hippylib2muq

Release 0.1.0

Ki-Tae Kim, Umberto Villa, Matthew Parno, Noemi Petra, Youssef M

Feb 18, 2022

CONTENTS

```
 _      ___ _____  _______    ___ _ _              __  __ _    _  ___
| |__ |_ _|  _ \|  _ \ \ / / (_) |__         |  \/  | | | |/ _ \
| '_ \ | || |_) | |_) \ V /| | | '_ \ ______| |\/| | | | | | | |
| | | || ||  __/|  __/ | | | | | |_) |______| |  | | |_| | |_| |
|_| |_|___|_|   |_|    |_| |_|_|_.__/       |_|  |_|\___/ \__\_\

Scalable Markov chain Monte Carlo Sampling Methods for Large-scale Bayesian
Inverse Problems Governed by PDEs
```

`hIPPYlib-MUQ` is a Python interface between two open source softwares, `hIPPYlib` and `MUQ`, which have complementary capabilities. hIPPYlib is an extensible software package aimed at solving deterministic and linearized Bayesian inverse problems governed by PDEs. MUQ is a collection of tools for solving uncertainty quantification problems. `hIPPYlib-MUQ` integrates these two libraries into a unique software framework, allowing users to implement the state-of-the-art Bayesian inversion algorithms in a seamless way.

Please look into example notebook below to get to know what `hIPPYlib-MUQ` does with step-by-step implementations.

CHAPTER

ONE

INSTALLATION

`hIPPYlib-MUQ` builds on hIPPYlib version 3.0.0 with FEniCS version 2019.1 and MUQ version 0.3.0. Installations of these packages are summarized here, but please see the detailed installation guides given in each github/bitbucket page.

Additional dependencies are

- jupyter, matplotlib (for tutorial notebooks)
- seaborn, statsmodels (for postprocessing)

1.1 Docker

We highly recommend to use our prebuilt Docker image, which is the easiest way to run `hIPPYlib-MUQ`. The docker image with the installation of all the dependencies is available here.

With Docker installed on your system, type:

```
docker run -ti --rm ktkimyu/hippylib2muq
```

Then, `hIPPYlib-MUQ` is available within the generated Docker container.

If you want to run `hIPPYlib-MUQ` using interactive notebooks, please type

```
docker run -ti --rm -p 8888:8888 ktkimyu/hippylib2muq 'jupyter-notebook --ip=0.0.0.0'
```

The notebook will be available at the following address in your web-browser. If you want to mount your local directory on docker container, add it with `-v` options, e.g., to mount your current directory on /home/fenics/shared/ in docker container, type

```
docker run -ti --rm -v $(pwd):/home/fenics/shared
           -p 8888:8888 ktkimyu/hippylib2muq 'jupyter-notebook --ip=0.0.0.0'
```

1.2 Conda

Conda is also a very convenient way to set up an enviroment to use hIPPYlib-MUQ. The script below builds a conda enviroment with FEniCS 2019 and MUQ. hIPPYlib 3.0.0 is also downloaded and installed via pip.

```
conda create -q -n hippylib2muq -c conda-forge fenics==2019.1.0 muq seaborn␣
↪statsmodels
conda activate hippylib2muq
git clone --depth 1 --branch 3.0.0 https://github.com/hippylib/hippylib.git
python hippylib/setup.py install
```

1.2.1 Installation of MUQ from source codes (Expert user/MUQ developers)

This requires cmake, the GNU Compiler Collection or Clang, and pybind11. On macOS, you can have these by installing Xcode Command Line Tools.

To compile and install MUQ, type

```
git clone https://bitbucket.org/mituq/muq2
cd muq2/build
cmake -DCMAKE_INSTALL_PREFIX=/your/muq2/install/directory -DMUQ_USE_PYTHON=ON ..
make
make install
```

Then Python static libraries are generated in /your/muq2/install/directory/lib folder.

You may append the path to this library folder, for example,

```
export PYTHONPATH=/your/muq2/install/directory/python:$PYTHONPATH
```

1.3 Build the hIPPYlib-MUQ documentation using Sphinx

You can build the documentation on your local machine by using sphinx (tested on version 2.3.0). Additional required packages are

- m2r
- sphinx_rtd_theme (current HTML theme)

If you want to use other HTML themes, install the corresponding package and modify the following line in conf.py in doc/source folder accordingly:

```
html_theme = 'name_of_the_theme'
```

All the packages above can be installed via pip or conda.

Once the required packages are installed, run make html from doc folder to build the documentation, then the document is available at doc/build/html/.

CHAPTER

TWO

CHANGELOG

2.1 Version 0.2.0, released on 11/18/2021

- Add two example python scripts
- Fix minor errors in `gaussian` module

2.2 Version 0.1.0, released on 11/12/2020

- Initial release under GPL3.

CHAPTER

THREE

HOW TO CONTRIBUTE

We welcome contributions at all levels: bugfixes, code improvements, simplifications, new capabilities, improved documentation, new examples/tutorials, etc.

Use a pull request (PR) toward the hippylib2muq:master branch to propose your contribution. If you are planning significant code changes, or have any questions, you should also open an issue before issuing a PR.

`hIPPYlib-MUQ` is an interface program between `hIPPYlib` and `MUQ`; contributions should be related to the interface; for contributing to each program, please refer to their own repositories (hIPPYlib and MUQ).

`hIPPYlib-MUQ` is maintained by https://github.com/hippylib, the main developer hub for the hIPPYlib project.

All new contributions must be made under the terms of the GPL3 license.

By submitting a pull request, you are affirming the Developer's Certificate of Origin at the end of this file.

3.1 Quick Summary

- Please create development branches off `hippylib2muq:master`.
- Please follow the *developer guidelines*, in particular with regards to documentation and code styling.
- Pull requests should be issued toward `hippylib2muq:master`. Make sure to check the items off the *Pull Request Checklist*.
- After approval, hIPPYlib-MUQ developers merge the PR in `hippylib2muq:master`.
- If you are also interested in the whole hIPPYlib project, we encourage you to join the hIPPYlib organization (see here); if you are interested in the whole MUQ project, please take a look at MUQ.
- Don't hesitate to *contact us* if you have any questions.

3.2 New Feature Development

- A new feature should be important enough that at least one person, the proposer, is willing to work on it and be its champion.
- The proposer creates a branch for the new feature (with suffix `-dev`), off the `master` branch, or another existing feature branch, for example:

```
# Clone assuming you have setup your ssh keys on GitHub:
git clone git@github.com:hippylib/hippylib2muq.git

# Alternatively, clone using the "https" protocol:
```

(continues on next page)

(continued from previous page)

```
git clone https://github.com/hippylib/hippylib2muq.git

# Create a new feature branch starting from "master":
git checkout master
git pull
git checkout -b feature-dev

# Work on "feature-dev", add local commits
# ...

# (One time only) push the branch to github and setup your local
# branch to track the github branch (for "git pull"):
git push -u origin feature-dev
```

- **We prefer that you create the new feature branch as a fork.** To allow hIPPYlib-MUQ developers to edit the PR, please enable upstream edits.
- The typical feature branch name is `new-feature-dev`, e.g. `optimal_exp_design-dev`. While not frequent in `hippylib2muq`, other suffixes are possible, e.g. `-fix`, `-doc`, etc.

3.3 Developer Guidelines

- *Keep the code lean and as simple as possible*
  - Well-designed simple code is frequently more general and powerful.
  - Lean code base is easier to understand by new collaborators.
  - New features should be added only if they are necessary or generally useful.
  - Code must be compatible with Python 3.
  - When adding new features add an example in the `example` or `application` folder and/or a new notebook in the `tutorial` folder.
  - The preferred way to export solutions for visualization in paraview is using `dl.XDMFFile`
  - The preferred way to save samples data is using h5py.
- *Keep the code general and reasonably efficient*
  - Main goal is fast prototyping for research.
  - When in doubt, generality wins over efficiency.
  - Respect the needs of different users (current and/or future).
- *Keep things separate and logically organized*
  - General usage features go in `hippylib2muq` (implemented in as much generality as possible), non-general features go into external apps/projects.
  - Inside `hippylib2muq`, compartmentalize between interface, mcmc, utility, etc.
  - Contributions that are project-specific or have external dependencies are allowed (if they are of broader interest), but should be `#ifdef`-ed and not change the code by default.
- Code specifics
  - All significant new classes, methods and functions have sphinx-style documentation in source comments.
  - Code styling should resemble existing code.

  - When manually resolving conflicts during a merge, make sure to mention the conflicted files in the commit message.

3.4 Pull Requests

- When your branch is ready for other developers to review / comment on the code, create a pull request towards `hippylib2muq:master`.
- Pull request typically have titles like:

  ```
  Description [new-feature-dev]
  ```

  for example:

  ```
  Bayesian Optimal Design of Experiments [oed-dev]
  ```

  Note the branch name suffix (in square brackets).
- Titles may contain a prefix in square brackets to emphasize the type of PR. Common choices are: `[DON'T MERGE]`, `[WIP]` and `[DISCUSS]`, for example:

  ```
  [DISCUSS] Bayesian Optimal Design of Experiments [oed-dev]
  ```
- Add a description, appropriate labels and assign yourself to the PR. The hIPPYlib team will add reviewers as appropriate.
- List outstanding TODO items in the description.

3.5 Pull Request Checklist

Before a PR can be merged, it should satisfy the following:

- [ ] Update `CHANGELOG`:
  - [ ] Is this a new feature users need to be aware of? New or updated application or tutorial?
  - [ ] Does it make sense to create a new section in the `CHANGELOG` to group with other related features?
- [ ] New examples/applications/tutorials:
  - [ ] All new examples/applications/tutorials run as expected.
- [ ] New capability:
  - [ ] All significant new classes, methods and functions have sphinx-style documentation in source comments.
  - [ ] Add new examples/applications/tutorials to highlight the new capability.
  - [ ] For new classes, functions, or modules, edit the corresponding `.rst` file in the `doc` folder.
  - [ ] If this is a major new feature, consider mentioning in the short summary inside `README` *(rare)*.
  - [ ] If this is a `C++` extension, the `package_data` dictionary in `setup.py` should include new files.

