the scikit-learn Orthogonal Matching Pursuit (OMP). Note that this example does not build a Polynomial Chaos Expansion to represent the function. By using the bare BCS call, it can work with arbitrary sets of basis functions.

Implementation

The Jupyter notebook is `bare_bcs.ipynb`

5.8. 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:

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

- $\tilde{\boldsymbol{\alpha}}$: 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:

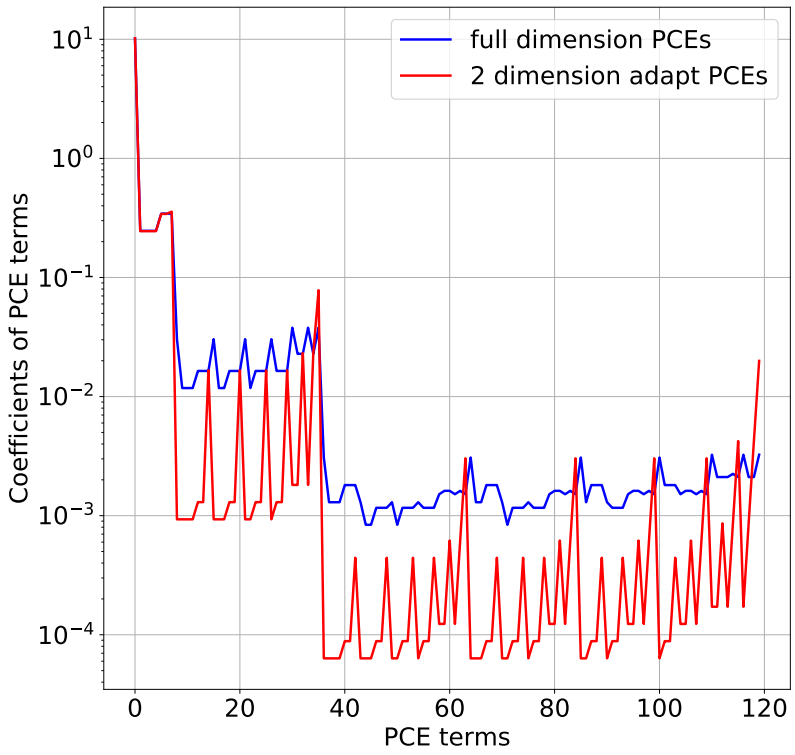


Figure 5-12. Coefficients comparison of adaptation method and full dimension PCE

Note that y axis of Figure 5-12 is plot in *log* scale, so the dominant coefficients of these two are very close. The PDF showed in Figure 5-13 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).

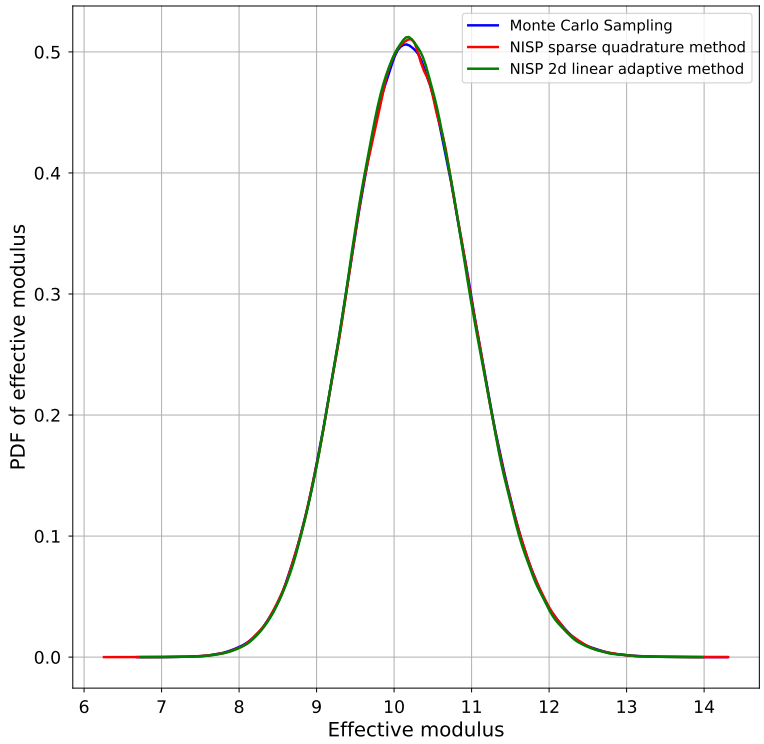


Figure 5-13. PDFs of effective modulus generated with different methods

5.9. 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

Sample Results

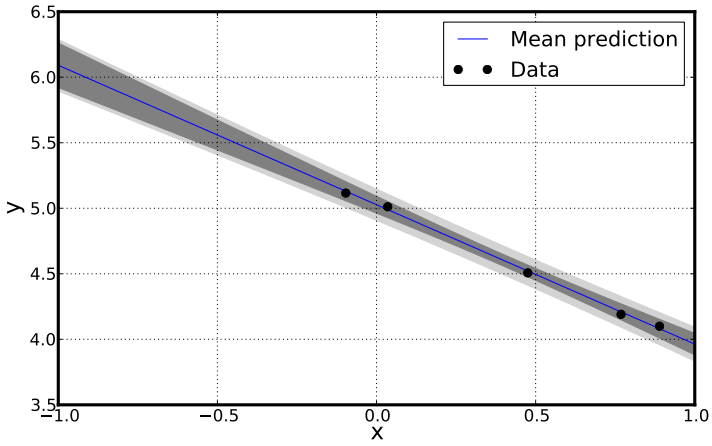


Figure 5-14. The pushed forward posterior distribution (dark grey) and posterior predictive distribution (light grey).

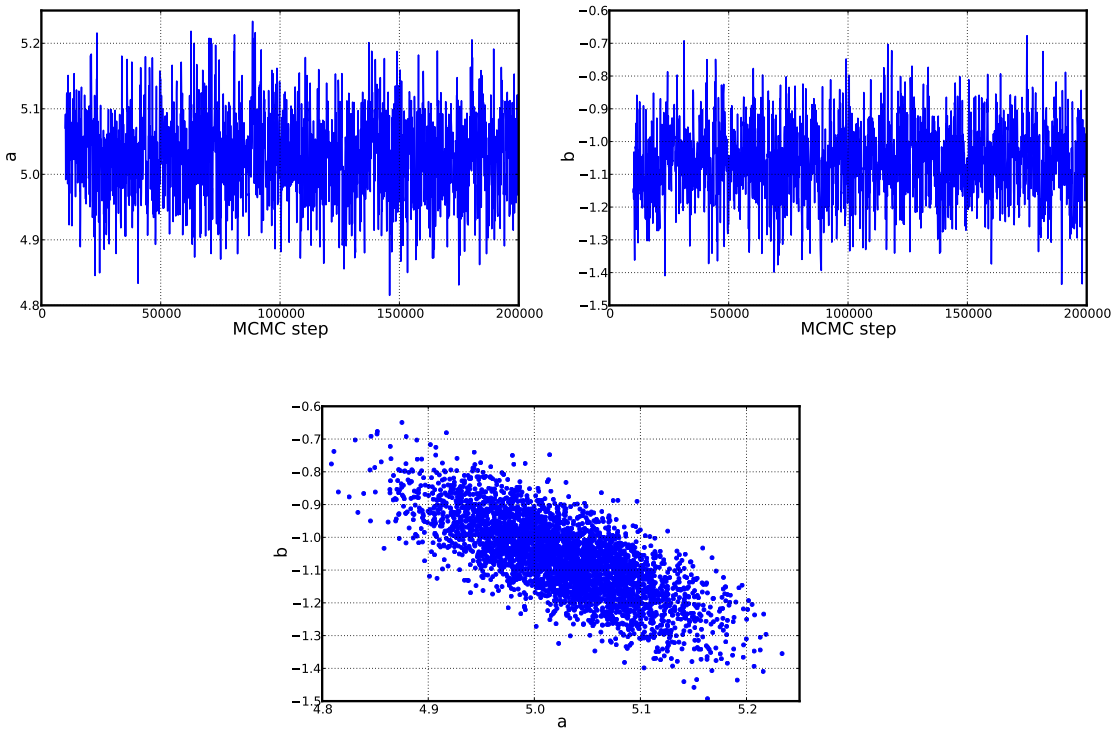


Figure 5-15. MCMC chains for parameters a and b, as well as a scatter plot for a and b

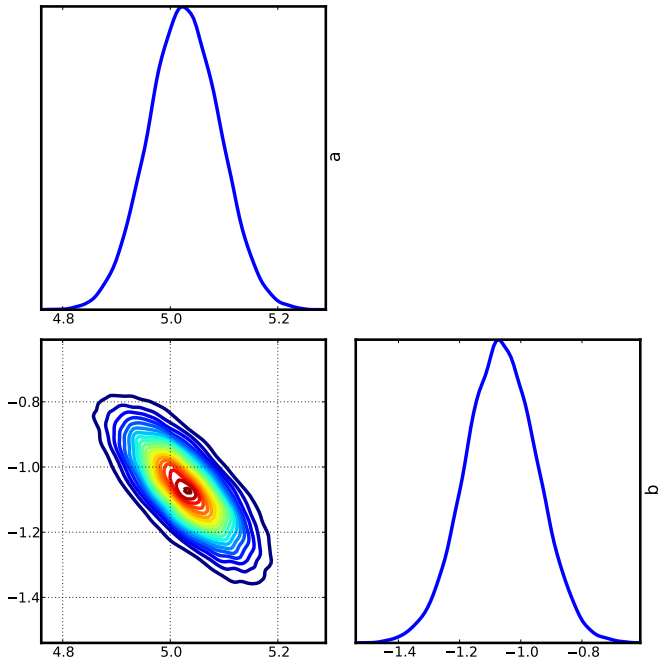


Figure 5-16. Marginal posterior distributions for all variables, as well as marginal joint posteriors

5.10. 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 [7], 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 (with a resampling step) 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 [62] for a relevant discussion).

