INL/EXT-21-63120

June 2021

Enhanced mechanical property evaluation using innovative data analytics capability

Zachary Prince
Lynn Munday
Dewen Yushu

INL/EXT-21-63120

Enhanced mechanical property evaluation using innovative data analytics capability

Zachary Prince
Lynn Munday
Dewen Yushu

June 2021

Idaho National Laboratory
Idaho Falls, Idaho 83415

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under U.S. Department of Energy-Idaho Operations Office
Contract DE-AC07-99ID13727

ABSTRACT

This report focuses on efforts to improve Multiphysics Object-Oriented Simulation Environment (MOOSE) for mechanical property evaluation using data analytics. These efforts involve improvements to the stochastic tools module (STM) for stochastic simulations of MOOSE multiphysics model and the development of inverse optimization capabilities. The report gives an overview of the STM and describes recent updates to its core capabilities and theory on its reduced-order model (ROM) schemes. Examples are also provided showing the impact of these updates and exhibits the usefulness of ROMs. An overview of the gradient based inverse optimziation algorithms are given along with examples of their application to source identification. Inverse optimization will provide a new methodology in STM for fitting model parameters to experimental data.

CONTENTS

FIGURES

1 Introduction

This report summarizes recent enhancements to Multiphysics Object-Oriented Simulation Environment (MOOSE) for rapid mechanical property evaluation using data analytics. These enhancements were developed on two fronts: through stochastic analysis/reduced-order model (ROM) capabilities and through inverse optimization. The first involves improvements and adding capabilities to the MOOSE stochastic tools module (STM); extensive documentation involving the STM can be found on the MOOSE website [1], some of which is presented in this report. The second involves the creation of a gradient based partial differential equation (PDE) constrained inverse optimization application named Isopod, which is a MOOSE-based application meant to be used in tandem with other MOOSE applications to perform force and material property inversion. Isopod will be merged into the STM at the end of fiscal year 2021 or early fiscal year 2022.

The STM is an open-source module within the MOOSE framework that any MOOSE-based application can link to in order to perform stochastic analysis and reduced-order modeling. Applications currently using the STM include BISON, Grizzly, and Griffin. The primary goals of STM are to:

- provide a MOOSE-like interface for performing stochastic analysis on MOOSE-based models,
- sample parameters, run applications, and gather data that is both efficient (memory and run-time) and scalable,
- perform uncertainty quantification (UQ) and sensitivity analysis (SA) with distributed data,
- train ROMs to develop fast-evaluating surrogates of high-fidelity multiphysics models, and
- provide a pluggable interface for the surrogates.

The updates included in Section 2 of this report will specifically address each of these goals.

This report also covers the implementation of inverse optimization algorithms in Isopod. Inverse optimization is an established mathematical framework to infer model parameters by minimizing the misfit between the experimental and simulation observables. By using inverse optimization, a user will be able to rapidly parameterize models to experimental data. The current optimization algorithms in Isopod are limited to material and force inversion problems constrained by the PDE for steady state heat conduction. An overview of the inverse optimization algorithms implemented in Isopod are given in Section 3 along with examples for parameter estimation.

2 Stochastic Tools Module and Surrogate Modeling

This section gives an overview of the STM capabilities and details updates/improvements over the past year. The primary focus of the STM is to efficiently perform stochastic simulations of multiphysics models developed in MOOSE-based applications. Stochastic simulations come in various forms with many applications, including UQ, SA, and optimization, but the idea is to run multiple variations of the same model while changing input parameters and gathering relevant results, or quantities of interest (QoIs). The number of simulations, or variations on the model, can vary wildly depending on the application, from dozens to billions. The STM provides a MOOSE-like interface for performing these stochastic simulations, allowing a short learning process for those who have used MOOSE applications before. Since the STM is within the MOOSE framework, it has a unique position over "black-box" software like DAKOTA [2] by being able to hold relevant data in memory, versus performing input/ouput file reading. This position allows the STM to be extremely efficient in memory usage and scalable by keeping data distributed and reusing model data where appropriate.

However, no matter how efficient the stochastic simulations are performed, the complexity of the multiphysics model is the most significant factor for run-time. Complex, high-fidelity models could require a large amount of computing resources to run a single simulation, performing stochastic simulations on such a model could prove intractable. Therefore, the STM includes the ability to train and utilize ROMs. ROMs serve as a surrogate to the high-fidelity full-order model (FOM), which is meant to be used in place of the FOM to perform UQ, SA, or optimization. The process involves a training phase which performs a relatively small number of simulations of the FOM to build a ROM surrogate that is much faster to evaluate and returns comparable results as the FOM. In the module, ROMs are known as surrogate models, these two terms are interchangeable in the context of this report.

Sections 2.1 and 2.2 describe updates to the core capabilities of the module including its sampling techniques, data handling system, and extendable model training. Section 2.3 introduces all the ROM methods implemented in STM. Section 2.4 provides several example problems showing the recently developed capabilities, focusing on the developed ROMs.

2.1 Batch Mode for Large Models

In order to run stochastic simulations of MOOSE models, the STM provides a *Sampler* system which is used to produce a matrix of perturbed parameters. These *Samplers* can either randomly generate perturbations (Monte Carlo or Latin Hypercube) or use pre-determined quantities (Cartesian grid or sparse quadrature) to evolve the parameters. STM leverages the *MultiApp* and *Transfer* system in MOOSE to run the MOOSE models with the perturbed parameters generated from *Samplers*. This system is described in detail in [3]. The idea is that a main STM app instantiates sub-apps of the same MOOSE model, transferring parameters from the *Sampler* to the parameters of the model, running the sub-apps, then concatenating the QoIs from the model.

Because of the high-dimensional nature of model parameters, there is a potential of multitudinous perturbations of model parameters needed; some problems use on the order of a billion samples. As such, the *Sampler* and stochastic *MultiApp* systems need to be efficient and scalable in run-time and memory. STM provides three different modes of operation: normal, batch-reset, and batch-restore. Normal mode works by instantiating an app for each sample, or row of the *Sampler* matrix. This is typical mode of operation for other *MultiApp* applications; however, instantiating a full app for each sample can become intractable in memory for large stochastic runs. Therefore, STM created the batch modes of operation. The idea is the main app instantiates a set number of apps, based on the number of processors used to run the study, and reuses that app for multiple sets of samples.

Previously, running a stochastic study with STM in batch mode would create an app for each processor used in the execution. This works well for small models since parallelizing samples is generally more scalable than parallelizing the model and having a number of apps equal to the number of processors is tractable in memory. However, for large models, it may be more efficient to run a single sample with multiple processors and have only one app for that group of processors. Therefore, two parameters were introduced to the *SamplerFullSolveMultiApp* and *Sampler* classes in the STM: `MultiApps/*/min_procs_per_app` and `Samplers/*/min_procs_per_row`. As the names suggest, these parameters will partition the samples and the sub-app runs such that that many processors will be used for parameter perturbation. It is required that these two parameters be set to the same value to ensure that the *Sampler* partitioning is equivalent to the *MultiApp* partitioning. As an example, say 1,000 samples are being run with 100 processors in batch mode, without setting these parameters 100 apps would be created running with 1 processor for 10 samples. With `min_procs_per_app/min_procs_per_row=10`, 10 apps would be created running with 10 processors for 100 samples. Section 2.4.1 shows a more detailed example with these parameters and how they affect memory usage and run-time.

2.2 Generalizing Surrogate Training

The STM surrogate system provides a means of training and evaluating ROMs. The system involves a two step procedure with different type of object for each step: *Trainers* for using predictor and QoI data to generate model data (training) and *Surrogates* for using model data to build a functional representation between the parameters and QoIs (evaluation). The training step usually involves sampling a MOOSE model using a stochastic *MuliApp* to generate the QoI, this data, along with the predictor data from the *Sampler*, is then used to generate the model data. The model data can then either be directly used by a *Surrogate* object or be outputted to a MOOSE readable file be used later by the *Surrogate* object. The second step is simply to use the *Surrogate* object as surrogate of the original high-fidelity MOOSE model. A common practice is to train a surrogate with a limited amount of samples, then use the surrogate to sample with many more perturbations for statistical analysis. However, *Surrogates* were designed to pluggable with any other object tied to the MOOSE framework.

Originally, *Trainers* were hard coded to use *Samplers* as predictor data and *VectorPostprocessors* as QoI data. However, this implementation was found to be too restrictive as it limits the system to the type of data being retrieved and how the data was produced. To address the first concern, MOOSE developed the *Reporter* system, which allows the output of any type of data from simulations. This is a much more general system than the postprocessor/vector-postprocessor system which only supported output of real numbers and vectors of real numbers. STM now utilizes *Reporters* to gather the model output QoI data into vector-type values that are distributed similarly to the *Sampler*. The surrogate *Trainers* were then refactored to be able to retrieve this distributed data. In addition to using the *Reporter* system, *Trainers* were also refactored so that either sampler values or model output values could be used as predictor data. This allows the creation of surrogates to be much more general; for instance, surrogates can now be trained to correlate QoIs, not just using the data used to produced the QoIs. New *Trainers* are now much easier to develop, even those that require complicated predictor and/or QoI values.

2.3 Surrogate Models

Along with core architecture of the surrogate system, various types of ROMs were added to STM. The following sub-sections give a general overview of these methods, this information can be found on the MOOSE-STM website:

- Section 2.3.1 Polynomial Chaos Expansion – `https://mooseframework.inl.gov/source/surrogates/PolynomialChaos.html`
- Section 2.3.2 Polynomial Regression – `https://mooseframework.inl.gov/source/surrogates/PolynomialRegressionSurrogate.html`, `https://mooseframework.inl.gov/source/surrogates/PolynomialRegressionTrainer.html`
- Section 2.3.3 Gaussian Processing – `https://mooseframework.inl.gov/source/surrogates/GaussianProcessTrainer.html`
- Section 2.3.4 Proper Orthogonal Decomposition – `https://mooseframework.inl.gov/source/surrogates/PODReducedBasisTrainer.html`, `https://mooseframework.inl.gov/source/surrogates/PODReducedBasisSurrogate.html`

2.3.1 Polynomial Chaos Expansion

Polynomial chaos (PC) is a surrogate modeling technique where a QoI that is dependent on input parameters is expanded as a sum of orthogonal polynomials [4]. Given a QoI Q dependent on a set of parameters $\vec{\xi}$, the PC expansion is:

$$Q(\vec{\xi}) = \sum_{i=1}^{P} q_i \Phi_i(\vec{\xi}), \tag{1}$$

where P is the multidimensional polynomial order and q_i are coefficients that are to be computed. These coefficients can be found using intrusive and non intrusive techniques. The intrusive technique is quite difficult to generalize and very computationally demanding. Since the polynomial basis is orthogonal, a non intrusive technique is developed where the coefficients are found by performing a Galerkin projection and integrating:

$$q_i = \frac{\left\langle Q(\vec{\xi})\Phi_i(\vec{\xi})\right\rangle}{\left\langle \Phi_i(\vec{\xi}), \Phi_i(\vec{\xi})\right\rangle}, \tag{2}$$

where,

$$\left\langle a(\vec{\xi})b(\vec{\xi})\right\rangle = \int_{-\infty}^{\infty} a(\vec{\xi})b(\vec{\xi})f(\vec{\xi})d\vec{\xi}. \tag{3}$$

The weight function ($f(\vec{\xi})$) and bases ($\Phi_i(\vec{\xi})$) are typically products of one-dimensional functions:

$$f(\vec{\xi}) = \prod_{d=0}^{D} f_d(\xi_d), \tag{4}$$

$$Q(\vec{\xi}) = \sum_{i=1}^{P} q_i \prod_{d=0}^{D} \phi^d_{k_{d,i}}(\xi_d). \tag{5}$$

Table 1: Common probability density functions and their corresponding orthogonal polynomials

Distribution	Density Function ($f_d(\xi)$)	Polynomial ($\phi_i(\xi)$)	Support
Normal	$\frac{1}{2\pi}e^{-\xi^2/2}$	Hermite	$[-\infty, \infty]$
Uniform	$\frac{1}{2}$	Legendre	$[-1, 1]$
Beta	$\frac{(1-\xi)^\alpha(1+\xi)^\beta}{2^{\alpha+\beta+1}B(\alpha+1,\beta+1)}$	Jacobi	$[-1, 1]$
Exponential	$e^{-\xi}$	Laguerre	$[0, \infty]$
Gamma	$\frac{\xi^\alpha e^{-\xi}}{\Gamma(\alpha+1)}$	Generalized Laguerre	$[0, \infty]$

The weighting functions are defined by the probability density function of the parameter and the polynomials are based on these distributions, Table 1 is a list of commonly used distributions and their corresponding orthogonal polynomials.

