The Parallel C++ Statistical Library 'QUESO': Quantification of Uncertainty for Estimation, Simulation and Optimization

Ernesto E. Prudencio* and Karl W. Schulz

Institute for Computational Engineering and Sciences (ICES),
The University of Texas at Austin, USA
{prudenci,karl}@ices.utexas.edu

Abstract. 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: it should (a) be *sufficiently abstract* in order to handle a large spectrum of models, (b) be *algorithmically extensible*, allowing an easy insertion of new and improved algorithms, and (c) take advantage of *parallel computing*, in order to handle realistic models. Such objectives demand a combination of an *object-oriented design* with robust software engineering practices. QUESO is written in C++, uses MPI, and leverages libraries already available to the scientific community. We describe some UQ concepts, present QUESO, and list planned enhancements.

Keywords: Software Design, Uncertainty Quantification, Parallel MCMC.

1 Introduction

QUESO stands for Quantification of Uncertainty for Estimation, Simulation and Optimization, and it is a library of statistical algorithms and programming classes for *research* on uncertainty quantification (UQ) of mathematical models and their predictions. We have three main objectives for QUESO. It should (a) be *model agnostic*, i.e., it should be able to handle a large spectrum of models; (b) be *algorithmically flexible*, allowing for easy insertion of new and improved algorithms; and (c) take advantage of *parallel computing*, enabling it to be used on realistic problems. Its design then follows three main principles. The library should (d) be *object-oriented*, naturally mapping into the code the *mathematical concepts* present in the models and algorithms; (e) *leverage* existing libraries and packages (e.g. GSL [7], Trilinos [10], PETSc [17], DAKOTA [6]); and (f) have its algorithms implemented in a way such that they become *independent* of the underlying vectors and matrices. Our decision to implement QUESO with the object oriented programming language C++ and with the message passing interface (MPI) standard is consistent with such objectives and principles. Class

* Corresponding author.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 398–407, 2012.

derivation, polymorphism and templating give QUESO the desired levels of abstractness and adaptability, allowing researchers to concentrate their efforts on algorithms rather than spending time on the details of underlying datatypes.

In Section 2 we present some concepts and algorithms supported by QUESO, paving the path for the description, in Section 3, of QUESO's main features and classes. We conclude with a list of planned enhancements in Section 4. Throughout the paper we use boldface letters to denote vector and matrix quantities.

2 Stochastic Models and Algorithms

In order to comprehend an actual phenomenon and to predict the future behavior of the actual system underlying it, one needs to (a) collect experimental data $\mathbf{d}$, and (b) construct a *computational model*, which refers to the combination of a mathematical model with a discretization procedure that enables one to compute an approximate solution using computer algorithms. At its core, a computational model (see Figure 1) is composed of two parts: a vector $\boldsymbol{\theta}$ of n parameters, and a set of governing equations $\mathbf{r}(\boldsymbol{\theta}, \mathbf{u}) = \mathbf{0}$, whose solution $\mathbf{u}$ represents the *state variables*, or model state. By *parameters* we designate various concepts, e.g.,

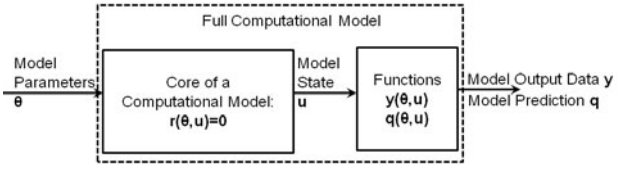

Fig. 1. Computational model: parameters $\boldsymbol{\theta}$, state $\mathbf{u}$, equations $\mathbf{r}$, output $\mathbf{y}$, QoIs $\mathbf{q}$

material properties, coefficients, constitutive parameters, boundary and initial conditions, external forces, parameters for describing the model inadequacy and characteristics of an experimental apparatus. The computational model includes functions for the calculation of *model output data* $\mathbf{y} = \mathbf{y}(\boldsymbol{\theta}, \mathbf{u})$, and the prediction of a vector $\mathbf{q} = \mathbf{q}(\boldsymbol{\theta}, \mathbf{u})$ of m quantities of interest (QoIs). Model output data is compared against experimental data during a model calibration, while QoIs are predicted during a model prediction and might not be directly measurable.

