MUQ (MIT Uncertainty Quantification):
Flexible Software for Connecting Algorithms and Applications

Matthew Parno, Andrew Davis, Patrick Conrad, and Youssef Marzouk

Center for Computational Engineering
Massachusetts Institute of Technology

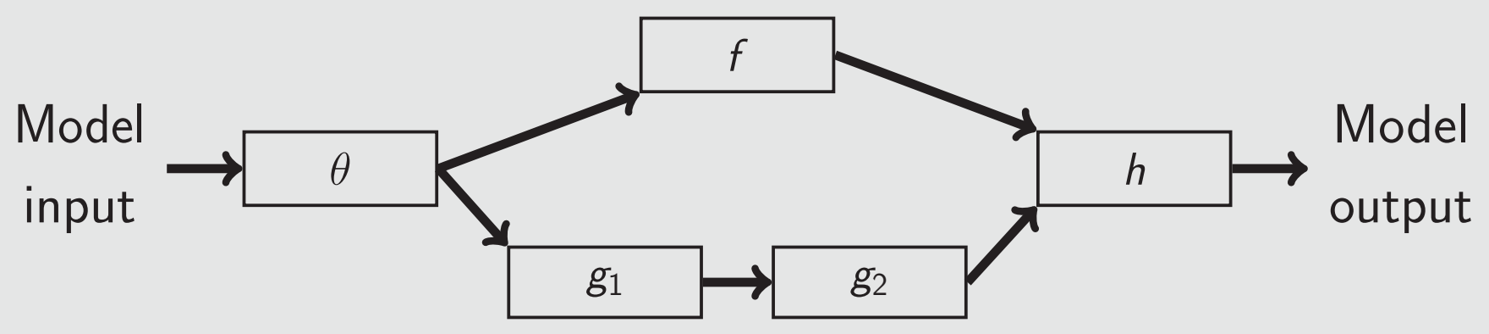
Motivation

MUQ (pronounced "muck") is a collection of C++ and Python libraries for accelerating both the application of existing uncertainty quantification (UQ) algorithms and the development of new approaches. To facilitate these tasks, MUQ provides interfaces for defining and coupling models with UQ-related algorithms. Throughout our code, we use well-founded software engineering techniques and stable external libraries. Our goal is to provide an efficient and stable platform to help users "MUQ" about—and much more—in UQ.

CAPABILITIES

Modelling Module

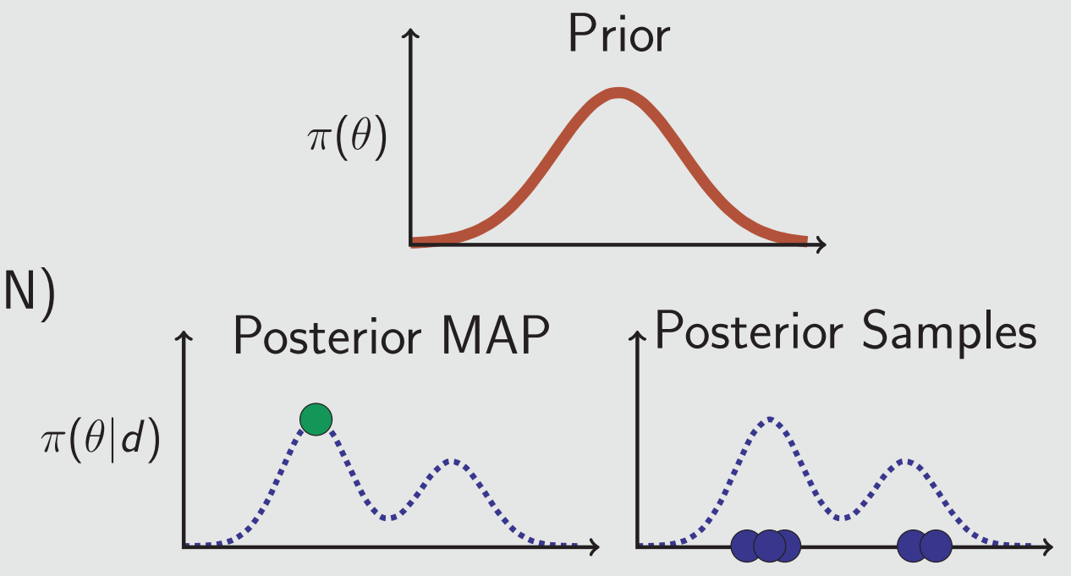
- Graph-based model construction
- Random variables
- Probability densities
- Differentiation