The expression in Eq. (2) can be integrated using many different techniques. One is performing a Monte Carlo integration,

$$q_i = \frac{1}{\left\langle \Phi_i(\vec{\xi}), \Phi_i(\vec{\xi}) \right\rangle} \frac{1}{N_{\text{mc}}} \sum_{n=1}^{N_{\text{mc}}} Q(\vec{\xi_n}) \Phi_i(\vec{\xi_n}), \tag{6}$$

or using numerical quadrature,

$$q_i = \frac{1}{\left\langle \Phi_i(\vec{\xi}), \Phi_i(\vec{\xi}) \right\rangle} \sum_{n=1}^{N_q} w_n Q(\vec{\xi_n}) \Phi_i(\vec{\xi_n}). \tag{7}$$

The numerical quadrature method is typically much more efficient that than the Monte Carlo method and has the added benefit of exactly integrating the polynomial basis. However, the quadrature suffers from the curse of dimensionality. The naive approach uses a Cartesian product of one-dimensional quadratures, which results in $(\max(k_i^d) + 1)^D$ quadrature points to be sampled. Sparse grids can help mitigate the curse of dimensionality significantly [5].

In PC, a tuple describes the combination of polynomial orders representing the expansion basis ($k_{d,i}$). Again, the naive approach would be to do a tensor product of highest polynomial order, but this is often wasteful since generating a complete monomial basis is usually optimal. Below demonstrates the difference between a tensor basis and a complete monomial basis:

$$\begin{aligned} &D = 2,\ k_{\max} = 2, \text{ Tensor product:} \\ &\Phi_0 = \phi_0^1\phi_0^2,\ \Phi_1 = \phi_1^1\phi_0^2,\ \Phi_2 = \phi_0^1\phi_1^2,\ \Phi_3 = \phi_1^1\phi_1^2 \\ &\Phi_4 = \phi_2^1\phi_0^2,\ \Phi_5 = \phi_0^1\phi_2^2,\ \Phi_6 = \phi_1^1\phi_2^2,\ \Phi_7 = \phi_2^1\phi_1^2,\ \Phi_8 = \phi_2^1\phi_2^2, \end{aligned} \tag{8}$$

$$\begin{aligned} &D = 2,\ k_{\max} = 2, \text{ Complete monomial:} \\ &\Phi_0 = \phi_0^1\phi_0^2,\ \Phi_1 = \phi_1^1\phi_0^2,\ \Phi_2 = \phi_0^1\phi_1^2,\ \Phi_3 = \phi_1^1\phi_1^2\Phi_4 = \phi_2^1\phi_0^2,\ \Phi_5 = \phi_0^1\phi_2^2. \end{aligned} \tag{9}$$

The tuple is generated and stored as matrix in the *Trainer* object, below is an example of this matrix with $D = 3$ and $k_{\max} = 3$:

$$k_{d,i} = \left[\begin{array}{c|ccc|cccccc|cccccccccc} 0 & 1 & 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 & 3 & 2 & 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 2 & 1 & 0 & 0 & 1 & 0 & 2 & 1 & 0 & 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 2 & 0 & 0 & 1 & 0 & 1 & 2 & 0 & 1 & 2 & 3 \end{array}\right] \tag{10}$$

Statistical moments are based on the expectation of a function of the quantity of interest:

$$E\left[g\left(Q(\vec{\xi})\right)\right] = \int_{-\infty}^{\infty} g\left(Q(\vec{\xi})\right) f(\vec{\xi}) d\vec{\xi}. \tag{11}$$

The first four statistical moments, and the most common ones, are defined as:

$$\text{Mean: } \mu = E[Q], \tag{12}$$

$$\text{Variance: } \sigma^2 = E\left[(Q-\mu)^2\right] = E[Q^2] - \mu^2, \tag{13}$$

$$\text{Skewness: Skew} = \frac{E\left[(Q-\mu)^3\right]}{\sigma^3} = \frac{E[Q^3] - 3\sigma^2\mu - \mu^3}{\sigma^3}, \tag{14}$$

$$\text{Kurtosis: Kurt} = \frac{E\left[(Q-\mu)^4\right]}{\sigma^4} = \frac{E[Q^4] - 4E[Q^3]\mu + 6\sigma^2\mu^2 + 3\mu^4}{\sigma^4}. \tag{15}$$

Because of the orthogonality of the polynomials, mean and variance are trivial to compute:

$$E\left[Q_{\text{pc}}\right] = q_0, \tag{16}$$

$$E\left[Q_{\text{pc}}^2\right] = \sum_{i=1}^{P} q_i^2 \prod_{d=1}^{D} \left\langle \phi_{k_{d,i}}^2 \right\rangle, \tag{17}$$

where $\left\langle \Phi_i^2 \right\rangle$ is known analytically. The higher order moments are significantly more taxing to compute since it does not take advantage of orthogonality:

$$E\left[Q_{\text{pc}}^3\right] = \sum_{i=1}^{P}\sum_{j=1}^{P}\sum_{k=1}^{P} q_i q_j q_k \prod_{d=1}^{D} \left\langle \phi_{k_{d,i}} \phi_{k_{d,j}} \phi_{k_{d,k}} \right\rangle, \tag{18}$$

$$E\left[Q_{\text{pc}}^4\right] = \sum_{i=1}^{P}\sum_{j=1}^{P}\sum_{k=1}^{P}\sum_{\ell=1}^{P} q_i q_j q_k q_\ell \prod_{d=1}^{D} \left\langle \phi_{k_{d,i}} \phi_{k_{d,j}} \phi_{k_{d,k}} \phi_{k_{d,\ell}} \right\rangle, \tag{19}$$

where the polynomial norms are computed using one-dimensional quadrature. We see here the number of operations required to compute Kurtosis is approximately DN^4. If the number of coefficients is sufficiently high, these moments would probably be best computed inexactly by sampling the PC surrogate.

The *PolynomialChaosTrainer* takes in a list of distributions and constructs a polynomial class based on their type. Given a sampler and a vector *Reporter* of results from sampling, it then loops through the MC or quadrature points to compute the coefficients. The statistical moments are then computed based on user preferences and the model can be evaluated using the `evaluate` function from any other moose object that has the reference. The algorithm uses the parallelization of the sampler to compute the coefficients, no other part of the algorithm is parallelized.

2.3.2 Polynomial Regression

Polynomial regression (PR) is a similar ROM technique as PC where a QoI Q is expanded as a sum of polynomials dependent on a set of parameters $\vec{\xi}$, as in Eq. (1) [6]. However, these polynomials are not necessarily orthogonal and do not depend on the parameters' probability distribution:

$$\Phi_i(\vec{\xi}) = \vec{\xi}^{\vec{i}} = \prod_{j=1}^{D} x_j^{i_j}, i = 1, ..., P. \tag{20}$$

In order determine the coefficients of the PR model, and ordinary least squares (OLS) procedure is used. OLS is performed by representing the parameter training data as a matrix and the QoI data as a vector:

$$\mathbf{X} = \begin{bmatrix} \xi_{1,1} & \xi_{1,2} & \dots & \xi_{1,D} \\ \xi_{2,1} & \xi_{2,2} & \dots & \xi_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ \xi_{N,1} & \xi_{N,2} & \dots & \xi_{N,D} \end{bmatrix} = \begin{bmatrix} \vec{\xi}_1^T \\ \vec{\xi}_2^T \\ \vdots \\ \vec{\xi}_N^T \end{bmatrix}, \tag{21}$$

$$\mathbf{y} = \begin{bmatrix} Q(\vec{\xi}_1) \\ Q(\vec{\xi}_2) \\ \vdots \\ Q(\vec{\xi}_N) \end{bmatrix}. \tag{22}$$

For convenience, a regression matrix can be defined as:

$$\mathbf{R} = \begin{bmatrix} \Phi_1(\vec{\xi}_1) & \Phi_2(\vec{\xi}_1) & \dots & \Phi_P(\vec{\xi}_1) \\ \Phi_1(\vec{\xi}_2) & \Phi_2(\vec{\xi}_2) & \dots & \Phi_P(\vec{\xi}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_N(\vec{\xi}_1) & \Phi_2(\vec{\xi}_N) & \dots & \Phi_P(\vec{\xi}_N) \end{bmatrix}. \tag{23}$$

Using the regression matrix $\mathbf{R}$ and an OLS approach, the unknown coefficients can be determined as follows:

$$\vec{q} \equiv \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_P \end{bmatrix} = \left(\mathbf{R}^T\mathbf{R}\right)^{-1} \mathbf{R}^T \mathbf{y}. \tag{24}$$

Finally, it must be mentioned that this method is only applicable if $N_p \leq N$ and keeping $N_p << N$ is recommended.

Unfortunately, the OLS approach is known to have some issues like:

- It is prone to overfit the data,
- It yields inaccurate results if the input variables are correlated,
- It is sensitive to outliers.

To tackle the problem, an L^2 regularization, or Tikhonov regularization, is adopted to make sure that the coefficients of the expansion do not have uncontrollably high values. This extended least squares regression is often referred to as Ridge Regression [7]. In this scenario the coefficients can be determined by solving:

$$\vec{q} = \left(\mathbf{R}^T\mathbf{R} + \lambda I\right)^{-1} \mathbf{R}^T \mathbf{y}, \tag{25}$$

where λ is a penalty parameter which penalizes coefficients with large magnitudes. As $\lambda \to 0$, Ridge regression converges to OLS.

2.3.3 Gaussian Processing

"Gaussian Processes for Machine Learning" [8] provides a well written discussion of Gaussian processing (GP), and a brief summary of its Chapters 1-5 are presented here for background. Given a set of inputs $\mathbf{X}$ for which we have made observations of the correspond outputs $\mathbf{y}$ using the system ($\mathbf{y} = Q(\mathbf{X})$). Given another set of inputs $\mathbf{X}_\star = \{\vec{\xi}_{\star 1}^T, \cdots, \vec{\xi}_{\star M}^T\}$ we wish to predict the associated outputs $\mathbf{y}_\star = Q(\mathbf{X}_\star)$ without evaluation of $Q(X_\star)$, which is presumed costly.

