The Parallel C++ Statistical Library for Bayesian Inference: QUESO

54

Damon McDougall, Nicholas Malaya, and Robert D. Moser

Abstract

The Parallel C++ Statistical Library for the Quantification of Uncertainty for Estimation, Simulation, and Optimization (QUESO) is a collection of statistical algorithms and programming constructs supporting research into the quantification of uncertainty of models and their predictions. QUESO is primarily focused on solving statistical inverse problems using Bayes' theorem, which expresses a distribution of possible values for a set of uncertain parameters (the posterior distribution) in terms of the existing knowledge of the system (the prior) and noisy observations of a physical process, represented by a likelihood distribution. The posterior distribution is not often known analytically and so requires computational methods. It is typical to compute probabilities and moments from the posterior distribution, but this is often a high-dimensional object, and standard Riemann-type methods for quadrature become prohibitively expensive. The approach QUESO takes in this regard is to rely on Markov chain Monte Carlo (MCMC) methods which are well suited to evaluating quantities such as probabilities and moments of high-dimensional probability distributions. QUESO's

D. McDougall (✉) • N. Malaya
Predictive Engineering and Computational Science, Institute for Computational and Engineering Sciences, The University of Texas at Austin, Austin, TX, USA
e-mail: damon@ices.utexas.edu; nick@ices.utexas.edu

R.D. Moser
Department of Mechanical Engineering, Institute for Computational and Engineering Sciences, The University of Texas at Austin, Austin, TX, USA

Predictive Engineering and Computational Science, Institute for Computational and Engineering Sciences, The University of Texas at Austin, Austin, TX, USA
e-mail: rmoser@ices.utexas.edu

R. Ghanem et al. (eds.), *Handbook of Uncertainty Quantification*,
DOI 10.1007/978-3-319-12385-1_57

intended use is as tool to assist and facilitate coupling uncertainty quantification to a specific application called a *forward problem*. While many libraries presently exist that solve Bayesian inference problems, QUESO is a specialized piece of software primarily designed to solve such problems by utilizing parallel environments demanded by large-scale forward problems. QUESO is written in C++, uses MPI, and utilizes libraries already available to the scientific community. Thus, the target audience of this library are researchers who have solid background in Bayesian methods, are comfortable with UNIX concepts and the command line, and have knowledge of a programming language, preferably C/C++.

Keywords

Bayesian inference • Inverse problems • Monte Carlo methods • Computational Markov chains • Mathematical software • Parallel computation • Parallel algorithms

Contents

1 Introduction

The Parallel C++ Statistical Library for the Quantification of Uncertainty for Estimation, Simulation, and Optimization (QUESO), is a collection of statistical algorithms and programming constructs supporting research into the uncertainty quantification (UQ) of models and their predictions. It has been designed with three objectives: (a) to be sufficiently abstract in order to handle a large spectrum of large-scale computationally intensive models; (b) to be extensible, allowing easy creation of custom-defined objects; and (c) leverage parallel computing through use of high-performance vector and matrix libraries. Such objectives demand a combination of an object-oriented design with robust software engineering practices. QUESO is written in C++, uses MPI, and utilizes libraries already available to the scientific community.

The purpose of this book chapter is not to teach uncertainty quantification methods, but rather to introduce the QUESO library so it can be used as a tool to assist and facilitate coupling UQ to a specific application (forward problem). Thus, the target audience of this chapter is researchers who have solid background in Bayesian methods, are comfortable with UNIX concepts and the command line, and have knowledge of a programming language, preferably C/C++.

The rest of the document is organized as follows. Section 2 has a brief discussion of statistical inverse problems, and in doing so, provides the impetus behind the QUESO library. Section 4 then discusses the types of problems the library is designed to solve, as well as introducing the notation used for the rest of this document. Several illustrative examples, including the new infinite-dimensional capability, are provided in Sect. 5 along with code snippets demonstrating typical software call-patterns. Section 6 discusses how the library design can easily be extended for bespoke user-defined random variables, probability distribution functions, and realizers. Section 7 discusses the design and internals of the library, as well as providing a software snapshot of the current library status. Finally, we conclude by discussing several areas in which to focus future QUESO development efforts.

All of the examples in this document are present in the QUESO source tree of the latest release, 0.53.0. Users should consult the website, libqueso.com, for the latest news and source code.

This chapter builds on the 2012 paper that introduced the QUESO library [1] and the current QUESO user's manual [2] by including a myriad of changes that have since been incorporated into the library.

2 Motivation

Statistical inverse problems using a Bayesian formulation model all quantities as random variables, where probability distributions of the quantities capture the uncertainty in their values. The solution to the inverse problem is then the probability distribution of the quantity of interest when all information available has

been incorporated in the model. This (posterior) distribution describes the degree of confidence about the quantity after the measurement has been performed [3].

Thus, the solution to the statistical inverse problem is given by Bayes' formula, which expresses the posterior distribution in terms of the prior distribution and the data represented through the likelihood function.

For all but toy problems, the likelihood function has an open form and its evaluation is highly computationally intensive. Worse, simulation-based posterior inference often requires a large number of these evaluations of the forward model. Therefore, fast and efficient sampling techniques are desirable for posterior inference.

It is often not straightforward to obtain explicit posterior point estimates of the solution, since it usually involves the evaluation of a high-dimensional integral with respect to a possibly non-smooth posterior distribution. In such cases, an alternative integration technique is the Markov chain Monte Carlo method where posterior moments may be estimated using the samples from a series of (correlated) random draws from the posterior distribution.

QUESO is designed in an abstract way so that it can be used by any computational model, as long as a likelihood function (in the case of statistical inverse problems) and a quantity of interest (QoI) function (in the case of statistical forward problems) are provided by the user application.

With this framework in mind, QUESO provides tools for both sampling algorithms for statistical inverse problems, following Bayes' formula, and statistical forward problems. It provides Markov chain Monte Carlo algorithms using the Metropolis-Hastings acceptance ratio [4, 5]: these are the multilevel Monte Carlo [6] method and DRAM [7]. QUESO is also capable of handling several chains in parallel computational environments.

3 Alternatives to QUESO

QUESO is certainly not the only quality statistical software library. There are many different libraries that can be used to solve Bayesian inference problems. QUESO is a specialized piece of software, primarily designed to solve such problems utilizing the, often required, parallel environment demanded by large-scale forward problems. This focus is simultaneously the QUESO's greatest strength and weakness, depending on user's target problem. For instance, QUESO would be less effective to use for serial problems than several alternative libraries, as there is significant turnaround time from learning how to build QUESO and link a custom forward code to it. In instances where parallelization is not necessary and the forward problem is relatively cheap to execute, there are good alternatives to QUESO. We now provide a simple survey of several other major libraries that we consider useful for problems of Bayesian inference, along with a brief discussion some unique strengths and weaknesses.

As discussed above, for inference problems that do not require parallelization, serial libraries can be leveraged with less development. An excellent example of this is PyMC [8]. A modern software package, PyMC is a python-based software library for Bayesian estimation and MCMC. Its strengths lie in its flexibility and excellent post-processing, especially when coupled with matplotlib [9]. emcee [10] is another python-based package, with a particular emphasis on Bayesian parameter estimation. Both of these libraries are useful for rapid software prototyping using serial inference problems.

On the other end of the spectrum, there are complete statistical software languages. These are often more mature software projects which are capable of much more general statistical computations than QUESO. However, these languages are often weaker for specialized problems, because they are not as well optimized for solving Bayesian inference problems, particularly at scale. The ultimate example of this is R [11]. R is a free software programming language and software environment for statistical computing and graphics. R is arguably the most general and complete source of open-source statistical packages in the world. It is not limited to Bayesian techniques and has packages across a wide range of topics in statistics. However, it is not easy to couple R with other codebases (for the forward problem, for instance). Furthermore, while some packages supporting parallelism are now being developed, the language is still primarily focused on serial computations. Another alternative is Stan [12]. Stan is a probabilistic programming language written in C++ implementing full Bayesian statistical inference.

Another major library is WinBUGS [13]. WinBUGS is statistical software for Bayesian analysis using MCMC methods. WinBUGS is of particular historical importance, as it was one of the earliest openly available MCMC libraries, with development starting the late 1980s. It is also unique in that it is developed for the Windows platform, instead of Linux. It is also primarily based on the Gibbs sampler algorithm.

Finally, the DAKOTA [14] toolkit is a very general library developed at Sandia National Laboratories, containing a vast array of algorithms with applications to uncertainty quantification, optimization, emulation, experimental design, prediction, and sensitivity analysis. DAKOTA is written in C++ and supported on Linux, OS X, and Windows and represents 20 years of advanced algorithms research. Furthermore, given DAKOTA's advanced certainty propagation algorithms, the QUESO and DAKOTA development teams are working together to establish a seamless integration of QUESO's algorithms into DAKOTA to give users a matured and coupled forward and inverse UQ software solution.

4 Formulation

Here we give a rigorous description of the types of problems that QUESO solves. This will crystallize both the terminology and notation in an attempt to make everything in this chapter self-contained.

4.1 The Forward Problem

Here we set out the auspices under which we will operate. We make two high-level assumptions: (1) we have access to a set of observations of some physical phenomenon; and (2) we have a mathematical model that attempts to model the observed physical phenomenon. Ensuring that the mathematical model is *valid* is an exercise left to the reader. We will denote the observed data by y and the mathematical model by $\mathcal{G}$. The model will certainly depend on various parameters, and we call the process of mapping these parameters to model output the *forward problem*. In many physical engineering applications, the forward problem is expensive and may involve the solution of a set of partial differential equations.

4.2 The Inverse Problem

In the subsection above, we described the forward problem. It may be the case that the mathematical model in the forward problem may depend on some parameters that are unknown and that we wish to estimate. We will refer to these unknown parameters as θ. The process of estimating θ given observations goes by many names, but is generally referred to as the *inverse problem*. There are several frameworks for solving inverse problems. We will focus only on the *Bayesian framework*, which we rigorously describe now.