Inference Module

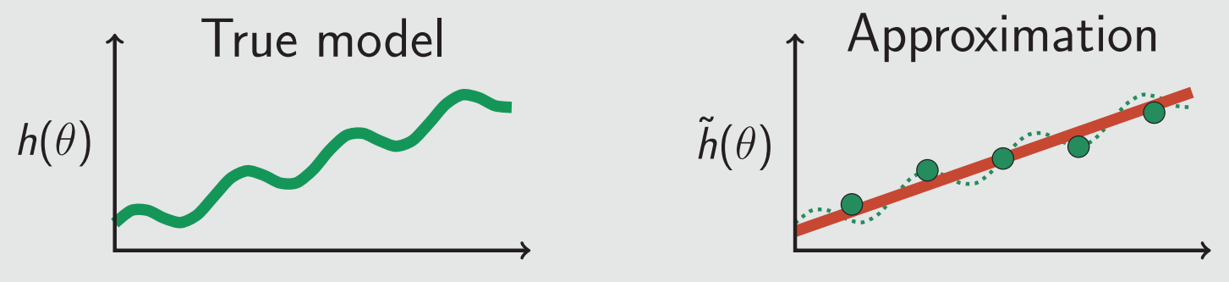
- Markov chain Monte Carlo
- Point estimates (MAP)
- Transport maps
- Importance sampling

- DRAM
- AMALA
- NUTS (via STAN)
- sMMALA
- and more...



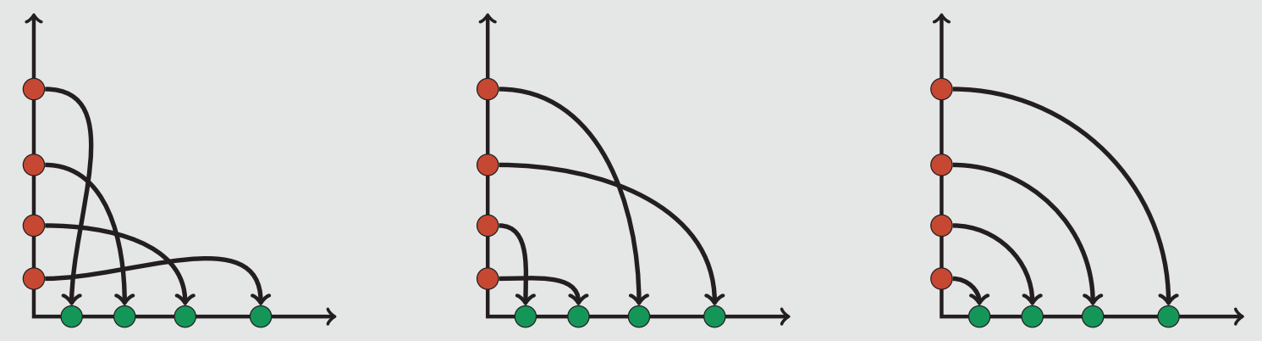
Approximation Module

- Polynomial chaos expansions
- Regression
- Incremental approximation



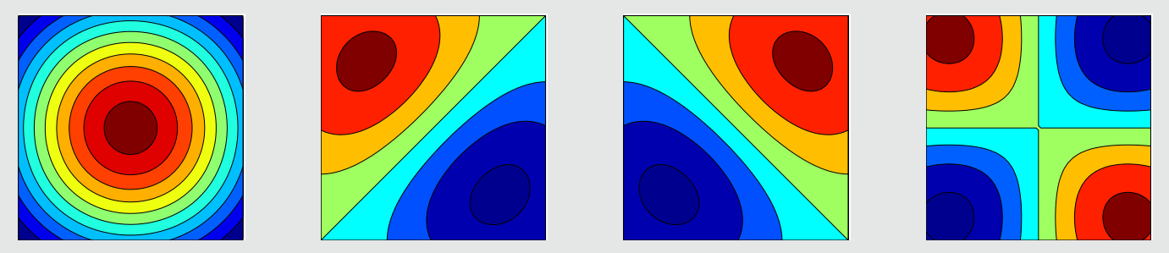
Optimization Module

- Constrained nonlinear programs
- Assignment problems
- Links to external optimizers