In overly simplistic terms, GP modeling is driven by the idea that trials which are "close" in their input parameter space will be "close" in their output space. Closeness in the parameter space is driven by the covariance function $k(\vec{\xi}, \vec{\xi}')$ (also called a kernel function, not to be confused with a MOOSE kernel). This covariance function is used to generate a covariance matrix between the complete set of parameters $\mathbf{X} \cup \mathbf{X}_\star = \{\vec{\xi}_1, \cdots, \vec{\xi}_N, \vec{\xi}_{\star 1}, \cdots, \vec{\xi}_{\star M}\}$, which can then be interpreted block-wise as various covariance matrices between $\mathbf{X}$ and $\mathbf{X}_\star$.

$$\begin{aligned}
\mathbf{K}(X \cup X_\star, X \cup X_\star) &= \left[\begin{array}{ccc|ccc}
k(\vec{\xi}_1, \vec{\xi}_1) & \cdots & k(\vec{\xi}_1, \vec{\xi}_N) & k(\vec{\xi}_1, \vec{\xi}_{\star 1}) & \cdots & k(\vec{\xi}_1, \vec{\xi}_{\star M}) \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
(\vec{\xi}_N, \vec{\xi}_1) & \cdots & k(\vec{\xi}_N, \vec{\xi}_N) & k(\vec{\xi}_N, \vec{\xi}_{\star 1}) & \cdots & k(\vec{\xi}_N, \vec{\xi}_{\star M}) \\
\hline
k(\vec{\xi}_{\star 1}, \vec{\xi}_1) & \cdots & k(\vec{\xi}_{\star 1}, \vec{\xi}_N) & k(\vec{\xi}_{\star 1}, \vec{\xi}_{\star 1}) & \cdots & k(\vec{\xi}_{\star 1}, \vec{\xi}_{\star M}) \\
\vdots & & \vdots & \vdots & & \vdots \\
k(\vec{\xi}_{\star M}, \vec{\xi}_1) & \cdots & k(\vec{\xi}_{\star M}, \vec{\xi}_N) & k(\vec{\xi}_{\star M}, \vec{\xi}_{\star 1}) & \cdots & k(\vec{\xi}_{\star M}, \vec{\xi}_{\star M})
\end{array}\right] \\
&= \left[\begin{array}{c|c}
\mathbf{K}(\mathbf{X}, \mathbf{X}) & \mathbf{K}(\mathbf{X}, \mathbf{X}_\star) \\
\hline
\mathbf{K}(\mathbf{X}_\star, \mathbf{X}) & \mathbf{K}(\mathbf{X}_\star, \mathbf{X}_\star)
\end{array}\right] \\
&= \left[\begin{array}{c|c}
\mathbf{K} & \mathbf{K}_\star \\
\hline
\mathbf{K}_\star^T & \mathbf{K}_{\star\star}
\end{array}\right].
\end{aligned} \tag{26}$$

The GP model consists of an infinite collection of functions, all of which agree with the training/observation data. Importantly the collection has closed forms for 2nd order statistics (mean and variance). When used as a surrogate, the nominal value is chosen to be the mean value. The method can be broken down into two step: definition of the prior distribution then conditioning on observed data.

A Gaussian process is a (potentially infinite) collection of random variables, such that the joint distribution of every finite selection of random variables from the collection is a Gaussian distribution.

$$\mathcal{GP}(\mu(\vec{\xi}), k(\vec{\xi}, \vec{\xi}')). \tag{27}$$

In an analogous way that a multivariate Gaussian is completely defined by its mean vector and its covariance matrix, a Gaussian Process is completely defined by its mean function and covariance function. The (potentially) infinite number of random variables within the Gaussian process correspond to the (potentially) infinite points in the parameter space our surrogate can be evaluated at.

We assume the observations (both training and testing) are pulled from an $N + M$ multivariate Gaussian distribution. The covariance matrix Σ is the result of the choice of covariance function.

$$\mathbf{y} \cup \mathbf{y}_\star \sim \mathcal{N}(\mu, \Sigma). \tag{28}$$

Note that μ and Σ are a vector and matrix respectively, and are a result of the mean and covariance functions applied to the sample points. Discussions of GP are typically presented under assumption that $\mu = 0$. This occurs without loss of generality since any sample can be made $\mu = 0$ by subtracting the sample mean (or a variety of other preprocessing options). Note that in a training/testing paradigm, the testing data $\mathbf{y}_\star$ is unknown, so determination of what to use as μ is based on the information from the training data $\mathbf{y}$ (or some other prior assumption). The covariance functions currently implemented in STM are listed below.

- Squared exponential – Also referred to as a radial basis function (RBF) this is a widely used, general purpose covariance function. Serves as a common starting point for many.
- Exponential – A simple exponential covariance function.
- Matern half integer – Implementation of the Matern class of covariance function, where the ν parameter takes on half-integer values.

With the prior formed as above, conditioning on the available training data Y is performed. This alters the mean and variance to new values $\mu_\star$ and $\Sigma_\star$, restricting the set of possible functions which agree with the training data.

$$\mu_\star = \mu + \mathbf{K}_\star \mathbf{K}^{-1}(\mathbf{y} - \mu) \tag{29}$$

$$\Sigma_\star = \mathbf{K}_{\star\star} - \mathbf{K}_\star^T \mathbf{K}^{-1} \mathbf{K}_\star, \tag{30}$$

$$\mathbf{y}_\star \sim \mathcal{N}(\mu_\star, \Sigma_\star). \tag{31}$$

When used as a surrogate, the nominal value is typically taken as the mean value, with $diag(\Sigma_\star)$ providing variances which can be used to generate confidence intervals.

While the only apparent decision in the above formulation is the choice of covariance function, most covariance functions will contain hyperparameters of some form which need to be selected in some manner. While each covariance function will have its own set of hyperparameters, a few hyperparameters of specific forms are present in many common covariance functions. Frequently Kernels consider the distance between two input parameters $\vec{\xi}$ and $\vec{\xi'}$. For system of only a single parameter this distance often takes the form of

$$\frac{|\xi - \xi'|}{\ell}. \tag{32}$$

In this form the ℓ factor set a relevant length scale for the distance measurements. When multiple input parameters are to be considered, it may be advantageous to specify M different length scales for each of the M parameters, resulting in a vector $\vec{\ell}$. For example distance may be calculated as

$$\sqrt{\sum_{i=1}^{M} \left(\frac{\xi_i - \xi_i'}{\ell_i}\right)^2}. \tag{33}$$

When used with standardized parameters, ℓ can be interpreted in units of standard deviation for the relevant parameter. Signal variance (σ_f^2) serves as an overall scaling parameter. Given a covariance function $\tilde{k}$ (which is not a function of σ_f^2), the multiplication of σ_f^2 yields a new valid covariance function.

$$k(x, x', \sigma_f) = \sigma_f^2 \tilde{k}(x, x') \tag{34}$$

This multiplication can also be pulled out of the covariance matrix formation, and simply multiply the matrix formed by $\tilde{k}$

$$\mathbf{K}(\xi, \xi', \sigma_f) = \sigma_f^2 \tilde{\mathbf{K}}(\xi, \xi') \tag{35}$$

Noise variance (σ_n^2) represents noise in the collected data, and is an additional σ_n^2 factor on the variance terms (when $\xi = \xi'$).

$$k(x, x', \sigma_f, \sigma_n) = \sigma_f^2\, \tilde{k}(x, x') + \sigma_n^2\, \delta_{x,x'} \tag{36}$$

In the matrix representation this adds a factor of σ_n^2 to diagonal of the noiseless matrix $\tilde{\mathbf{K}}$

$$\mathbf{K}(x, x', \sigma_f, \sigma_n) = \sigma_f^2\, \tilde{\mathbf{K}}(x, x') + \sigma_n^2 \mathbf{I} \tag{37}$$

Due to the addition of σ_n^2 along the diagonal of the K matrix, this hyperparameter can aid in the the inversion of the covariance matrix. For this reason adding a small amount of σ_n^2 may be preferable, even when you believe the data to be noise free.

2.3.4 Proper Orthogonal Decomposition

Proper Orthogonal Decomposition (POD) is an intrusive reduced basis (RB) method which, unlike non-intrusive surrogates such as PR or PC, is capable of considering the physics of the full-order problem at surrogate level [9]. Therefore, it is often referred to as a physics-based but still data-driven approach. The intrusiveness, however, decreases the range of problems which this method can be used for. Currently in the STM, this surrogate model can deal with parameterized scalar-valued linear steady-state PDEs with affine parameter dependence only. The POD *Trainer* object is responsible for two steps in the generation of the surrogate model: generation of reduced subspaces and generation of reduced operators. It must be mentioned that in POD-RB literature, the training phase is often referred to as offline phase and in this section the two expressions are used interchangeably.

Before the details of the above-mentioned steps are discussed, a short overview is given about the problems considered. A scalar-valued linear steady-state PDE can be expressed in operator notation as:

$$\mathcal{A}u = b, \tag{38}$$

where u is the solution, $\mathcal{A}$ is a linear operator and b is a source term. The linear operator and the source terms may depend on uncertain parameters which are denoted by $\xi_i, i = 0, ..., D$ and organized into a parameter vector $\vec{\xi} = [\xi_1, ..., \xi_D]^T$. Therefore, Eq. (38) can be expressed as:

$$\mathcal{A}(\vec{\xi})u = b(\vec{\xi}). \tag{39}$$

This also means that the solution itself is the function of these parameters $u = u(\vec{\xi})$. To make an efficient surrogate, operator $\mathcal{A}(\vec{\xi})$ and source $b(\vec{\xi})$ should have an affine parameter dependence:

$$\mathcal{A}(\vec{\xi}) = \sum_{i=1}^{N_A} f_i^A(\vec{\xi})\mathcal{A}_i, \quad \text{and} \quad b(\vec{\xi}) = \sum_{i=1}^{N_b} f_i^b(\vec{\xi}) b_i, \tag{40}$$

or in other words, the operators have to be decomposable as the sums of products of parameter-dependent scalar functions and parameter-independent constituent operators. By plugging the decompositions back to Eq. (39), the problem takes the following form:

$$\left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi})\mathcal{A}_i\right) u = \sum_{i=1}^{N_b} f_i^b(\vec{\xi}) b_i. \tag{41}$$

Therefore, before starting the construction of a POD-RB surrogate model, the user must identify the these decompositions first. At input level, this can be done by utilizing the *Tagging* system. For each constituent operator a separate vector tag has to be created and the tags need to be supplied to the trainer object through the `tag_names` input parameter. Furthermore, an indicator shall be added to each tag through the `tag_types` input to show if the tag corresponds to a source term (b_i) or an operator ($\mathcal{A}_i$). As a last step, this system is discretized in space using the finite element method to obtain:

$$\left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi})\mathbf{A}_i\right)\mathbf{u} = \sum_{i=1}^{N_b} f_i^b(\vec{\xi})\mathbf{b}_i, \tag{42}$$

where $\mathbf{A}_i$ and $\mathbf{b}_i$ are finite element matrices and vectors, while $\mathbf{u}$ denotes the vector containing the values of the degrees of freedom. This model is referred to as FOM in subsequent sections. Furthermore, let $\mathbf{u}(\vec{\xi^*})$ denote a solution vector which is obtained by solving Eq. (42) with $\vec{\xi} = \vec{\xi^*}$.

Even though it is not explicitly stated above, $\mathbf{u}$ may contain solutions for multiple variables, hence it can be expressed as $\mathbf{u} = [\mathbf{u}_1; ...;\ \mathbf{u}_{N_v}]$, where N_v is the total number of variables. It is assumed that each variable has $N_i,\ i = 1, ..., N_v$ spatial degrees of freedom, thus the size of the full-order system is $\sum\limits_{i=1}^{N_v} N_i$. As a first step in this process, Eq. (42) is solved using N_s different parameter samples and the solution vectors for each variable defined on the `var_names` input parameter are saved into snapshot matrices

$$\mathbf{S}_i = [\mathbf{u}_i(\vec{\xi_1}), ..., \mathbf{u}_i(\vec{\xi}_{N_s})]. \tag{43}$$

The next step in this process is to use these snapshots to create reduced sub-spaces for each variable. This can be done by performing POD on the snapshot matrices, which consists of the following four steps for each variable:

1. Creation of correlation matrices: The correlation matrices ($\mathbf{C}_i$) can be computed using the snapshot matrices as $\mathbf{C}_i = \mathbf{S}_i^T\mathbf{W}_i\mathbf{S}_i$, where $\mathbf{W}_i$ is a weighting matrix. At this moment only $\mathbf{W}_i = \mathbf{I}$ is supported, where $\mathbf{I}$ is the identity matrix.

2. Eigenvalue decomposition of the correlation matrices: The eigenvalue decompositions of the correlation matrices is obtained as: $\mathbf{C}_i = \mathbf{V}_i\mathbf{\Lambda}_i\mathbf{V}_i^T$, where matrix $\mathbf{V}_i$ and matrix $\mathbf{\Lambda}_i$ contain the eigenvectors and eigenvalues of $\mathbf{C}_i$, respectively.