3.6 Contact Information

- Contact the hIPPYlib-MUQ team by posting to the GitHub issue tracker. Please perform a search to make sure your question has not been answered already.

3.7 Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Acknowledgement: This file is made based on CONTRIBUTING.md of hIPPYlib which used MFEM team contributing guidelines file as template.

CHAPTER

FOUR

BAYESIAN QUANTIFICATION OF PARAMETER UNCERTAINTY

4.1 I. Estimating the posterior pdf of the coefficient parameter field in an elliptic PDE

In this example we tackle the problem of quantifying the uncertainty in the solution of an inverse problem governed by an elliptic PDE via the Bayesian inference framework. Hence, we state the inverse problem as a problem of statistical inference over the space of uncertain parameters, which are to be inferred from data and a physical model. The resulting solution to the statistical inverse problem is a posterior distribution that assigns to any candidate set of parameter fields, our belief (expressed as a probability) that a member of this candidate set is the "true" parameter field that gave rise to the observed data.

4.1.1 Bayes's Theorem

The posterior probability distribution combines the prior pdf $\mu_{\text{prior}}(m)$ over the parameter space, which encodes any knowledge or assumptions about the parameter space that we may wish to impose before the data are considered, with a likelihood pdf $\pi_{\text{like}}(\ |\ m)$, which explicitly represents the probability that a given parameter m might give rise to the observed data $\in \mathbb{R}^{n_t}$, namely:

$$d\mu_{\text{post}}(m|) \propto \pi_{\text{like}}(\ |\ m)\, d\mu_{\text{prior}}(m). \tag{4.1}$$

Note that infinite-dimensional analog of Bayes's formula requires the use Radon-Nikodym derivatives instead of probability density functions.

The prior

We consider a Gaussian prior with mean m_{prior} and covariance , $\mu_{\text{prior}} \sim \mathcal{N}(m_{\text{prior}},)$. The covariance is given by the discretization of the inverse of differential operator $\mathcal{A}^{-2} = (-\gamma\Delta + \delta I)^{-2}$, where γ, $\delta > 0$ control the correlation length and the variance of the prior operator. This choice of prior ensures that it is a trace-class operator, guaranteeing bounded pointwise variance and a well-posed infinite-dimensional Bayesian inverse problem.

The likelihood

$$= \mathbf{f}(m) + \mathbf{e}, \quad \mathbf{e} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Gamma}_{\text{noise}})$$

$$\pi_{like}(\ |\ m) \propto \exp\left(-\tfrac{1}{2}\ \|\ \mathbf{f}(m) -\ \|^2_{\mathbf{\Gamma}^{-1}_{\text{noise}}}\right)$$

Here $\mathbf{f}$ is the parameter-to-observable map that takes a parameter m and maps it to the space observation vector .

In this application, $\mathbf{f}$ consists in the composition of a PDE solve (to compute the state u) and a pointwise observation of the state u to extract the observation vector .

The posterior

$$d\mu_{\text{post}}(m\ |\ ) \propto \exp\left(-\tfrac{1}{2}\ \|\ \mathbf{f}(m) -\ \|^2_{\mathbf{\Gamma}^{-1}_{\text{noise}}} - \tfrac{1}{2}\ \|\ m - m_{\text{prior}}\ \|^2_{-1}\right)$$

4.1.2 The Laplace approximation to the posterior: $\nu \sim \mathcal{N}(,)$

The mean of the Laplace approximation posterior distribution, , is the parameter maximizing the posterior, and is known as the maximum a posteriori (MAP) point. It can be found by minimizing the negative log of the posterior, which amounts to solving a deterministic inverse problem) with appropriately weighted norms,

$$:= \arg\min_m \mathcal{J}(m)\ :=\ \left(\frac{1}{2}\|\mathbf{f}(m) - \|^2_{\mathbf{\Gamma}^{-1}_{\text{noise}}} + \frac{1}{2}\|m - m_{\text{prior}}\|^2_{-1}\right).$$

The posterior covariance matrix is then given by the inverse of the Hessian matrix of $\mathcal{J}$ at , namely

$$= \left(()+^{-1}\right)^{-1},$$

provided that () is positive semidefinite.

The generalized eigenvalue problem

In what follows we denote with , , $\in \mathbb{R}^{n\times n}$ the matrices stemming from the discretization of the operators (), , with respect to the unweighted Euclidean inner product. Then we considered the symmetric generalized eigenvalue problem

$$V =^{-1} V\Lambda,$$

where $\Lambda = (\lambda_i) \in \mathbb{R}^{n\times n}$ contains the generalized eigenvalues and the columns of $V \in \mathbb{R}^{n\times n}$ the generalized eigenvectors such that $V^{T-1}V = I$.

Randomized eigensolvers to construct the approximate spectral decomposition

When the generalized eigenvalues $\{\lambda_i\}$ decay rapidly, we can extract a low-rank approximation of by retaining only the r largest eigenvalues and corresponding eigenvectors,

$$\approx^{-1} \Lambda_r^{T-1}.$$

Here, $\in \mathbb{R}^{n\times r}$ contains only the r generalized eigenvectors of that correspond to the r largest eigenvalues, which are assembled into the diagonal matrix $\Lambda_r = (\lambda_i) \in \mathbb{R}^{r\times r}$.

The approximate posterior covariance

Using the Sherman–Morrison–Woodbury formula, we write

$$= \left(+^{-1}\right)^{-1} =^{-1} -D_r^T + \mathcal{O}\left(\sum_{i=r+1}^{n} \frac{\lambda_i}{\lambda_i + 1}\right),$$

where $D_r := (\lambda_i/(\lambda_i + 1)) \in \mathbb{R}^{r\times r}$. The last term in this expression captures the error due to truncation in terms of the discarded eigenvalues; this provides a criterion for truncating the spectrum, namely that r is chosen such that λ_r is small relative to 1.

Therefore we can approximate the posterior covariance as

$$\approx -D_r^T$$

Drawing samples from a Gaussian distribution with covariance $Gpost$

Let $\mathbf{x}$ be a sample for the prior distribution, i.e. $\mathbf{x} \sim \mathcal{N}(\mathbf{0},)$, then, using the low rank approximation of the posterior covariance, we compute a sample $\mathbf{v} \sim \mathcal{N}(\mathbf{0},)$ as

$$\mathbf{v} = \left\{\left[(\Lambda_r+)^{-1/2} - \right]^{T-1} + \mathbf{I}\right\}\mathbf{x}$$

4.1.3 Full posterior sampling via Markov chain Monte Carlo (MCMC)

The posterior can be fully explored by using MCMC algorithms, the most popular method for sampling from a probability distribution. In this example, some of the advanced MCMC algorithms are considered and compared in terms of efficiency and accuracy.

The preconditioned Crank-Nicolson algorithm (pCN)

The pCN algorithm is perhaps the simplest MCMC method that is well-defined in the infinite dimensional setting ensuring a mixing rates independent of the dimension of the discretized parameter space.

The algorithm proceeds as follows (see *[Cotter et al. (2013)] [Pinski et al. (2015)]* for the details):

1. Given $m^{(k)}$, propose $v^{(k+1)} = m_{\text{prop}} + \sqrt{1-\beta^2}(m^{(k)} - m_{\text{prop}}) + \beta\xi^{(k)}, \quad \xi^{(k)} \sim \mathcal{N}(0, \mathcal{C}_{\text{prop}})$
2. Set $m^{(k+1)} = v^{(k+1)}$ with probability $a(m^{(k)}, v^{(k+1)}) = \min\left(1, \frac{\mu_{\text{post}}(v^{(k+1)})q(v^{(k+1)}, m^{(k)})}{\mu_{\text{post}}(m^{(k)})q(m^{(k)}, v^{(k+1)})}\right)$

where $q(m, v) \sim \mathcal{N}\left(m_{\text{prop}} + \sqrt{1-\beta^2}(m - m_{\text{prop}}), \beta^2\mathcal{C}_{\text{prop}}\right)$ with proposal mean m_{prop} and covariance $\mathcal{C}_{\text{prop}}$ and β is a parameter controlling the step length of the proposal.

The preconditioned Metropolis adjusted Langevin algorithm (MALA)

The MALA algorithm is built on two mechanisms: the overdamped Langevin diffusion to propose a move and the Metropolis–Hastings algorithm to accept or reject the proposal move *[Roberts and Tweedie (1996)]*.

The preconditioned MALA algorithm is described as follows:

1. Given $m^{(k)}$, propose $v^{(k+1)} = m^{(k)} + \tau\mathcal{A}_{\text{prop}}\nabla \log \mu_{\text{post}}(m^{(k)}) + \sqrt{2\tau\mathcal{A}_{\text{prop}}}\xi^{(k)}, \quad \xi^{(k)} \sim \mathcal{N}(0, \mathcal{I})$
2. Set $m^{(k+1)} = v^{(k+1)}$ with probability $a(m^{(k)}, v^{(k+1)}) = \min\left(1, \frac{\mu_{\text{post}}(v^{(k+1)})q(v^{(k+1)}, m^{(k)})}{\mu_{\text{post}}(m^{(k)})q(m^{(k)}, v^{(k+1)})}\right)$

where $q(m, v) \sim \mathcal{N}\left(m + \tau\mathcal{A}_{\text{prop}}\nabla \log \mu_{\text{post}}(m), 2\tau\mathcal{A}_{\text{prop}}\right)$ with a proposal covariance $\mathcal{A}_{\text{prop}}$ and τ is a step size.

The Delayed Rejection (DR)

The basic idea of the delayed rejection is to use a sequence of stages in each iteration. Unlike the basic Metropolis-Hastings algorithm, if a candidate is rejected, a new move is proposed. The acceptance rate for the new proposal move is adjusted so that the stationary distribution is preserved. For the details, see *[Mira (2001)]*.