In general, the performance of TMCMC as implemented in UQTk heavily depends on the maximum allowable coefficient of variation of the sample weights. This can be controlled using the `MCMC` class member function `initTMCMCCv`. Based on numerical experiments, the UQTk default value of 0.1 should be adjusted down whenever the apparent bias in the resulting posterior samples is insufficiently high (i.e. when the generated ensemble does not adhere to the structure inherent in the posterior PDF). This situation seems to arise whenever the discrepancy between the prior and posterior PDFs is high (as dictated by the likelihood). However, the need to adjust the coefficient of variation is reduced with (a) greater number of TMCMC samples, and/or (b) longer MCMC chains (to encourage mixing as controlled via the `initTMCMCMFactor` member function).

This example involves a driver python script, `tmcmc_bimodal.py`, that invokes the program (based on provided C++ code) `tmcmc_bimodal.x`. This program 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. 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 running `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

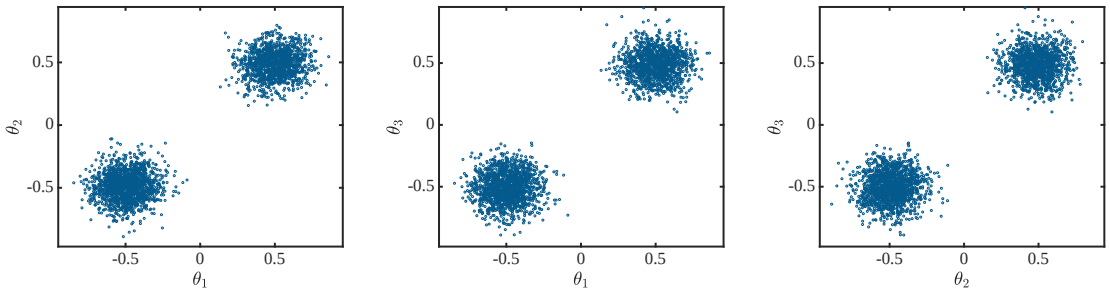


Figure 5-17. 2-dimensional scatter plots of posterior samples

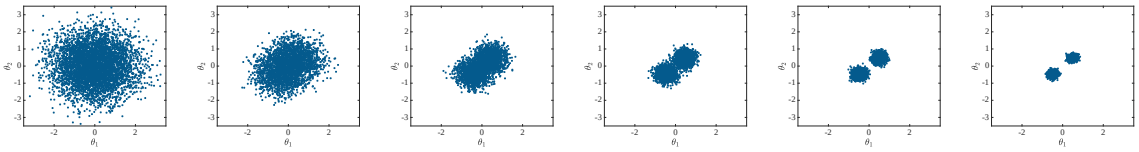


Figure 5-18. 2-dimensional scatter plots of intermediate TMCMC samples, from prior to posterior

Sample Results

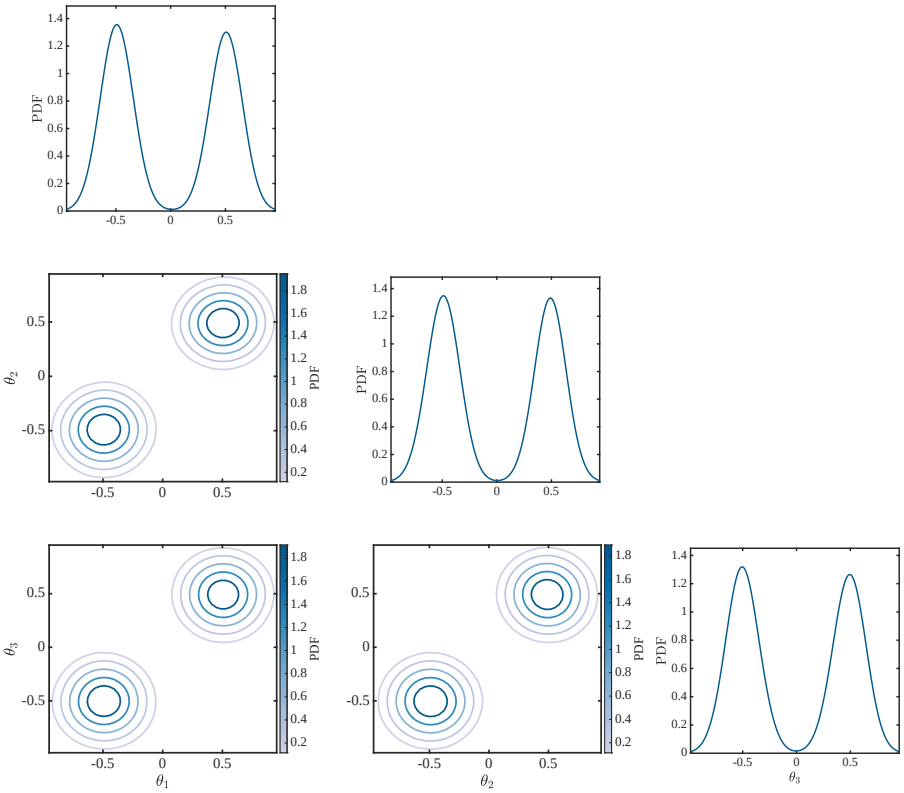


Figure 5-19. Marginal posterior distributions for all variables, as well as marginal joint posteriors

5.11. 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. [46]. 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, 49] 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 $\psi_1(\xi_i), \psi_2(\xi_i), \ldots$. 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 4.2.11.

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 5-20 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 ranges 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.

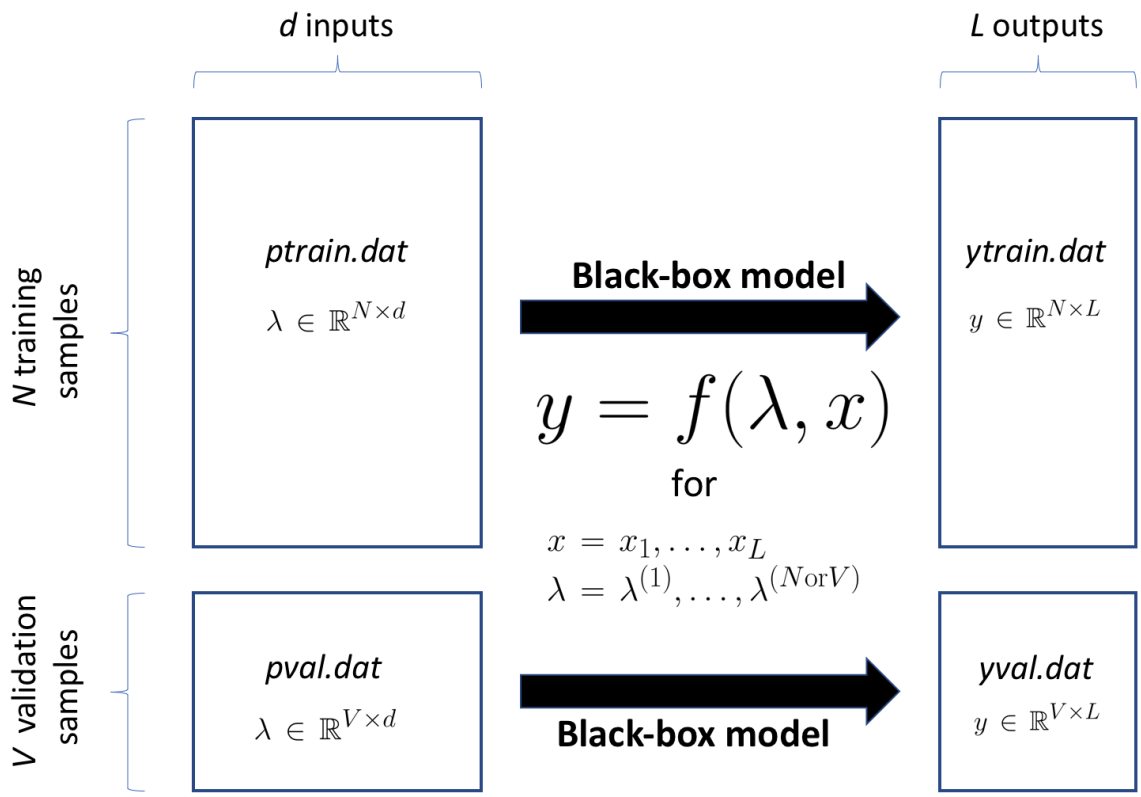


Figure 5-20. Sketch of the expected input-output structure of the black-box model.