There are many possible sources of uncertainty on procedures (a) and (b) above. First, $\mathbf{d}$ need not be equal to the actual values of observables because of errors in the measurement process. Second, the values of the input parameters to the phenomenon might not be precisely known. Third, the appropriate set of equations governing the phenomenon might not be well understood.

In deterministic models, all parameters are assigned numbers, and no parameter is related to the parametrization of a random variable (RV) or field. As a consequence, a deterministic model assigns a number to each of the components of quantities $\mathbf{u}$, $\mathbf{y}$ and $\mathbf{q}$. In stochastic models, however, at least one parameter is

assigned a probability density function (PDF) or is related to the parametrization of a RV or field, causing $\mathbf{u}$, $\mathbf{y}$ and $\mathbf{q}$ to become random. Parameters that are not directly measurable need to be *estimated* through the solution of an *inverse problem* (IP) [12,14], where $\mathbf{d}$ is given and one estimates the values of $\boldsymbol{\theta}$ that cause $\mathbf{y}$ to best fit $\mathbf{d}$. A computational model might also be used in a *forward problem* (FP), where $\boldsymbol{\theta}$ is given and one computes $\mathbf{u}$, $\mathbf{y}$ and/or $\mathbf{q}$.

QUESO supports a Bayesian approach [11,16], i.e., the posterior PDF

$$\pi_{\text{post}}(\boldsymbol{\theta}|\mathbf{d}, M) = \frac{\pi_{\text{like}}(\mathbf{d}|\boldsymbol{\theta}, M) \cdot \pi_{\text{prior}}(\boldsymbol{\theta}|M)}{\pi(\mathbf{d}|M)} \tag{1}$$

is the solution of statistical IPs, combining the prior information $\pi_{\text{prior}}(\boldsymbol{\theta}|M)$ about the parameters with the likelihood $\pi_{\text{like}}(\mathbf{d}|\boldsymbol{\theta}, M)$ of observing the data $\mathbf{d}$ given parameter values $\boldsymbol{\theta}$. The letter M designates a *model class* [2,4,5] and represents all the assumptions and mathematical statements that are involved in the modeling of the system. The choice of a particular $\boldsymbol{\theta} \in \mathbb{R}^n$ specifies a particular model in the set M of models. The denominator

$$\pi(\mathbf{d}|M) = \int \pi_{\text{like}}(\mathbf{d}|\boldsymbol{\theta}, M) \cdot \pi_{\text{prior}}(\boldsymbol{\theta}|M)\ d\boldsymbol{\theta} \tag{2}$$

is called the *evidence* [3] for M provided by $\mathbf{d}$, and it can be used for ranking *competing candidate model classes* that reflect different modeling choices. Given M, $\mathbf{d}$, and a conditional PDF $\pi(\mathbf{q}|\boldsymbol{\theta}, \mathbf{d}, M_j)$ of $\mathbf{q}$, the predictive PDF of $\mathbf{q}$ is [2]

$$\pi_{\text{predicted}}(\mathbf{q}|\mathbf{d}, M) = \int \pi(\mathbf{q}|\boldsymbol{\theta}, \mathbf{d}, M) \cdot \pi_{\text{post}}(\boldsymbol{\theta}|\mathbf{d}, M)\ d\boldsymbol{\theta}. \tag{3}$$

Stochastic algorithms are used to generate samples from (1). Metropolis Hastings (MH) [13,9] is one of them. Given (a) the target PDF $\pi_{\text{target}} : B \subset \mathbb{R}^n \to \mathbb{R}_+$, up to a multiplicative constant, (b) the number $n_{\text{pos}} \geqslant 2$ of positions in the chain, (c) an initial guess $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^n$, and (d) a symmetric positive definite proposal covariance matrix $\mathbf{C} \in \mathbb{R}^{n \times n}$. MH runs as follows,

01. Do {
02. Generate candidate $\mathbf{z} \in \mathbb{R}^n$ by sampling from $q(\boldsymbol{\theta}^{(k)}, \mathbf{z})$;
03. If $\mathbf{z} \notin supp(\pi_{\text{target}})$ then $\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)}$;
04. If $\mathbf{z} \in supp(\pi_{\text{target}})$ then {
05. Compute acceptance probability $\alpha(\boldsymbol{\theta}^{(k)}, \mathbf{z})$;
06. Generate sample $0 < \tau \leqslant 1$ from uniform RV defined over $(0, 1]$;
07. If $\alpha \geqslant \tau$ then $\boldsymbol{\theta}^{(k+1)} = \mathbf{z}$; else $\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)}$;
09. }
10. Set $k = k + 1$;
11. } while $(k + 1 < n_{\text{pos}})$,

where $q : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_+$ is a proposal distribution [8,12], e.g.