4.1.4 This tutorial shows

- Definition of the component of an inverse problem (the forward problem, the prior, and the misfit functional) using hIPPYlib
- Computation of the maximum a posterior MAP point using inexact Newton-CG algorithm
- Low-rank based approximation of the posterior covariance under the Laplace Approximation
- Sampling from the prior distribution and Laplace Approximation using hIPPYlib
- Construction of a MUQ workgraph using a PDE model defined in hIPPYlib
- Exploring the full posterior using the MCMC methods implemented in MUQ
- Convergence diagnostics of MCMC simulation results and their comparison

4.1.5 Mathematical tools used

- Finite element method
- Derivation of gradient and Hessian via the adjoint method
- Inexact Newton-CG
- Randomized eigensolvers
- Bayes' formula
- MCMC methods

4.1.6 List of software used

, and their interfaces are the main software framework in this tutorial. Additional tools used are:

- , A parallel finite element element library for the discretization of partial differential equations
- , A set of data structures and routines for scalable and efficient linear algebra operations and solvers
- , A python package for linear algebra
- , A python package for visualizing the results

4.1.7 References

. MCMC methods for functions: modifying old algorithms to make them faster. Statistical Science, 424-446.

. Algorithms for Kullback–Leibler approximation of probability measures in infinite dimensions. SIAM Journal on Scientific Computing, 37(6), A2733-A2757.

. Exponential convergence of Langevin distributions and their discrete approximations. Bernoulli, 2(4), 341-363.

. On Metropolis-Hastings algorithms with delayed rejection. Metron, 59(3-4), 231-241.

4.2 II. hIPPYlib-MUQ integration

The main objective of this example is to illustrate the interface between and .

We make use of to

- Define the forward model, prior distribution, and likelihood function
- Compute the MAP point by solving a deterministic inverse problem
- Construct the Laplace Approximation to the posterior distribution with a low-rank based approximation of the covariace operator.

The main classes and functions of employed in this example are

- `hippylib::PDEVariationalProblem` : forward, adjoint and incremental problems solvers and their derivatives evaluations
- `hippylib::BiLaplacianPrior` : a biLaplacian Gaussian prior model
- `hippylib::GaussianLRPosterior` : the low rank Gaussian approximation of the posterior (used for generating starting points of MCMC simulations)

is used to sample from the posterior by implementing MCMC methods with various kernels and proposals.

The main classes and functions used here are

- `muq.Modeling::PyModPiece` : an abstract interface for defining vector-valued models
- `muq.Modeling::PyGaussianBase` : an abstract interface for implementing Gaussian distributions
- `muq.Modeling::WorkGraph` : a graph or a frame of connected `muq.Modeling::PyModPiece` (or `muq.Modeling::WorkPiece`) classes
- `muq.SamplingAlgorithms::CrankNicolsonProposal` : the pCN proposal
- `muq.SamplingAlgorithms::MALAProposal` : the MALA proposal
- `muq.SamplingAlgorithms::MHKernel` : the Metropolis-Hastings transition kernel
- `muq.SamplingAlgorithms::DRKernel` : the delayed rejection kernel
- `muq.SamplingAlgorithms::SingleChainMCMC` : a single chain MCMC sampler

To interface and for this example, `hippylib2muq` provides the following classes:

- `hippylib2muq::Param2LogLikelihood` : a child of `muq::PyModPiece` which wraps `hippylib::PDEVariationalProblem` and `hippylib:PointwiseStateObservation` (solving the forward problem, mapping from parameters to log likelihood and evaluating its derivative)
- `hippylib2muq::BiLaplaceGaussian` : a child of `muq.Modeling::PyGaussianBase` which wraps `hippylib::BiLaplacianPrior`

- `hippylib2muq::LAPosteriorGaussian` : a child of `muq.Modeling::PyGaussianBase` which wraps `hippylib::GaussianLRPosterior`

4.3 III. Implementation

4.3.1 1. Load modules

```
from __future__ import absolute_import, division, print_function

import math
import matplotlib.pyplot as plt
%matplotlib inline

import muq.Modeling_ as mm
import muq.SamplingAlgorithms as ms

import dolfin as dl
import hippylib as hp

import hippylib2muq as hm
import numpy as np

import logging
logging.getLogger('FFC').setLevel(logging.WARNING)
logging.getLogger('UFL').setLevel(logging.WARNING)
dl.set_log_active(False)

np.random.seed(seed=1)
```

4.3.2 2. Generate the true parameter

This function generates a random field with a prescribed anisotropic covariance function.

```
def true_model(prior):
    noise = dl.Vector()
    prior.init_vector(noise, "noise")
    hp.parRandom.normal(1., noise)
    mtrue = dl.Vector()
    prior.init_vector(mtrue, 0)
    prior.sample(noise, mtrue)
    return mtrue
```

4.3.3 3. Set up the mesh and finite element spaces

We compute a two dimensional mesh of a unit square with nx by ny elements. We define a P2 finite element space for the *state* and *adjoint* variable and P1 for the *parameter*.

```
ndim = 2
nx   = 32
ny   = 32
mesh = dl.UnitSquareMesh(nx, ny)
```

(continues on next page)

(continued from previous page)

```
Vh2 = dl.FunctionSpace(mesh, 'Lagrange', 2)
Vh1 = dl.FunctionSpace(mesh, 'Lagrange', 1)
Vh  = [Vh2, Vh1, Vh2]
print("Number of dofs: STATE={0}, PARAMETER={1}, ADJOINT={2}".format(
      Vh[hp.STATE].dim(), Vh[hp.PARAMETER].dim(), Vh[hp.ADJOINT].dim()) )
```

```
Number of dofs: STATE=4225, PARAMETER=1089, ADJOINT=4225
```

4.3.4 4. Set up the forward problem

Let Ω be the unit square in $\mathbb{R}^2$, and Γ_D, Γ_N be the Dirichlet and Neumann portitions of the boundary $\partial\Omega$ (that is $\Gamma_D \cup \Gamma_N = \partial\Omega$, $\Gamma_D \cap \Gamma_N = \emptyset$). The forward problem reads

$$\begin{cases} \nabla \cdot (e^m \nabla u) = f & \text{in } \Omega \\ u = u_D & \text{on } \Gamma_D, \\ e^m \nabla u \cdot \boldsymbol{n} = 0 & \text{on } \Gamma_N, \end{cases}$$

where $u \in \mathcal{V}$ is the state variable, and $m \in \mathcal{M}$ is the uncertain parameter. Here Γ_D corresponds to the top and bottom sides of the unit square, and Γ_N corresponds to the left and right sides. We also let $f = 0$, and $u_D = 1$ on the top boundary and $u_D = 0$ on the bottom boundary.

To set up the forward problem we use the `hp::PDEVariationalProblem` class, which requires the following inputs

- the finite element spaces for the state, parameter, and adjoint variables `Vh`
- the pde in weak form `pde_varf`
- the boundary conditions `bc` for the forward problem and `bc0` for the adjoint and incremental problems.

The `hp::PDEVariationalProblem` class offer the following functionality:

- solving the forward/adjoint and incremental problems
- evaluate first and second partial derivative of the forward problem with respect to the state, parameter, and adojnt variables.

```
def u_boundary(x, on_boundary):
    return on_boundary and ( x[1] < dl.DOLFIN_EPS or x[1] > 1.0 - dl.DOLFIN_EPS)

u_bdr  = dl.Expression("x[1]", degree=1)
u_bdr0 = dl.Constant(0.0)
bc  = dl.DirichletBC(Vh[hp.STATE], u_bdr, u_boundary)
bc0 = dl.DirichletBC(Vh[hp.STATE], u_bdr0, u_boundary)

f = dl.Constant(0.0)

def pde_varf(u,m,p):
    return dl.exp(m)*dl.inner(dl.nabla_grad(u), dl.nabla_grad(p))*dl.dx - f*p*dl.dx

pde = hp.PDEVariationalProblem(Vh, pde_varf, bc, bc0, is_fwd_linear=True)
```

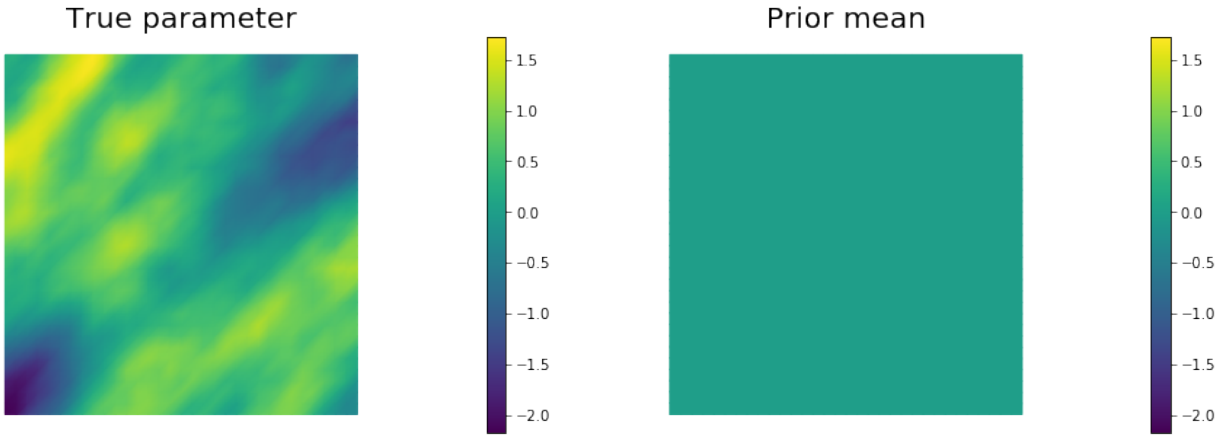
4.3.5 5. Set up the prior

To obtain the synthetic true parameter m_{true} we generate a realization from the prior distribution.