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 run. 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 5-20)
`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`.

5.12. 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 $X = (X_1, \cdots, X_n) : \Omega \to \mathcal{X} \subset \mathbb{R}^n$ 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 $X_{-i} = (X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_n) : \Omega \to \mathcal{X}_{-i} \subset \mathbb{R}^{n-1}$, 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 $Y = f(X) : \Omega \to \mathbb{R}$, 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

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

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

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

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

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

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 [55] and

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

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

$$S_i + S^T_{-i} = S_{-i} + S^T_i = 1 \tag{5.28}$$

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^T_i 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 [43] and Kucherenko *et al* [32]. 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 $p_{X_{-i}|X_i}(x'_{-i}|x^k_i)$, 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

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

In the second approach, S_i^T is estimated as

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

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`.

  `verb`: verbosity level

  `nd`: number of significant digits for ascii output

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

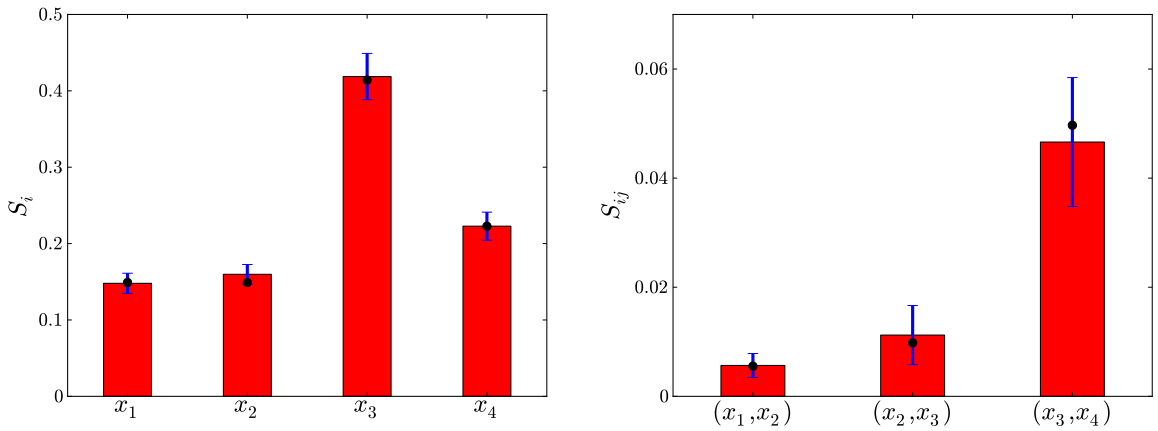


Figure 5-21. First-order (left frame) and joint (right frame) Sobol indices for the model given in Eq. (5.35). The black circles show the theorerical values, computed analytically, and the error bars correspond to $\pm\sigma$ computed based on an ensemble of 10 runs.

- `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 5-21 and 5-22 show results based on an ensemble of 10 runs. To generate these results run the example workflow:

```
python gsatest.py
```

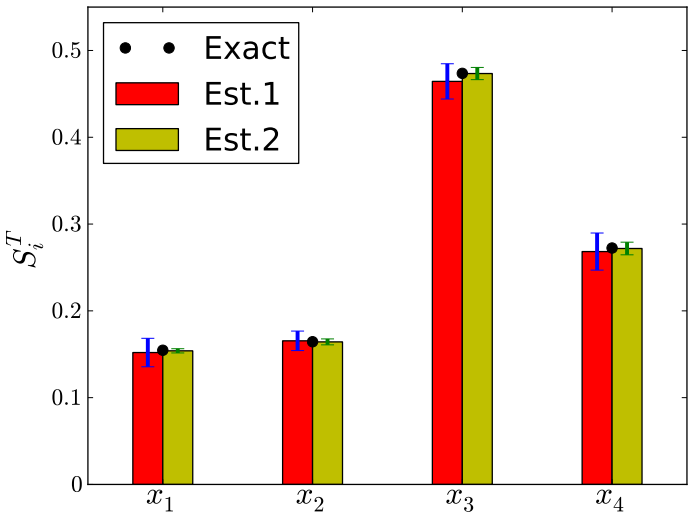


Figure 5-22. Total-order Sobol indices for the model given in Eq. (5.35). The red bars shows results based on Eq. (5.33) while the yellow bars are based on Eq. (5.34). The black circles show the theorerical values, computed analytically, and the error bars correspond to $\pm\sigma$ computed based on 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).

5.13. 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 [36] 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:

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

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 [27]

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

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**:

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

Figure 5-23 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
```

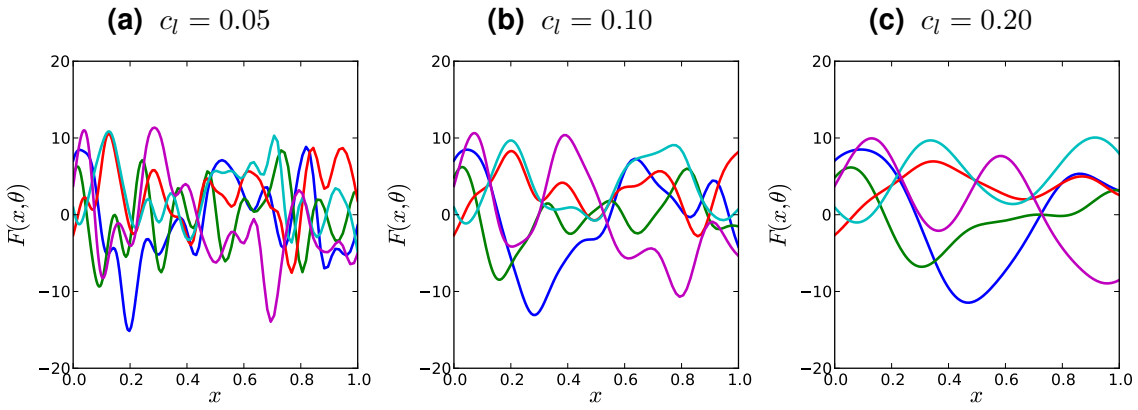


Figure 5-23. Sample 1D random field realizations for several correlation lengths c_l.

Once the RF realizations are generated the covariance matrix is discarded and a "numerical" covariance matrix is estimated based on the available realizations. Figure 5-24 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 density 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.

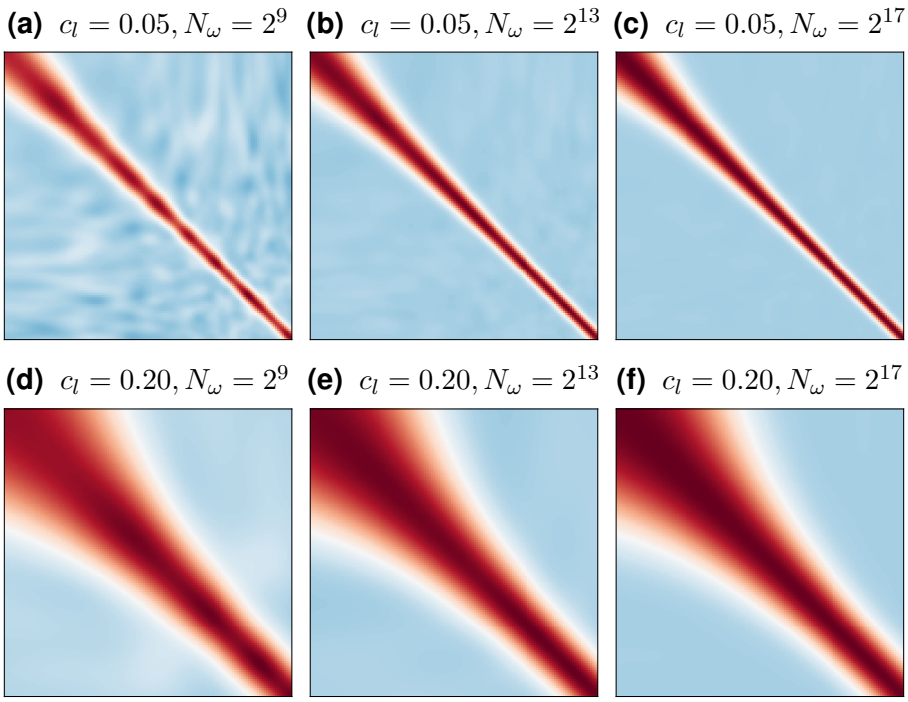


Figure 5-24. Illustration of covariance matrices computed from 1D RF realizations. Red corresponds to large values, close to 1, while blue corresponds to small values, close to 0.

Figure 5-25 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 5-26 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

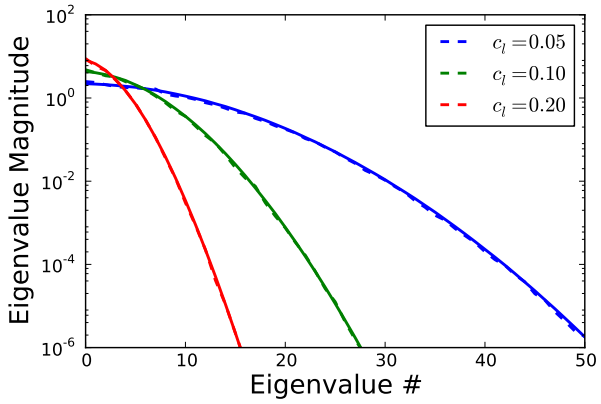


Figure 5-25. KL eigenvalues estimated with two sets of RF realizations: $2^9 = 512$ **(dashed lines) and** $2^{17} = 131072$ **(solid lines).**

```
./mkplots.py numKLevec 0.05 512 on
./mkplots.py numKLevec 0.05 8192 off
./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. 5-26c, 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).