Geostatistics Module

- Covariance kernels
- Karhunen-Loève (KL) decompositions



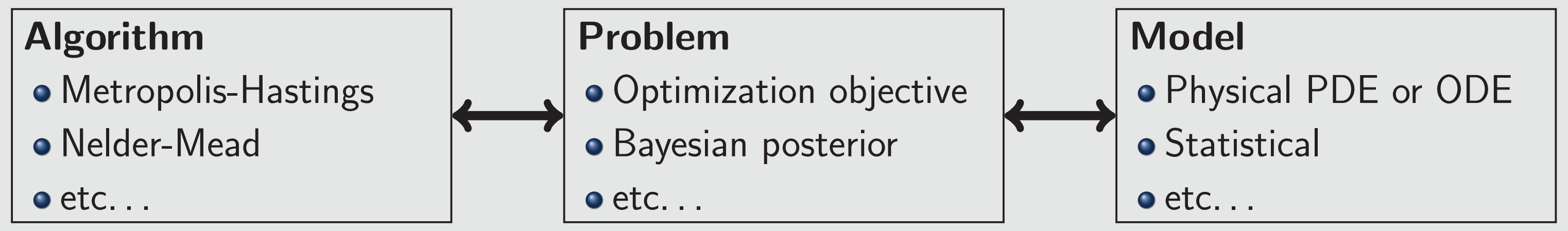
Utilities Module and Development Tools

- Tools for HDF5 i/o
- Vector type translation
- Random number generation
- Multi-indices
- Linear solver and eigensolvers

Trilinos boost sundials CMake Jenkins

Separating models from algorithms

It is challenging to provide a flexible interface between scientific models and a wide variety of algorithms. This is because different algorithms exploit different aspects of model structure, and models can provide varying levels of information (e.g., gradients, Hessians, block structure). In MUQ, we have generally adopted a three-component system to define the model-algorithm interface.



Algorithm
- Metropolis-Hastings
- Nelder-Mead
- etc...

Problem
- Optimization objective
- Bayesian posterior
- etc...

Model
- Physical PDE or ODE
- Statistical
- etc...

Benefits of this approach:
- Models are independent of algorithms, which allows greater flexibility on both sides.
- Problems can extract algorithm-specific structure from models and provide meaningful defaults.
- Model/problem approximations can be employed without changing algorithms.

Inheritance and extendability

A good software library will allow a user to complete complicated tasks with a minimal amount of code. In MUQ, we try to achieve this using abstract classes. Users implement a few member functions that define core functionality, and code from the parent class provides additional functionality. This type of object orientated programming makes it easy to extend MUQ with new models and new algorithms.

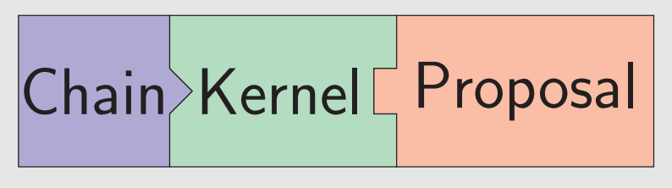
Example implementation: MCMC

There are 3 key components to most MCMC algorithms:

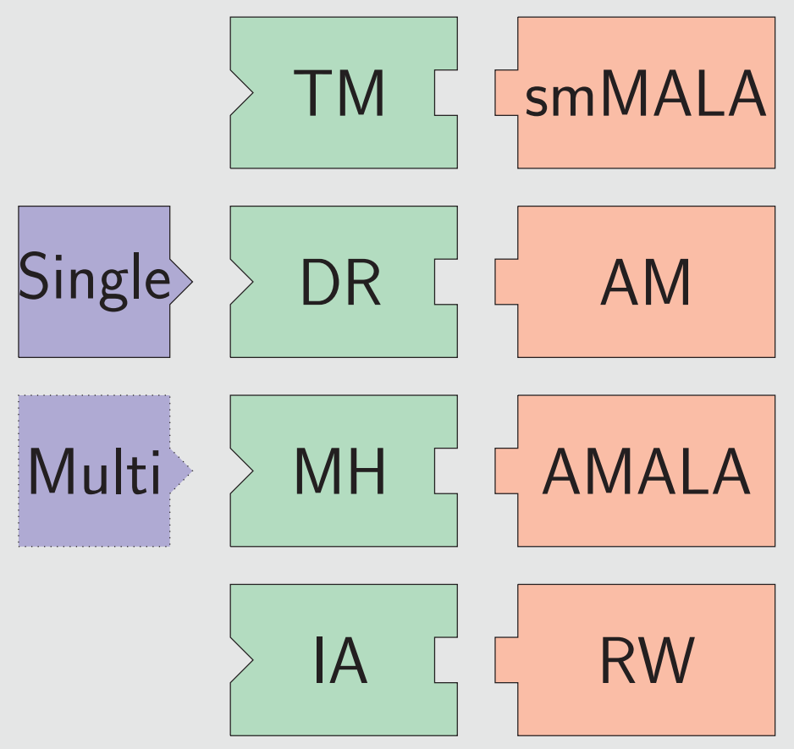
1. The chain.
2. The kernel.
3. The proposal.