As noted above, we are given a set of observations y. This dataset is corrupted by errors made during the experiment. These errors could be human errors, equipment errors, or errors in the setup of the experimental scenario. In complete generality, it is difficult to say with certainty what statistical distribution these errors follow. In a lot of experimental cases, however, a Gaussian distribution with some, perhaps unknown, variance is quite a reasonable characterization.

The unknown parameters themselves might have some inherent constraining property. For example, if the unknown parameter were a concentration of a contaminant underground then it is not possible for this unknown parameter to be negative. The constraint varies depending on the physical domain, but it is rarely the case one knows *nothing* about the unknown parameters. This information can be translated to constraints on a prior distribution.

To regroup, we have a statistical distribution governing the behavior of the experimental errors given the unknown parameters, $\mathbb{P}(y|\theta)$. We also have some prior distribution on the unknown parameters $\mathbb{P}(\theta)$. The Bayesian solution to the inverse problem of finding θ is the distribution of θ given y, $\mathbb{P}(\theta|y)$. By Bayes' rule, this can be written as follows:

$$\mathbb{P}(\theta|y) \propto \mathbb{P}(y|\theta)\mathbb{P}(\theta).$$

The left-hand side is referred to as the posterior distribution. The right-hand side is the product of the likelihood distribution and the prior distribution. `QUESO` solves

the Bayesian inverse problem by providing samples that are distributed according to the posterior distribution using Markov chain Monte Carlo. This chapter does not provide the details of how MCMC works. The authors refer the reader to the expansive body of available literature on the topic cited throughout this work.

4.3 Prediction

The prediction step in the Bayesian framework is that of estimating some quantity $\mathcal{Q}(\theta)$ dependent on the unknown parameters. This is usually referred to as a *statistical forward problem*. QUESO is equipped to solve statistical forward problems, but throughout this chapter we will focus mainly on the statistical inverse problem.

5 Examples

5.1 A Template Example

Here we walk through a template example. This template should be general enough to serve as a good starting point for most Bayesian inverse problems. Before we step through the example, here it is in its entirety:

```
#include <queso/GslVector.h>
#include <queso/GslMatrix.h>
#include <queso/UniformVectorRV.h>
#include <queso/StatisticalInverseProblem.h>
#include <queso/ScalarFunction.h>
#include <queso/VectorSet.h>

template<class V = QUESO::GslVector, class M = QUESO::GslMatrix>
class Likelihood : public QUESO::BaseScalarFunction<V, M>
{
public:

  Likelihood(const char * prefix, const QUESO::VectorSet<V, M> & domain)
    : QUESO::BaseScalarFunction<V, M>(prefix, domain)
  {
    // Setup here
  }

  virtual ~Likelihood()
  {
    // Deconstruct here
  }

  virtual double lnValue(const V & domainVector, const V * domainDirection,
      V * gradVector, M * hessianMatrix, V * hessianEffect) const
  {
    // 1) Run the forward code at the point domainVector
    //    domainVector[0] is the first element of the parameter vector
    //    damainVector[1] is the second element of the parameter vector
    //    and so on
```

```
    //
    // 2) Compare to data, y
    //    Usually we compute something like
    //    || MyModel(domainVector) - y ||^2 / (sigma * sigma)
    //
    // 3) Return below

    double misfit = 0.0;

    return -0.5 * misfit;
  }

  virtual double actualValue(const V & domainVector, const V * domainDirection,
      V * gradVector, M * hessianMatrix, V * hessianEffect) const
  {
    return std::exp(this->lnValue(domainVector, domainDirection, gradVector,
          hessianMatrix, hessianEffect));
  }

private:
  // Maybe store the observed data, y, here.
};

int main(int argc, char ** argv) {
  MPI_Init(&argc, &argv);

  // Step 0 of 5: Set up environment
  QUESO::FullEnvironment env(MPI_COMM_WORLD, argv[1], "", NULL);

  // Step 1 of 5: Instantiate the parameter space
  QUESO::VectorSpace<> paramSpace(env,
      "param_", 1, NULL);

  double min_val = 0.0;
  double max_val = 1.0;

  // Step 2 of 5: Set up the prior
  QUESO::GslVector paramMins(paramSpace.zeroVector());
  paramMins.cwSet(min_val);
  QUESO::GslVector paramMaxs(paramSpace.zeroVector());
  paramMaxs.cwSet(max_val);

  QUESO::BoxSubset<> paramDomain("param_", paramSpace, paramMins, paramMaxs);

  // Uniform prior here.  Could be a different prior.
  QUESO::UniformVectorRV<> priorRv("prior_", paramDomain);

  // Step 3 of 5: Set up the likelihood using the class above
  Likelihood<> lhood("llhd_", paramDomain);

  // Step 4 of 5: Instantiate the inverse problem
  QUESO::GenericVectorRV<> postRv("post_", paramSpace);

  QUESO::StatisticalInverseProblem<> ip("", NULL, priorRv, lhood, postRv);

  // Step 5 of 5: Solve the inverse problem
  QUESO::GslVector paramInitials(paramSpace.zeroVector());

  // Initial condition of the chain
  paramInitials[0] = 0.0;
```

```
  paramInitials[1] = 0.0;

  QUESO::GslMatrix proposalCovMatrix(paramSpace.zeroVector());

  for (unsigned int i = 0; i < 2; i++) {
    // Might need to tweak this
    proposalCovMatrix(i, i) = 0.1;
  }

  ip.solveWithBayesMetropolisHastings(NULL, paramInitials, &proposalCovMatrix);

  MPI_Finalize();

  return 0;
}
```

Notice that this template example is fairly short, weighing in at roughly 100 lines of boilerplate C++ code. Incorporating a specific physical model into the likelihood will certainly increase the size of the statistical application. In the meantime, we will walk through the boilerplate setup that will be common to many use cases.

We will start with the `main` function. This is where most of the setup takes place. Firstly, since `QUESO` uses `MPI`, we must call the `MPI_Init` function before using any of the classes in `QUESO`. The next line,

```
QUESO::FullEnvironment env(MPI_COMM_WORLD, argv[1], "",
    NULL);
```

sets up the `QUESO` environment. The constructor parameters are, in order, an `MPI` communicator and could be a custom sub-communicator; the filename of a `QUESO` input file; a prefix, if a different from the default is desired, for input file options specific to the `QUESO` environment; and an optional `EnvOptionsValues` object so that the user can set environment options programmatically. The next thing we do is define the dimension of the state space by created a object representing a vector space:

```
QUESO::VectorSpace<> paramSpace(env, "param_", 1, NULL);
```

In this particular example, the dimension of the state space is 1. The constructor parameters here are the `QUESO` environment; a prefix, if a different from the default is desired, for input file options specific to this parameter space object; and a vector of strings to name components of the vectors belonging to this vector space. Now we are in a position to set up the domain of the statistical inverse problem. `QUESO` only supports box domains, but the bounds for the box may be arbitrary. We store the bounds for the domain in `GslVector` objects like so:

```
QUESO::GslVector paramMins(paramSpace.zeroVector());
paramMins.cwSet(min_val);
QUESO::GslVector paramMaxs(paramSpace.zeroVector());
paramMaxs.cwSet(max_val);
```

Here `min_val` and `max_val` will be specific to the user's problem. The box domain uses these bounds and is constructed as follows:

```
QUESO::BoxSubset<> paramDomain("param_", paramSpace,
    paramMins,
    paramMaxs);
```

We have finished setting up the domain of the statistical inverse problem. Recall the ingredients we need for a well-posed statistical inverse problem; a prior distribution and a likelihood distribution. QUESO supports many statistical distributions that can all be used as a prior, and the user may choose to implement their own prior distribution if (see Sect. 6) such customization is needed. The following line creates an object representing a uniform random variable:

```
QUESO::UniformVectorRV<> priorRv("prior_",
    paramDomain);
```

This object contains all the necessary information to fully define a uniformly distributed random variable, namely, its probability density function and mechanisms by which one can make draws with this density. The second ingredient needed for a statistical inverse problem is the definition of a likelihood distribution, and this is done now:

```
Likelihood<> lhood("llhd_", paramDomain);
```

This line may look different to the one for your specific application, as it is intended to interact with a specific physical model. The `Likelihood` class is a custom-defined class. We will come back to the full `Likelihood` class in Sects. 5.3 and 5.2 explain how it is implemented. For now, we will continue with the setup of the inverse problem and all the necessary code needed to initialize the sampling. We construct a placeholder object that represents a posterior random variable:

```
QUESO::GenericVectorRV<> postRv("post_", paramSpace);
```

QUESO will operate on this object during the sampling. After QUESO has finished its sampling, this object is then available to you for post-processing. Next, we pass the prior, likelihood, and posterior over to the `StatisticalInverseProblem` class like so:

```
QUESO::StatisticalInverseProblem<> ip("", NULL,
    priorRv, lhood, postRv);
```

We are now ready to finalize the setup of the inverse problem. We do this by giving QUESO an initial condition for the sampler:

```
QUESO::GslVector paramInitials(
    paramSpace.zeroVector());
paramInitials[0] = 0.0;
paramInitials[1] = 0.0;
```

We also give QUESO an initial covariance matrix:

```
QUESO::GslMatrix proposalCovMatrix(
    paramSpace.zeroVector());
```

```
for (unsigned int i = 0; i < 1; i++) {
  proposalCovMatrix(i, i) = 0.1;
}
```

The closer this matrix is to the covariance between parameters under the posterior measure, the better the Markov chain will perform. Providing a bad initial covariance does not change the posterior distribution in the limit of infinite samples. Finally, we begin sampling with the following call:

```
ip.solveWithBayesMetropolisHastings(NULL,
    paramInitials, &proposalCovMatrix);
```

5.2 Defining the Likelihood Distribution

As can be observed in the example illustrated above, the user must pass a likelihood to `QUESO`. `QUESO` expects, as a likelihood, anything that subclasses the `BaseScalarFunction` abstract base class. This base class has two pure virtual functions that must be implemented in any subclass. These functions are `lnValue()` and `actualValue()`. The function `lnValue` takes a number of parameters, the most important of which is `const V & domainVector`. When the user implements this function, it should return the natural logarithm of the likelihood distribution evaluated at the point `domainVector`. A concrete example of this can be seen in the next subsection. The function `actualValue` should return exactly the likelihood distribution evaluated at the point `domainVector`. For most practical applications, this function will usually just return `std::exp` of `lnValue`, but the user has the freedom to implement a more optimized computation if one is needed.