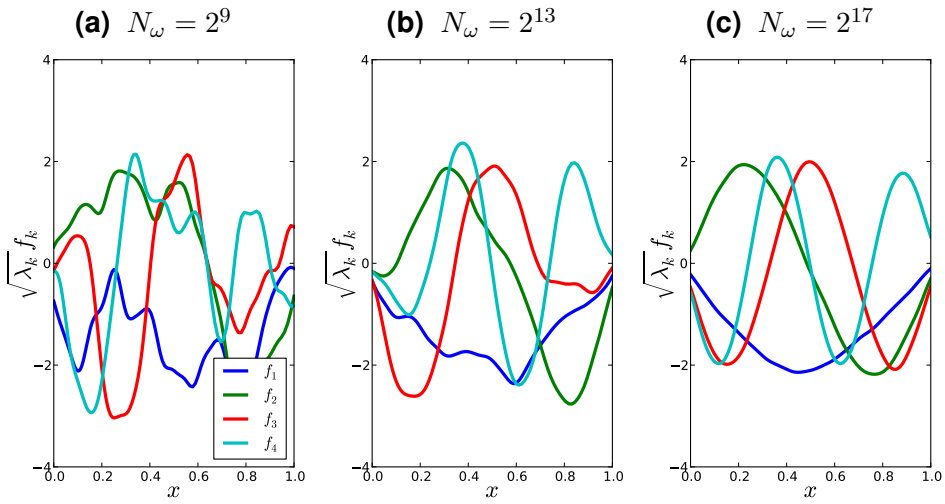


Figure 5-26. Illustration of first 4 **KL modes, computed based on a numerical covariance matrices using three sets of RF realizations with** $c_l = 0.05$

Figure 5-27 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. 5-27b, are close to the ones based on a much larger number of realizations, $N_\omega = 2^{17} = 131072$ in Fig. 5-27c.

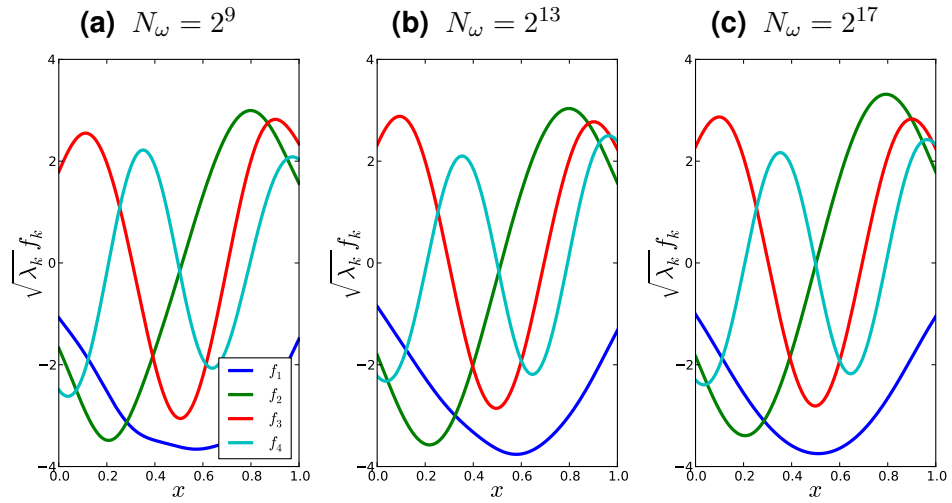


Figure 5-27. Illustration of first 4 KL modes, computed based on a numerical covariance matrices using three sets of RF realizations with $c_l = 0.20$

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 5-28 and 5-29. 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
```

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

Figures 5-30 and 5-31 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 occurring for smaller correlation lengths, in Fig. 5-30, compared to RF with larger correlation lengths, e.g. the example shown in Fig. 5-31. The plots shown in Figs. 5-30 and 5-31 were generated with

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

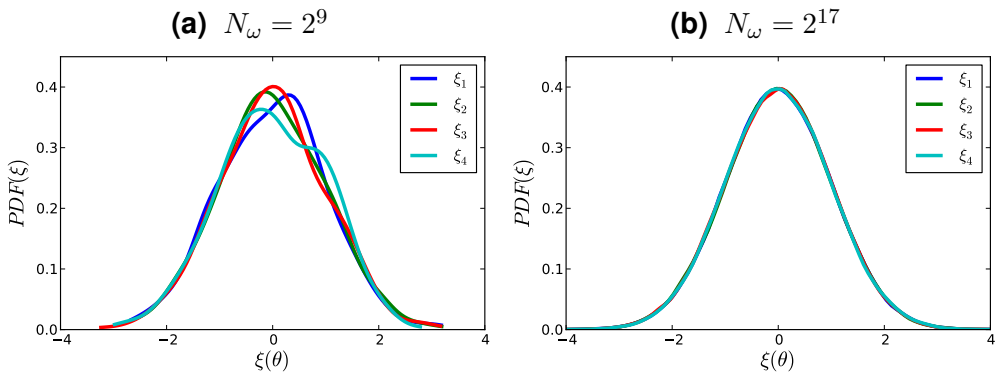


Figure 5-28. Probability densities for ξ_k obtained via KDE using samples obtained by projecting RF realizations onto KL modes. Results correspond to $c_l = 0.05$.

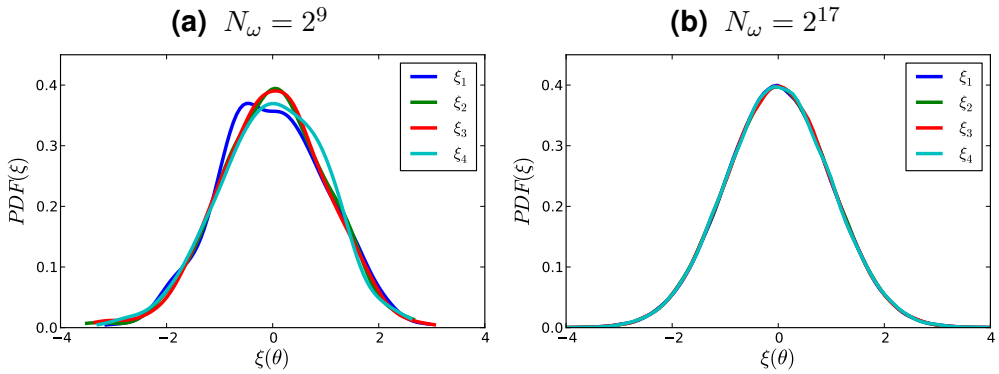


Figure 5-29. Probability densities for ξ_k obtained via KDE using samples obtained by projecting RF realizations onto KL modes. Results correspond to $c_l = 0.20$.

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 [27]. The first coordinate is computed via

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

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**:

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

Figure 5-32 shows the 2D computational grid created with these parameters. This figure was generated with the Python script "pl2Dsgrid.py"

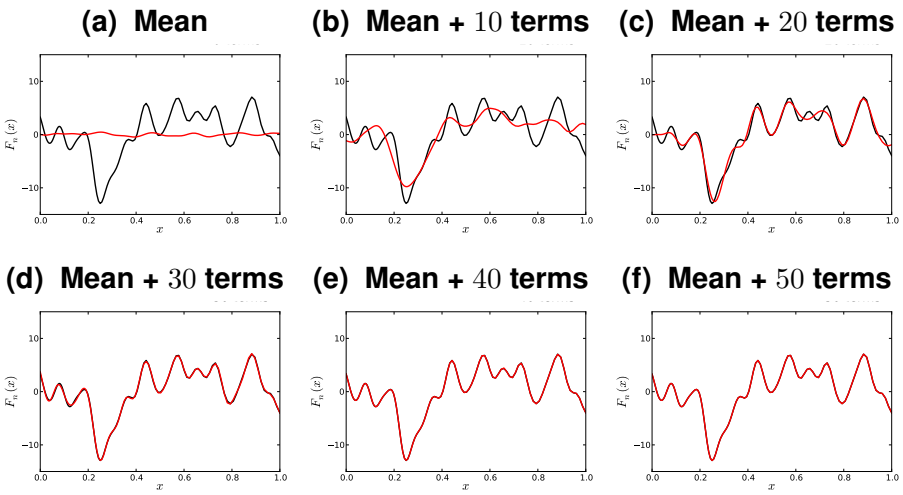


Figure 5-30. Reconstructing realizations with an increasing number of KL expansion terms for $c_l = 0.05$

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

Figure 5-33 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. 5-33a,5-33b)
`./mkplots.py samples2D 0.2 4096 2` (Figs. 5-33c,5-33d)
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 5-34 and 5-35 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. 5-35 are close to the KL modes corresponding to the analytical covariance matrix (results not shown), while the results in Fig. 5-34 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. 5-34)
`./mkplots.py numKLevec2D 0.1 65536` (Fig. 5-35)

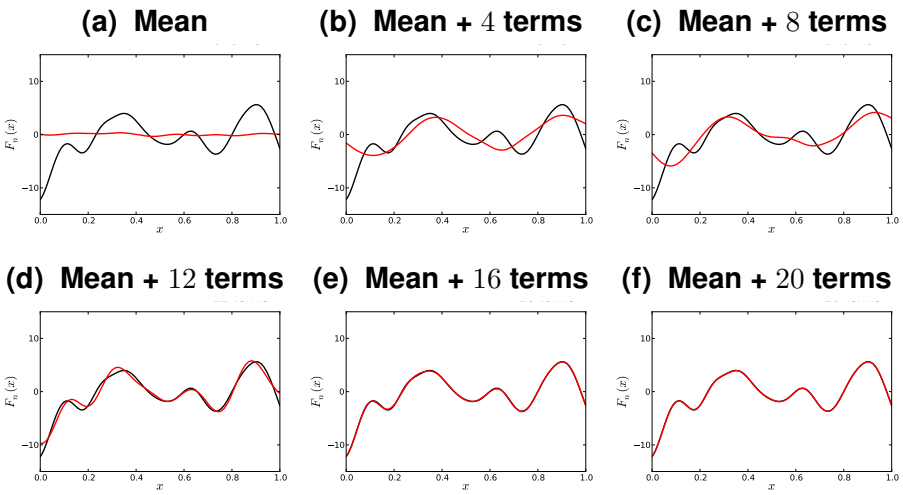


Figure 5-31. Reconstructing realizations with an increasing number of KL expansion terms for $c_l = 0.10$

Figure 5-36 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. 5-36 were generated with

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

Figures 5-37 and 5-38 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 occurring for smaller correlation lengths, in Fig. 5-37, compared to RF with larger correlation lengths, e.g. the example shown in Fig. 5-38. The plots shown in Figs. 5-37 and 5-38 were generated with