Here we assume a Gaussian prior, $\mu_{\text{prior}} \sim \mathcal{N}(0,)$ with zero mean and covariance matrix $= \mathcal{A}^{-2}$, which is implemented by `hp::BiLaplacianPrior` class that provides methods to apply the regularization (precision) operator to a vector or to apply the prior covariance operator.

```
gamma = .1
delta = .5

theta0 = 2.
theta1 = .5
alpha  = math.pi/4

anis_diff = dl.CompiledExpression(hp.ExpressionModule.AnisTensor2D(), degree = 1)
anis_diff.set(theta0, theta1, alpha)

prior = hp.BiLaplacianPrior(Vh[hp.PARAMETER], gamma, delta, anis_diff, robin_bc=True)
print("Prior regularization: (delta_x - gamma*Laplacian)^order: "
      "delta={0}, gamma={1}, order={2}".format(delta, gamma,2))

mtrue = true_model(prior)

objs = [dl.Function(Vh[hp.PARAMETER],mtrue), dl.Function(Vh[hp.PARAMETER],prior.mean)]
mytitles = ["True parameter", "Prior mean"]
hp.nb.multi1_plot(objs, mytitles)
plt.show()
```

```
Prior regularization: (delta_x - gamma*Laplacian)^order: delta=0.5, gamma=0.1, order=2
```



4.3.6 6. Set up the likelihood and generate synthetic observations

To setup the observation operator $\mathcal{B} : \mathcal{V} \mapsto \mathbb{R}^{n_t}$, we generate n_t (`ntargets` in the code below) random locations where to evaluate the value of the state.

Under the assumption of Gaussian additive noise, the likelihood function π_{like} has the form

$$\pi_{\text{like}}(\,|\,m) \propto \exp\left(-\frac{1}{2}\|\mathcal{B}\,u(m) - \|^2_{\Gamma^{-1}_{\text{noise}}}\right),$$

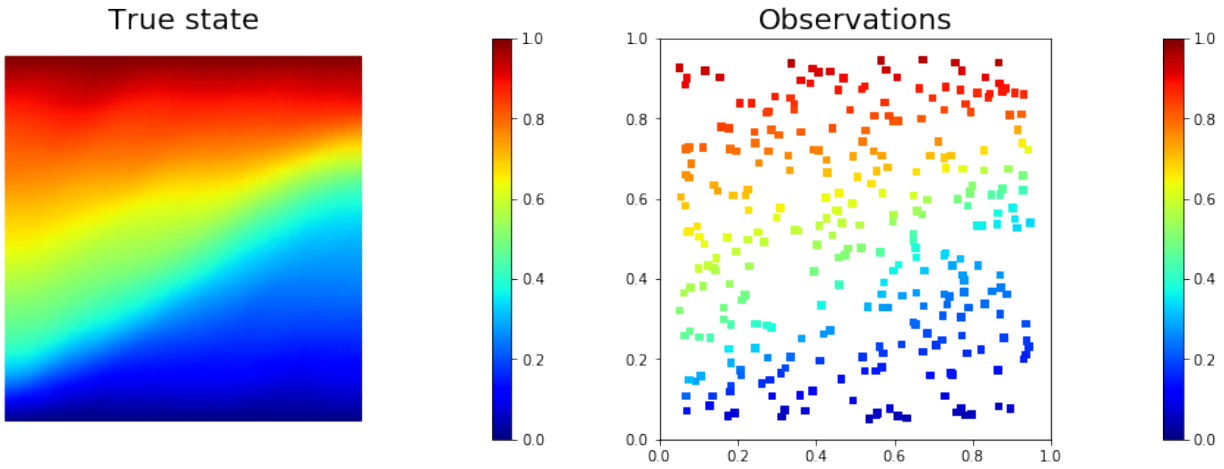
where $u(m)$ denotes the solution of the forward model at a given parameter m.

The class `hp::PointwiseStateObservation` implements the evaluation of the log-likelihood function and of its partial derivatives w.r.t. the state u and parameter m.

To generate the synthetic observation, we first solve the forward problem using the true parameter m_{true}. Synthetic observations are obtained by perturbing the state variable at the observation points with a random Gaussian noise. `rel_noise` is the signal to noise ratio.

```
ntargets  = 300
rel_noise = 0.005
targets   = np.random.uniform(0.05, 0.95, [ntargets, ndim])
print("Number of observation points: {0}".format(ntargets))

misfit = hp.PointwiseStateObservation(Vh[hp.STATE], targets)

utrue = pde.generate_state()
x = [utrue, mtrue, None]
pde.solveFwd(x[hp.STATE], x)
misfit.B.mult(x[hp.STATE], misfit.d)
MAX = misfit.d.norm("linf")
noise_std_dev = rel_noise * MAX
hp.parRandom.normal_perturb(noise_std_dev, misfit.d)
misfit.noise_variance = noise_std_dev*noise_std_dev

model = hp.Model(pde, prior, misfit)

vmax = max( utrue.max(), misfit.d.max() )
vmin = min( utrue.min(), misfit.d.min() )

plt.figure(figsize=(15,5))
hp.nb.plot(dl.Function(Vh[hp.STATE], utrue), mytitle="True state",
           subplot_loc=121, vmin=vmin, vmax=vmax, cmap="jet")
hp.nb.plot_pts(targets, misfit.d, mytitle="Observations",
               subplot_loc=122, vmin=vmin, vmax=vmax, cmap="jet")
plt.show()
```

```
Number of observation points: 300
```



4.3.7 7. Compute the MAP point

We used the globalized Newtown-CG method to compute the MAP point.

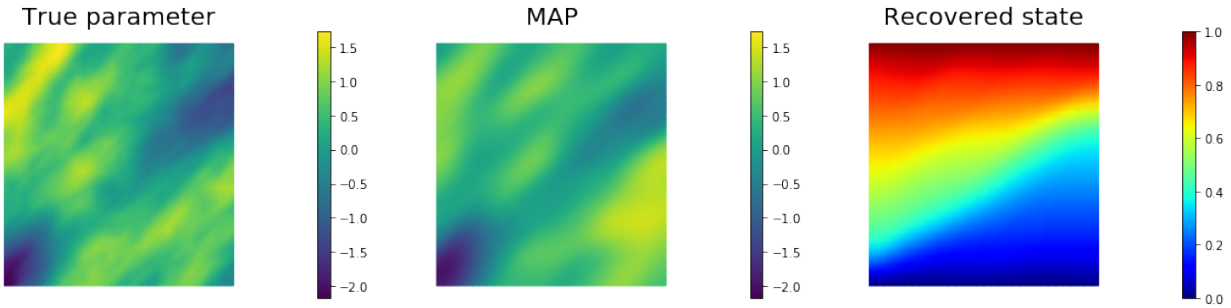
```
m = prior.mean.copy()
solver = hp.ReducedSpaceNewtonCG(model)
solver.parameters["rel_tolerance"]  = 1e-6
solver.parameters["abs_tolerance"]  = 1e-12
solver.parameters["max_iter"]       = 25
solver.parameters["GN_iter"]        = 5
solver.parameters["globalization"]  = "LS"
solver.parameters["LS"]["c_armijo"] = 1e-4

x = solver.solve([None, m, None])

if solver.converged:
    print( "\nConverged in ", solver.it, " iterations.")
else:
    print( "\nNot Converged")

print( "Termination reason:  ", solver.termination_reasons[solver.reason] )
print( "Final gradient norm: ", solver.final_grad_norm )
print( "Final cost:          ", solver.final_cost )

plt.figure(figsize=(18,4))
mtrue_min = dl.Function(Vh[hp.PARAMETER],mtrue).vector().min()
mtrue_max = dl.Function(Vh[hp.PARAMETER],mtrue).vector().max()
hp.nb.plot(dl.Function(Vh[hp.PARAMETER],mtrue), subplot_loc=131, mytitle="True␣
↪parameter",
           vmin=mtrue_min, vmax=mtrue_max)
hp.nb.plot(dl.Function(Vh[hp.PARAMETER], x[hp.PARAMETER]), subplot_loc=132,mytitle=
↪"MAP",
           vmin=mtrue_min, vmax=mtrue_max)
hp.nb.plot(dl.Function(Vh[hp.STATE], x[hp.STATE]), subplot_loc=133,mytitle="Recovered␣
↪state", cmap="jet")
plt.show()
```

```
It  cg_it   cost        misfit      reg         (g,dm)      ||g||L2     alpha       tolcg
  1     2   6.95e+03    6.94e+03    9.30e-01   -7.91e+04    1.78e+05    1.00e+00    5.00e-01
```

(continues on next page)

(continued from previous page)

```
  2      3    2.81e+03    2.81e+03    1.36e+00   -8.27e+03    5.53e+04    1.00e+00    5.00e-01
  3      4    9.25e+02    9.21e+02    3.58e+00   -3.70e+03    2.53e+04    1.00e+00    3.77e-01
  4     10    3.39e+02    3.30e+02    9.32e+00   -1.30e+03    9.45e+03    1.00e+00    2.30e-01
  5      1    2.71e+02    2.61e+02    9.32e+00   -1.37e+02    1.41e+04    1.00e+00    2.81e-01
  6     13    1.73e+02    1.56e+02    1.63e+01   -1.96e+02    3.73e+03    1.00e+00    1.45e-01
  7     16    1.44e+02    1.20e+02    2.40e+01   -5.63e+01    1.78e+03    1.00e+00    1.00e-01
  8     12    1.41e+02    1.16e+02    2.45e+01   -6.92e+00    1.14e+03    1.00e+00    8.01e-02
  9     43    1.34e+02    9.87e+01    3.50e+01   -1.47e+01    8.67e+02    1.00e+00    6.98e-02
 10      3    1.34e+02    9.86e+01    3.50e+01   -1.31e-01    3.77e+02    1.00e+00    4.60e-02
 11     42    1.34e+02    9.85e+01    3.51e+01   -8.26e-02    8.90e+01    1.00e+00    2.24e-02
 12     59    1.34e+02    9.85e+01    3.51e+01   -7.70e-04    8.86e+00    1.00e+00    7.06e-03

Converged in  12  iterations.
Termination reason:   Norm of the gradient less than tolerance
Final gradient norm:  0.10248108935885225
Final cost:           133.60199155509807
```



4.3.8 8. Compute the low-rank based Laplace approximation of the posterior (LA-posterior)

We used the *double pass* algorithm to compute a low-rank decomposition of the Hessian Misfit. In particular, we solve

$$\mathbf{v}_i = \lambda_i^{-1}\mathbf{v}_i.$$

The effective rank of the Hessian misfit is the number of eigenvalues above the red line ($y = 1$). The effective rank is independent of the mesh size.