MUQ defines each component through an abstract base class.

Typical MCMC algorithms:



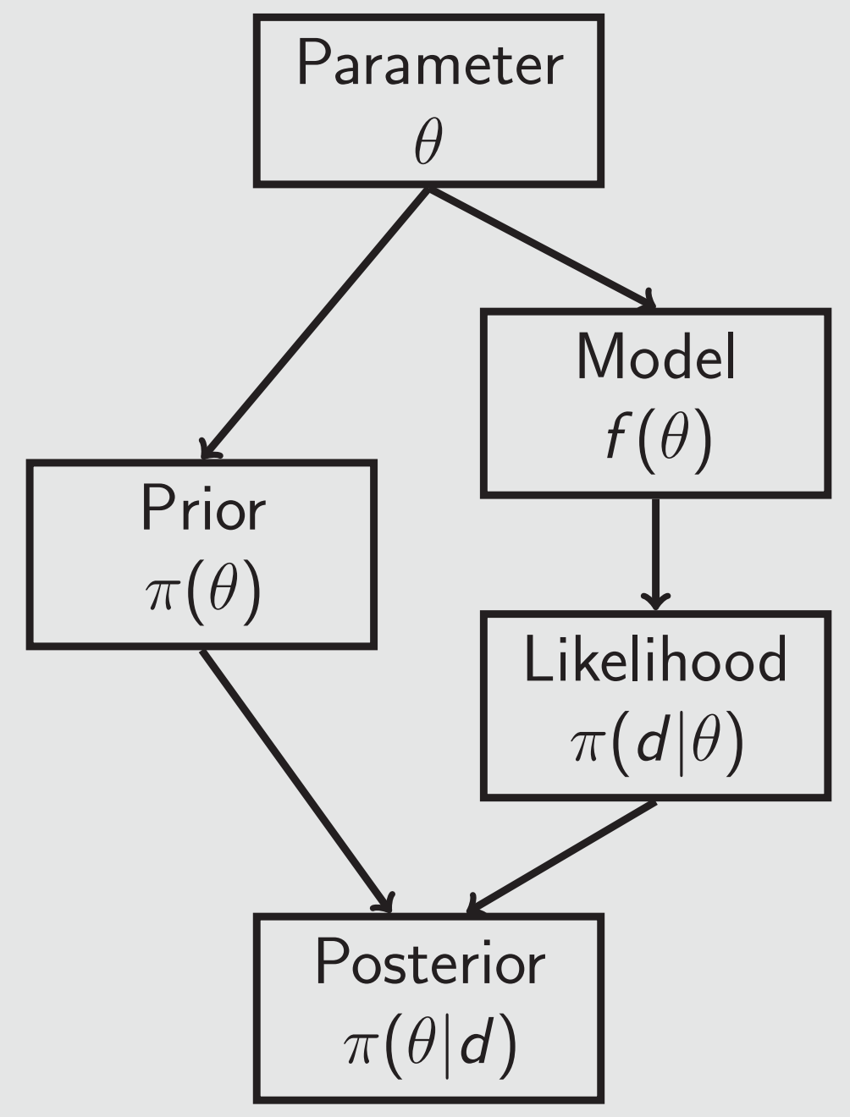
Other configurations are also possible. Hybrid methods might use more than one kernel and/or several proposals



CODE EXAMPLES

Combining model components

Assume we want to create a Bayesian posterior density, $\pi(\theta|d) \propto \pi(d|\theta)\pi(\theta)$, where $\pi(\theta) = N(\mu, \Gamma)$, $d = f(\theta) + \epsilon$, and $\epsilon \sim N(0, 1)$. The graphical model for the posterior and the corresponding code are given below.