```
./mkplots.py pltKLrecon2D 0.2 3  85 12 (Fig. 5-37)
./mkplots.py pltKLrecon2D 0.5 37 36 5  (Fig. 5-38)
```

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 5-39 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 5-40 shows 2D RF realizations with correlation lengths $c_l = 0.5°$ and $c_l = 2°$. These figure were generated with

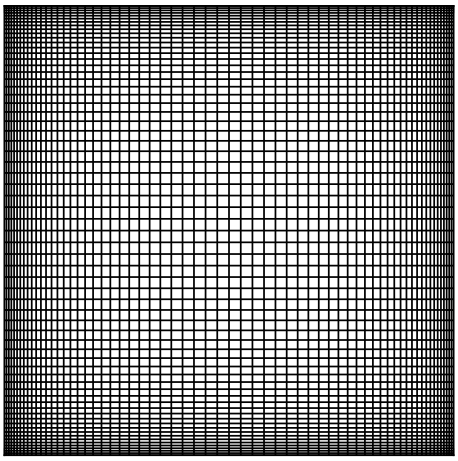


Figure 5-32. Structured grid employed for 2D RF examples.

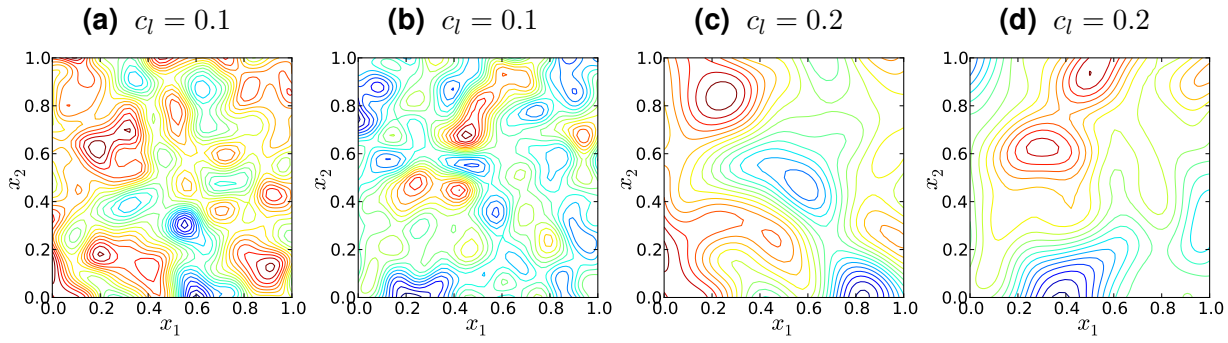


Figure 5-33. Sample 2D random field realizations for $c_l = 0.1$ and $c_l = 0.2$.

`./mkplots.py samples2Du 0.5 4096 2` (Figs. 5-40a,5-40b)
`./mkplots.py samples2Du 2.0 4096 2` (Figs. 5-40c,5-40d)

Figure 5-41 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. 5-42. 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 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 5-43 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. 5-41)
`./mkplots.py anlKLevec2Du SqExp 0.5` (Fig. 5-42)
`./mkplots.py numKLevec2Du 2.0 65536` (Fig. 5-43)

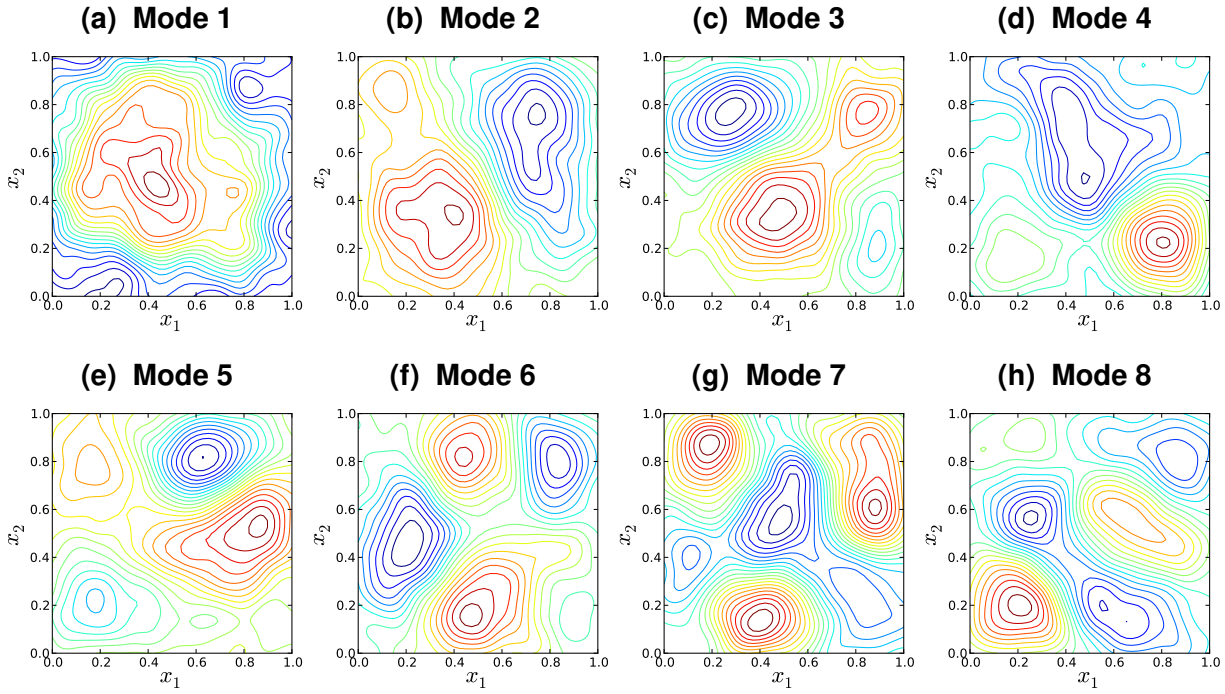


Figure 5-34. Illustration of first 8 KL modes, computed based on a numerical covariance matrix estimated using 2^{12} 2D RF realizations with $c_l = 0.1$

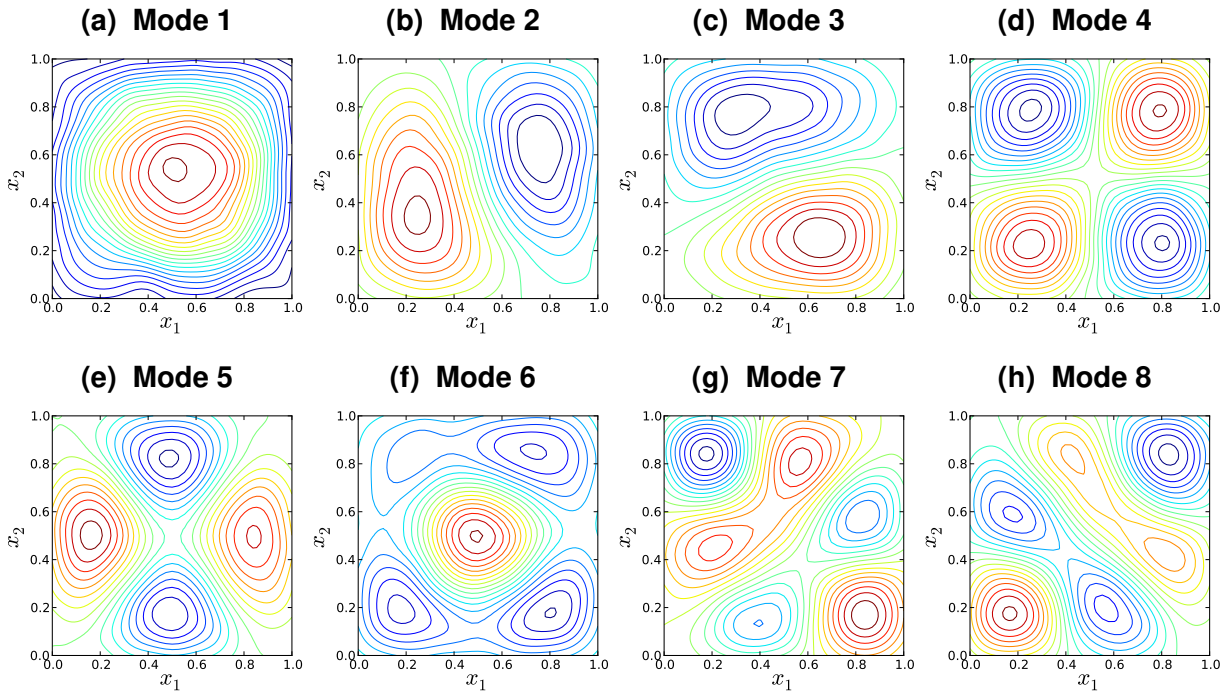


Figure 5-35. Illustration of first 8 KL modes, computed based on a numerical covariance matrix estimated using 2^{16} 2D RF realizations with $c_l = 0.1$

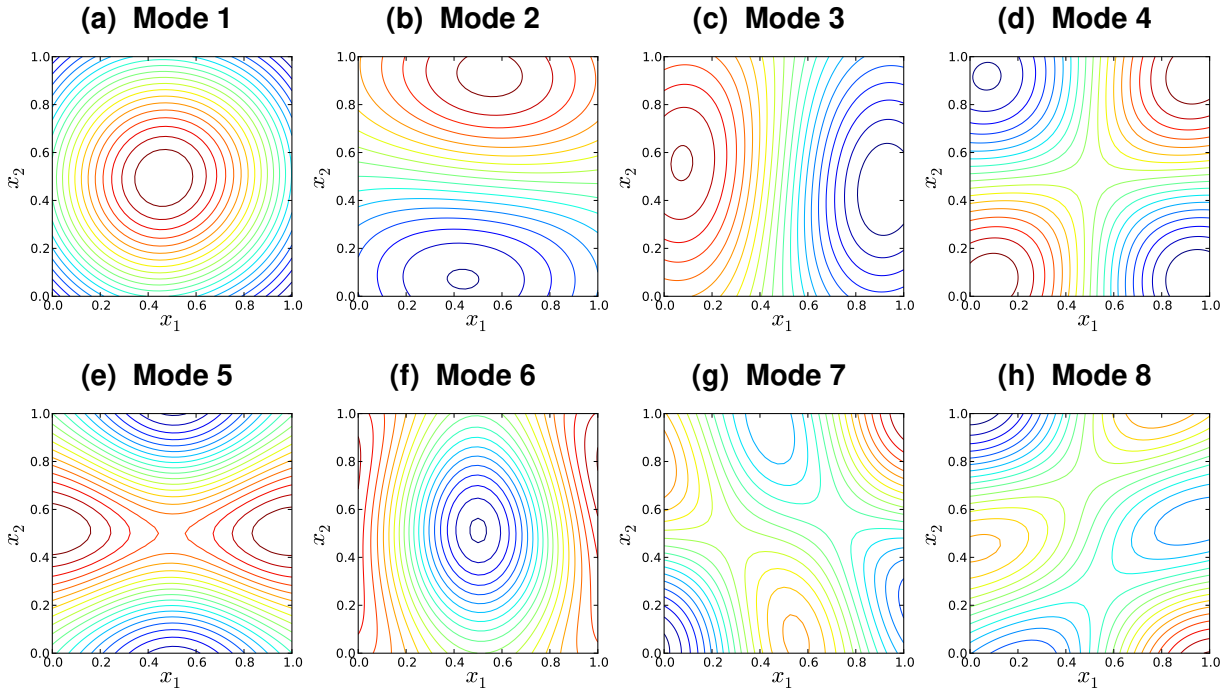


Figure 5-36. Illustration of first 8 KL modes, computed based on a numerical covariance matrix estimated using 2^{12} 2D RF realizations with $c_l = 0.5$

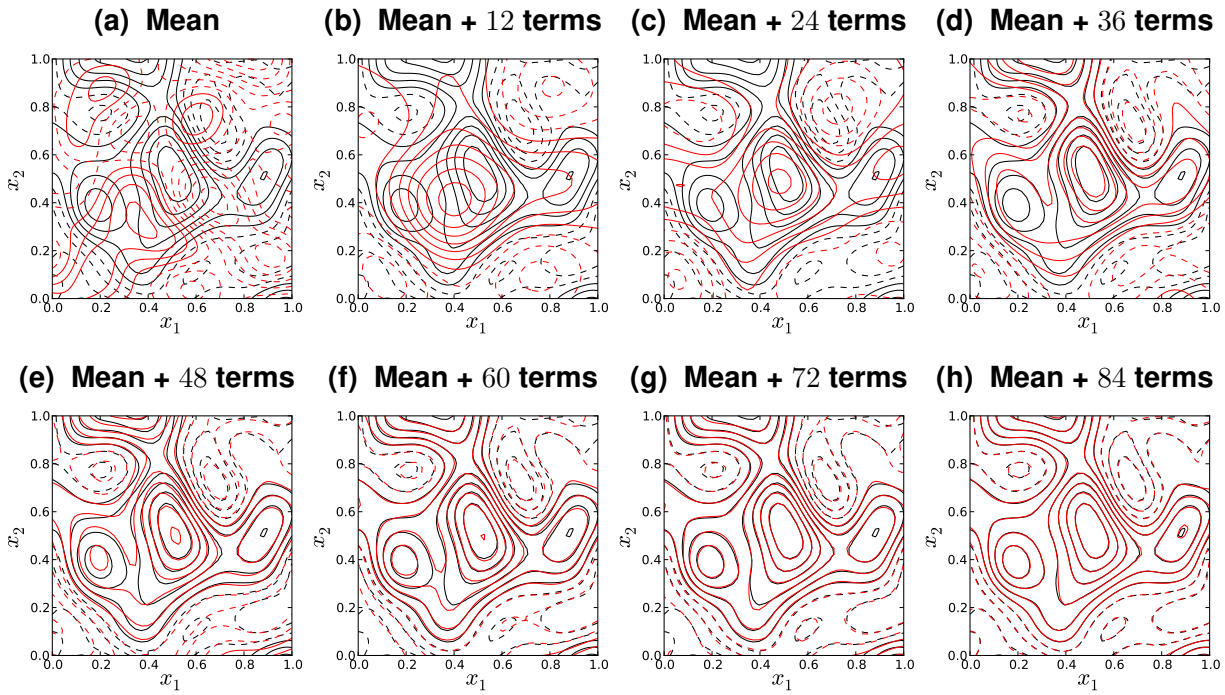


Figure 5-37. Reconstructing 2D realizations with an increasing number of KL expansion terms for $c_l = 0.2$

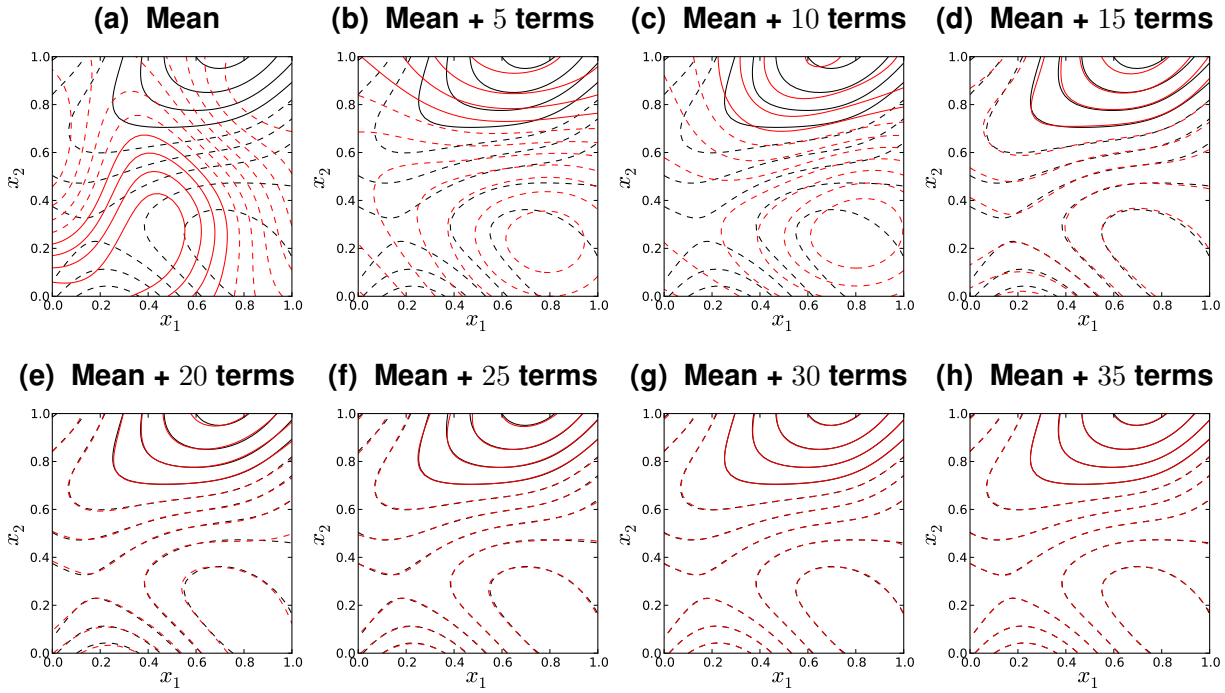


Figure 5-38. Reconstructing 2D realizations with an increasing number of KL expansion terms for $c_l = 0.5$

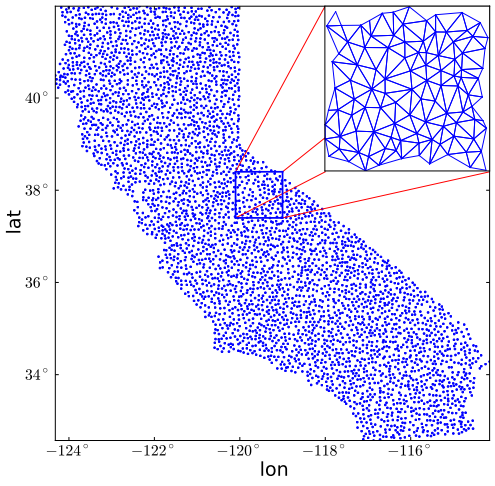


Figure 5-39. Unstructured grid generated via Delaunay traingulation overlaid over California.

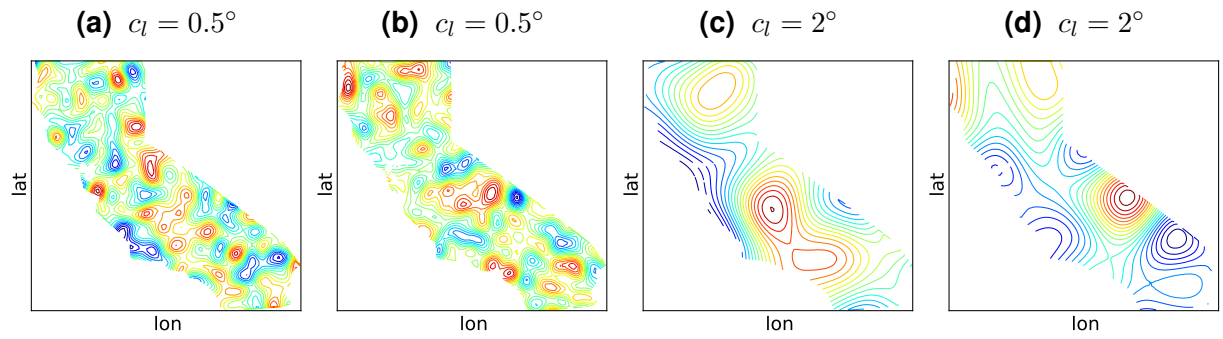


Figure 5-40. Sample 2D random field realizations on an unstructured grid for $c_l = 0.5°$ and $c_l = 2°$.

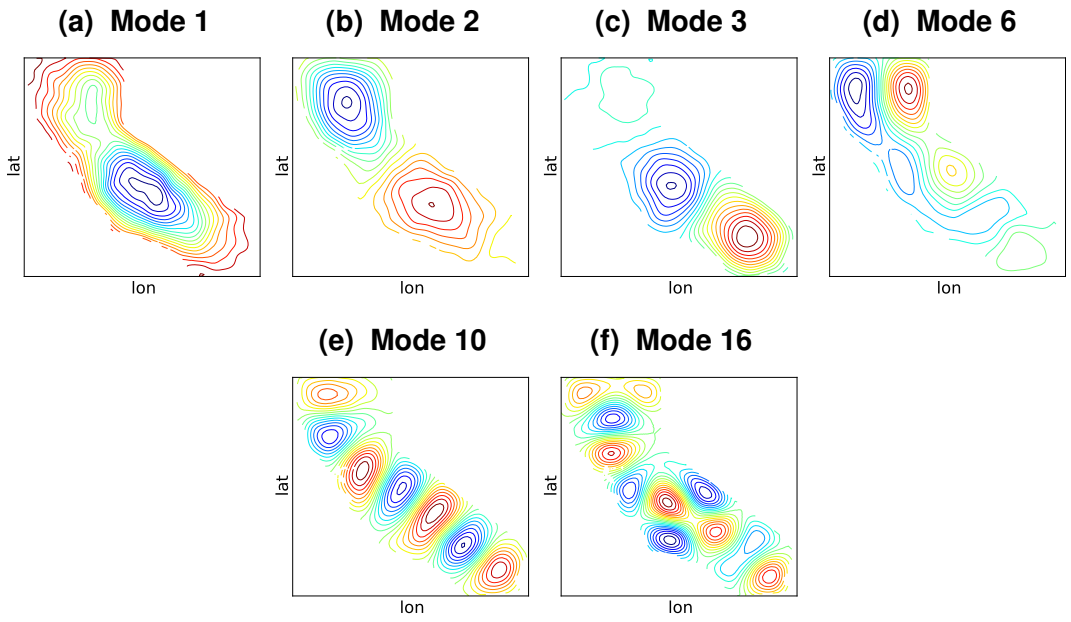


Figure 5-41. Illustration of select KL modes, computed based on a numerical covariance matrix estimated using 2^{16} 2D RF realizations on an unstructured grid with $c_l = 0.5°$.

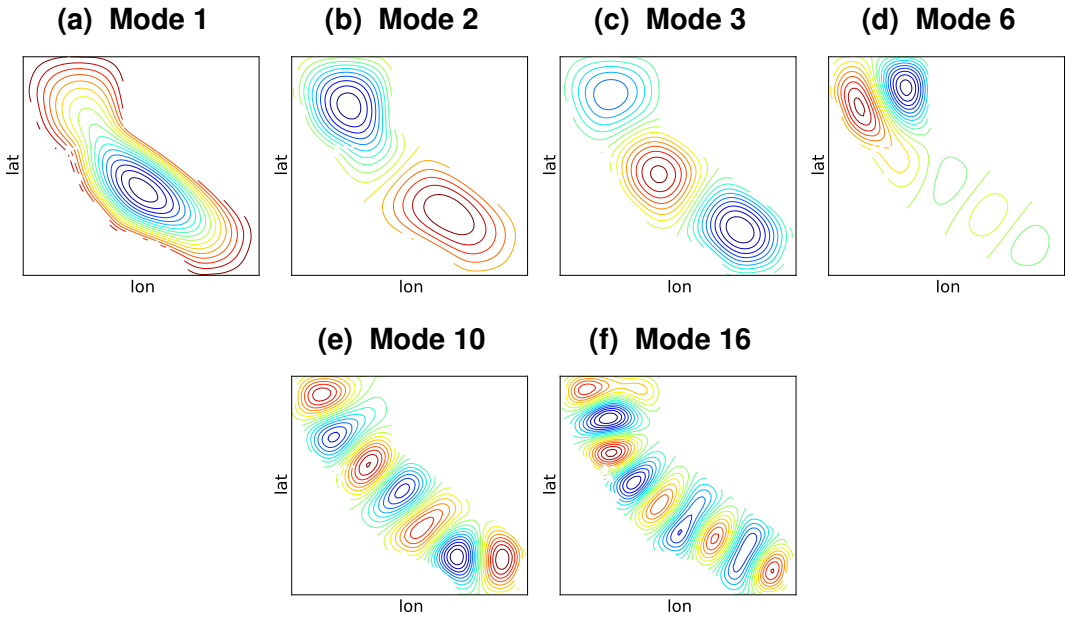


Figure 5-42. Illustration of select KL modes, computed based on an analytical covariance matrix for 2D RF realizations on an unstructured grid with $c_l = 0.5°$ and a square-exponential form.