3. Determining the dimension of the reduced subspace: Based on the magnitude of the eigenvalues ($\lambda_{i,k},\ k = 1, ..., N_s$) in $\mathbf{\Lambda}_i$, one can compute how many basis functions are needed to reconstruct the snapshots with a given accuracy. The rank of the subspace (r_i) can be determined as:

$$r_i = \underset{1\leq r_i\leq N_s}{\arg\min}\left(\frac{\sum_{k=1}^{r_i}\lambda_{i,k}}{\sum_{k=1}^{N_s}\lambda_{i,k}} > 1 - \tau_i\right), \tag{44}$$

   where τ is a given parameter describing the allowed error in the reconstruction of the snapshots.

4. The reconstruction of the basis vectors for each variable: For this, the eigenvalues and eigenvectors of the correlation matrices are used together with the snapshots as:

$$\Phi_{i,k} = \frac{1}{\sqrt{\lambda_{i,k}}}\sum_{j=1}^{N_s}\mathbf{V}_{i,k,j}S_{i,j}, \tag{45}$$

   where $\Phi_{i,k}$ is the k-th ($k = 1, ..., r_i$) basis function of the reduced subspace for variable $\mathbf{u}_i$. Moreover, $\mathbf{V}_{i,k,j}$ denottes the j-th element of the k-th eigenvector of correlation matrix $\mathbf{C}_i$. It is important to remember that $\Phi_{i,k}$ has a global support in space, and shall not be mistaken for the local basis functions (ϕ) of the finite element approximation. The global basis vectors can be also referred to as POD modes and the two expressions are used interchangeably from here on.

Finally, the solutions of different variables in the full-order model can be approximated as the parameter-dependent linear combination of these basis functions:

$$\mathbf{u}_i(\vec{\xi}) \approx \sum_{k=1}^{r_i} \Phi_{i,k} c_{i,k}(\vec{\xi}), \quad \text{or} \quad \mathbf{u}_i(\vec{\xi}) \approx \mathbf{\Phi}_i \mathbf{c}_i(\vec{\xi}) \tag{46}$$

where $\mathbf{\Phi}_i = [\Phi_{i,1}, ..., \Phi_{i,r_i}]$ is a matrix with the POD modes as columns and $\mathbf{c}_i = [c_{i,1}, ..., c_{i,r_i}]^T$ are the expansion coefficients. In essence, these coefficients describe the coordinates of the approximate solution in the reduced subspace. To approximate the full solution vector $\mathbf{u}(\vec{\xi})$ using its components ($\mathbf{u}_i(\vec{\xi})$-s), a segregated approach is used as follows:

$$\mathbf{u}(\vec{\xi}) = \begin{bmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{N_v} \end{bmatrix} \approx \begin{bmatrix} \mathbf{\Phi}_1 \mathbf{c}_1(\vec{\xi}) \\ \vdots \\ \mathbf{\Phi}_{N_s} \mathbf{c}_{N_s}(\vec{\xi}) \end{bmatrix} = \begin{bmatrix} \mathbf{\Phi}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{\Phi}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{\Phi}_{N_s} \end{bmatrix} \begin{bmatrix} \mathbf{c}_1(\vec{\xi}) \\ \mathbf{c}_2(\vec{\xi}) \\ \vdots \\ \mathbf{c}_{N_s}(\vec{\xi}) \end{bmatrix} = \mathbf{\Phi}\mathbf{c}(\vec{\xi}). \tag{47}$$

It is important to mention that in this approximation the unknowns are the elements of $\mathbf{c}(\vec{\xi})$ vector. In most of the cases, the size of this vector ($\sum\limits_{i=1}^{N_v} r_i$) is considerably smaller than the size of the original solution vector ($\sum\limits_{i=1}^{N_v} N_i$).

To generate reduced operators, Eq. (47) is plugged into Eq. (42) first:

$$\left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi}) \mathbf{A}_i \right) \mathbf{\Phi}\mathbf{c}(\vec{\xi}) = \sum_{i=1}^{N_b} f_i^b(\vec{\xi}) \mathbf{b}_i. \tag{48}$$

Since the size of $\mathbf{c}$ is smaller than the size of $\mathbf{u}$, this equation is under-determined. To solve this problem, a Galerkin projection is used on the system:

$$\mathbf{\Phi}^T \left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi}) \mathbf{A}_i \right) \mathbf{\Phi}\mathbf{c}(\vec{\xi}) = \mathbf{\Phi}^T \sum_{i=1}^{N_b} f_i^b(\vec{\xi}) \mathbf{b}_i. \tag{49}$$

By pulling the basis matrices into the summation, the following form is obtained.

$$\left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi}) \mathbf{\Phi}^T \mathbf{A}_i \mathbf{\Phi} \right) \mathbf{c}(\vec{\xi}) = \sum_{i=1}^{N_b} f_i^b(\vec{\xi}) \mathbf{\Phi}^T \mathbf{b}_i, \tag{50}$$

where the reduced operators $\mathbf{A}_i^r = \mathbf{\Phi}^T \mathbf{A}_i \mathbf{\Phi}$ and source terms $\mathbf{b}_i^r = \mathbf{\Phi}^T \mathbf{b}_i$ can be precomputed once the basis functions are available. Therefore, the reduced equation system that is solved to obtain $\mathbf{c}(\vec{\xi})$ is

$$\left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi}) \mathbf{A}_i^r \right) \mathbf{c}(\vec{\xi}) = \sum_{i=1}^{N_b} f_i^b(\vec{\xi}) \mathbf{b}_i^r. \tag{51}$$

It is important to note that this equation system is of size $\sum_{i=1}^{N_v} r_i \times \sum_{i=1}^{N_v} r_i$, therefore it can be solved much faster than the original full-order system which is of size $\sum_{i=1}^{N_v} N_i \times \sum_{i=1}^{N_v} N_i$. The computation of the reduced operators consists of two step in the current implementation:

1. Computing the effect of the full-order operator on the global basis functions: This step includes the creation of $\mathbf{A}_i\mathbf{\Phi}$. In practice, this is done by plugging in the basis function into a *PODFullSolveMultiApp* object which evaluates the residual for a given vector tag (defined using the `tag_names` input argument). The tagged residual is then transferred back to the trainer using a *PODResidualTransfer* object. In case when the residual from a kernel contains contributions to both the system matrix and the source term (e.g. Dirichlet BC or time derivative), certain input-level tricks can be used to separate these.

2. Projection of the residual vectors: this step consists of computing the $\mathbf{\Phi}^T(\mathbf{A}_i\mathbf{\Phi})$ inner products.

As a final note, it must emphasized that even though obtaining snapshots and creating reduced operators is a computationally expensive procedure, it has to be carried out only once. After this initial investment every new evaluation for a new parameter sample involves the summation and scaling of small dense matrices, which is of low computational cost.

As of now, the training phase is implemented in a semi-parallel manner. This means that the snapshot generation, correlation matrix generation, base generation and the computation of the reduced operators are all executed in parallel. However, the eigenvalues and eigenvectors of the correlation matrices are obtained in serial. Therefore, this phase may experience considerable slowdown when the number of snapshots is large (above ∼2000).

The surrogate object, *PODReducedBasisSurrogate*, takes the reduced operators and bases from the trainer and assembles the reduced equation system for a new parameter sample ($\vec{\xi}^*$):

$$\left(\sum_{i=1}^{N_A} f_i^A(\vec{\xi}^*)\mathbf{A}_i^r\right)\mathbf{c}(\vec{\xi}^*) = \sum_{i=1}^{N_b} f_i^b(\vec{\xi}^*)\mathbf{b}_i^r. \tag{52}$$

Following this, the reduced equation system is solved for $\mathbf{c}(\vec{\xi}^*)$ and an approximate solution is reconstructed as

$$\mathbf{u}(\vec{\xi}^*) \approx \mathbf{\Phi}\mathbf{c}(\vec{\xi}^*). \tag{53}$$

It must be mentioned that in case of Dirichlet boundary conditions (either nodal or in weak form) there is a contribution in $\mathbf{A}_{Dir}^r$ and $\mathbf{b}_{Dir}^r$ as well. However, in this case a penalty parameter (γ) is used to enforce the boundary condition at reduced order level. Therefore, the slightly modified equation system can be written as:

$$\left(\sum_{i=1}^{N_A-1} f_i^A(\vec{\xi}^*)\mathbf{A}_i^r + \gamma f_{Dir}^A(\vec{\xi}^*)\mathbf{A}_{Dir}^r\right)\mathbf{c}(\vec{\xi}^*) = \sum_{i=1}^{N_b-1} f_i^b(\vec{\xi}^*)\mathbf{b}_i^r + \gamma f_{Dir}^b(\vec{\xi}^*)\mathbf{b}_{Dir}^r. \tag{54}$$

the magnitude of γ can be set using the `penalty` parameter in the input file. It is important to note that by increasing the magnitude of γ, the condition number of the reduced equation may deteriorate, therefore the overly high values are not recommended. It is also important to mention that the size of the reduced system can be modified by changing the number of bases per variable in the input file. To do this one can use `change_rank` and `new_ranks` input parameters. This feature makes it possible to test the accuracy of the surrogate with different subspace sizes without the need of rerunning the training procedure.

2.4 Examples

This section presents some results from the updates of STM. The first, Section 2.4.1, shows the memory and run-time performance of running samples on a relatively large MOOSE model. The second, Section 2.4.2, applies the non-intrusive ROM techniques to a simple heat conduction problem. Section 2.4.3 shows an example of the intrusive POD ROM. Finally, Section 2.4.4 performs a parameter study of a multiphysics problem using the PC surrogate. More documentation on these examples can be found on the following pages of the MOOSE website:

- Section 2.4.1: `https://mooseframework.inl.gov/modules/stochastic_tools/batch_mode.html`
- Section 2.4.2: `https://mooseframework.inl.gov/modules/stochastic_tools/examples/poly_chaos_surrogate.html`, `https://mooseframework.inl.gov/modules/stochastic_tools/examples/poly_regression_surrogate.html`, `https://mooseframework.inl.gov/modules/stochastic_tools/examples/gaussian_process_surrogate.html`
- Section 2.4.3: `https://mooseframework.inl.gov/modules/stochastic_tools/examples/pod_rb_surrogate.html`
- Section 2.4.4: `https://mooseframework.inl.gov/modules/combined/examples/stm_thermomechanics.html`

2.4.1 Batch Mode Scaling Study

This example is meant to demonstrate the batch mode system in the STM and the new capability of specifying a minimum number of processors per sample. The *SamplerFullSolveMultiApp* and *SamplerTransientMultiApp* are capable of running sub-apps in "batch" mode. In normal operation these to object create one sub-app for every row of data in the supplied *Sampler* object. In batch mode one sub-app is created per processor, or group of processors, and re-used to solve for each row of data. In general, there are three modes of operation for the stochastic tools *MultiApp* objects.

1. `normal`: One sub-app is created for each row of data supplied by the *Sampler* object.
2. `batch-reset`: One sub-application is created for each group of processors, this sub-app is destroyed and re-created for each row of data supplied by the *Sampler* object.
3. `batch-restore`: One sub-app is created for each group of processors, this sub-app is backed up after initialization. Then for each row of data supplied by the *Sampler* object the sub-app is restored to the initial state prior to execution.

All three modes are available when using *SamplerFullSolveMultiApp*, the "batch-reset" mode is not available for *SamplerTransientMultiApp* because the sub-application have state that must be maintained as simulation time progresses. The primary benefit to using a batch mode is to improve performance of a simulation by reducing the memory of the running application. The performance gains depend on the type of sub-app being executed as well as the number of samples being evaluated. The following highlight the performance improvements that may be expected for full solve sub-apps.

The first example demonstrates the performance improvements to expect when using *SamplerFullSolveMultiApp* with the use of one processor. In this case, the sub-app solves a steady-state diffusion on a unit cube domain with Dirichlet boundary conditions on the left, $x = 0$, and right,

$y = 1$, sides of the domain. This first demonstration uses a relatively small mesh with 10^3 3D elements. The main app does not perform a solve; it performs a stochastic analysis using Monte Carlo sampling to perturb the values of the two Dirichlet conditions on the sub-applications to vary with a uniform distribution. The example is executed to demonstrate memory performance of the various modes of operation: “normal”, “batch-reset”, and “batch-restore”. Each mode is executed with an increasing number of Monte Carlo samples by setting the `num_rows` parameter of the *MonteCarlo* object. Figure 1 show the resulting memory use at the end of the simulation for each mode of operation with increasing sample numbers.