Python Code:

```
# Create the model components
para   = VectorPassthroughModel()
prior  = GaussianDensity(mu,gamma)
mod    = ForwardModel()
likely = GaussianDensity()

# Combine the components
graph = ModGraph()
graph.AddNode(para,"Parameter")
graph.AddNode(prior,"Prior")
graph.AddNode(likely,"Likelihood")
graph.AddNode(mod,"Model")
graph.AddNode(DensityProduct(),"Posterior")

graph.AddEdge("Parameter","Prior",0)
graph.AddEdge("Parameter","Model",0)
graph.AddEdge("Model","Likelihood",0)
graph.AddEdge("Prior","Posterior",0)
graph.AddEdge("Likelihood","Posterior",1)
```

Defining a new MCMC proposal

A fundamental quantity in Markov chain Monte Carlo (MCMC) algorithms is the proposal density. In code, the proposal does two things: (1) generates a random sample of the proposal, and (2) evaluates the proposal density. The code below shows an implementation of a simple Gaussian random walk proposal.

C++ Code:

```
class MyPropoposal : public MCMCProposal {
public:
  /** Construct the proposal using information in props  */
  MyProposal(shared_ptr<AbstractSamplingProblem> prob, ptree& props) :
  MCMCProposal(prob,props){
    // extract the proposal variance from the parameters
    propVar = props.get("MCMC.MyProposal.Var",1.0);
    // tell the problem what information this proposal needs
    prob->SetStateComputations(false, false, false);
  };
  virtual ~MyProposal() = default;

  /** This function generates a sample of the proposal */
  virtual shared_ptr<MCMCState> DrawProposal(shared_ptr<MCMCState> currentState,
                                             shared_ptr<HDF5LogEntry> logEntry){
    Eigen::VectorXd propState = currentState->state;
    propState += sqrt(propVar)*RandomGenerator::GetNormalRandomVector(dim);
    return prob->ConstructState(propState,0,logEntry);
  };

  /** This function evaluates the log proposal density for a pair of points. */
  virtual double ProposalDensity(shared_ptr<MCMCState> currentState,
                                 shared_ptr<MCMCState> proposedState){
    auto diff = proposedState->state-currentState->state;
    return -0.5*diff.squaredNorm()/propVar;
  };
private:and/or
  double propVar;
};
```

Property tree parameters

Nearly all algorithms have tunable parameters. In MUQ, these are defined through either a boost property tree (in C++) or a dictionary (in Python). In either case, parameter names are matched with parameter values. These pairs can also be easily store in structured xml files.

C++ code:

```
boost::property_tree::ptree params;
params.put("MCMC.Kernel","DR");
params.put("MCMC.Verbosity",3);
params.put("Dummy",100.0);
```

Python code:

```
params = dict()
params['MCMC.Kernel'] = DR
params['MCMC.Verbosity']=3
params['Dummy']=100.0
```

XML file:

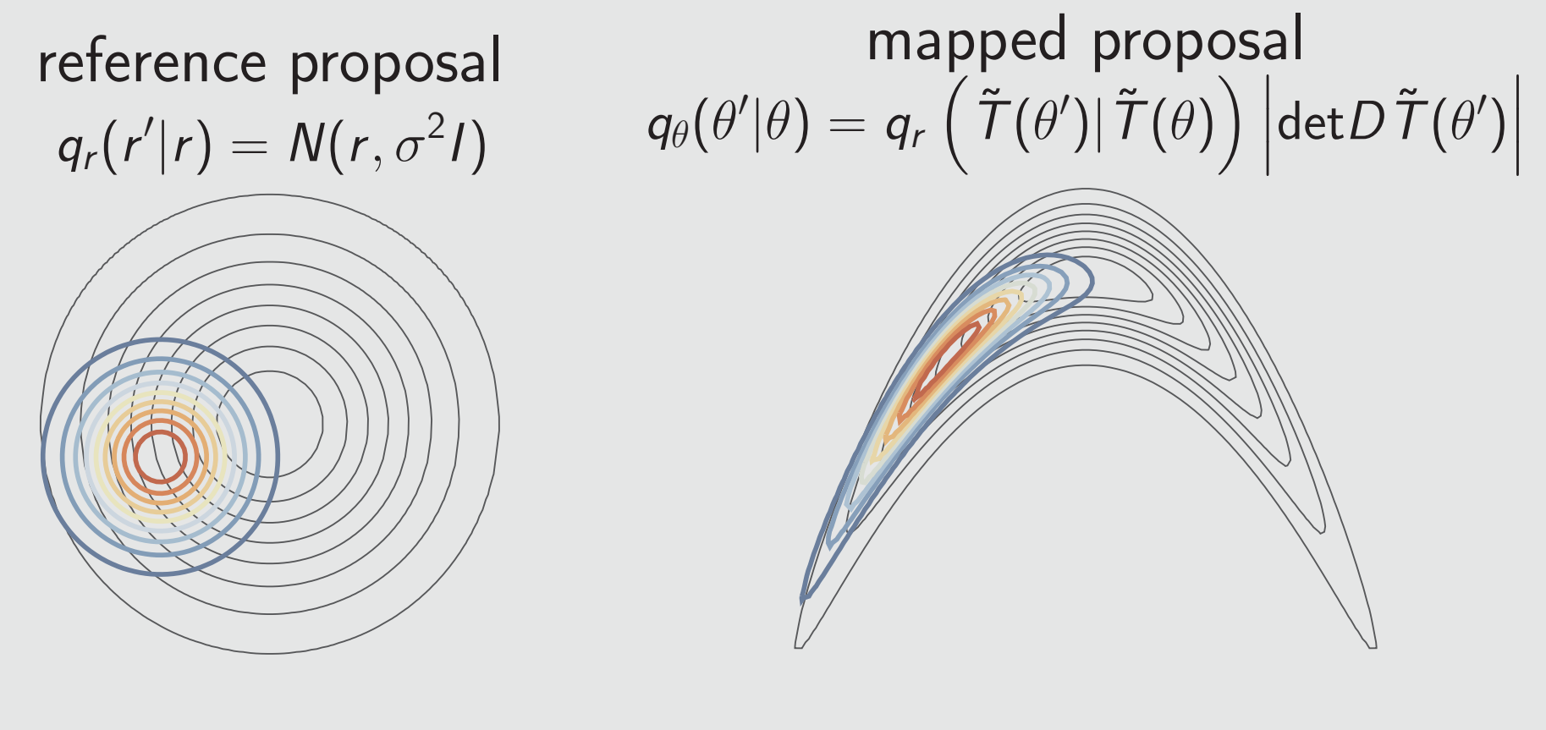
```
<MCMC>
  <Kernel>DR</Kernel>
  <Verbosity>3</Verbosity>
</MCMC>
<Dummy>100.0</Dummy>
```

USE IN RESEARCH

Map-accelerated MCMC

Idea:
- Use nonlinear variable transformations to "precondition" target density.
- Apply any MCMC kernel to transformed problem.

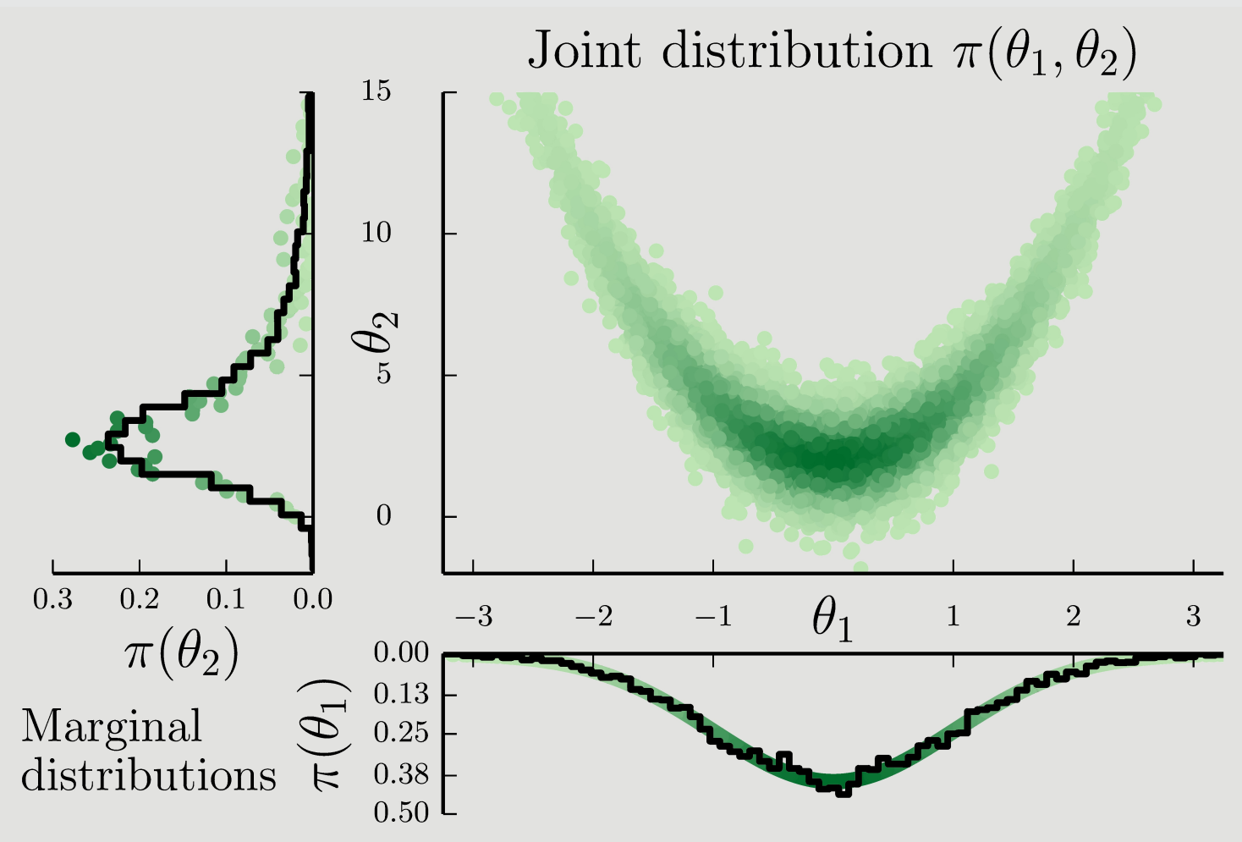
The exchangeability of MCMC kernels and proposals makes testing many methods easy.