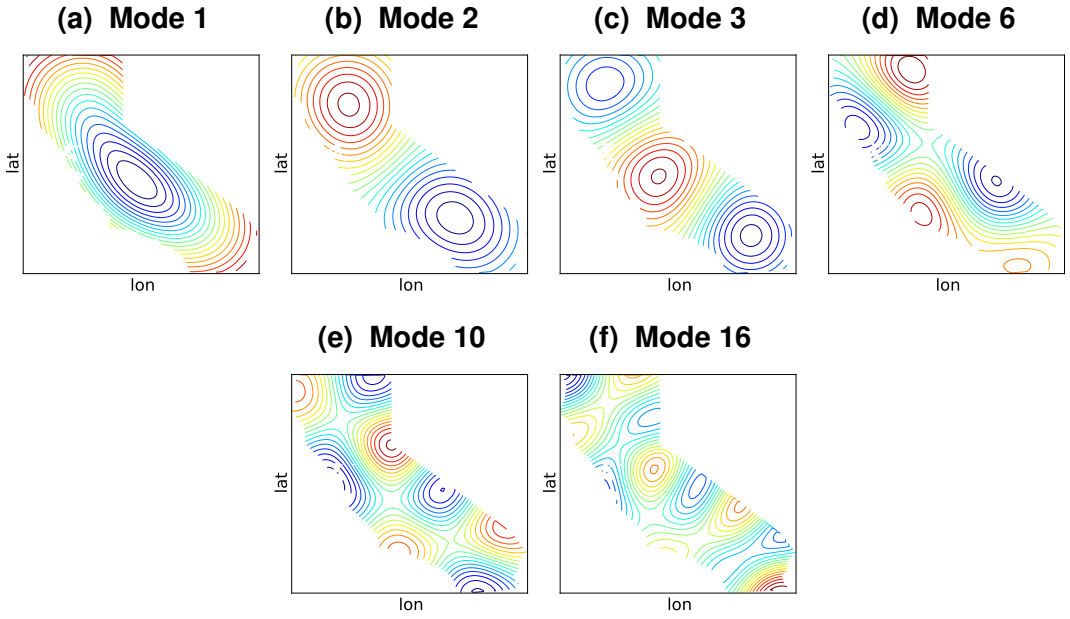


Figure 5-43. Illustration of select KL modes, computed based on a numerical covariance matrix estimated using 2^{16} 2D RF realizations on an unstructured grid with $c_l = 2°$.

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 developers at `uqtk-developers@software.sandia.gov`, or share your questions directly with the UQTk Users list, at `uqtk-users@software.sandia.gov`. We also maintain an announcement list `uqtk-announce@software.sandia.gov` for announcements about UQTk. To sign up for these mailing lists, please visit the UQTk website at `https://www.sandia.gov/UQToolkit/`.

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] V. Barthelmann, E. Novak, and K. Ritter. High-dimensional polynomial interpolation on sparse grids. *Adv. Compu. Math.*, 12:273–288, 2000.

[3] J.M. Bernardo and A.F.M. Smith. *Bayesian Theory*. Wiley Series in Probability and Statistics. John Wiley & Sons Ld, Chichester, England, 2000.

[4] E. Candès, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory*, 52(2):489–509, 2006.

[5] Emmanuel J Candes, Michael B Wakin, and Stephen P Boyd. Enhancing sparsity by reweighted ? 1 minimization. *Journal of Fourier analysis and applications*, 14(5-6):877–905, 2008.

[6] B. P. Carlin and T. A. Louis. *Bayesian Methods for Data Analysis*. Chapman and Hall/CRC, Boca Raton, FL, 2011.

[7] 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.

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

[9] P. Conrad and Y. Marzouk. Adaptive smolyak pseudospectral approximations. *SIAM Journal on Scientific Computing*, 35(6):A2643–A2670, 2013.

[10] T. Crestaux, O. Le Maître, and J.M. Martinez. Polynomial chaos expansion for sensitivity analysis. *Reliability Engineering & System Safety*, 94(7):1161–1172, 2009.

[11] 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.

[12] D. Donoho. Compressed sensing. *IEEE Trans. Inform. Theory*, 52(4):1289–1306, 2006.

[13] A. Doostan and H. Owhadi. A non-adapted sparse approximation of PDEs with stochastic inputs. *J. Comput. Phys.*, 230(8):3015–3034, 2011.

[14] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall CRC, 2 edition, 2003.

[15] 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.

[16] 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).

[17] Thomas Gerstner and Michael Griebel. Dimension adaptive tensor product quadrature. *Computing*, 71:2003, 2003.

[18] R.G. Ghanem and P.D. Spanos. *Stochastic Finite Elements: A Spectral Approach*. Springer Verlag, New York, 1991.

[19] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall, London, 1996.