A typical Gaussian likelihood distribution will look something like this:

```
template<class V, class M>
double
Likelihood<V = QUESO::GslVector,
    M = QUESO::GslMatrix>::lnValue(
    const V & domainVector,
    const V * domainDirection, V * gradVector,
    M * hessianMatrix, V * hessianEffect) const
{
  double misfit = 0.0;
  unsigned int vec_len = domainVector.sizeLocal()

  for (unsigned int i = 0; i < vec_len; i++) {
    misfit += domainVector[i] - observation[i];
  }

  return -0.5 * misfit;
}
```

To avoid numerical problems computing the acceptance probability in an MCMC algorithm, `QUESO` will call `lnValue` instead of `actualValue` to do the accept-reject step in log space.

5.3 Ball Drop Example

This section presents an example of how to use `QUESO` as an application that solves a statistical inverse problem (SIP) and a statistical forward problem (SFP), where the solution of the former serves as input to the later. This example will use the canonical "ball drop" problem, a standard problem in uncertainty quantification. The objective of the SIP is to infer the acceleration due to gravity on an object in free fall near the surface of the Earth. During the SFP, the distance traveled by a projectile launched at a given angle and altitude is calculated using the calibrated magnitude of the acceleration of gravity gathered during the SIP. As expressed in Sect. 4, both the inference and forward problem will be performed using a Bayesian methodology, and so, the resulting quantities of interest (QoIs) will be expressed as probability distributions.

5.4 Statistical Inverse Problem

A deterministic mathematical model for the vertical motion of an object in free fall near the surface of the Earth is given by

$$h(t) = -\frac{1}{2}gt^2 + v_0 t + h_0. \tag{54.1}$$

where, v_0 $[m/s]$ is the initial velocity, h_0 $[m]$ is the initial altitude, $h(t)$ $[m]$ is the altitude with respect to time, t $[s]$ is the elapsed time, and g $[m/s^2]$ is the magnitude of the acceleration due to gravity (the parameter which cannot be directly measured and will be statistically inferred).

This model is an expression of a high-fidelity model, Newton's second law of motion. However, the model is imperfect, as it does not account resistive force of air resistance, for example.

5.4.1 Experimental Data

The experimental data will be generated from an identical object falling from several different heights, each with zero initial velocity (see Fig. 54.1). We collect data, $\mathbf{y}$, of the time taken for the ball to impact the ground starting from various different initial heights. Each experimental observation error is treated as Gaussian with some known mean and variance standard deviation, σ. The error is a result of measurement uncertainties, such as estimates of the actual height the object was dropped from, the human error introduced by operating a stopwatch for time measurement, and any other possible sources of error. The actual observation values can be found in the accompanying source code that will follow shortly.

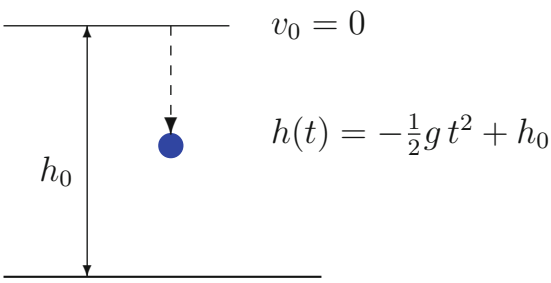

Fig. 54.1 An object falls from altitude h_0 with zero initial velocity ($v_0 = 0$)

5.4.2 The Prior, Likelihood, and Posterior

In Bayesian inference, the prior probability signifies the modeler's expectation of the result of an experiment before any data is provided. In this problem, a prior must be provided for the parameter g. Near the surface of the Earth, an object in free fall in a vacuum will accelerate at approximately $9.8\,\mathrm{m/s^2}$, independent of its mass. For this gravitational inference problem, we will place a uniform prior on our unknown variable θ, over the interval [8,11]:

$$\mathbb{P}(\theta) = \mathscr{U}(8, 11). \tag{54.2}$$

We select a Gaussian likelihood function that assigns greater probabilities to parameter values that result in model predictions close to the data:

$$\mathbb{P}(\mathbf{y}|\theta) \propto \exp\left(-\frac{1}{2}\left(\mathscr{G}(\theta) - \mathbf{y}\right)^T \mathbf{C}^{-1}\left(\mathscr{G}(\theta) - \mathbf{y}\right)\right), \tag{54.3}$$

where $\mathbf{C}$ is a given covariance matrix, $\mathbf{y}$ is the experimental data, and $\mathscr{G}(\theta)$ is the model output.

Using the deterministic model for the acceleration of gravity (Eq. 54.1) with no initial velocity, the observations $\mathbf{y}$, and equation (54.3), we have

$$\theta \overset{\text{def.}}{=} g, \quad \mathscr{G}(\theta) = \begin{bmatrix} \sqrt{\frac{2h_1}{g}} \\ \sqrt{\frac{2h_2}{g}} \\ \vdots \\ \sqrt{\frac{2h_{n_d}}{g}} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_{n_d} \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \sigma_{n_d}^2 \end{bmatrix}, \tag{54.4}$$

where $n_d = 14$ is the number of observations. We now invoke Bayes' formula in order to obtain the posterior PDF $\mathbb{P}(\theta|\mathbf{y})$:

$$\mathbb{P}(\theta|\mathbf{y}) \propto \mathbb{P}(\mathbf{y}|\theta)\mathbb{P}(\theta). \tag{54.5}$$

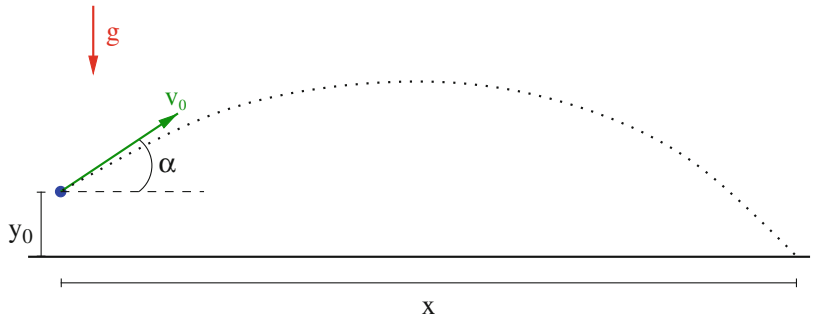

Fig. 54.2 Object traveling with projectile motion

5.5 Statistical Forward Problem

Projectile motion refers to the motion of an object projected into the air at an angle, e.g., a soccer ball being kicked, a baseball being thrown, or an athlete long jumping. In the absence of a propulsion system and neglecting air resistance, the only force acting on the object is proportional to a constant gravitational acceleration g. A deterministic two-dimensional mathematical model for the vertical motion of an object projected from near the surface of the Earth is given by

$$v_x = v_{0x}, \tag{54.6}$$

$$v_y = v_{0y} - gt, \tag{54.7}$$

$$x = v_{0x}t, \tag{54.8}$$

$$h = h_0 + v_{0y}t - \frac{1}{2}gt^2, \tag{54.9}$$

where h_0 is the initial height, $x = x(t)$ is the distance traveled by the object, $\boldsymbol{v_0} = (v_{0x}, v_{0y})$ is the initial velocity, $v_{0x} = v_0\cos(\alpha)$, $v_{0y} = v_0\sin(\alpha)$, and $v_0 = \|\boldsymbol{v_0}\|^2$. Figure 54.2 displays the projectile motion of an object in these conditions.

In this example, we assume that h_0, α, and v_0 are all known, with $h_0 = 0$, $\alpha = \pi/4$, $v_0 = 5$, and g is the result of the SIP described in Sect. 5.4.

Since the result of the SIP is a PDF on g, the output of the mathematical model (54.6) will be a random variable, and our forward problem result will also be statistical in nature.

5.5.1 The Input Random Variable, QoI, and Output Random Variable

The input for the statistical forward problem is the random variable g, the acceleration of gravity. This is the solution (posterior PDF) of the inverse problem described in Sect. 5.4. The QoI for this example is the distance x traveled by an object in projectile motion.

Combining the expressions in Eq. (54.6) and rearranging them, the QoI function for x is

$$x = \frac{v_0 \cos\alpha}{g}\left(v_0 \sin\alpha + \sqrt{(v_0 \sin\alpha)^2 + 2g\, y_0}\right). \tag{54.10}$$

Here x is the distance traveled and our quantity of interest (QoI).

5.6 Example Code

The source code for the SIP and the SFP is composed of several files. Three of them are common for both problems, `gravity_main.C`, `gravity_compute.h`, and `gravity_compute.C`; they combine both problems and use the solution of the SIP (the posterior PDF for the gravity) as an input for the SFP. We present only the statistical inverse problem here. The forward problem is very similar to the inverse problem, and the user is encouraged to visit the source tree (https://libqueso.com) for the full treatment.

The files common to the inverse (and forward) problem are in Listings 1 and 2. Two files specifically handle the SIP: `gravity_likelihood.h` and `gravity_likelihood.C`. These are displayed in Listings 3 and 4.