$$q(\mathbf{x}, \mathbf{y}) \propto e^{-\frac{1}{2}\left\{[\mathbf{y}-\mathbf{x}]^T \cdot [\mathbf{C}]^{-1} \cdot [\mathbf{y}-\mathbf{x}]\right\}},$$

$supp(\cdot)$ denotes the support of a function, and

$$\alpha(\boldsymbol{\theta}^{(k)}, \mathbf{z}) = \frac{\pi_{\text{target}}(\mathbf{z})}{\pi_{\text{target}}(\boldsymbol{\theta}^{(k)})} \cdot \frac{q(\mathbf{z}, \boldsymbol{\theta}^{(k)})}{q(\boldsymbol{\theta}^{(k)}, \mathbf{z})}.$$

A sample $\mathbf{z}$ is given by $\boldsymbol{\theta}^{(k)} + \mathbf{C}^{1/2}\mathcal{N}(0, I)$, where $\mathcal{N}(0, I)$ designates a Gaussian RV of zero mean and unit covariance matrix. MH can be improved in different ways, e.g. delayed rejection adaptive Metropolis [8] and stochastic Newton.

Stochastic algorithms are also needed to compute high dimensional integrals like (2). One example is the adaptive multilevel sampling [4] that simultaneously generates samples and computes (2) through a sequence of intermediate PDFs

$$\pi_i(\boldsymbol{\theta}|\mathbf{d}, M) \propto \pi_{\text{like}}(\mathbf{d}|\boldsymbol{\theta}, M)^{\alpha_i} \cdot \pi_{\text{prior}}(\boldsymbol{\theta}|M), \ i = 0, 1, \ldots, L,$$

where $0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{L-1} < \alpha_L = 1$ are adaptively selected exponents. A *load balanced* parallel version is proposed in [15].

Many other statistical calculations are also important, e.g. Monte Carlo (MC) for (3), autocorrelation, kernel density estimations, and accuracy assessment.

3 The QUESO Design and Implementation

Section 2 identified many mathematical entities present in the description of statistical problems and in some algorithms used for their solution. As part of the design, QUESO attempts to conceptually implement these entities in order to allow algorithmic researchers to manipulate them at the library level, as well as for algorithm users (the modelers interested in UQ) to manipulate them at the application level. Examples of entities are vector space $\mathbb{R}^n$; vector subset $B \subset \mathbb{R}^n$; vector $\boldsymbol{\theta} \in B$; matrix $\mathbf{C} \in \mathbb{R}^n \times \mathbb{R}^n$; function $\pi : \mathbb{R}^n \to \mathbb{R}_+$, e.g. joint PDF; function $\pi : \mathbb{R} \to \mathbb{R}_+$, e.g. marginal PDF; function $\pi : \mathbb{R} \to [0, 1]$, e.g. cumulative distribution function; realizer function; function $\mathbf{q} : \mathbb{R}^n \to \mathbb{R}^m$; sequences of scalars; and sequences of vectors. QUESO tries to naturally map such entities through an object-oriented design. Indeed, QUESO C++ classes include vector spaces, subsets, scalar sequences, PDFs, and RVs.

An application using QUESO will fall into three categories: a statistical IP, a statistical FP, or combinations of both. In each problem the user might deal with up to five vectors of potentially very different sizes: parameters $\boldsymbol{\theta}$, state $\mathbf{u}$, output $\mathbf{y}$, data $\mathbf{d}$ and QoIs $\mathbf{q}$. Figure 2 shows the software stack of a typical application that uses QUESO. Even though QUESO deals directly with $\boldsymbol{\theta}$ and $\mathbf{q}$ only, it is usually the case the one of the other three vectors ($\mathbf{u}$, $\mathbf{y}$ and $\mathbf{d}$) will have the biggest number of components and will therefore dictate the size of the minimum parallel environment to be used in a problem. So, for example, even though one processor might be sufficient for handling $\boldsymbol{\theta}$, $\mathbf{y}$, $\mathbf{d}$ and $\mathbf{q}$, eight processors at least might be necessary to solve for $\mathbf{u}$. QUESO currently only requires that the amounts n and m can be handled by the memory available to one processor, which allows the analysis of problems with thousands of parameters and QoIs, a large amount even for state of the art UQ algorithms.