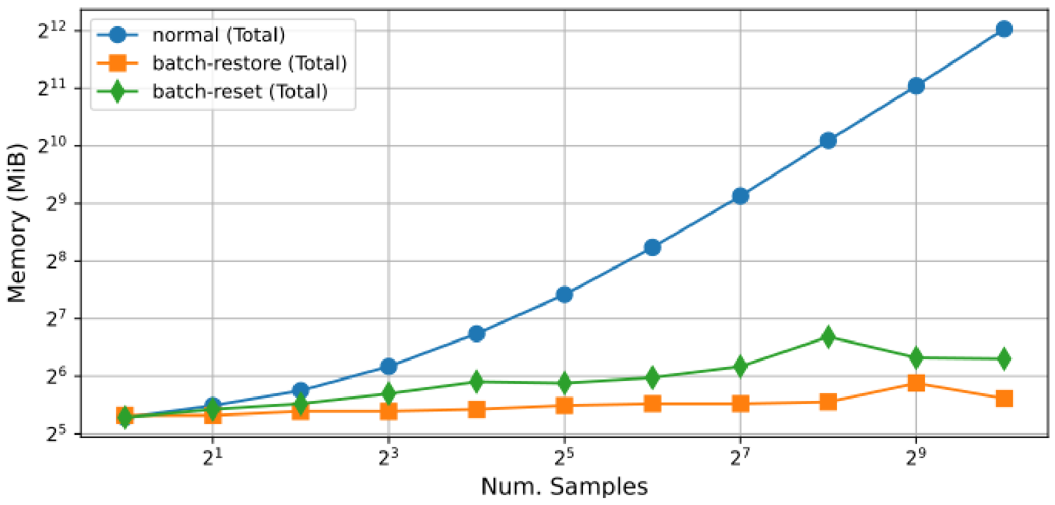


Figure 1: Total memory at the end of the simulation using a *SamplerFullSolveMultiApp* with increasing number of Monte Carlo samples for the three available modes of operation running on a single processor.

An important feature of the various modes of operation is that run-time is not negatively impacted by changing the mode; in some cases using a batch mode can actually decrease total simulation run time. The total run time results for the full solve problem is shown in Figure 2. The time shown in these plots is the total simulation time, which encompasses both the simulation initialization and solve. The differences in speed are mainly due to the installation and destruction of the sub-application. When running in “batch-reset” mode, each data sample causes the sub-app to be created and destroyed during the solve, causing the slowest performance. The “normal” mode creates all sub-apps up front, and the “batch-restore” method uses the backup-restore capability to save the state of the sub-applications, thus does not require as many instantiations and has the lowest run-time. For this example, the solve portion is minimal as such the sub-app creation time plays a large role. As the solve time increases time gains can be expected to be minimal.

The second part of this example utilizes the `min_procs_per_app` input parameter for *SamplerFullSolveMultiApp* for a large version of the previous diffusion problem. Figure 3 shows the total memory usage for various selections of the parameter and varying problem size. Figure 4 shows the corresponding run-time for these problems. All of these simulations were run with 100 processors (`mpiexec -n 100`). The results show that for small problems using more processors has minimal impact on memory and negatively affect run-time. However, for the larger problems the memory savings become much more apparent and the effect on run-time is minimal.

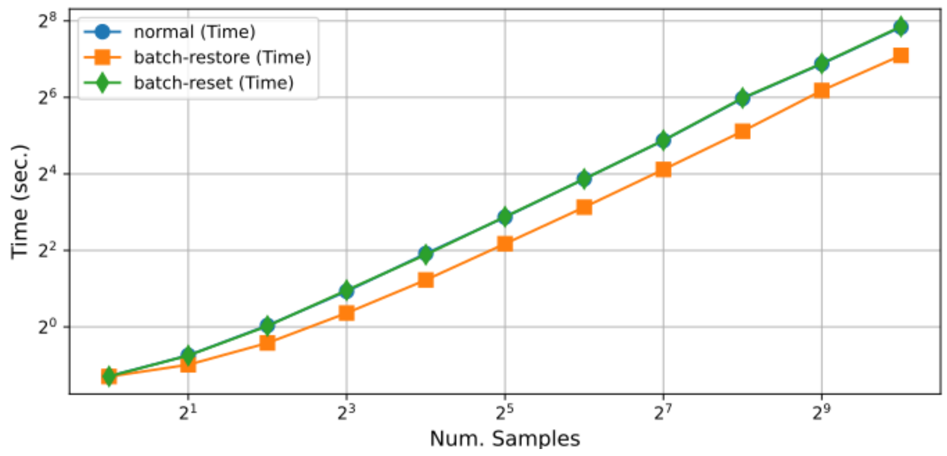
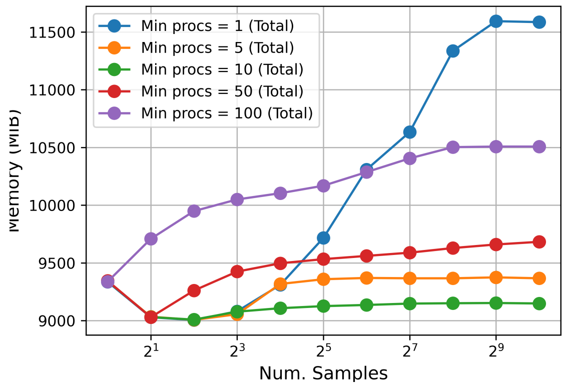
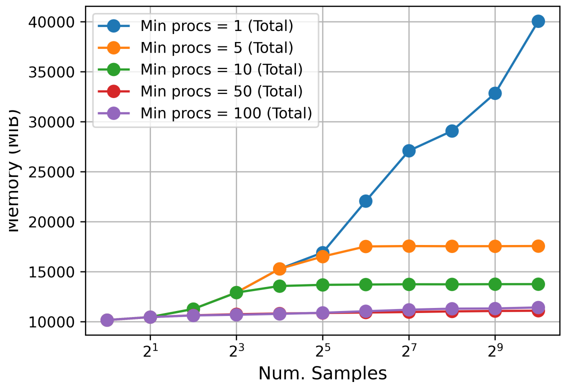


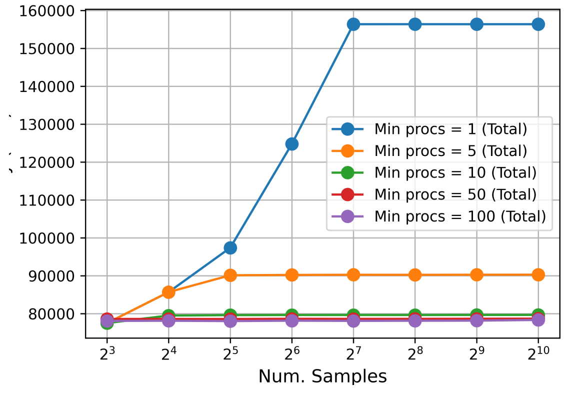
Figure 2: Total execution time of a simulation using a *SamplerFullSolveMultiApp* with increasing number of Monte Carlo samples for the three available modes of operation running on a single processor.



(a) 10^4 Elements

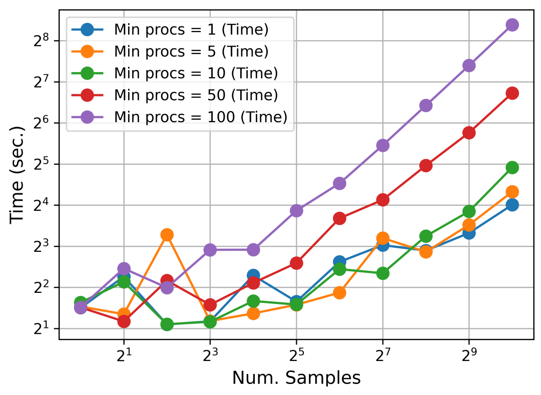


(b) 10^5 Elements

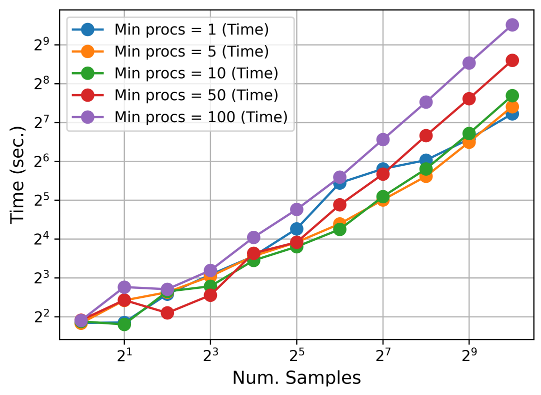


(c) 10^6 Elements

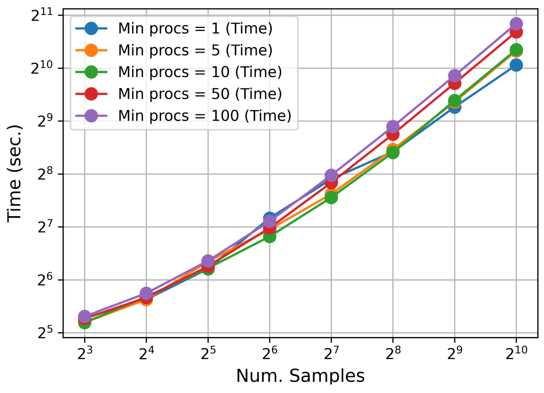
Figure 3: Total memory usage for stochastic sampling of steady-state diffusion problem with varying number of elements and minimum processors per sample.



(a) 10^4 Elements



(b) 10^5 Elements



(c) 10^6 Elements

Figure 4: Execution time for stochastic sampling of steady-state diffusion problem with varying number of elements and minimum processors per sample.

2.4.2 Comparing Surrogate Models – Simple Heat Conduction

This example applies three of the non-intrusive ROM techniques in STM (PC, PR, and GP) to a simple heat conduction problem. The purpose is to demonstrate how to build ROMs and what they can be used for. This example uses a one-dimensional heat conduction problem as the full-order model which has certain uncertain parameters. The model equation is as follows:

$$-k\frac{d^2T}{dx^2} = q\,, \quad x \in [0, L], \tag{55}$$

$$\begin{gathered} \left.\frac{dT}{dx}\right|_{x=0} = 0 \\ T(x = L) = T_\infty \end{gathered} \tag{56}$$

The QoI is the average temperature:

$$\bar{T} = \frac{\int_0^L T(x)dx}{L}, \tag{57}$$

For demonstration, each of the uncertain parameters will have a uniform probability distribution ($\mathcal{U}(a, b)$). Where a and b are the max and minimum bounds of the uniform distribution, respectively. The uncertain parameters for this model problem are described in Table 2.

Table 2: Uncertain parameter definition for 1D heat conduction example

Parameter	Symbol	Distribution
Conductivity	k	$\sim \mathcal{U}(1, 10)$
Volumetric Heat Source	q	$\sim \mathcal{U}(9000, 11000)$
Domain Size	L	$\sim \mathcal{U}(0.01, 0.05)$
Right Boundary Temperature	T_∞	$\sim \mathcal{U}(290, 310)$

This simple model problem has analytical descriptions for the field temperature and average temperature:

$$T(x, k, q, L, T_\infty) = \frac{q}{2k}\left(L^2 - x^2\right) + T_\infty, \tag{58}$$

$$\bar{T}(k, q, L, T_\infty) = \frac{qL^2}{3k} + T_\infty, \tag{59}$$

$$\tag{60}$$

With the quadratic feature of the field temperature, using quadratic elements in the discretization will actually yield the exact solution.