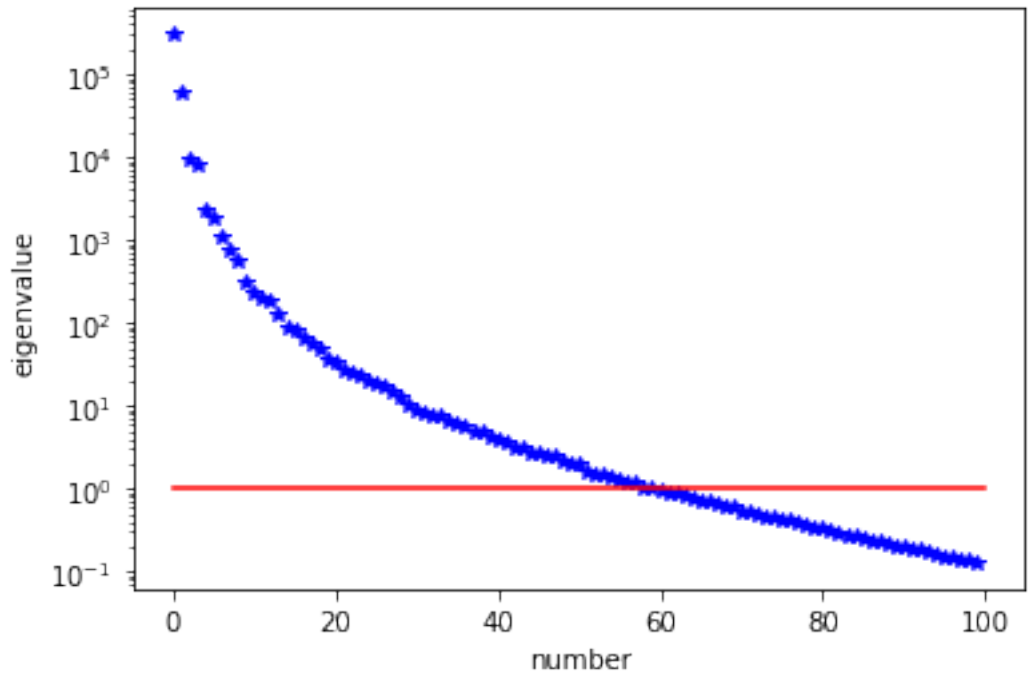
```
model.setPointForHessianEvaluations(x, gauss_newton_approx=False)
Hmisfit = hp.ReducedHessian(model, misfit_only=True)
k = 100
p = 20
print( "Single/Double Pass Algorithm. Requested eigenvectors: "\
       "{0}; Oversampling {1}.".format(k,p) )

Omega = hp.MultiVector(x[hp.PARAMETER], k+p)
hp.parRandom.normal(1., Omega)
lmbda, V = hp.doublePassG(Hmisfit, prior.R, prior.Rsolver, Omega, k)

nu = hp.GaussianLRPosterior(prior, lmbda, V)
nu.mean = x[hp.PARAMETER]

plt.plot(range(0,k), lmbda, 'b*', range(0,k+1), np.ones(k+1), '-r')
plt.yscale('log')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('number')
plt.ylabel('eigenvalue')
plt.show()
```

```
Single/Double Pass Algorithm. Requested eigenvectors: 100; Oversampling 20.
```



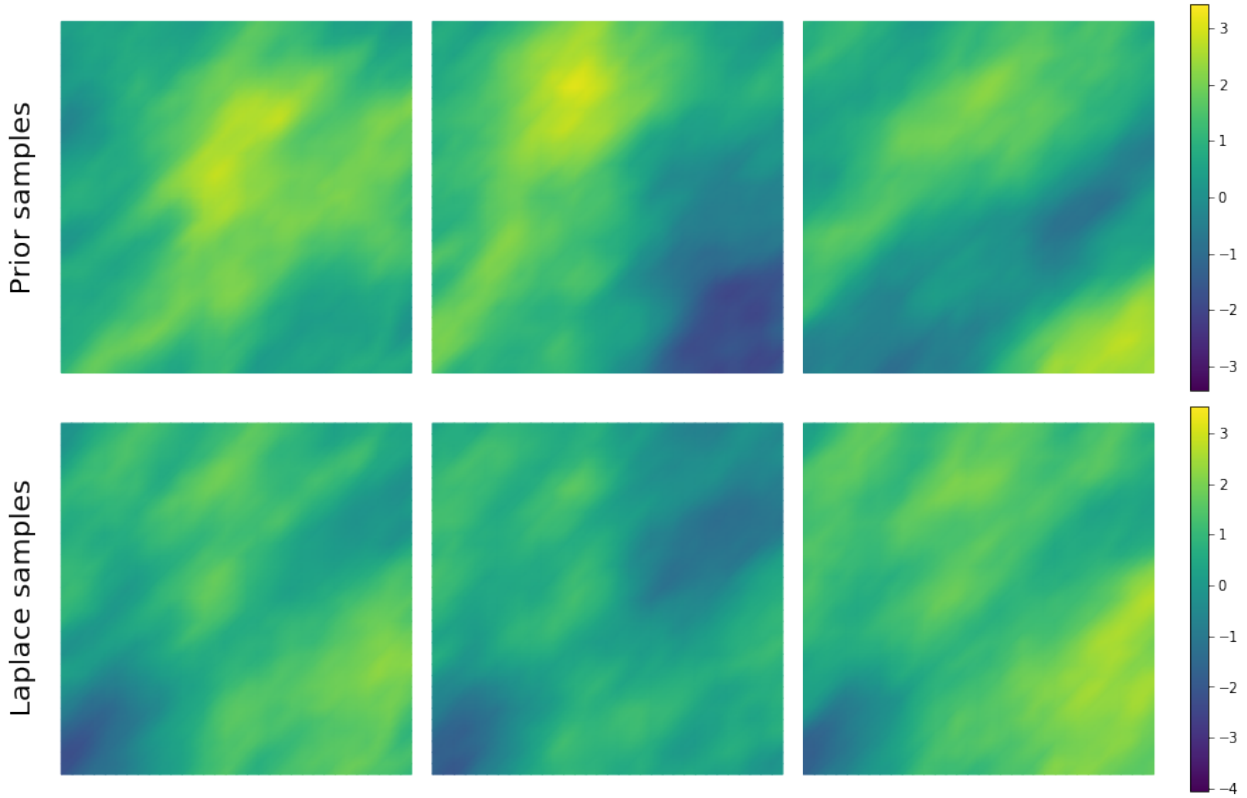
4.3.9 9. Drawing samples from the prior distribution and Laplace Approximation

```
nsamples = 3
noise = dl.Vector()
nu.init_vector(noise,"noise")
s_prior = dl.Function(Vh[hp.PARAMETER], name="sample_prior")
s_post = dl.Function(Vh[hp.PARAMETER], name="sample_post")

post_pw_variance, pr_pw_variance, corr_pw_variance =\
    nu.pointwise_variance(method="Exact")

pr_max   =  2.5*math.sqrt( pr_pw_variance.max() ) + prior.mean.max()
pr_min   = -2.5*math.sqrt( pr_pw_variance.max() ) + prior.mean.min()
ps_max   =  2.5*math.sqrt( post_pw_variance.max() ) + nu.mean.max()
ps_min   = -2.5*math.sqrt( post_pw_variance.max() ) + nu.mean.min()

fig = plt.figure(figsize=(18,8))
for i in range(nsamples):
    hp.parRandom.normal(1., noise)
    nu.sample(noise, s_prior.vector(), s_post.vector())

    impr = hp.nb.plot(s_prior, subplot_loc=231+i, vmin=pr_min, vmax=pr_max,␣
↪colorbar=None)
    imps = hp.nb.plot(s_post, subplot_loc=234+i, vmin=ps_min, vmax=ps_max,␣
↪colorbar=None)
```

(continues on next page)

(continued from previous page)

```
fig.tight_layout()
fig.subplots_adjust(left=0.15, right=0.8)
pos_impr = impr.axes.get_position().get_points()
pos_imps = imps.axes.get_position().get_points()
height_im = impr.axes.get_position().size[1]
cbaxes_pr = fig.add_axes([pos_impr[1,0]+0.01, pos_impr[0,1], 0.01, height_im])
cbaxes_ps = fig.add_axes([pos_imps[1,0]+0.01, pos_imps[0,1], 0.01, height_im])
fig.colorbar(impr, cbaxes_pr)
fig.colorbar(imps, cbaxes_ps)
fig.text(0.15, pos_impr[0,1]+0.125, 'Prior samples', fontsize=20, rotation=90)
fig.text(0.15, pos_imps[0,1]+0.1, 'Laplace samples', fontsize=20, rotation=90)
plt.show()
```



4.3.10 10 Define a quantify of interest

As a quantity of interest, we consider the log of the flux through the bottom boundary:

$$q(m) = \ln\left\{\int_{\Gamma_b} e^m \nabla u \cdot \mathbf{n}\, ds\right\},$$

where the state variable u denotes the pressure, and $\mathbf{n}$ is the unit normal vector to Γ_b (the bottom boundary of the domain).

```
class FluxQOI(object):
    def __init__(self, Vh, dsGamma):
        self.Vh = Vh
```