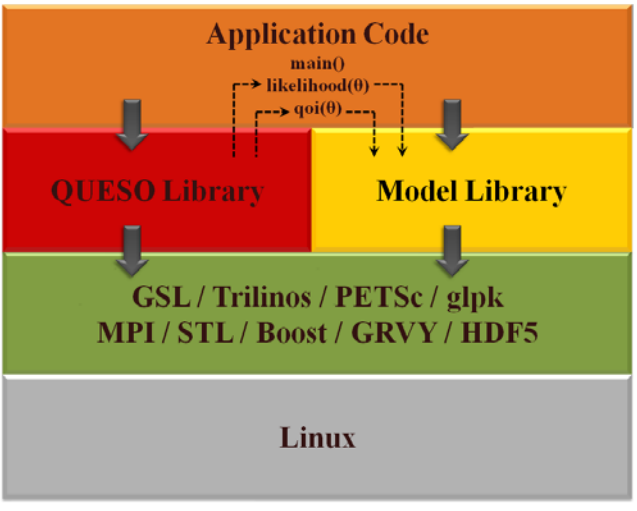

Fig. 2. An application software stack. QUESO requires the input of a likelihood routine $\pi_{\text{like}} : \mathbb{R}^n \to \mathbb{R}_+$ for IPs and of a QoI routine $\mathbf{q} : \mathbb{R}^n \to \mathbb{R}^m$ for FPs. These application level routines provide the bridge between the statistical algorithms in QUESO, physics knowledge in the model library, and relevant calibration and validation data.

QUESO currently supports three modes of parallel execution: an application user may simultaneously run (a) multiple instances of a problem where the physical model requires a single processor, or (b) multiple instances of a problem where the physical model requires multiple processors, or (c) independent sets of types (a) and (b). For example, suppose an user wants to use the MH algorithm to solve a statistical IP, and that 1,024 processors are available. If the physical model is simple enough to be handled efficiently by a single processor, then the user can run 1,024 chains simultaneously, as in case (a). If the model is more complex and requires, say, 16 processors, then the user can run 64 chains simultaneously, as in case (b), with 16 processors per chain. QUESO treats this situation by using only 1 of the 16 processors to handle the chain. When a likelihood evaluation is required, all 16 processors call the likelihood routine simultaneously. Once the likelihood returns its value, QUESO puts 15 processors into idle state until the routine is called again or the chain completes. Case (c) is useful, for instance, in the case of a computational procedure involving two models, where a group of processors can be split into two groups, each handling one model. Once the two model analysis end, the combined model can use the full set of processors. The parallel capabilities of QUESO have been exercised on the Ranger system of the TACC [18] with up to 1,024 processors [5].

Classes in QUESO can be divided in four main groups:

- core: environment (and options), vector, matrix;
- templated basic: vector sets (and subsets, vector spaces), scalar function, vector function, scalar sequence, vector sequence;

- templated statistical: vector realizer, vector RV, statistical IP (and options), MH solver (and options), statistical FP (and options), MC solver (and options), sequence statistical options; and
- miscellaneous: C and FORTRAN interfaces.

The templated basic classes are necessary for the definition and description of other entities, such as RVs, Bayesian solutions of IPs, sampling algorithms and chains. In the following we briefly explain 10 QUESO classes.

Environment Class: This class sets up the environment underlying the use of the QUESO library by an executable. The constructor of the environment class requires a communicator, the name of an options input file, and the eventual prefix of the environment in order for the proper options to be read (multiple environments can coexist, as explained further below). The environment class (a) assigns rank numbers, other than the world rank, to nodes participating in a parallel job, (b) provides MPI communicators for generating a sequence of vectors in a distributed way, (c) provides functionality to read options from the options input file (whose name is passed in the constructor of this environment class), (d) opens output files for messages that would otherwise be written to the screen (one output file per allowed rank is opened and allowed ranks can be specified through the options input file).

Let $S \geqslant 1$ be the number of problems a QUESO environment will be handling at the same time, in parallel. S has default value of 1 and is an option read by QUESO from the input file provided by the user. The QUESO environment class manages five types of communicators, referred to as *world* (MPI_WORLD_COMM); *full* (communicator passed to the environment constructor, of size F and usually equal to the world communicator); *sub* (communicator of size F/S that contains the number of MPI nodes necessary to solve a statistical IP or a statistical FP); *self* (MPI_SELF_COMM, of size 1); and *inter0* (communicator of size S formed by all MPI nodes that have subrank 0 in their respective subcommunicators).