Listing 1 File `gravity_main.C`.

```
#include <gravity_compute.h>

int main(int argc, char* argv[])
{
  // Initialize QUESO environment
  MPI_Init(&argc,&argv);
  QUESO::FullEnvironment* env =
    new QUESO::FullEnvironment(MPI_COMM_WORLD, argv[1],"",NULL);

  // Call application
  computeGravityAndTraveledDistance(*env);

  // Finalize QUESO environment
  delete env;
  MPI_Finalize();

  return 0;
}
```

Listing 2 File `gravity_compute.C`. The first part of the code (lines 4–44) handles the statistical forward problem, whereas the second part of the code (lines 53–76) handles the statistical forward problem.

```
1  void computeGravityAndTraveledDistance(const QUESO::FullEnvironment& env) {
2    // Statistical inverse problem (SIP): find posterior PDF for 'g'
3
4    // SIP Step 1 of 6: Instantiate the parameter space
5    QUESO::VectorSpace<QUESO::GslVector,QUESO::GslMatrix> paramSpace(env,
6        "param_", 1, NULL);
7
```

```
8   // SIP Step 2 of 6: Instantiate the parameter domain
9   QUESO::GslVector paramMinValues(paramSpace.zeroVector());
10  QUESO::GslVector paramMaxValues(paramSpace.zeroVector());
11  paramMinValues[0] = 8.;
12  paramMaxValues[0] = 11.;
13
14  QUESO::BoxSubset<QUESO::GslVector,QUESO::GslMatrix> paramDomain("param_",
15      paramSpace, paramMinValues, paramMaxValues);
16
17  // SIP Step 3 of 6: Instantiate the likelihood object to be used by QUESO.
18  Likelihood<QUESO::GslVector, QUESO::GslMatrix>lhood("like_", paramDomain);
19
20  // SIP Step 4 of 6: Define the prior RV
21  QUESO::UniformVectorRV<QUESO::GslVector,QUESO::GslMatrix> priorRv
22      ("prior_", paramDomain);
23
24  // SIP Step 5 of 6: Instantiate the inverse problem
25  QUESO::GenericVectorRV<QUESO::GslVector,QUESO::GslMatrix>
26    postRv("post_", // Extra prefix before the default "rv_" prefix
27           paramSpace);
28
29  QUESO::StatisticalInverseProblem<QUESO::GslVector,QUESO::GslMatrix>
30    ip("",          // No extra prefix before the default "ip_" prefix
31       NULL,
32       priorRv,
33       lhood,
34       postRv);
35
36  // SIP Step 6 of 6: Solve the inverse problem, that is,
37  // set the 'pdf' and the 'realizer' of the posterior RV
38  QUESO::GslVector paramInitials(paramSpace.zeroVector());
39  priorRv.realizer().realization(paramInitials);
40
41  QUESO::GslMatrix proposalCovMatrix(paramSpace.zeroVector());
42  proposalCovMatrix(0,0) = std::pow(std::abs(paramInitials[0]) / 20.0, 2.0);
43
44  ip.solveWithBayesMetropolisHastings(NULL, paramInitials,
45    &proposalCovMatrix);
46
47  // Statistical forward problem (SFP): find the max distance
48  // traveled by an object in projectile motion; input pdf for 'g'
49  // is the solution of the SIP above.
50
51  // SFP Step 1 of 6: Instantiate the parameter *and* qoi spaces.
52  // SFP input RV = FIP posterior RV, so SFP parameter space
53  // has been already defined.
54  QUESO::VectorSpace<QUESO::GslVector,QUESO::GslMatrix> qoiSpace(env,
55      "qoi_", 1, NULL);
56
57  // SFP Step 2 of 6: Instantiate the parameter domain
58  // NOTE: Not necessary because input RV of the SFP = output RV of SIP.
59  // Thus, the parameter domain has been already defined.
60
61  // SFP Step 3 of 6: Instantiate the qoi object to be used by QUESO.
62  Qoi<QUESO::GslVector, QUESO::GslMatrix, QUESO::GslVector,QUESO::GslMatrix>
63    qoi("qoi_", paramDomain, qoiSpace);
64
65  // SFP Step 4 of 6: Define the input RV
66  // NOTE: Not necessary because input RV of SFP=output RV of SIP (postRv).
```

```
67     // SFP Step 5 of 6: Instantiate the forward problem
68     QUESO::GenericVectorRV<QUESO::GslVector, QUESO::GslMatrix> qoiRv("qoi_",
69         qoiSpace);
70
71     QUESO::StatisticalForwardProblem<QUESO::GslVector, QUESO::GslMatrix,
72       QUESO::GslVector, QUESO::GslMatrix> fp("", NULL, postRv, qoi, qoiRv);
73
74     // SFP Step 6 of 6: Solve the forward problem
75     fp.solveWithMonteCarlo(NULL);
76   }
```

Listing 3 File `gravity_likelihood.h`.

```
template<class V, class M>
class Likelihood : public QUESO::BaseScalarFunction<V, M>
{
public:
  Likelihood(const char * prefix, const QUESO::VectorSet<V, M> & domain);
  virtual ~Likelihood();
  virtual double lnValue(const V & domainVector, const V * domainDirection,
      V * gradVector, M * hessianMatrix, V * hessianEffect) const;
  virtual double actualValue(const V & domainVector, const V * domainDirection,
      V * gradVector, M * hessianMatrix, V * hessianEffect) const;

private:
  std::vector<double> m_heights; // heights
  std::vector<double> m_times;   // times
  std::vector<double> m_stdDevs; // uncertainties in time measurements
};
```

Listing 4 File `gravity_likelihood.C`.

```
#include <gravity_likelihood.h>

template<class V, class M>
Likelihood<V, M>::Likelihood(const char * prefix,
    const QUESO::VectorSet<V, M> & domain)
  : QUESO::BaseScalarFunction<V, M>(prefix, domain),
    m_heights(0),
    m_times(0),
    m_stdDevs(0)
{
  // Observational data
  double const heights[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110,
                            120, 130, 140};

  double const times  [] = {1.41, 2.14, 2.49, 2.87, 3.22, 3.49, 3.81, 4.07,
                            4.32, 4.47, 4.75, 4.99, 5.16, 5.26};

  double const stdDevs[] = {0.020, 0.120, 0.020, 0.010, 0.030, 0.010, 0.030,
                            0.030, 0.030, 0.050, 0.010, 0.040, 0.010, 0.09};

  std::size_t const n = sizeof(heights) / sizeof(*heights);
  m_heights.assign(heights, heights + n);
  m_times.assign(times, times + n);
  m_stdDevs.assign(stdDevs, stdDevs + n);
}

template<class V, class M>
```

```
Likelihood<V, M>::~Likelihood()
{
  // Deconstruct here
}

template<class V, class M>
double
Likelihood<V, M>::lnValue(const V & domainVector, const V * domainDirection,
    V * gradVector, M * hessianMatrix, V * hessianEffect) const
{
  double g = domainVector[0];

  double misfitValue = 0.0;
  for (unsigned int i = 0; i < m_heights.size(); ++i) {
    double modelTime = std::sqrt(2.0 * m_heights[i] / g);
    double ratio = (modelTime - m_times[i]) / m_stdDevs[i];
    misfitValue += ratio * ratio;
  }

  return -0.5 * misfitValue;
}

template<class V, class M>
double
Likelihood<V, M>::actualValue(const V & domainVector,
    const V * domainDirection, V * gradVector, M * hessianMatrix,
    V * hessianEffect) const
{
  return std::exp(this->lnValue(domainVector, domainDirection, gradVector,
      hessianMatrix, hessianEffect));
}

template class Likelihood<QUESO::GslVector, QUESO::GslMatrix>;
```

5.7 Running the Gravity Example with Several Processors

QUESO requires MPI, so any compilation of the user's statistical application will look like this:

```
mpicxx -I/path/to/boost/include -I/path/to/gsl/include \
       -I/path/to/queso/include -L/path/to/queso/lib \
       YOURAPP.C -o YOURAPP -lqueso
```

This will produce a file in the current directory called YOURAPP. To run this application with QUESO in parallel, you can use the standard mpirun command:

```
mpirun -np N ./YOURAPP
```

Here N is the number of processes you would like to give to QUESO. They will be divided equally among the number of chains requested (see env_numSubEnvironments below). If the number of requested chains does not divide the number of processes, an error is thrown.

Even though the application described in Sect. 5.6 is a serial code, it is possible to run it using more than one processor, i.e., produce multiple chains. Supposing

the user's workstation has $N_p = 8$ processors, then, the user my choose to have $N_s = 1, \ldots, 8$ subenvironments. This complies with the requirement that the total number of processors in the environment (eight) must be a multiple of the specified number of subenvironments (one). Each subenvironment has only one processor because the forward code is serial.

Thus, to build and run the application code with $N_p = 8$, and $N_s = 8$ subenvironments, the must set the variable `env_numSubEnvironments = 8` in the input file and enter the following command lines:

```
mpirun -np 8 ./gravity_gsl gravity_inv_fwd.inp
```

The steps above will create a total number of eight raw chains, of size defined by the variable `ip_mh_rawChain_size`. `QUESO` internally combines these eight chains into a single chain of size $8 \times$ `ip_mh_rawChain_size` and saves it in a file named according to the variable `ip_mh_rawChain_dataOutputFileName`. `QUESO` also provides the user with the option of writing each chain—handled by its corresponding processor—in a separate file, which is accomplished by setting the variable `ip_mh_rawChain_dataOutputAllowedSet = 0 1 ...Ns-1`.

Note: Although the discussion in the previous paragraph refers to the raw chain of a SIP, the analogous is true for the filtered chains (SIP), and for the samples employed in the SFP (`ip_mh_filteredChain_size`, `fp_mc_qseq_size` and `fp_mc_qseq_size`, respectively). See the `QUESO` user's manual for further details.

5.8 Data Post-processing and Visualization

5.8.1 Statistical Inverse Problem

`QUESO` supports both python and Matlab for post-processing. This section illustrates several forms of visualizing `QUESO` output and discusses the results computed by `QUESO` with the code of Sect. 5.6. For Matlab-ready commands for post-processing the data generated by `QUESO`, refer to the `QUESO` user's manual.

It is quite simple to plot, using Matlab, the chain of positions used in the DRAM algorithm implemented within `QUESO`. Figure 54.3a, b show what raw and filtered chain output look like, respectively.

Predefined Matlab and numpy/matplotlib functions exist for converting the raw or filtered chains into histograms. The resulting output can be seen in Fig. 54.4a, b, respectively.

There are also standard built-in functions in Matlab and SciPy to compute kernel density estimates. Resulting output for the raw and filtered chains can be seen in Fig. 54.5a, b, respectively.