(continues on next page)

(continued from previous page)

```
        self.dsGamma = dsGamma
        self.n = dl.Constant((0.,1.))

        self.u = None
        self.m = None
        self.L = {}

    def form(self, x):
        return dl.exp(x[hp.PARAMETER])*dl.dot( dl.grad(x[hp.STATE]), self.n)*self.
↪dsGamma

    def eval(self, x):
        u = hp.vector2Function(x[hp.STATE], self.Vh[hp.STATE])
        m = hp.vector2Function(x[hp.PARAMETER], self.Vh[hp.PARAMETER])
        return np.log( dl.assemble(self.form([u,m])) )

class GammaBottom(dl.SubDomain):
    def inside(self, x, on_boundary):
        return ( abs(x[1]) < dl.DOLFIN_EPS )

GC = GammaBottom()
marker = dl.MeshFunction("size_t", mesh, 1)
marker.set_all(0)
GC.mark(marker, 1)
dss = dl.Measure("ds", subdomain_data=marker)
qoi = FluxQOI(Vh,dss(1))
```

4.3.11 11. Exploring the posterior using MCMC methods

Define the parameter-to-observable map in MUQ

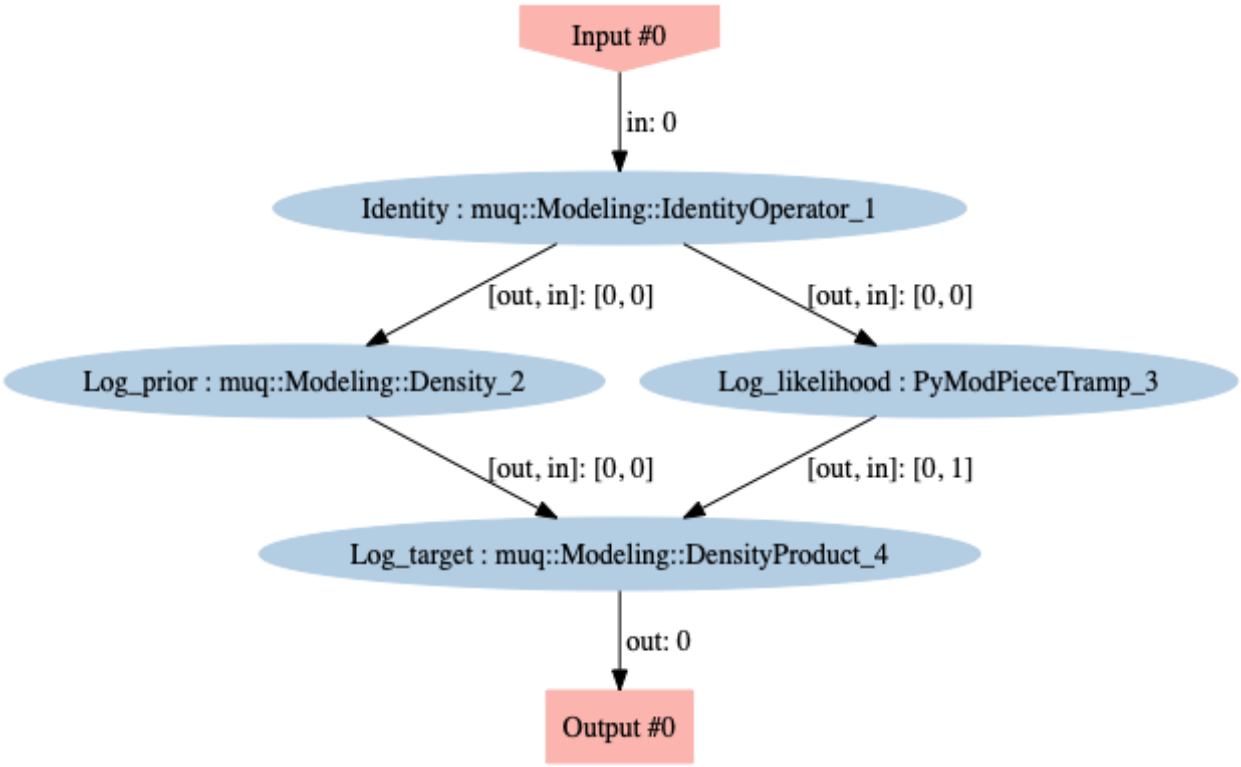
Overall, we want a mapping from parameter coefficients vector to the log target, $J(m) = -\frac{1}{2} \parallel \mathbf{f}(m) - \parallel^2_{\mathbf{\Gamma}^{-1}_{\text{noise}}} - \frac{1}{2} \parallel m - m_{\text{prior}} \parallel^2_{\mathcal{C}^{-1}}$. To do so, we generate a MUQ WorkGraph of connected ModPieces.

```
# a place holder ModPiece for the parameters
idparam = mm.IdentityOperator(Vh[hp.PARAMETER].dim())

# log Gaussian Prior ModPiece
gaussprior = hm.BiLaplaceGaussian(prior)
log_gaussprior = gaussprior.AsDensity()

# parameter to log likelihood Modpiece
param2loglikelihood = hm.Param2LogLikelihood(model)

# log target ModPiece
log_target = mm.DensityProduct(2)

workgraph = mm.WorkGraph()

# Identity operator for the parameters
workgraph.AddNode(idparam, 'Identity')

# Prior model
workgraph.AddNode(log_gaussprior, "Log_prior")
```

(continues on next page)

(continued from previous page)

```
# Likelihood model
workgraph.AddNode(param2loglikelihood, "Log_likelihood")

# Posterior
workgraph.AddNode(log_target, "Log_target")

workgraph.AddEdge("Identity", 0, "Log_prior", 0)
workgraph.AddEdge("Log_prior", 0, "Log_target", 0)

workgraph.AddEdge("Identity", 0, "Log_likelihood", 0)
workgraph.AddEdge("Log_likelihood", 0, "Log_target", 1)

workgraph.Visualize("workgraph.png")

# Construct the problem
problem = ms.SamplingProblem(workgraph.CreateModPiece("Log_target"))
```



Set up MCMC methods

We run five different MCMC methods:

- **pCN**: Metropolis-Hastings kernel + pCN proposal with $m_{\text{prop}} = m_{\text{prior}} = 0$ and $\mathcal{C}_{\text{prop}} =$
- **MALA**: Metropolis-Hastings kernel + MALA proposal with $\mathcal{A}_{\text{prop}} =$
- **h-pCN**: Metropolis-Hastings kernel + pCN proposal with $m_{\text{prop}} =$ and $\mathcal{C}_{\text{prop}} =$
- **h-MALA**: Metropolis-Hastings kernel + MALA proposal with $\mathcal{A}_{\text{prop}} =$

- **DR (h-pCN/h-MALA)**: Delayed rejection kernel + two stage proposals (h-pCN proposal as first stage and h-MALA proposal as second stage)

where is the covariance of the LA-posterior.

We set the value of parameters (β for pCN and τ for MALA) such that the acceptance rates are about 20-35% and 50-60% for pCN and MALA, respectively.

```
# Construct options for MH kernel and MCMC sampler
options = dict()
options['NumSamples'] = 22000  # Number of MCMC steps to take
options['BurnIn'] = 2000  # Number of steps to throw away as burn in
options['PrintLevel'] = 0  # in {0,1,2,3} Verbosity of the output

method_list = dict()

# pCN
opts = options.copy()
opts.update( {'Beta':0.005} )
gauss_pcn = hm.BiLaplaceGaussian(prior)
prop = ms.CrankNicolsonProposal(opts, problem, gauss_pcn)
kern = ms.MHKernel(opts, problem, prop)
sampler = ms.SingleChainMCMC(opts, [kern])

method_list['pCN'] = {'Options': opts, 'Sampler': sampler}

# MALA
opts = options.copy()
opts.update( {'StepSize':0.000006} )
gauss_mala = hm.BiLaplaceGaussian(prior, use_zero_mean=True)
prop = ms.MALAProposal(opts, problem, gauss_mala)
kern = ms.MHKernel(opts, problem, prop)
sampler = ms.SingleChainMCMC(opts, [kern])

method_list['MALA'] = {'Options': opts, 'Sampler': sampler}

# h-pCN
opts = options.copy()
opts.update( {'Beta':0.55} )
gauss_hpcn = hm.LAPosteriorGaussian(nu)
prop = ms.CrankNicolsonProposal(opts, problem, gauss_hpcn)
kern = ms.MHKernel(opts, problem, prop)
sampler = ms.SingleChainMCMC(opts, [kern])

method_list['h-pCN'] = {'Options': opts, 'Sampler': sampler}

# h-MALA
opts = options.copy()
opts.update( {'StepSize':0.1} )
gauss_hmala = hm.LAPosteriorGaussian(nu, use_zero_mean=True)
prop = ms.MALAProposal(opts, problem, gauss_hmala)
kern = ms.MHKernel(opts, problem, prop)
sampler = ms.SingleChainMCMC(opts, [kern])

method_list['h-MALA'] = {'Options': opts, 'Sampler': sampler}

# DR (h-pCN/h-MALA)
opts = options.copy()
opts.update( {'Beta':1.0, 'StepSize':0.1} )
```

(continues on next page)

(continued from previous page)

```
gauss_dr1 = hm.LAPosteriorGaussian(nu)
gauss_dr2 = hm.LAPosteriorGaussian(nu, use_zero_mean=True)
prop1 = ms.CrankNicolsonProposal(opts, problem, gauss_dr1)
prop2 = ms.MALAProposal(opts, problem, gauss_dr2)
kern = ms.DRKernel( opts, problem, [prop1, prop2], [1.0, 1.0] )
sampler = ms.SingleChainMCMC(opts, [kern])

method_list['DR (h-pCN/h-MALA)'] = {'Options': opts, 'Sampler': sampler}

hm.print_methodDict(method_list)
```

```
Method              Kernel     Proposal   Beta or Step-size
-------------------------------------------------------------
pCN                 mh         pcn           5.0e-03
MALA                mh         mala          6.0e-06
h-pCN               mh         pcn           5.5e-01
h-MALA              mh         mala          1.0e-01
DR (h-pCN/h-MALA)   dr         pcn           1.0e+00
                               mala          1.0e-01
```