A *subenvironment* in QUESO is the smallest collection of processors necessary for the proper run of the model code. An *environment* in QUESO is the collection of all subenvironments, if there is more than one subenvironment. So, for instance, if the model code requires 16 processors to run and the user decides to run 64 Markov chains in parallel, then the environment will consist of a total of $F = 1024$ processors and $S = 64$ subenvironments, each subenvironment with $F/S = 16$ processors. Any given computing node in a QUESO run has potentially five different ranks. Each subenvironment is assigned a subid varying from 0 (zero) to $S - 1$, and is able to handle a statistical IP and/or a statistical FP. That is, each subenvironment is able to handle a *sub* Markov chain (a sequence) of vectors and/or a *sub* MC sequence of output vectors. The *sub* sequences form an unified sequence in a distributed way. QUESO takes care of the unification of results for the application programming and for output files.

Vector Set Class: The vector set class is fundamental for the proper handling of many mathematical entities. Indeed, the definition of a scalar function like $\pi : \mathbf{B} \subset \mathbb{R}^n \to \mathbb{R}$ requires the specification of the domain $\mathbf{B}$, which is a *subset* of the *vector space* $\mathbb{R}^n$, which is itself a *set*. The relationship among the classes set, subset and vector space is sketched in Figure 3. An attribute of the *subset* class is the *vector space* which it belongs to, and in fact a reference to a vector space is required by the constructor of the subset class. The power of an object-oriented design is clearly featured here. The *intersection* subset derived class is useful for handling a posterior PDF (1), since its domain is the intersection of the domain of the prior PDF with the domain of the likelihood function.

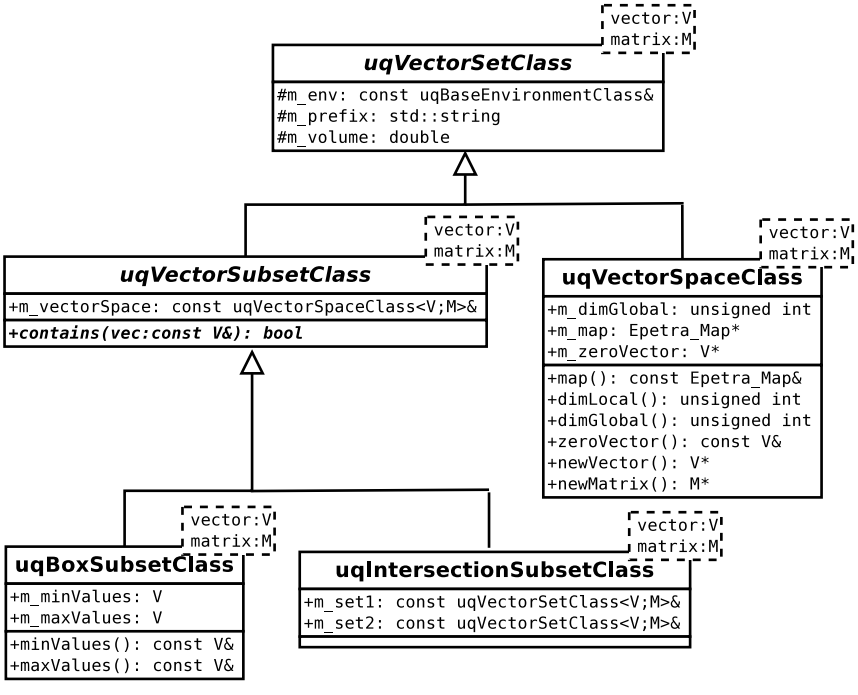

Fig. 3. The class diagram for vector set, vector subset and vector space classes

Scalar Function and Vector Function Classes: PDFs are examples of scalar functions. QUESO currently supports basic PDFs such as uniform and Gaussian. See Diagram 4. The definition of a vector function $\mathbf{q} : \mathbf{B} \subset \mathbb{R}^n \to \mathbb{R}^m$ requires only the extra specification of the image vector space $\mathbb{R}^m$.

Scalar Sequence and Vector Sequence Classes: The scalar sequence class contemplates *scalar* samples generated by an algorithm, as well as operations that can be done over them, e.g., calculation of means, variances, and convergence indices. Similarly, the vector sequence class contemplates *vector* samples and operations such as means, correlation matrices and covariance matrices.