The input file used to solve the one-dimensional heat conduction model can be found in Listing 1. With this input the uncertain parameters are defined as:

1. $k \rightarrow$ `Materials/conductivity/prop_values`
2. $q \rightarrow$ `Kernels/source/value`
3. $L \rightarrow$ `Mesh/xmax`
4. $T_\infty \rightarrow$ `BCs/right/value`

The input file used to train the surrogate models can be found in Listing 2. The uncertainty in each parameter is represented by a *Distribution* in the main STM input which a *Sampler* then evaluates to create a sampling matrix. The objects in blocks *Controls*, *MultiApps*, *Transfers*, and *Reporters* are responsible for managing the communication between main and sub-apps, execution of the sub-apps and the collection of the results.

The next step is to setup the *Trainers*. The required parameters for a PC trainer are:

- `distributions` specify the type of polynomials to use for the expansion, it is very import that these distributions match the distributions given to the sampler.
- `sampler` is the object that will provide sample points which are given to the sub app during execution.
- `response` specifies the result vector for storing the computed values.
- `order` defines the maximum order of the PC expansion, this parameter ultimately defines the accuracy and complexity of the surrogate model.

The parameters for PR are:

- `sampler` is the object that will provide sample points which are given to the sub app during execution.
- `response` specifies the result vector for storing the computed values.
- `max_degree` describes the maximum order of polynomial used in the regression. This is similar to the `order` parameter for PC.
- `regression_type` determines the algorithm used to compute the coefficients of the regression, `ols` is default OLS scheme and specifying `ridge` will perform Ridge regression.

Defining the GP trainer is slightly more complicated. First the *Covariance* function must be defined, *SquaredExponentialCovariance* is used for this example. Because the `signal_variance` and `length_factor` are very difficult to optimize by hand, *GaussianProcessTrainer* uses a TAO optimization algorithm to determine these values. The parameters for the GP trainer are:

- `covariance_function` is the name of the *Covariance* object used for training.
- `standardize_params/standardize_data` centers and scales the parameter/response data if set to `true`.
- `tune_parameters` determines which parameters in the *Covariance* object to optimize.
- `tuning_max/tuning_min` determines the absolution max/min values that the tuned *Covariance* parameters can be set to. This is useful for ensuring the optimization algorithm produces useful results.
- `tao_options` is an advanced parameter which feeds options to the optimization solver.

All *Trainer* objects create variables that *Surrogates* can use for evaluation. Saving this data will allow the surrogate to be run separately in another input. *SurrogateTrainerOutput* is used to output training data.

Evaluating surrogate models typically occurs in a separate run with a separate input file, see Listing 3. In this file, the *Surrogate* model can be loaded by inputting the training data file with the `filename` parameter. Evaluating a surrogate model occurs within objects that obtain the surrogate object's reference and call the `evaluate` function. In this example, *EvaluateSurrogate* is used to evalaute the surrogate with a new sampler. The results of evaluating the surrogate can then be used to compute statistics. Table 3 shows the results of computing the statistics from evaluating each surrogate.

Table 3: Statistics results from evaluating trained ROMs of heat conduction problem.

Model	Mean	Standard Deviation
Reference	300.893	5.8545
PC	300.866	4.9773
PR	300.896	5.8565
GP	300.802	5.9920

2.4.3 Intrusive POD Training

This example is meant to demonstrate how a POD reduced basis surrogate model is trained and used on a parametric problem. The full-order model is a one energy group, fixed-source diffusion-reaction problem, adopted from [10]. The geometry for this problem is presented in Figure 5. The problem has four different material regions, from which three (1, 2 and 3) act as fixed sources. The

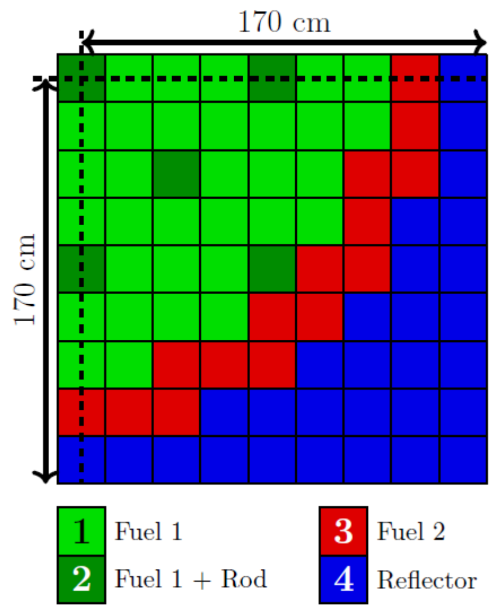


Figure 5: The geometry of the diffusion-reaction problem used in this example

fixed-source diffusion-reaction problem with space dependent coefficients can be expressed as:

$$-\nabla \cdot [D(\mathbf{r})\nabla\psi(\mathbf{r})] + \Sigma_a(\mathbf{r})\psi(\mathbf{r}) = q(\mathbf{r})\,, \quad \mathbf{r} \in \Omega, \tag{61}$$

where $D(\mathbf{r})$ is the diffusion coefficient, $\Sigma_a(\mathbf{r})$ is the reaction coefficient, $q(\mathbf{r})$ is the fixed source term and field variable $\psi(\mathbf{r})$ is the solution of interest. Furthermore, Ω denotes the internal domain, without the boundaries, which can be partitioned into four sub-domains corresponding to the four material regions (Ω_1, Ω_2, Ω_3 and Ω_4). This equation needs to be supplemented with boundary conditions. For the symmetry lines (dashed lines in Figure 5, denoted by $\partial\Omega_{sym}$) a homogeneous Neumann condition is used:

$$-D(\mathbf{r})\nabla\psi(\mathbf{r}) \cdot \mathbf{n}(\mathbf{r}) = 0\,, \quad \mathbf{r} \in \partial\Omega_{sym}, \tag{62}$$

while the rest of the boundaries (in the reflector, denoted by $\partial\Omega_{refl}$) are treated with homogeneous Dirichlet conditions:

$$\psi(\mathbf{r}) = 0\,, \quad \mathbf{r} \in \partial\Omega_{refl}. \tag{63}$$

This problem is parametric in a sense that the solution depends on the values of the coefficients and the source: $\psi = \psi\left(\mathbf{r}, D(\mathbf{r}), \Sigma_a(\mathbf{r}), q(\mathbf{r})\right)$. In this example, material region-wise constant coefficients and source terms are considered yielding eight uncertain parameters altogether (assuming that Region 4 does not have a source). The material properties in each region have uniform distributions ($\mathcal{U}(a,b)$) specified in Table 4 with a and b being the lower and upper bounds.

Table 4: The distributions of the uncertain parameters used in [10]

Parameter	Symbol	Distribution
Diffusion coefficient in Region 1 (cm)	D_1	$\sim \mathcal{U}(0.2, 0.8)$
Diffusion coefficient in Region 2 (cm)	D_2	$\sim \mathcal{U}(0.2, 0.8)$
Diffusion coefficient in Region 3 (cm)	D_3	$\sim \mathcal{U}(0.2, 0.8)$
Diffusion coefficient in Region 4 (cm)	D_4	$\sim \mathcal{U}(0.15, 0.6)$
Reaction coefficient in Region 1 $\left(cm^{-1}\right)$	$\Sigma_{a,1}$	$\sim \mathcal{U}(0.0425, 0.17)$
Reaction coefficient in Region 2 $\left(cm^{-1}\right)$	$\Sigma_{a,2}$	$\sim \mathcal{U}(0.065, 0.26)$
Reaction coefficient in Region 3 $\left(cm^{-1}\right)$	$\Sigma_{a,3}$	$\sim \mathcal{U}(0.04, 0.16)$
Reaction coefficient in Region 4 $\left(cm^{-1}\right)$	$\Sigma_{a,4}$	$\sim \mathcal{U}(0.005, 0.02)$
Fixed-source in Region 1 $\left(\frac{n}{cm^3 \cdot s}\right)$	q_1	$\sim \mathcal{U}(5, 20)$
Fixed-source in Region 2 $\left(\frac{n}{cm^3 \cdot s}\right)$	q_2	$\sim \mathcal{U}(5, 20)$
Fixed-source in Region 3 $\left(\frac{n}{cm^3 \cdot s}\right)$	q_3	$\sim \mathcal{U}(5, 20)$

It is important to mention that POD-RB surrogates are only efficient when the original problem has an affine decomposition. Luckily, the problem at hand has an affine decomposition in the following form:

$$-\sum_{i=1}^{4} \nabla \cdot \left[D_i(\mathbf{r}) \nabla \psi(\mathbf{r})\right] + \sum_{i=1}^{4} \Sigma_{a,i}(\mathbf{r}) \psi(\mathbf{r}) = \sum_{i=1}^{3} q_i(\mathbf{r}), \quad \mathbf{r} \in \Omega, \tag{64}$$

where $D_i(\mathbf{r})$, $\Sigma_{a,i}(\mathbf{r})$ and $q_i(\mathbf{r})$ take the values of D_i, $\Sigma_{a,i}$ and q_i when $\mathbf{r} \in \Omega_i$ and 0 otherwise.

The first step towards creating a POD-RB surrogate model is the generation of a full-order problem which can solve Eq. (61) with fixed parameters, see Listing 4. There are three important factors that need to be considered while preparing the input file for this problem:

1. The user must specify vector tags in the *Problem* block for each component in the affine decomposition of the system. In this example, 8 vector tags are specified for the eight uncertain parameters. These do not introduce extra work in the full-order model, however they help to identify the affine components throughout the training phase.

2. The input file has to reflect the affine decomposition of the problem. This means that the *Kernels*, *BCs*, and *Materials* have to be created in a way that they correspond to the components in the affine decomposition. Note that a separate kernel has been created for every single term in the decomposition. The vector tags in the 'Problem' block are then applied to these kernels to ensure that the affine components are correctly identified throughout the training phase.

3. The values for each uncertain parameter have to be set to 1 by default. This is necessary because the *PODReducedBasisTrainer* uses the same input file to create the affine constituent operators. This ensures that the mentioned operators are not influenced by the parameter-dependent multipliers. Of course, these values are not fixed and are changed by the main application throughout the simulations to values aligned with those in Table 4. However, the default values in the input file should be set to one.

The input file used for training a POD surrogate model can be found in Listing 5. The first step in training is the collection of data, which involves the repeated execution of the full-order problem with different parameter combinations and the saving of the full solution vectors. These solution vectors are often referred to as snapshots and this naming is preferred in this example as well. This step is managed by the main training input file which creates parameter samples, transfers them to the sub-app and collects the results from the completed computations. The snapshot collection phase starts with the definition of the distributions in the *Distributions* block. As a next step, the underlying distributions are sampled to create parameter combinations. This is done using a *LatinHypercubeSampler* defined in the *Samplers* block. It is visible that 100 samples are prepared, meaning that 100 snapshots will be collected for the generation of the surrogates. To be able to create the reduced operators for the surrogate model, a custom *MultiApp*, *PODFullSolveMultiApp*, has been created. This object is responsible for executing sub-problems using different combinations of parameter values provided by the sampler. The secondary function of this object is to create the action of the full-order operators on the basis functions of the reduced subspace. Therefore, this object has to be executed twice in the same simulation. It is visible that unlike a regular *SamplerFullSolveMultiApp*, this custom object has to know certain parameters of the trainer as well. In terms of the *Transfers* block, besides sending the actual parameter samples to the sub-apps, in this intrusive procedure, the snapshots need to be collected from the sub-apps, the basis functions need to be sent back to different sub-applications and the action of the operators on the basis functions need to be collected as well. This requires four transfer objects. The two custom types (*PODSamplerSolutionTransfer* and *PODResidualTransfer*) are specifically used to support *PODReducedBasisTrainer* at this moment. Next, the *PODReducedBasisTrainer* is set up in the *Trainers* block. The trainer stores the snapshots and uses them to create basis functions for the reduced subspaces. Furthermore, it is also responsible for creating the reduced-order operators, therefore it needs to be executed twice in the training process. The trainer object needs to know what variable needs to be reduced and the names of the vector tags from the sub-app to be able to identify the affine constituent operators. Furthermore, using the `tag_types` input argument, the user has to specify if the reduced affine constituent operator acts on the variable or not. The ordering must be the same as the names of the vector tags. The meaning of the energy retention limits is discussed in *PODReducedBasisTrainer*. As a last step in the training process, the basis functions, reduced operators and every necessary information for the surrogate are saved into an `.rd` file. This file can be then used to construct a surrogate model without the need to repeat the training process.