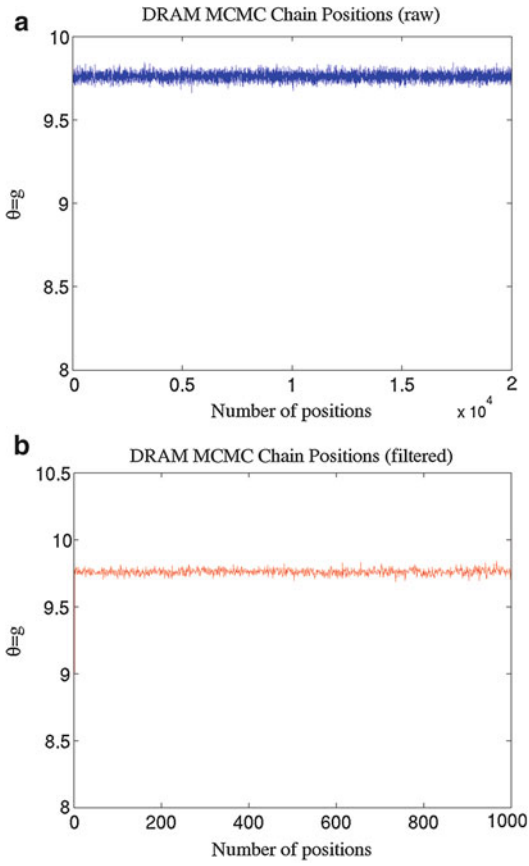

Fig. 54.3 MCMC raw chain with 20,000 positions and a filtered chain with lag of 20 positions (**a**) Raw chain. (**b**) Filtered chain

5.9 Infinite-Dimensional Inverse Problems

QUESO has functional but limited support for solving infinite-dimensional inverse problems. Infinite-dimensional inverse problems are problems for which the posterior distribution is formally defined on a function space. After implementation, this distribution will lie on a discrete space, but the MCMC algorithm used is robust to mesh refinement of the underlying function space.

There is still substantial work to be done to bring the formulation of these class of inverse problems in QUESO in line with that of the finite-dimensional counterpart described above, but what currently exists in QUESO is usable. The reason for the departure in design pattern to that of the finite-dimensional code is that for infinite-dimensional problems, QUESO must be agnostic to any underlying vector type representing the random functions that are sampled. To achieve this, a finite element back end is needed to represent functions. There are many choices of finite element libraries that are freely available to download and use, and the design of the infinite-dimensional part of QUESO is such that addition of new back ends should

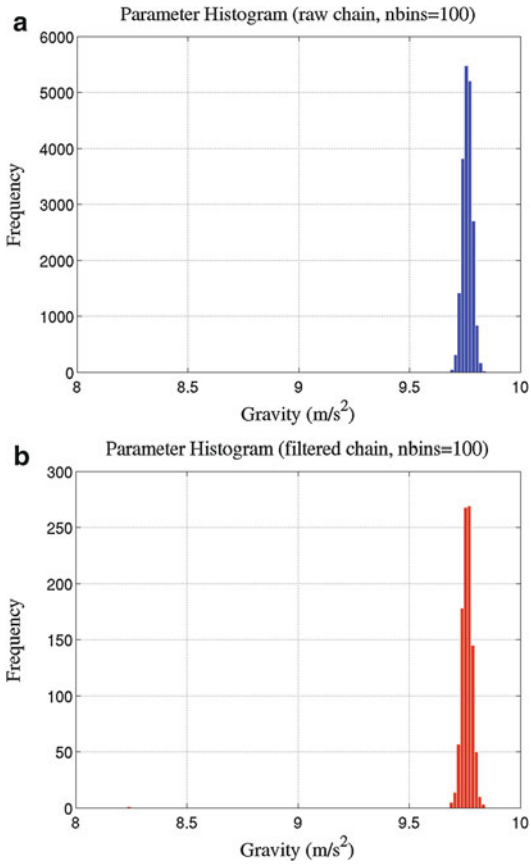

Fig. 54.4 Histograms of parameter $\theta = g$. (**a**) Raw chain. (**b**) Filtered chain

be attainable without too much effort. `libMesh` is the default and only choice currently available in `QUESO`. `libMesh` is open source and freely available to download and use. Visit the `libMesh` website for further details: http://libmesh.github.io

We proceed with showing a concrete example of how to formulate an infinite-dimensional inverse problem in `QUESO`.

First, we assume the user has access to a `libMesh::Mesh` object on which their forward problem is defined. In what follows, we shall call this object `mesh`.

5.9.1 Defining the Prior

Currently, the only measure you can define is a Gaussian measure. This is because Gaussian measures are well-defined objects on function space and their properties are well understood.

To define a Gaussian measure on function space, one needs a mean function and a covariance operator. `QUESO` has a helper object to help the user build functions

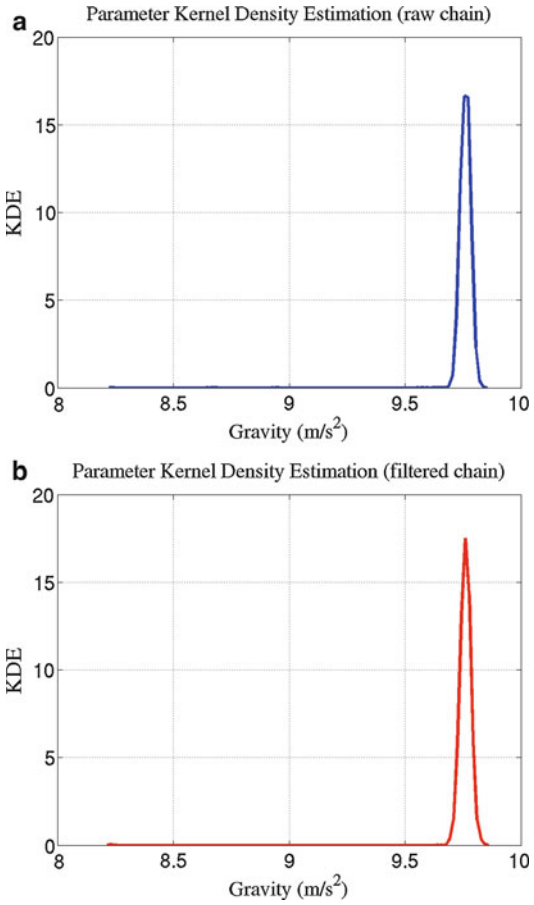

Fig. 54.5 Kernel density estimation (**a**) Raw chain. (**b**) Filtered chain

and operators called `FunctionOperatorBuilder`. This object has properties that are set by the user that define the type and order of the finite elements used by `libMesh` to represent functions:

```
// Use a helper object to define some of the properties
of our samples
QUESO::FunctionOperatorBuilder fobuilder;
fobuilder.order = "FIRST";
fobuilder.family = "LAGRANGE";
fobuilder.num_req_eigenpairs = num_pairs;
```

This object will be passed to the constructors of functions and operators and will instruct `libMesh`, in this case, to use first-order Lagrange finite elements. The `num_req_eigenpairs` variable dictates how many eigenpairs to solve for in an eigenvalue problem needed for the construction of random functions. The more eigenpairs used in the construction of Gaussian random functions, the more high-

frequency information is present in the function. The downside to asking for a large number of eigenpairs is that the solution of the eigenvalue problem will take longer. Solving the eigenvalue problem, however, is a one-time cost. The details of the construction of Gaussian random fields can be found in [15–17]. To define a function, one can do the following:

```
QUESO::LibMeshFunction mean(fobuilder, mesh);
```

This function is initialized to be exactly zero everywhere. For more fine-grained control over point values, one can access the internal `libMesh EquationSystems` object using the `get_equation_systems()` method.

Specifying a Gaussian measure on a function space is often more convenient to do in terms of the precision operator rather than the covariance operator. Currently, the only precision operators available in `QUESO` are powers of the Laplacian operator. However, the design of the class hierarchy for precision operators in `QUESO` should be such that implementation of other operators is easily achievable. To create a Laplacian operator in `QUESO` one can do the following:

```
QUESO::LibMeshNegativeLaplacianOperator
                                precision(fobuilder, mesh);
```

The Gaussian measure can then be defined by the mean and precision above (where the precision can be taken to a power) as such:

```
QUESO::InfiniteDimensionalGaussian
                    mu(env, mean, precision, alpha, beta);
```

Here `beta` is the coefficient of the precision operator, and `alpha` is the power to raise the precision operator to.

5.9.2 Defining the Likelihood

Defining the likelihood is very similar to the ball drop example. We have to subclass `InfiniteDimensional LikelihoodBase` and implement the `evaluate(FunctionBase & flow)` method. This method should return the logarithm of the likelihood distribution evaluated at the point `flow`.

One's specific likelihood implementation will vary from problem to problem, but an example, which is actually independent of `flow`, is shown here for completeness:

```
double
Likelihood::evaluate(QUESO::FunctionBase & flow)
{
  const double obs_stddev = this->obs_stddev();
  const double obs = gsl_ran_gaussian(this->r, obs_stddev);
  return obs * obs / (2.0 * obs_stddev * obs_stddev);
}
```

The reader is reminded that a full working implementation of this example is available in the source tree. See http://libqueso.com.

5.9.3 Sampling the Posterior

The following code will use the prior and the likelihood defined above to set up the inverse problem and start sampling:

```
QUESO::InfiniteDimensionalMCMCSamplerOptions opts(env, "");

// Set the number of iterations to do
opts.m_num_iters = 1000;

// Set the frequency with which we save samples
opts.m_save_freq = 10;

// Set the RWMH step size
opts.m_rwmh_step = 0.1;

// Construct the sampler, and set the name of the output file (will only
// write HDF5 files)
QUESO::InfiniteDimensionalMCMCSampler s(env, mu, llhd, &opts);

for (unsigned int i = 0; i < opts.m_num_iters; i++) {
  s.step();
  if (i % 100 ==  0) {
    std::cout << "sampler iteration: " << i << std::endl;
    std::cout << "avg acc prob is: " << s.avg_acc_prob() << std::endl;
    std::cout << "l2 norm is: " << s.llhd_val() << std::endl;
  }
}
```

The infinite-dimensional inverse problem work is still considered experimental but should produce meaningful results for a large class of simple problems. Work is ongoing to bring the user interface in line with that of the finite-dimensional inverse problem API.