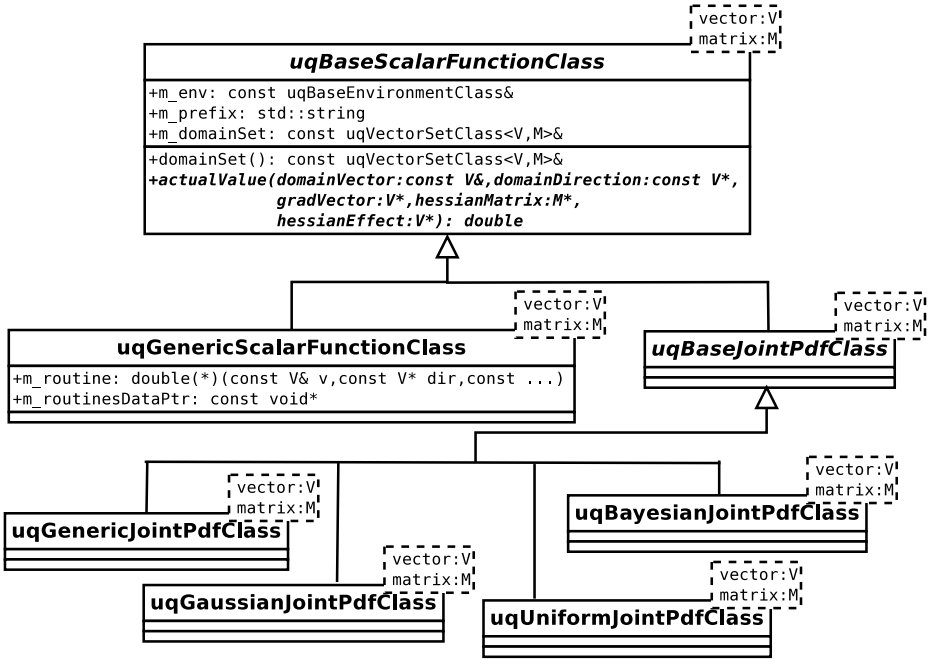

Fig. 4. The class diagram for the scalar function class

Vector Realizer Class: A *realizer* is an object that, simply put, contains a *realization()* operation that returns a sample of a vector RV. QUESO currently supports basic realizers such as uniform and Gaussian. It also contains a *sequence realizer* class for storing samples of a MH algorithm, for instance.

Vector Random Variable Class: Vector RVs are expected to have two basic functionalities: compute the value of its PDF at a point, and generate realizations following such PDF. The joint PDF and vector realizer classes allow a straightforward definition and manipulation of vector RVs. QUESO currently supports basic vector RVs such as uniform and Gaussian. A derived class called *generic vector RV* allows QUESO to store the solution of an statistical IP: a *Bayesian joint PDF* becomes the PDF of the posterior RV, while a *sequence vector realizer* becomes the realizer of the same posterior RV. QUESO also allows users to form new RVs through the concatenation of existing RVs.

Statistical Inverse and Forward Problem Classes: For QUESO, a statistical IP has two input entities, a prior RV and a likelihood routine, and one output entity, the posterior RV. Similarly, a statistical FP has two input entities, an input RV and a QoI routine, and one output entity, the output RV. QUESO differentiates the entities that allow us to define a problem from the entities that allow us to solve it. Indeed, QUESO defines the MH and MC sequence generator classes. The former expects the specification of a target distribution, a

proposal covariance matrix, and the initial position in the chain, while the latter expects the specification of an input RV, a QoI function, and an output RV. The proper definition, by QUESO, of more basic entities allows an easy specification of more complex entities such as statistical problems and solvers.

Software Engineering: We utilize various community tools to manage the QUESO development cycle. Source code traceability is provided via subversion and the GNU autotools suite is used to provide a portable, flexible build system, with the standard `configure; make; make check; make install` steps. We employ an active regression testing and utilize the BuildBot system in order to have a continuous integration analysis of source code commits. We also utilize the Redmine project management system, which provides a web-based mechanism to manage milestone developments, issues, bugs, and source code changes.

4 Conclusions and Future Directions

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 *quickly* prototype new algorithms in a sophisticated computational environment, rather than first coding and testing them with a scripting language and only then recoding in a C++/MPI environment. It also allows them 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.