To evaluate surrogate models, a new input file has to be created, see Listing 6. In this example, the same distributions are defined for the parameters as used in the training phase. Therefore, the content of the *Distributions* block is identical to the one in the trainer input file. As a next step, new samples are generated using these distributions. Again, a *LatinHypercubeSampler* is employed for this task, however this time the number of samples is increased to 1000 since the surrogates are orders of magnitudes faster than the full-order model. A *PODReducedBasisSurrogate* is created in the *Surrogates* block. It is constructed using the information available within the corresponding `.rd` file and allows the user to change of the rank of the sub-spaces used for different variables through `change_rank` and `new_ranks` parameters. These surrogate models can be evaluated at the points defined in the testing sample batch by a *PODSurrogateTester* object in the *VectorPostprocessors* block. In this case, QoI is the nodal L^2 norm of the solution for ψ.

In the remainder of this section, a short analysis is provided for the results obtained using the example input files. As already mentioned, the problem has 8 uncertain parameters and altogether 100 parameter samples are generated using *LatinHypercubeSampler* to obtain snapshots for the training. Three examples of the snapshots are presented in Figure 6. It is visible that depending on the actual parameter combination, the profile of the solution can change considerably.

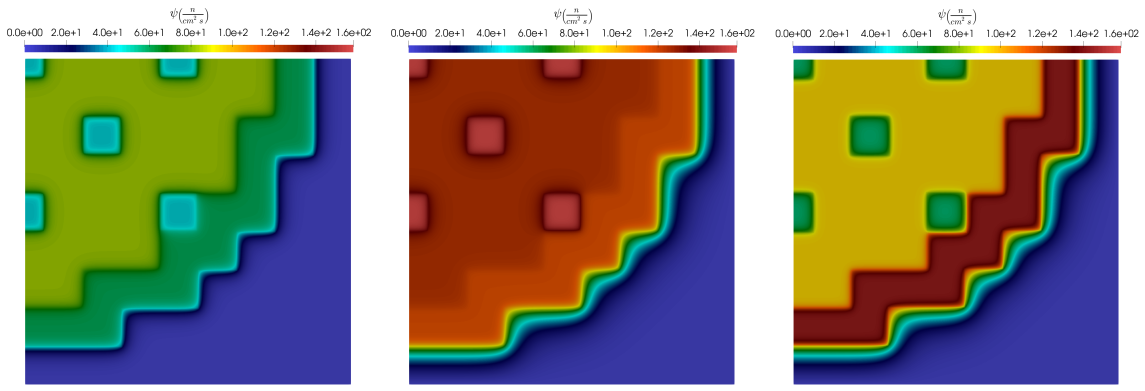


Figure 6: Various snapshots of the FOM during training POD surrogate.

After all of the snapshots are obtained, the basis functions of the reduced subspaces are extracted. In this scenario, an energy retention limit of 0.999 999 999 is used in the trainer which will keep 55 basis functions for the reduced subspace. The decay of the eigenvalues of the snapshot correlation matrix is shown in Figure 7. The reduced operators are then computed using these 55 basis functions.

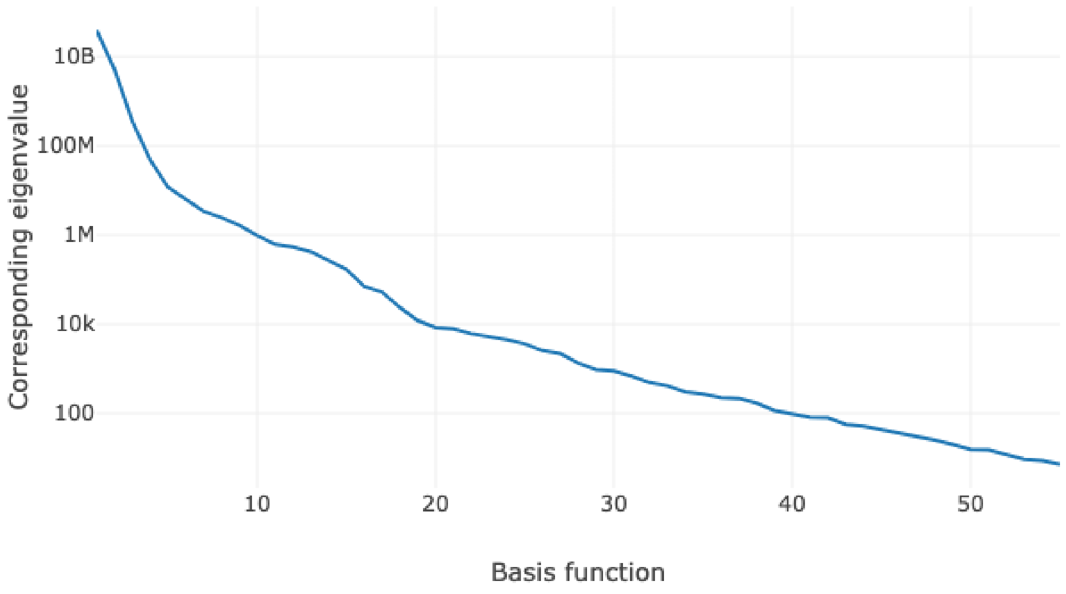


Figure 7: Screen plot of the eigenvalues of the correlation matrix.

As a next step, two surrogate models are prepared using the `change_rank` and `new_ranks` parameters of *PODReducedBasisSurrogate* to change the size of the reduced system. The first surrogate model has 1 basis function, while the other has 8. Both models are then run on a 1000 sample test set and the nodal L^2 norms of the approximate solutions are saved. Additionally, a full-order model was executed on the same test set and the results are saved to serve as reference values. Figure 8 presents the results with the surrogate model built with 1 basis function only. It is visible that the distribution of the QoI (nodal L^2 norm) on the test set is considerably different than the reference distribution.

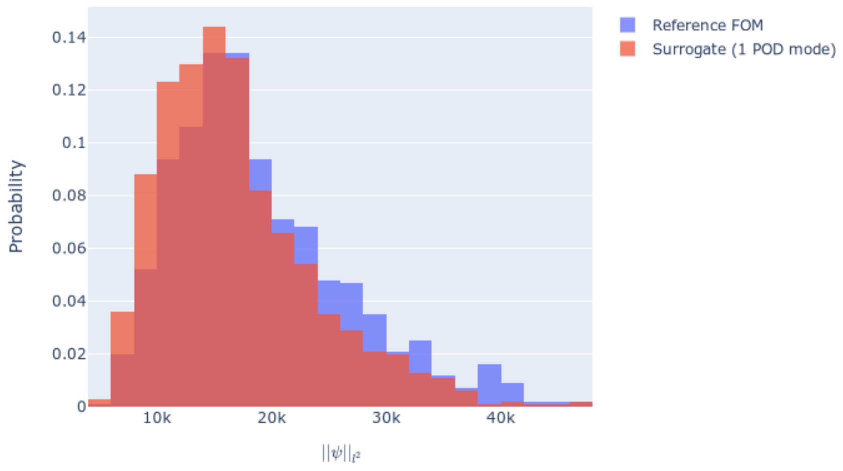


Figure 8: The histogram of the QoI for the FOM reference and the surrogate built with 1 basis function.

Figure 9 shows the distribution of the QoI obtained by a surrogate with 8 basis functions. It is visible that the difference between the reference values and those from the surrogate is negligible.

To see the convergence of the results from the surrogate to those of the full-order model, the surrogate model is run multiple times with different ranks and the following error indicators are computed for each sample in the test set:

$$e_i = \frac{||\psi_{Ref} - \psi_{Surr}||_{l^2}}{||\psi_{Ref}||_{l^2}}. \quad i = 1, ..., 1000. \tag{65}$$

Then, the maximum and average relative errors are recorder as function of the number of basis functions used. Figure 10 shows the results. It is visible that by increasing the number of basis functions, both error indicators decrease rapidly.

Lastly, the computation time FOM on the test set is compared to the combined cost of training and evaluating a POD surrogate model in Table 5. The test has been carried out on one processor only, not using the parallel capabilities of the *MultiApp* system. The results show that it is beneficial to create a POD surrogate if more than 148 evaluations are needed. This assumes that the full-order evaluation time can be equally distributed among the 1000 test samples (0.779 s/sample). By dividing the training time with this number we get a critical sample number above which the generation of a surrogate model is a better alternative.

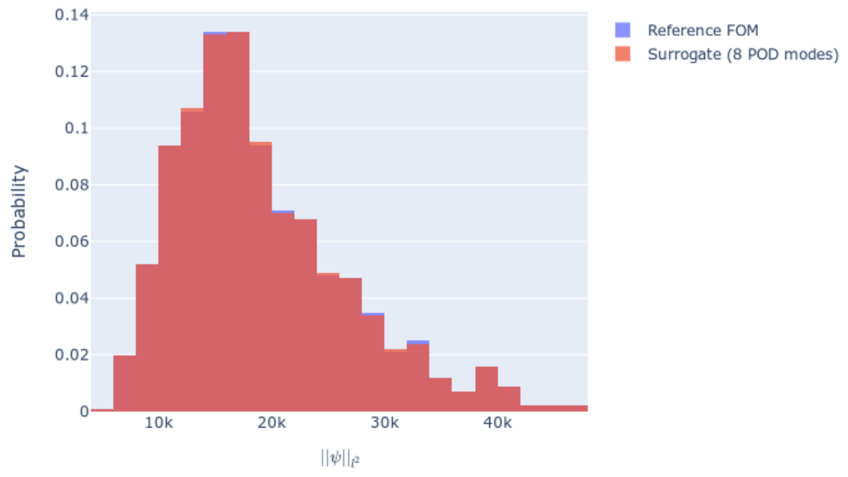


Figure 9: The histogram of the QoI for the FOM reference and the surrogate built with 8 basis function.

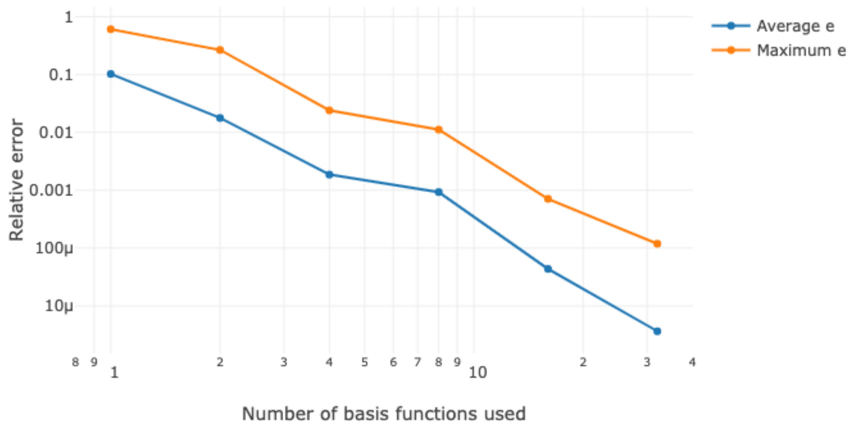


Figure 10: The POD convergence of averaged quantities of interest.

Table 5: The computation time of the full-order solutions on the test set compared to the cost of training a surrogate and evaluating it on the same test set.

Process	Execution time (s)
Evaluation of the FOM with 1000 sample test set	779.5
Training a POD surrogate using 100 samples	116.2
Evaluating POD surrogate with 1000 sample test set (4 basis functions)	0.592
Evaluating POD surrogate with 1000 sample test set (8 basis functions)	0.937
Evaluating POD surrogate with 1000 sample test set (16 basis functions)	1.576

2.4.4 Parameter Study and Surrogate Modeling of Multiphysics Problem

The purpose of this section is to present a multiphysics example using STM. The intention is to showcase the capabilities of the module to produce statistically relevant results including uncertainty propagation and sensitivity, as well as the module's surrogate modeling infrastructure. The problem of interest is a thermomechanics model using a combination of the heat conduction and tensor mechanics modules. The problem involves multiple uncertain material properties and multiple QoIs. Using both Monte Carlo sampling and polynomial chaos surrogate modeling, the effect of these properties' uncertainties are quantified with uncertainty propagation and global sensitivity analysis.