[20] G. H. Golub and J. H. Welsch. Calculation of Gauss quadrature rules. *Math. Comp.*, 23:221–230, 1969.

[21] M. Griebel. Sparse grids and related approximation schemes for high dimensional problems. In *Proceedings of the Conference on Foundations of Computational Mathematics*, Santander, Spain, 2005.

[22] H. Haario, E. Saksman, and J. Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, 7:223–242, 2001.

[23] Heikki Haario, Marko Laine, Antonietta Mira, and Eero Saksman. Dram: Efficient adaptive mcmc. *Statistics and Computing*, 16(4):339–354, 2006.

[24] Chris Hans. Bayesian Lasso regression. *Biometrica*, 96:835–845, 2009.

[25] 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.

[26] F.B. Hildebrand. *Introduction to Numerical Analysis*. Dover, 1987.

[27] K.A Hoffmann and S.T. Chiang. *Computational Fluid Dynamics*, volume 1, chapter 9, pages 358–426. EES, 2000.

[28] John D Jakeman, Michael S Eldred, and Khachik Sargsyan. Enhancing ℓ_1-minimization estimates of polynomial chaos expansions using basis selection. *Journal of Computational Physics*, 289:18–34, 2015.

[29] Michiel JW Jansen. Analysis of variance designs for model output. *Computer Physics Communications*, 117(1):35–43, 1999.

[30] S. Ji, Y. Xue, and L. Carin. Bayesian compressive sensing. *IEEE Transactions on Signal Processing*, 56(6):2346–2356, 2008.

[31] 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.

[32] S. Kucherenko, S. Tarantola, and P. Annoni. Estimation of global sensitivity indices for models with dependent variables. *Computer Physics Communications*, 183:937–946, 2012.

[33] O.P. Le Maître and O.M. Knio. *Spectral Methods for Uncertainty Quantification*. Springer, New York, NY, 2010.

[34] 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.

[35] 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.

[36] E.J. Nyström. Über die praktische auflösung von integralgleichungen mit anwendungen auf randwertaufgaben. *Acta Mathematica*, 54(1):185–204, 1930.

[37] J. Oakley and A. O'Hagan. Bayesian inference for the uncertainty distribution of computer model outputs. *Biometrika*, 89(4):769–784, 2002.

[38] Mark Orr. Introduction to radial basis function networks. *Technical Report, Center for Cognitive Science, University of Edinburgh*, 1996.

[39] Trevor Park and George Casella. The Bayesian Lasso. *Journal of the American Statistical Association*, 103(482):681–686, 2008.

[40] H. Rabitz, O. F. Alis, J. Shorter, and K. Shim. Efficient input-output model representations. *Comp. Phys. Comm.*, 117:11–20, 1999.