Pseudo marginal MCMC with local approximations

Idea:
- Focus MCMC on parameters of interest
- Marginalize unneeded parameters with importance sampling
- Approximating noisy likelihood dramatically reduces model evaluations.

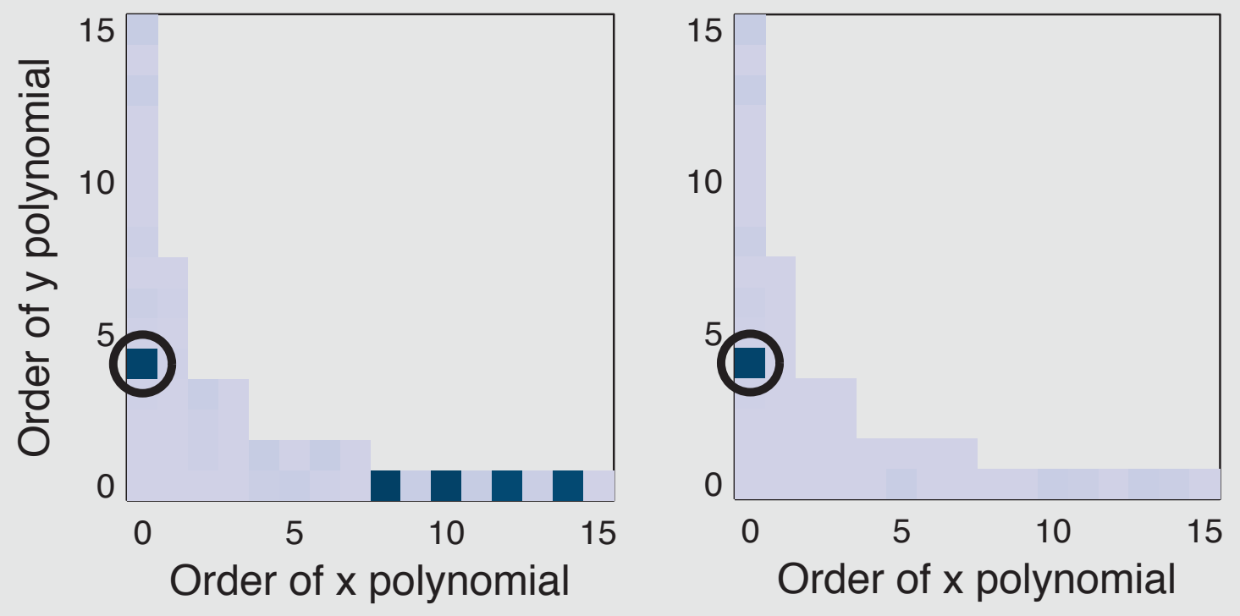
Performing marginalization in a problem class allows the algorithm and model to be independent of pseudo-marginal application.