6 Extensibility

QUESO is written in C++. The choice of the language inspired design decisions that the user can take advantage of. One such benefit of having a well-defined inverse problem setup and workflow is that the user is offered the freedom to extend many of the abstract base classes in QUESO. A good example of this we have seen already is the specification of the likelihood distribution by subclassing BaseScalarFunction.

In this section we will take this a step further and show how to extend some of the other classes in QUESO to define a custom prior measure. All of the classes we deal with here have their relationships with other objects discussed in Sect. 7.2.

6.1 Custom Priors

We will look at one of the existing measures in QUESO to get a feel for a how a measure QUESO is built. Take, for example, the Gamma distribution.

In QUESO, the user will interact with a Gamma measure by instantiating a GammaVectorRV class. This object has two main members that QUESO is

interested in, an object representing a probability distribution function and an object called a "realizer" through which random variates are drawn. These classes are called `GammaJointPdf` and `GammaVectorRealizer`, respectively.

The user does not, usually, need to interact with the probability distribution function or the realizer; these are objects that `QUESO` will utilize during the execution of the Markov chain Monte Carlo procedure.

6.1.1 PDF Objects

Probability distribution functions are represented by C++ objects. If the user wishes to create a custom prior measure, for example, then they will also have to implement a probability distribution class. The probability distribution class must derive from the `BaseJointPdf`. The `BaseJoinPdf` class subclasses from `BaseScalarFunction`, as we have seen before, and therefore any probability distribution class must implement the `lnValue` and `actualValue` methods. These methods have exactly the same purpose as when the user defines their likelihood. That is, `lnValue` returns the log of the probability distribution function evaluated at `domainVector`, and `actualValue` returns the actual value of the distribution evaluated at `domainVector`.

`BaseJointPdf` has an extra method called `computeLogOfNomalization Factor` and so this must also be implemented. This method computes the logarithm of the normalizing constant of the probability distribution. If it is known analytically, the user can implement it here. For many distributions, this is not known analytically. In these circumstances one can use the `numSamples` argument to approximate this quantity using samples from the distribution instead. A basic algorithm for computing the log of the normalizing constant from samples is implemented in the `commonComputeLogOfNormalizationFactor` method of `BaseJointPdf`. Indeed the computation of the log of the normalization constant for the Gamma distribution is handed off to this method:

```
template<class V, class M>
double
GammaJointPdf<V, M>::computeLogOfNormalizationFactor(
    unsigned int numSamples,
    bool updateFactorInternally) const
{
  value =
    BaseJointPdf<V,M>::commonComputeLogOfNormalizationFactor(
      numSamples, updateFactorInternally);
  return value;
}
```

Notice that when we defined a custom likelihood object, we only subclassed `BaseScalarFunction` and not `BaseJointPdf`. This is because for most applications, the likelihood is not a probability distribution since it does not integrate to 1. Furthermore, it avoids needing to implement the `computeLogOfNormalizationFactor` method. This is because the normalizing constant is usually not known analytically, and computing it by samples is often intractable for large engineering problems. Note, however, that

the normalizing constant for the likelihood is not needed since MCMC methods do not require knowledge of any normalizing constant in order to draw random samples. This is crystallized in the following section.

6.1.2 Realizer Objects

Realizer objects are objects that `QUESO` interacts with to draw random samples from the appropriate distribution. A realizer object must subclass `BaseVectorRealizer` and must therefore implement the `realization(V & nextValues) const` method. This method fills the `nextValues` vector with a random draw from the associated distribution. The size of the vector `nextValues` is equal to the dimension of the state space on which the measure is defined.

In the case of the Gamma distribution, `QUESO` falls back to `GSL` to draw samples that are Gamma distributed.

A warning to the user: it is possible to define a measure on a space that is improper. In this case drawing realizations from the associated realizer object produces meaningless results.

6.1.3 Random Variable Objects

Random variable objects, named `*VectorRV` in `QUESO`, are encapsulating objects that hold references to the associated probability distribution function object and the associated realizer object. A random variable object must subclass `BaseVectorRV` which implements the getter methods `realizer()` and `pdf()` that return references to the realizer object and PDF object, respectively.

The user never has to deal with constructing the PDF object or the realizer object explicitly. Construction of these objects is handled by the random variable object's constructor.

7 The QUESO Design and Implementation

7.1 Software Engineering

High-quality software is essential for developing, analyzing, and scaling up new UQ algorithmic ideas involving complex simulation codes running on HPC platforms. `QUESO` helps researchers to bootstrap statistical inverse problems for large-scale models widely seen in the physics and engineering domains in parallel compute environments. With ongoing effort to enhance the API in terms of extensibility (see Sect. 10.3), in the future it will be possible to quickly prototype new algorithms in a sophisticated computation environment, rather than first coding and testing them with a scripting language and only then recoding in a C++/MPI environment. `QUESO` also allows researchers to more naturally translate the mathematical language present in algorithms to a concrete program in the library and to concentrate their efforts on algorithmic, load balancing, and parallel scalability issues.

We utilize various community tools to manage the `QUESO` development cycle. Source code traceability is provided via Git, and the GNU Autotools suite is used to

provide a portable, flexible build system, with the standard GNU package pattern: `configure; make; make check; make install` steps. We also utilize GitHub for project management, which provides a web-based mechanism to manage releases, milestone developments, issues, bugs, and source code changes. In case the build system or application development processes change, please consult the website (http://libqueso.com) for a detailed and up-to-date guide on how to build and install `QUESO`.

As of the latest `QUESO` release, 0.53.0, the library is comprised of approximately 73,000 source lines of code, with the vast majority of this instantiated across approximately 200 C/C++ source files and headers. At a minimum, `QUESO` compilation requires `MPI` and linkage against two external libraries: `boost` and `GSL`. `QUESO` also has several optional dependencies that enable additional functionality: `Teuchos`, `GRVY`, `HDF5`. The optional infinite-dimensional capabilities of `QUESO` in particular require `libMesh` and `HDF5`.

We employ an active regression testing, with approximately thirty regression tests, and can test latest GitHub builds using Travis-CI in order to have a continuous integration analysis of source code commits.

Contributing `QUESO` has been made easy with the recent explosion in popularity of GitHub. We employ the feature branch model by Driessen (http://nvie.com/posts/a-successful-git-branching-model), and further instructions for contribution to `QUESO` can be found by mirroring some of the other contributions we have merged (https://github.com/libqueso/queso/issues).

7.2 `QUESO` Internals

In this subsection, we show and discuss several of the inheritance diagrams behind the principle objects in the `QUESO` library. This is in order to:

- Document the `QUESO` internal structure
- Provide context for leveraging the existing `QUESO` objects in extending the library (as in Sect. 6).

This subsection addresses some of the C++ objects for the finite-dimensional Bayesian inverse problem. Objects associated with the infinite-dimensional problem exist and are available on the online documentation, but are not discussed here since development work to get the finite- and infinite-dimensinoal APIs consistent with each other is ongoing.

`BaseScalarFunction` is a templated base class for handling generic scalar functions. This provides a high-level interface and member functions for the `QUESO` generic class, `BaseJointPDF`, which is discussed below.

`BaseJointPdf` is a templated (base) class for handling joint PDFs. For example, Fig. 54.6 shows the inheritance of the Gamma joint PDF class, which is a derived class from the `BaseScalarFunction` class. `QUESO` presently has several provided joint PDFs for a wide variety of statistical distributions, including: `InvLogitGaussianJointPdf`, `ConcatenatedJointPdf`,

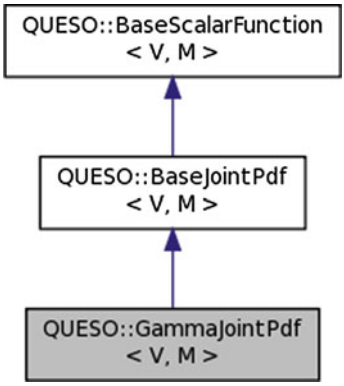

Fig. 54.6 Class Reference for the Gamma Joint PDF

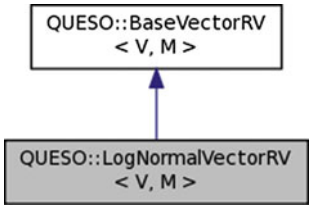

Fig. 54.7 Class reference for the LogNormalVectorRV

`GaussianJointPdf`, `BaseJointPdf`, `BayesianJointPdf`, `LogNormalJointPdf`, `PoweredJointPdf`, `BetaJointPdf`, `GammaJointPdf`, `InverseGammaJointPdf`, `WignerJointPdf`, `GenericJointPdf`, `UniformJointPdf`, `JeffreysJointPdf`, `GenericScalarFunction`, and `ConstantScalarFunction`. However, implementing a new distribution is intended to be straightforward and is detailed in Sect. 6.

Another useful internal `QUESO` object, `BaseVectorRV`, is a templated base class for handling vector random variables. For example, Fig. 54.7 shows the inheritance diagram of the `LogNormalRV` class, which is a class that contains member functions and associated utilities to provide a random vector of draws from a `LogNormal` distribution.

Presently included in `QUESO` are the following: `GaussianVectorRV`, `GenericVectorRV`, `BetaVectorRV`, `GammaVectorRV`, `InverseGammaVectorRV`, `InvLogitGaussianVectorRV`, `JeffreysVectorRV`, `LogNormalVectorRV`, `UniformVectorRV`, and `WignerVectorRV`. In other words, nearly all canonical distributions from classical statistics are already available in the library. However, as stated above, `QUESO` is designed with extensibility in mind, and the user can implement any `*VectorRV` by deriving from the `BaseVectorRV` class. In principle, this permits a series of draws from any distribution.