Run MCMC methods

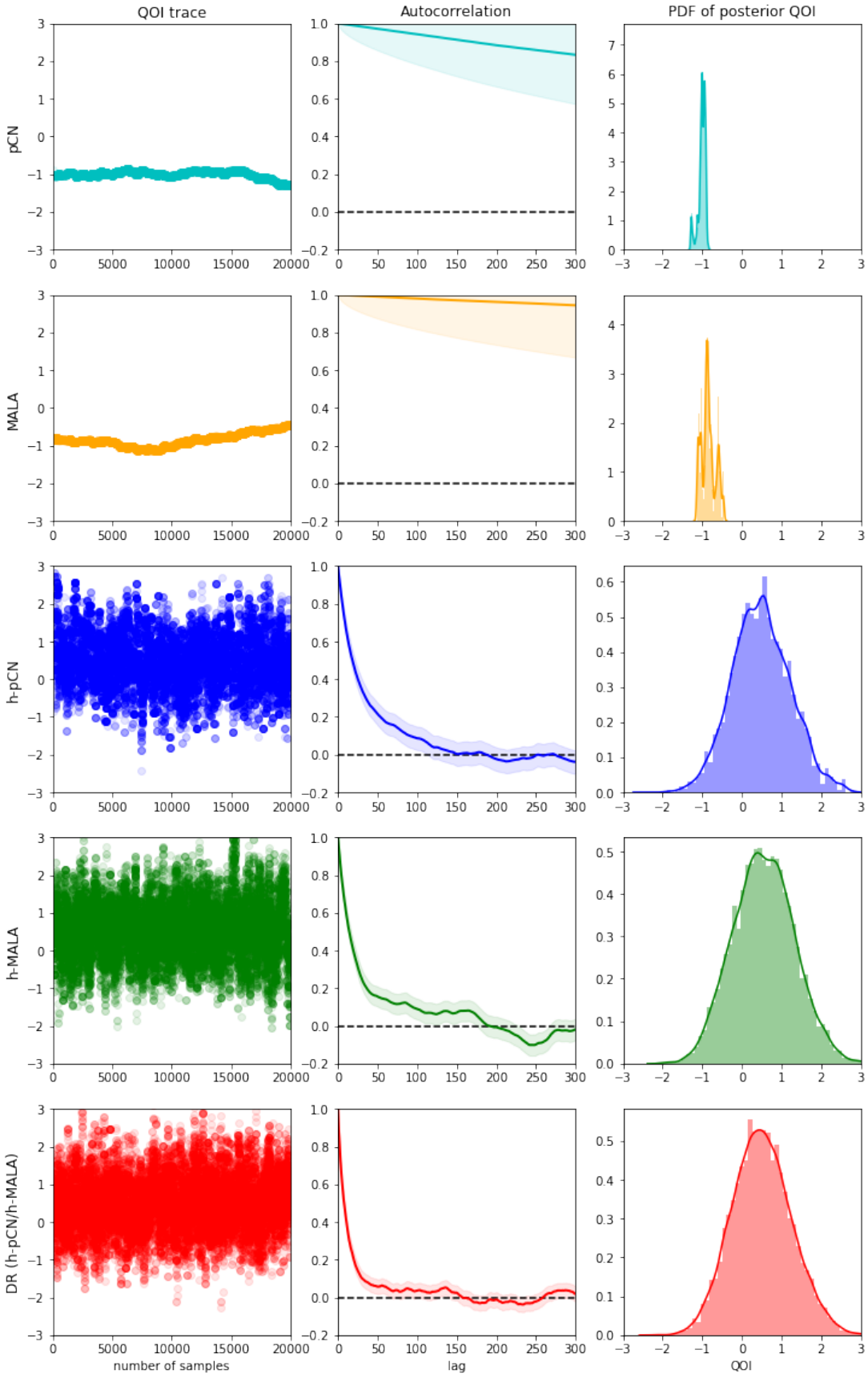
```
# Generate starting sample vector for all the MCMC simulations
noise = dl.Vector()
nu.init_vector(noise, "noise")
hp.parRandom.normal(1., noise)
pr_s = model.generate_vector(hp.PARAMETER)
post_s = model.generate_vector(hp.PARAMETER)
nu.sample(noise, pr_s, post_s, add_mean=True)
x0 = hm.dlVector2npArray(post_s)

# Implement MCMC simulations
for mName, method in method_list.items():
    # Run the MCMC sampler
    sampler = method['Sampler']
    samps = sampler.Run([x0])

    # Save the computed results
    method['Samples'] = samps
    method['ElapsedTime'] = sampler.TotalTime()

    kernel = sampler.Kernels()[0]
    if "AcceptanceRate" in dir(kernel):
        method['AcceptRate'] = kernel.AcceptanceRate()
    elif "AcceptanceRates" in dir(kernel):
        method['AcceptRate'] = kernel.AcceptanceRates()

    print("Drawn ", options['NumSamples'] - options['BurnIn'] + 1,
          "MCMC samples using", mName)

print("\n")
print("Parameter space dimension:", Vh[hp.PARAMETER].dim())
print("Number of samples:", options['NumSamples'] - options['BurnIn'] + 1)
```

(continues on next page)

(continued from previous page)

```
# Keep track of the quantity of interest
qoi_dataset = hm.track_qoiTracer(pde, qoi, method_list)
hm.print_qoiResult(method_list, qoi_dataset)
hm.plot_qoiResult(method_list, qoi_dataset, max_lag=300)
```

```
Drawn  20001 MCMC samples using pCN
Drawn  20001 MCMC samples using MALA
Drawn  20001 MCMC samples using h-pCN
Drawn  20001 MCMC samples using h-MALA
Drawn  20001 MCMC samples using DR (h-pCN/h-MALA)


Parameter space dimension: 1089
Number of samples: 20001

=================================================
 Summary of convergence diagnostics (single chain)
=================================================

Method              E[QOI]    AR      ESS   ES/min
-------------------------------------------------
pCN                -1.006   0.252     8.8     0.6
MALA               -0.844   0.550     5.7     0.1
h-pCN               0.491   0.258   191.1    12.8
h-MALA              0.553   0.543   305.3     5.4
DR (h-pCN/h-MALA)   0.505   0.582   550.8     7.8
```


QOI trace
Autocorrelation
PDF of posterior QOI
pCN
MALA
h-pCN
h-MALA
DR (h-pCN/h-MALA)
number of samples
lag
QOI

Acknowledgment: This work is supported by the National Science Foundation under grants ACI-1550487, ACI-1550547, and ACI-1550593.

CHAPTER

FIVE

HIPPYLIB2MUQ

5.1 hippylib2muq package

5.1.1 Subpackages

hippylib2muq.interface package

Submodules

hippylib2muq.interface.gaussian module

This module provides a set of wrappers that expose some functions of `hippylib` (the prior Gaussian distributions and the low-rank based Laplace approximation to the posterior distribution) to `muq`.

class hippylib2muq.interface.gaussian.**BiLaplaceGaussian**(*hp_prior*, *use_zero_mean=False*)

Bases: `muq.Modeling.PyGaussianBase`

The prior Gaussian distribution with Laplacian-like covariance operator

A class interfacing between `hippylib::BiLaplacianPrior` and `muq::GaussianBase`.

ApplyCovSqrt(*x*)

Apply the square root of covariance matrix to `x`.

Parameters **x** (*numpy::ndarray*) – input vector

ApplyCovariance(*x*)

Apply the covariance matrix to `x`.

Parameters **x** (*numpy::ndarray*) – input vector

ApplyPrecSqrt(*x*)

Apply the square root of precision matrix to `x`.

Parameters **x** (*numpy::ndarray*) – input vector

ApplyPrecision(*x*)

Apply the precision matrix to `x`.

Parameters **x** (*numpy::ndarray*) – input vector

SampleImpl(*inputs*)

Draw a sample from the prior distribution. This is an overloaded function of `muq::PyGaussianBase` The argument `inputs` is not used, but should be given when `SampleImpl` is called.

Parameters **inputs** (*numpy::ndarray*) – input vector

class hippylib2muq.interface.gaussian.**LAPosteriorGaussian**(*lapost*, *use_zero_mean=False*)

Bases: muq.Modeling.PyGaussianBase

Low-rank based Laplace approximation to the posterior distribution

A class interfacing between hippylib::GaussianLRPosterior and muq:PyGaussianBase.

ApplyCovSqrt(*x*)

Apply the square root of covariance matrix to x.

Parameters **x** (*numpy::ndarray*) – input vector

ApplyCovariance(*x*)

Apply the covariance matrix to x.

Parameters **x** (*numpy::ndarray*) – input vector

ApplyPrecision(*x*)

Apply the precision matrix to x.

Parameters **x** (*numpy::ndarray*) – input vector

SampleImpl(*inputs*)

Draw a sample from the approximated posterior distribution. This is an overloaded function of muq::PyGaussianBase.

Parameters **inputs** (*numpy::ndarray*) – input vector

class hippylib2muq.interface.gaussian.**LaplaceGaussian**(*hp_prior*, *use_zero_mean=False*)

Bases: muq.Modeling.PyGaussianBase

The prior Gaussian distribution with Laplacian-like covariance operator

An interface class between hippylib::LaplaceGaussian and muq::GaussianBase. This class is appropriate for 1D (parameter) problems.

ApplyCovariance(*x*)

Apply the covariance matrix to x.

Parameters **x** (*numpy::ndarray*) – input vector

ApplyPrecision(*x*)

Apply the precision matrix to x.