Planned features for QUESO include (a) convergence diagnostics and statistical accuracy assessments on the fly, for the optimization of computational effort, (b) more basic distributions (e.g. log-normal), (c) Gaussian random fields, (d) stochastic collocation algorithms [1], (e) parallel vectors for parameters and/or QoIs, (f) graphical user interface, (g) real time interaction capabilities, and (h) robustness (resiliency) w.r.t. node crashing, via the interaction with fault tolerant versions of MPI. *Fault tolerance* is critical for UQ methods due to their greater computational requirements compared to single deterministic simulations. Sampling algorithms, the current focus of QUESO, *need themselves to be fault tolerant*, since they are the drivers, not any of the model simulations. Also, they have the nice property of allowing statistical explorations to continue even if a group of nodes fails. The more nodes are used, the smaller the potential impact of a node failure in the overall sampling mechanism. Also, because checkpoints deal with parameter vectors and QoIs, as opposed to full state vectors, it becomes easier to handle a potential statistical bias due to a node failure.

Acknowledgments. This work has been supported by the National Nuclear Security Administration, U.S D.O.E., under Award Number DE-FC52-08NA28615. The first author was also partially supported by Sandia National Laboratories, under Contracts 1017123 and 1086312, and by King Abdullah University

of Science and Technology (KAUST), under the Academic Excellence Alliance program. The authors would also like to thank fruitful discussions with many researchers, including P. Bauman, S. H. Cheung, M. Eldred, O. Ghattas, J. Martin, K. Miki, F. Nobile, T. Oliver, C. Simmons, L. Swiler, R. Tempone, G. Terejanu and L. Wilcox.

References

1. Babuška, I., Nobile, F., Tempone, R.: A stochastic collocation method for elliptic partial differential equations with random input data. SIAM J. Num. Anal. (2007)
2. Beck, J.L., Katafygiotis, L.S.: Updating of a model and its uncertainties utilizing dynamic test data. In: Proc. 1st International Conference on Computational Stochastic Mechanics, pp. 125–136 (1991)
3. Beck, J.L., Yuen, K.V.: Model selection using response measurements: A Bayesian probabilistic approach. ASCE Journal of Eng. Mechanics 130, 192–203 (2004)
4. Cheung, S.H., Beck, J.L.: New Bayesian updating methodology for model validation and robust predictions of a target system based on hierarchical subsystem tests. CMAME (2010) (accepted for publication)
5. Cheung, S.H., Oliver, T.A., Prudencio, E.E., Prudhomme, S., Moser, R.D.: Bayesian uncertainty analysis with applications to turbulence modeling. Reliability Engineering & System Safety (2011) (in press)
6. Eldred, M.S., et al.: DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis (1994-2009), `http://www.cs.sandia.gov/DAKOTA/`
7. Galassi, M., et al.: GNU Scientific Library (1996-2009), `http://www.gnu.org/software/gsl/`
8. Haario, H., Laine, M., Mira, A., Saksman, E.: DRAM: Efficient adaptive MCMC. Stat. Comput. 16, 339–354 (2006)
9. Hastings, W.K.: Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57(1), 97–109 (1970)
10. Heroux, M.: Trilinos (2009), `http://www.trilinos.gov/`
11. Hoeting, J.A., Madigan, D., Raftery, A.E., Volinsky, C.T.: Bayesian model averaging: a tutorial (with discussion). Statistical Science 14, 382–417 (1999)
12. Kaipio, J., Somersalo, E.: Statistical and Computational Inverse Problems, Applied Mathematical Sciences, vol. 160. Springer (2005)
13. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equations of state calculations by fast computing machines. Journal of Chemical Physics 21(6), 1087–1092 (1953)
14. Prudencio, E.E., Cai, X.C.: Parallel multilevel restricted Schwarz preconditioners with pollution removing for PDE-constrained optimization. SIAM J. Sci. Comp. 29, 964–985 (2007)
15. Prudencio, E.E., Cheung, S.H.: Parallel adaptive multilevel sampling algorithms for the Bayesian analysis of mathematical models (2011) (submitted)
16. Robert, C.: The Bayesian Choice, 2nd edn. Springer (2004)
17. Smith, B.: PETSc (2009), `http://www.mcs.anl.gov/petsc/`
18. TACC: Texas advanced computing center (2008), `http://www.tacc.utexas.edu/`