Another important base class contained within `QUESO` is the realizer object, `BaseVectorRealizer`. A realizer is an object that, simply put, contains a `realization()` operation that returns a sample of a random variable. `BaseVectorRealizer` is therefore an abstract base class that provides the necessary interface for sampling from random variables. As before, the realizer

object contains most of the common statistical distributions. It also contains a sequence realizer class for storing samples of a MH algorithm.

8 Algorithms

8.1 DRAM

A simple Metropolis-Hastings sampling algorithm [4] can be improved by adding both "Delayed Rejection" [18–21] and "Adaptive Metropolis". Taken together, these form the "DRAM" algorithm, which is available in `QUESO`. In particular, the `QUESO` implements the DRAM algorithm of Haario, Laine, Mira, and Saksman [7].

A "vanilla" Metropolis-Hastings sampler involves a proposal at each step, and accepts or rejects this proposal based on the ratio between proposal and prior likelihoods. Typically, the proposal is drawn from some fixed distribution, such as a Gaussian distribution, with fixed covariance and a mean centered at the value of the current state of the chain. However, this has several deficiencies. Should the proposal variance be set too high, many proposals will be rejected. This is undesirable, as it increases the auto-correlation of the chain. Furthermore, should the target distribution deviate greatly from the proposal distribution, the proposal will not match the local shape of the distribution, resulting in poor sampling.

Delayed rejection attempts to circumvents these issues. Before rejecting a sample, a series of back-up proposals each with successively smaller jumps in state space are pushed through the Metropolis-Hastings acceptance probability rejection. They are tested in order of decreasing jump size, and if one of them is accepted, the sampler continues. If they are all rejected, the sampler rejects the sample and starts again.

Conversely, when the proposal variance is too small to efficiently sample the target distribution, the sampler will randomly walk through regions of higher likelihood in the posterior distribution, without efficiently sampling the tails. This results in too high an acceptance rate.

In order to mitigate this, Adaptive Metropolis sampling continuously adapts the proposal covariance. This is accomplished by using the covariance of the history of the Markov chain as the proposal covariance matrix of the Gaussian proposal distribution instead of the arbitrary proposal covariance imposed at the start. Adapting the proposal to match the posterior covariance structure results in a better chain performance than a static proposal covariance.

8.2 Multilevel

Multilevel Monte Carlo [6] is an algorithm available in `QUESO` that attempts to sample probability distributions with multiple modes. Sampling multi-modal distributions is a heavily researched topic. The way multilevel Monte Carlo attempts to solve the problem of metastability in Markov chains is by "heating up" the posterior distribution to flatten out some of the modes, allowing a Markov chain to

sample the flattened distribution and then "cooling down" the posterior distribution before doing a final sampling run. The idea is identical to that of simulated tempering or simulated annealing, except that the multilevel algorithm allows for convenient and efficient computation of the posterior normalizing constant. This constant is usually intractable to compute but is essential for Bayesian model selection purposes.

8.3 Preconditioned Crank-Nicolson

The preconditioned Crank-Nicolson proposal [15] is used by `QUESO` for solving infinite-dimensional Bayesian inverse problems (Sect. 5.9). This particular form of proposal is typical for sampling on formally infinite-dimensional spaces since the Metropolis-Hastings acceptance probability remains unchanged when the state undergoes mesh refinement, a popular technique in large-scale engineering models involving the solution of partial different equations by finite element methods.

9 Input File

Here we provide some of the default input file options `QUESO` recognizes. For detailed descriptions of the behavior of each option and how they interact with other options, consult the online `QUESO` documentation. For example, for the description of each DRAM option, consult the documentation for the `MhOptionsValues` object. For the description of each `FullEnvironment` option, see the documentation for the `EnvOptionsValues` object. The documentation for these is available at http://libqueso.com (Tables 54.1, 54.2, 54.3, 54.4, and 54.5).

Table 54.1 Input file options for a QUESO environment

Option name	Default	Description
`env_help`		Produces help message for environment class
`env_numSubEnvironments`	1	Number of subenvironments
`env_subDisplayFileName`	`"."`	Output filename for sub-screen writing
`env_subDisplayAllowAll`	0	Allows all subenvironments to write to output file
`env_subDisplayAllowedSet`	`""`	Subenvironments that will write to output file
`env_displayVerbosity`	0	Sets verbosity
`env_syncVerbosity`	0	Sets synchronized verbosity
`env_seed`	0	Set seed

Table 54.2 Input file options for a QUESO statistical inverse problem

Option name	Default	Description
`ip_help`		Produces help message for statistical inverse problem
`ip_computeSolution`	1	Computes solution process
`ip_dataOutputFileName`	"."	Name of data output file
`ip_dataOutputAllowedSet`	""	Subenvironments that will write to data output file

Table 54.3 Input file options for a QUESO DRAM solver

Option name	Default value
`mh_dataOutputFileName`	"."
`mh_dataOutputAllowAll`	0
`mh_initialPositionDataInputFileName`	"."
`mh_initialPositionDataInputFileType`	"m"
`mh_initialProposalCovMatrixDataInputFileName`	"."
`mh_initialProposalCovMatrixDataInputFileType`	"m"
`mh_rawChainDataInputFileName`	"."
`mh_rawChainDataInputFileType`	"m"
`mh_rawChainSize`	100
`mh_rawChainGenerateExtra`	0
`mh_rawChainDisplayPeriod`	500
`mh_rawChainMeasureRunTimes`	1
`mh_rawChainDataOutputPeriod`	0
`mh_rawChainDataOutputFileName`	"."
`mh_rawChainDataOutputFileType`	"m"
`mh_rawChainDataOutputAllowAll`	0
`mh_filteredChainGenerate`	0
`mh_filteredChainDiscardedPortion`	0.
`mh_filteredChainLag`	1
`mh_filteredChainDataOutputFileName`	"."
`mh_filteredChainDataOutputFileType`	"m"
`mh_filteredChainDataOutputAllowAll`	0
`mh_displayCandidates`	0
`mh_putOutOfBoundsInChain`	1
`mh_tkUseLocalHessian`	0
`mh_tkUseNewtonComponent`	1
`mh_drMaxNumExtraStages`	0
`mh_drDuringAmNonAdaptiveInt`	1
`mh_amKeepInitialMatrix`	0
`mh_amInitialNonAdaptInterval`	0
`mh_amAdaptInterval`	0
`mh_amAdaptedMatricesDataOutputPeriod`	0
`mh_amAdaptedMatricesDataOutputFileName`	"."
`mh_amAdaptedMatricesDataOutputFileType`	"m"
`mh_amAdaptedMatricesDataOutputAllowAll`	0
`mh_amEta`	1.
`mh_amEpsilon`	1×10^{-5}
`mh_enableBrooksGelmanConvMonitor`	0
`mh_BrooksGelmanLag`	100

Table 54.4 Input file options for a QUESO multilevel solver

Option name	Default value
ml_restartOutput_levelPeriod	0
ml_restartOutput_baseNameForFiles	"."
ml_restartOutput_fileType	"m"
ml_restartInput_baseNameForFiles	"."
ml_restartInput_fileType	"m"
ml_stopAtEnd	0
ml_dataOutputFileName	"."
ml_dataOutputAllowAll	0
ml_loadBalanceAlgorithmId	2
ml_loadBalanceTreshold	1.0
ml_minEffectiveSizeRatio	0.85
ml_maxEffectiveSizeRatio	0.91
ml_scaleCovMatrix	1
ml_minRejectionRate	0.50
ml_maxRejectionRate	0.75
ml_covRejectionRate	0.25
ml_minAcceptableEta	0.
ml_totallyMute	1
ml_initialPositionDataInputFileName	"."
ml_initialPositionDataInputFileType	"m"
ml_initialProposalCovMatrixDataInputFileName	"."
ml_initialProposalCovMatrixDataInputFileType	"m"
ml_rawChainDataInputFileName	"."
ml_rawChainDataInputFileType	"m"
ml_rawChainSize	100
ml_rawChainGenerateExtra	0
ml_rawChainDisplayPeriod	500
ml_rawChainMeasureRunTimes	1
ml_rawChainDataOutputPeriod	0
ml_rawChainDataOutputFileName	"."
ml_rawChainDataOutputFileType	"m"
ml_rawChainDataOutputAllowAll	0
ml_filteredChainGenerate	0
ml_filteredChainDiscardedPortion	0.
ml_filteredChainLag	1
ml_filteredChainDataOutputFileName	"."
ml_filteredChainDataOutputFileType	"m"
ml_filteredChainDataOutputAllowAll	0
ml_displayCandidates	0
ml_putOutOfBoundsInChain	1
ml_tkUseLocalHessian	0
ml_tkUseNewtonComponent	1

(continued)

Table 54.4 (continued)

ml_drMaxNumExtraStages	0
ml_drScalesForExtraStages	0
ml_drDuringAmNonAdaptiveInt	1
ml_amKeepInitialMatrix	0
ml_amInitialNonAdaptInterval	0
ml_amAdaptInterval	0
ml_amAdaptedMatricesDataOutputPeriod	0
ml_amAdaptedMatricesDataOutputFileName	"."
ml_amAdaptedMatricesDataOutputFileType	"m"
ml_amAdaptedMatricesDataOutputAllowAll	0
ml_amEta	1.
ml_amEpsilon	1.e-5

Table 54.5 Input file options for a QUESO pCN solver

Option name	Default value
infmcmc_dataOutputDirName	"chain"
infmcmc_dataOutpuFileName	"out.h5"
infmcmc_num_iters	1000
infmcmc_save_freq	1
infmcmc_rwmh_step	1e-2

10 Conclusions

We conclude this chapter with a discussion of several of the areas the QUESO development team is investing time into implementing, extending, and improving along with some of the newest features recently made available in v0.53.0. Previously, we have covered only the basics of how to interact with QUESO and to provide a resource that is accessible and can be used to bootstrap a user's statistical inverse problem quickly and efficiently. With this in mind, there are still many areas in which QUESO can improve to become more user friendly, consistent, and extensible. In what follows, we discuss some major areas of development that would likely encourage widespread adoption of QUESO in the computational applied mathematics and engineering community.