Parameters **x** (*numpy::ndarray*) – input vector

SampleImpl(*inputs*)

Draw a sample from the prior distribution. This is an overloaded function of muq::PyGaussianBase. The argument inputs is not used, but should be given when SampleImpl is called.

Parameters **inputs** (*numpy::ndarray*) – input vector

hippylib2muq.interface.modpiece module

This module provides a set of wrappers that bind some `hippylib` functionalities such that they can be used by `muq`.

Please refer to ModPiece for the detailes of member functions defined here.

class hippylib2muq.interface.modpiece.**LogBiLaplaceGaussian**(*prior*)
Bases: `muq.Modeling.PyModPiece`

Log-bi-Laplace prior

This class evaluates log of the bi-Laplacian prior.

EvaluateImpl(*inputs*)
Evaluate the log of bi-Laplacian prior.

Parameters **inputs** (*numpy::ndarray*) – input vector

class hippylib2muq.interface.modpiece.**Param2LogLikelihood**(*model*)
Bases: `muq.Modeling.PyModPiece`

Parameter to log-likelihood map

This class implements mapping from parameter to log-likelihood.

ApplyHessianImpl(*outWrt*, *inWrt1*, *inWrt2*, *inputs*, *sens*, *vec*)
Apply Hessian to `vec` for given `sens` and `inputs`.

Parameters

- **outWrt** (*int*) – output dimension; should be 0
- **inWrt1** (*int*) – input dimension; should be 0
- **inWrt2** (*int*) – input dimension; should be 0
- **inputs** (*numpy::ndarray*) – parameter values
- **sens** (*numpy::ndarray*) – sensitivity values
- **vec** (*numpy::ndarray*) – input vector Hessian applies to

ApplyJacobianImpl(*outDimWrt*, *inDimWrt*, *inputs*, *vec*)
Apply Jacobian to `vec` for given `inputs`.

Parameters

- **outDimWrt** (*int*) – output dimension; should be 0
- **inDimWrt** (*int*) – input dimension; should be 0
- **inputs** (*numpy::ndarray*) – parameter values
- **vec** (*numpy::ndarray*) – input vector Jacobian applies to

EvaluateImpl(*inputs*)
Evaluate the log-likelihood for given `inputs`.

Parameters **inputs** (*numpy::ndarray*) – parameter values

GradientImpl(*outDimWrt*, *inDimWrt*, *inputs*, *sens*)
Compute gradient; apply the transpose of Jacobian to `sens` for given `inputs`.

Parameters

- **outDimWrt** (*int*) – output dimension; should be 0
- **inDimWrt** (*int*) – input dimension; should be 0

- **inputs** (*numpy::ndarray*) – parameter values
- **sens** (*numpy::ndarray*) – input vector the transpose of Jacobian applies to

JacobianImpl(*outDimWrt*, *inDimWrt*, *inputs*)
Compute the Jacobian for given `inputs`.

Parameters

- **outDimWrt** (*int*) – output dimension; should be 0
- **inDimWrt** (*int*) – input dimension; should be 0
- **inputs** (*numpy::ndarray*) – parameter values

class hippylib2muq.interface.modpiece.**Param2obs**(*model*)
Bases: `muq.Modeling.PyModPiece`

Parameter to observable map

This class implements mapping from parameter to observations.

EvaluateImpl(*inputs*)
Evaluate the observations for given `inputs`.

Parameters **inputs** (*numpy::ndarray*) – parameter values

GradientImpl(*outDimWrt*, *inDimWrt*, *inputs*, *sens*)
Compute gradient; apply the transpose of Jacobian to `sens`.

Parameters

- **outDimWrt** (*int*) – output dimension; should be 0
- **inDimWrt** (*int*) – input dimension; should be 0
- **inputs** (*numpy::ndarray*) – parameter values
- **sens** (*numpy::ndarray*) – input vector the transpose of Jacobian applies to

Module contents

hippylib2muq.mcmc package

Submodules

hippylib2muq.mcmc.diagnostics module

This module provides a convergence diagnostic for samples drawn from MCMC methods.

class hippylib2muq.mcmc.diagnostics.**MultPSRF**(*ndof*, *nsamps*, *nchain*)
Bases: `object`

Computing the Multivariate Potential Scale Reduction Factor

This class is to compute the Multivariate Potential Scale Reduction Factor (MPSRF) described in [Brooks1998]. Note that MPSRF is the square-root version, i.e., $\hat{R}^p$ where $\hat{R}^p$ is defined by Equation (4.1) in [Brooks1998].

compute_mpsrf()
Compute MPSRF.

print_result()
Print the description and the result of MCMC chains and its diagnostic.

update_W(*samps*)
Update the within-sequence varance matrix W for a chain `samps`.

Parameters **samps** (*numpy:ndarray*) – a sequence of samples generated

class hippylib2muq.mcmc.diagnostics.**PSRF**(*nsamps*, *nchain*, *calEss=False*, *max_lag=None*)
Bases: `object`

Computing the Potential Scale Reduction Factor and the effective sample size

This class is to compute the Potential Scale Reduction Factor (PSRF) and the effective sample size (ESS) as described in [Brooks1998] and [Gelman2014]. Note that PSRF is the square-root version of $\hat{R}$ where $\hat{R}$ is defined by Equation (1.1) defined in [Brooks1998].

compute_PsrfEss(*plot_acorr=False*, *write_acorr=False*, *fname=None*)
Compute PSRF and ESS

Parameters

- **plot_acorr** (*bool*) – if True, plot the autocorrelation function
- **write_acorr** (*bool*) – if True, write the autocorrelation function to a file
- **fname** (*string*) – file name for the autocorrelation function result

print_result()
Print the description and the result of MCMC chains and its diagnostic.

update_W(*sample*)
Update the within-sequence varance W for a chain `samps`.

Parameters **samps** (*numpy:ndarray*) – a sequence of samples generated

hippylib2muq.mcmc.qoi module

This module contains some functions related to the quantity of interest.

hippylib2muq.mcmc.qoi.**cal_qoiTracer**(*pde*, *qoi*, *muq_samps*)
This function is for tracing the quantity of interest.

Parameters

- **pde** (*hippylib:PDEProblem*) – a hippylib:PDEProblem instance
- **qoi** – the quantity of interest; it should contain the member function named as `eval` which evaluates the value of qoi
- **muq_samps** – samples generated from `muq` sampler

hippylib2muq.mcmc.qoi.**track_qoiTracer**(*pde*, *qoi*, *method_list*, *max_lag=None*)
This function computes the autocorrelation function and the effective sample size of the quantity of interest.

Parameters

- **pde** (*hippylib:PDEProblem*) – a hippylib:PDEProblem instance
- **qoi** – the quantity of interest; it should contain the member function
- **method_list** (*dictionary*) – a dictionary containing MCMC methods descriptions with samples generated from muq sampler
- **max_lag** (*int*) – maximum of time lag for computing the autocorrelation function

Module contents

hippylib2muq.utility package

Submodules

hippylib2muq.utility.conversion module

This module provides type conversions for the proper use of dependent packages.

hippylib2muq.utility.conversion.**const_dlVector**(*mat*, *dim*)
Construct and initialize a dolfin vector so that it is compatible with matrix A for the multiplication $Ax = b$.

Parameters

- **mat** (*dolfin:matrix*) – a dolfin:matrix
- **dim** (*int*) – 0 for b and 1 for x

Returns an initialized dolfin:vector which is compatiable with mat

hippylib2muq.utility.conversion.**dlVector2npArray**(*vec*)
Convert a dolfin:vector to a numpy:ndarray.

Parameters **vec** (*dolfin:vector*) – a dolfin:vector

Returns converted numpy:ndarray

hippylib2muq.utility.conversion.**npArray2dlVector**(*arr*, *vec*)
Assign values of numpy:ndarray to dolfin:vector.

Parameters

- **arr** (*numpy:ndarray*) – a numpy:ndarray
- **vec** (*dolfin:vector*) – a dolfin:vector assigned by arr

hippylib2muq.utility.postprocessing module

This module provides postprocessing related functions.

hippylib2muq.utility.postprocessing.**plot_qoiResult**(*method_list*, *qoi_dataset*, *max_lag=None*)
Plot the result of MCMC simulations for the quantity of interest

Parameters

- **method_list** (*dictionary*) – the discriptions of MCMC methods used
- **qoi_dataset** (*dictionary*) – a dictionary returned from a call of hippymuq:track_qoiTracer
- **max_lag** (*int*) – maximum of time lag for computing the autocorrelation function

hippylib2muq.utility.postprocessing.**print_methodDict**(*method_list*)
Print the method descriptions formatted for the MCMC simulation.

For MCMC kernel, abbreviations mean –

Name	MCMC kernel
mh	Metropolis-Hastings
dr	Delayed Rejection

For MCMC proposal, abbreviations mean –

Name	MCMC proposal
pcn	Preconditioned Crank-Nicolson
mala	preconditioned Metropolis Adjusted Langevin Algorithm

Note that this auxiliary function is only for the MCMC kernels and proposals listed in the above tables, but other MCMC methods such as Dimension-independent likelihood-informed MCMC are also available for use.

Parameters **method_list** (*dictionary*) – the discriptions of MCMC methods

hippylib2muq.utility.postprocessing.**print_qoiResult**(*method_list*, *qoi_dataset*)
Print the result of MCMC simulations for the quantity of interest.

Parameters

- **method_list** (*dictionary*) – the discriptions of MCMC methods used
- **qoi_dataset** (*dictionary*) – a dictionary returned from a call of hippymuq:track_qoiTracer

Module contents

5.1.2 Module contents

CHAPTER

SIX

INDICES AND SEARCH

- genindex
- modindex
- search

BIBLIOGRAPHY

[Brooks1998] Brooks and Gelman, 1998, General Methods for Monitoring Convergence of Iterative Simulations.

[Gelman2014] Gelman et al., 2014, Bayesian Data Analysis, pp 286-287.

PYTHON MODULE INDEX

h

INDEX

J

L

M

N

P

S

T

U