[41] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[42] M. Rosenblatt. Remarks on a multivariate transformation. *Annals of Mathematical Statistics*, 23(3):470 – 472, 1952.

[43] A. Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145:280–297, 2002.

[44] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. John Wiley & Sons, 2004.

[45] Andrea Saltelli, Paola Annoni, Ivano Azzini, Francesca Campolongo, Marco Ratto, and Stefano Tarantola. Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index. *Computer Physics Communications*, 181(2):259–270, 2010.

[46] 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.

[47] 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.

[48] K. Sargsyan, C. Safta, B. Debusschere, and H. Najm. Multiparameter spectral representation of noise-induced competence in Bacillus subtilis. *IEEE/ACM Trans. Comp. Biol. and Bioinf.*, 9(6):1709–1723, 2012.

[49] 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.

[50] D.W. Scott. *Multivariate Density Estimation. Theory, Practice and Visualization*. Wiley, New York, 1992.

[51] B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986.

[52] D. S. Sivia and J. Skilling. *Data Analysis: A Bayesian Tutorial, Second Edition*. Oxford University Press, 2006.

[53] D.S. Sivia. *Data Analysis: A Bayesian Tutorial*. Oxford Science, 1996.

[54] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics Dokl.*, 4:240–243, 1963.

[55] I. M. Sobol. Sensitivity estimates for nonlinear mathematical models. *Math. Modeling and Comput. Exper.*, 1:407–414, 1993.

[56] I. M. Sobol. Theorems and examples on high dimensional model representation. *Reliability Engineering and System Safety*, 79:187–193, 2003.

[57] Bruno Sudret. Global sensitivity analysis using polynomial chaos expansions. *Reliability Engineering & System Safety*, 93(7):964–979, 2008.

[58] M. E. Tipping. Sparse Bayesian learning and the relevance vector machine. *Journal of Machine Learning Research*, 1:211–244, 2001.

[59] M. E. Tipping and A. C. Faul. Fast marginal likelihood maximisation for sparse Bayesian models. In C. M. Bishop and J. Frey, editors, *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, 1991.

[60] D. Xiu and J. S. Hesthaven. High-order collocation methods for differential equations with random inputs. *SIAM J. Sci. Comp.*, 27(3):1118–1139, 2005.

[61] Dongbin Xiu. Efficient collocational approach for parametric uncertainty analysis. *Communications in computational physics*, 2(2):293–309, 2007.

[62] 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

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
CA Technical Library	8551	cateclib@sandia.gov

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.