Adaptive non-intrusive polynomial chaos

Idea:
- Approximate forward model with polynomial expansion.
- Adaptively choose expansion terms and quadrature points to minimize number of necessary model evaluations.

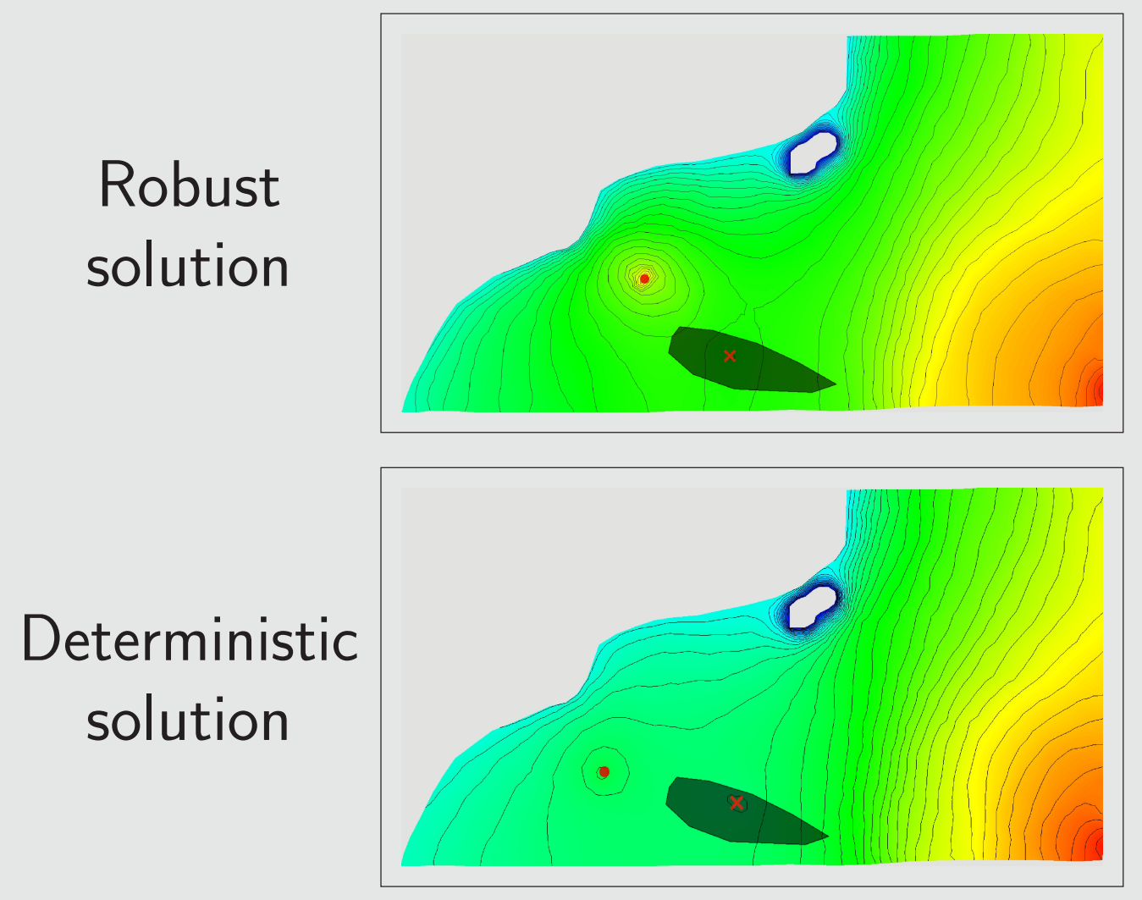
The polynomial approximation has the same interface as the original model, making it easy to swap with the original model in any graph.



Robust optimization of pumping rates

Idea:
- Use posterior samples to evaluate expected head.
- Control pumping rates to restrict contaminant movement.
- Optimization performed with sample average approximation (SAA) or stochastic approximation (SA).

The same model implementation can be used for both optimization and inference (just in different graphs), reducing repeated code.



Where can I get MUQ?

Want more? Download MUQ and find links to our documentation at

bitbucket.org/mituq/muq

Supported by the US Department of Energy, Office of Advanced Scientific Computing Research (ASCR), part of the SciDAC Institute for the Quantification of Uncertainty in Extreme Scale CompuTations (QUEST).

Email contacts: ymarz@mit.edu, mparno@mit.edu