10.1 QUESO-Provided Likelihoods

In many large-scale physics and engineering-based experimental settings, it is often the case that observations of a physical quantity are performed several times. These observations are then averaged to homogenize the effect of experimental observation error. In the case of independent experimental errors, this average will be normally distributed. Therefore, a reasonable choice for a likelihood in many applications would be a Gaussian.

At present, the user must derive from `BaseScalarFunction` and implement `lnValue` explicitly. This is a tedious task if all that is needed is the standard Gaussian error in the Euclidean 2-norm:

$$\mathbb{P}(y|\theta) = Z \exp\left(\frac{1}{2}\left(\mathscr{G}(\theta) - y\right)^{\top} \Sigma^{-1} \left(\mathscr{G}(\theta) - y\right)\right), \tag{54.11}$$

where Z is a normalizing constant.

A recently released and much leaner approach is to provide an abstract base class of `BaseScalarFunction` called `BaseGaussianLikelihood` with a pure virtual method called `evaluateModel` that asks for the output of the map $\mathscr{G}$ at the point `domainVector`. Equipped with an implementation of `lnValue` that computes the log of (54.11), the user would only need to provide Σ and y, which can be passed in from the constructor. An example follows:

```
template<class V, class M>
class Likelihood : public
    QUESO::GaussianLikelihoodScalarCovariance<V, M>
{
public:

  Likelihood(const char * prefix,
      const QUESO::VectorSet<V, M> & domain,
      const V & observations, double variance)
    : QUESO::GaussianLikelihoodScalarCovariance<V, M>(
        prefix, domain,
        observations, variance)
  { }

  virtual ~Likelihood() { }

  virtual void evaluateModel(const V & domainVector,
      const V * domainDirection,
      V & modelOutput, V * gradVector,
      M * hessianMatrix,
      V * hessianEffect) const
  {
    // Evaluate model and fill up the modelOutput
    // variable
    int dim = modelOutput.sizeLocal();
    for (unsigned int i = 0; i < dim; i++) {
      modelOutput[i] = 1.0;  // Replace this with
                             // the output from
                             // your model
    }
  }
};
```

Here the user would pass an instance of `Likelihood` to `StatisticalInverseProblem`, as per usual.

Extensions of this idea are also available, where one wishes to treat Σ as a hyperparameter to be sampled along with θ in so-called "hierarchical Bayesian" methods. The design described above is easily applied to this situation.

Ongoing work is being invested in developing other pre-made likelihood objects representing other likelihood forms that are commonly used.

10.2 Emulators

The two main forms of emulation used in the statistical modeling community are Gaussian processes and generalized polynomial chaos. These are both important methods in statistical inference as they can considerably reduce the computational cost of computing the posterior.

Gaussian process emulators, similar to the ready-made Gaussian likelihoods discussed in the previous section, are also a form of baked likelihood, but where the user is not required to implement a method returning the output of $\mathcal{G}$. For Gaussian process emulators, the user would only need to instantiate an emulator with a specific dataset and observational error covariance matrix. The rest of the statistical application the user writes is identical to any other statistical application and the output (samples) is processed as per usual.

Generalized polynomial chaos methods require different algorithms for solution, since no Markov chain Monte Carlo is done. This type of emulator is not currently on the `QUESO` development road map for the near future, but contributions in the area are more than welcome.

As of `QUESO` v0.53.0, the only supported emulator is a linear interpolation of model output values. Interested users should consult the documentation and, in particular, the example called `4d_interp.C`.

10.3 API Considerations

As mentioned in the infinite-dimensional example, the infinite-dimensional and finite-dimensional APIs are not aligned. Although the user interacts with only one of these APIs at any given time, an aligned API structure exposes the opportunity for algorithms designed on function space, which tend to be more robust algorithms, to be used in the finite-dimensional setting. Moreover, an aligned API eases the maintenance, documentation, and testing burden.

Currently, there are only two (finite-dimensional) algorithms the user can use, DRAM and multilevel. At present, there is no organized structure that Markov chains (`MetropolisHastingsSG` objects) inherit from, meaning that there is a significant hurdle involved in bootstrapping one's own MCMC algorithm for the purposes of testing and research. Just as above, a consistent class hierarchy for MCMC algorithms would ease the burden for software maintenance.

A rather cumbersome design choice made early on in the development of QUESO was the hot-swappability of vector and matrix implementations for all of QUESO's classes. The net result of this is that any QUESO class that involves an operation with a vector or a matrix is templated around the type of that vector or matrix. This was done to ensure that optimized code could be generated that dealt with the specifics of each vector and matrix library. Assuming that, in high-performance uncertainty quantification, likelihood evaluations are the dominating cost of Markov chain Monte Carlo sampling, one need not encumber the QUESO API with such templates. Furthermore, a hierarchical class structure for vector and matrix types exists in QUESO and therefore necessitates the run-time overhead of virtual table lookups. Efforts are currently ongoing to enrich the vector and matrix class hierarchy in QUESO sufficiently such that the particulars of vector and matrix implementations still remain opaque but significantly shorten unnecessarily long class names with a negligibly small impact on run-time performance. This enrichment would also allow QUESO to pick a high-tuned vector/matrix implementation at configure time for high-performance problems in exascale compute environments. For example, QUESO's build system could default to using PETSc vectors optimized for multi-core architectures, while the user need not deal explicitly with MPI calls. All parallel logic would be handled under the hood. This offers a pleasing software experience while maintaining performance.

Python has become a very popular environment to do post-processing and visualization in multi-core HPC systems. A desirable feature to have in QUESO would be the automatic generation of python bindings. This would offer the possibility to do uncertainty quantification in statistical inverse problems as a quick-turnaround experiment for cheap forward models in an interpreted language environment. This implementation will likely leverage the Simplified Wrapper and Interface Generator (SWIG) which is not limited to Python and can provide interfaces to many modern programming languages, such as Perl, Python, Ruby, and Tcl.

10.4 Exascale

Uncertainty quantification has pushed the limits of current computational power by requiring many evaluations of large-scale engineering systems described by partial differential equations. Utilizing more information about the system can significantly increase the performance of MCMC algorithms [22–24]. In particular QUESO does not currently implement MCMC algorithms that use gradient or Hessian information to construct proposal distributions. However, the design of the API for the pure virtual methods in `BaseScalarFunction` allows this information to be passed to QUESO easily, in the form of a pointer `V * gradVector`. For more details on the parameters passed to the `lnValue` function, the reader is directed to the QUESO documentation which be found online here: http://libqueso.com.

References

1. Prudencio, E.E., Schulz, K.W.: Euro-Par 2011: Parallel Processing Workshops, pp. 398–407. Springer (2012). http://dx.doi.org/10.1007/978-3-642-29737-3_44
2. Estacio-Hiroms, K.C., Prudencio, E.E.: Quantification of Uncertainty for Estimation, Simulation, and Optimization (QUESO), User's Manual (2008). Unpbulished, http://www.libqueso.com/.
3. Kaipio, J., Somersalo, E.: Statistical and Computational Inverse Problems, vol. 160. Springer, New York (2005)
4. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: J. Chem. Phys. **21**(6), 1087 (1953). doi:10.1063/1.1699114. http://link.aip.org/link/JCPSA6/v21/i6/p1087/s1&Agg=doi
5. Hastings, W.K.: Biometrika **57**(1), 97 (1970). doi:10.1093/biomet/57.1.97. http://biomet.oxfordjournals.org/cgi/doi/10.1093/biomet/57.1.97
6. Cheung, S.H., Prudencio, E.E.: Int. J. Uncertain. Quantif. **2**(3), p. 215–237 (2012)
7. Haario, H., Laine, M., Mira, A., Saksman, E.: Stat. Comput. **16**(4), 339 (2006). doi:10.1007/s11222-006-9438-0. http://link.springer.com/10.1007/s11222-006-9438-0
8. Patil, A., Huard, D., Fonnesbeck, C.J.: J. Stat. Softw. **35**(4), p. 1–81. (2010)
9. Hunter, J.D.: Comput. Sci. Eng. **9**(3), 90 (2007)
10. Foreman-Mackey, D., Hogg, D.W., Lang, D., Goodman, J.: Publ. Astron. Soc. Pac. **125**(925), 306 (2013)
11. Core Team, R.: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria (2013). ISBN 3-900051-07-0, http://www.R-project.org/.
12. Stan Development Team: Stan: a c++ library for probability and sampling, version 2.5.0 (2014). http://mc-stan.org/
13. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: Stat. Comput. **10**(4), 325 (2000)
14. Adams, B.M., Hart, W.E., Eldred, M.S., Dunlavy, D.M., Hough, P.D., Giunta, A.A., Griffin, J.D., Martinez-Canales, M.L., Watson, J.P., Kolda, T.G.: DAKOTA, a Multilevel Parellel Object-oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 4.0 Uers's Manual (2006)
15. Cotter, S.L., Roberts, G.O., Stuart, A.M., White, D.: http://arxiv.org/abs/1202.0709 (2012)
16. Bogachev, V.I.: Gaussian Measures. American Mathematical Society, Providence (1998)
17. Lifshits, M.A.: Gaussian Random Functions. Springer Netherlands (1995)
18. Mira, A.: **LIX**(3–4), 231 (2001)
19. Mira, A.: Stat. Sci. **16**(4), 340 (2002). doi:10.1214/ss/1015346319. http://projecteuclid.org/euclid.ss/1015346319
20. Tierney, L., Mira, A.: Stat. Med. **18**, 2507 (1999)
21. Green, P.J., Mira, A.: Biometrika **88**, 1035 (2001)
22. Girolami, M., Calderhead, B.: J. R. Stat. Soc.: Ser. B (Stat. Methodol.) **73**(2), 123 (2011). doi:10.1111/j.1467-9868.2010.00765.x. http://doi.wiley.com/10.1111/j.1467-9868.2010.00765.x
23. Martin, J., Wilcox, L., Burstedde, C., Ghattas, O.: SIAM J. Sci. Comput. **34**(3), 1460 (2012)
24. Bui-thanh, T., Ghattas, O., Higdon, D.: SIAM J. Sci. Comput. **34**(6), 2837 (2012)