The problem of interest involves a steady-state thermomechanics model. The geometry is a 3-D finite hollow cylinder with two concentric layers of different material properties, seen in Figure 11. Due to symmetry, only 1/8 of the cylinder is represented by the mesh. The inner surface of the ring is exposed to a surface heat source that has a cosine shape along the axis of the cylinder with its peak being at the center. The end and outside of the cylinder have convective boundary conditions. The cylinder is free to displace in all directions due to the thermal expansion. The relevant material and geometric properties are listed in Table 6. This table also lists the "uncertain" parameters that will be described later. For reference, the temperature and displacement profiles are shown in Figure 12 where the uncertain properties are set to some arbitrary values.

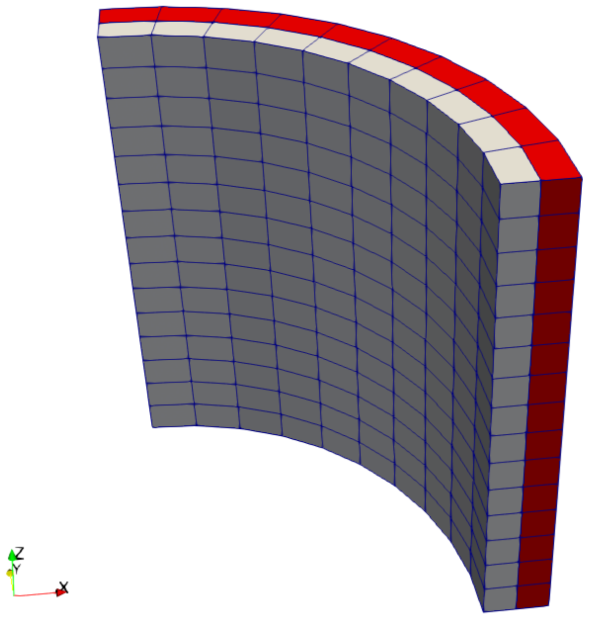
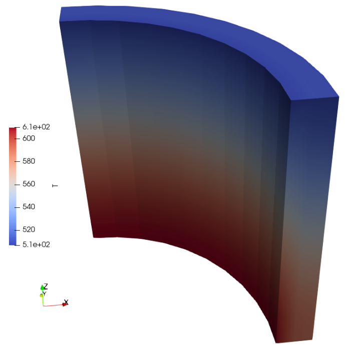


Figure 11: Thermomechanics model problem geometry

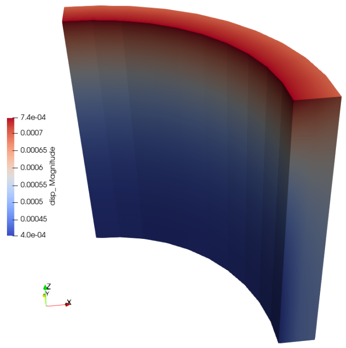
A total of nine properties of the model are not known exactly, but with some known probability of values occurring. The probabilities are represented by each parameter's probability distribution. All parameters have a independent uniform distribution, $\mathcal{U}(a, b)$, where a and b are the lower and upper limit values for the property, respectively. Table 7 lists the details of each of the property's distribution.

Table 6: Material properties for thermomechanics cylinder

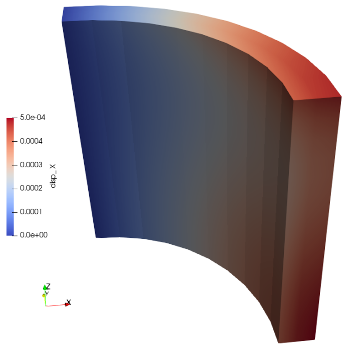
Property	Symbol	Value	Units
Half Cyliner Height	L	3	m
Inner Radius	R	1.0	m
Inner Ring Width	r_1	0.1	m
Outer Ring Width	r_2	0.1	m
Outer Heat Transfer Coef.	h_{outer}	10	W/m^2·K
Outer Free Temperature	$T_{\infty,\text{outer}}$	300	K
End Heat Transfer Coef.	h_{end}	10	W/m^2·K
End Free Temperature	$T_{\infty,\text{end}}$	300	K
Heat Source Magnitude	Q_t	Uncertain	W
Inner Thermal Conductivity	k_1	Uncertain	W/m·K
Outer Thermal Conductivity	k_2	Uncertain	W/m·K
Inner Young's Modulus	Y_1	Uncertain	Pa
Outer Young's Modulus	Y_2	Uncertain	Pa
Inner Poisson's Ratio	ν_1	Uncertain	
Outer Poisson's Ratio	ν_2	Uncertain	
Inner Thermal Expansion Coef.	α_1	Uncertain	1/K
Outer Thermal Expansion Coef.	α_2	Uncertain	1/K
At Rest Temperature	T_0	300	K



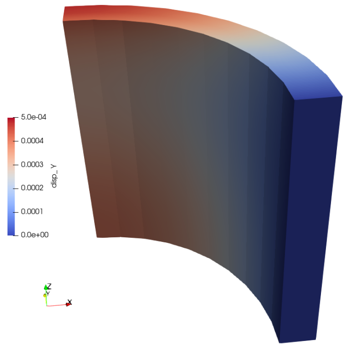
(a) Temperature (K)



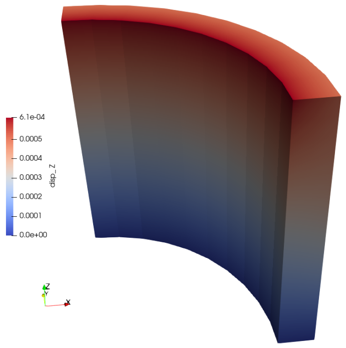
(b) Displacement (m)



(c) X-Displacement (m)



(d) Y-Displacement (m)



(e) Z-Displacement (m)

Figure 12: Example temperature and displacement profiles of thermomechanics problem.

Table 7: Uniform distribution parameters for uncertain properties

Index	Property	a	b
1	k_1	20	30
2	k_2	90	110
3	Q_t	9000	11000
4	α_1	1.0×10^{-6}	3.0×10^{-6}
5	α_2	0.5×10^{-6}	1.5×10^{-6}
6	Y_1	2.0×10^{5}	2.2×10^{5}
7	Y_2	3.0×10^{5}	3.2×10^{5}
8	ν_1	0.29	0.31
9	ν_2	0.19	0.21

There are a total of ten QoIs for the model, which involve temperature and displacement:

1. Temperature at center of inner surface — $T_{1,c}$
2. Temperature at center of outer surface — $T_{2,c}$
3. Temperature at end of inner surface — $T_{1,e}$
4. Temperature at end of outer surface — $T_{2,e}$
5. x-displacement at center of inner surface — $\delta_{x,1,c}$
6. x-displacement at center of outer surface — $\delta_{x,2,c}$
7. x-displacement at end of inner surface — $\delta_{x,1,e}$
8. x-displacement at end of outer surface — $\delta_{x,2,e}$
9. z-displacement at end of inner surface — $\delta_{z,1}$
10. z-displacement at end of outer surface — $\delta_{z,2}$

Figure 13 shows geometrically where these QoIs are located in the model.

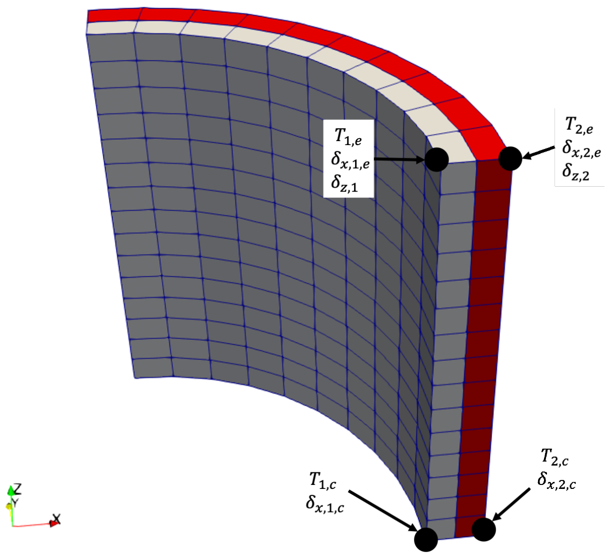


Figure 13: Geometric description of quantities of interest

This example uses the statistics and Sobol sensitivity capabilities available in STM. The goal of this exercise is to understand how the uncertainty in the parameters affects the the resulting QoIs. This is done through sampling the model at different perturbations of the parameters and performing statistical calculations on resulting QoI values. Two methods are used to perform this analysis. First is using the sampler system to perturb the uncertain properties and retrieve the QoIs which will undergo the analysis. The second is training a polynomial chaos surrogate and using that reduced order model to sample and perform the analysis. The idea is that many evaluations of the model are necessary to compute accurate statistical quantities and surrogate modeling speeds up this computation by requiring much fewer full model evaluations for training and is significantly faster to evaluate once trained.

Using Latin hypercube sampling, the thermomechanics model was run with a total of 100,000 samples. A order four polynomial chaos surrogate was training using a Smolyak sparse quadrature for a total of 7,344 runs of the full model. Table 8 shows the run-time for sampling the full order model and training and evaluating the surrogate. We see here that cumulative time for training and evaluating the surrogate is much smaller than just sampling the full order model, this is because building the surrogate required far fewer evaluations of the full model and evaluating the surrogate is much faster than evaluating the full model.

Table 8: Stochastic run-time results for thermomechanics problem

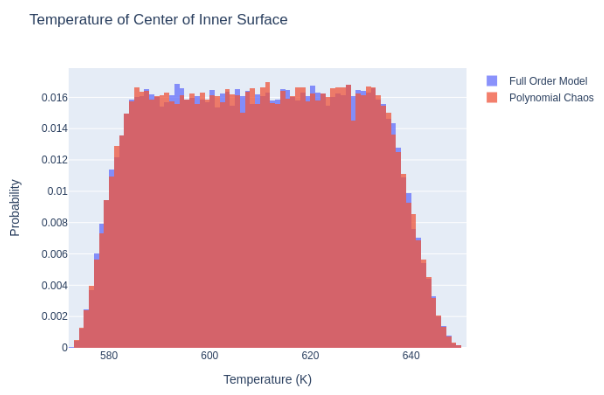
Simulation	Samples	CPU Time
Full-Order Sampling	100,000	176 hr
Polynomial Chaos — Training	7,344	13.7 hr
Polynomial Chaos — Evaluation	100,000	6.8 s

Table 9 shows the statistical results of sampling the thermomechanis model and the polynomial chaos surrogate. μ and σ represent the mean and standard deviation of the QoI, and CI is the confidence interval. Note that the confidence interval for the PC statistics is not relevant since these values were found analytically using integration techniques. Figure 14 compares several of the probability distributions of the QoIs between sampling the FOM and the PC surrogate.

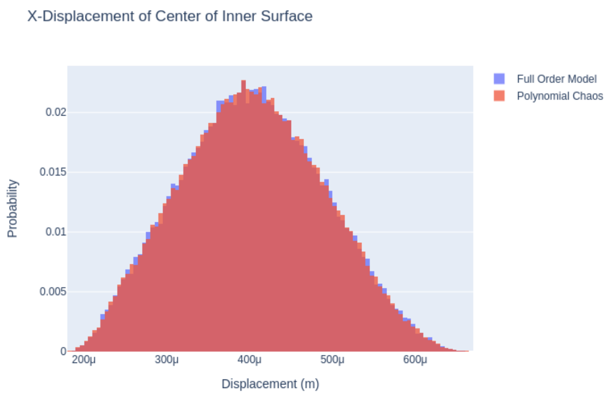
Table 9: Statistics results for themomechanics problem

QoI	μ	95% CI	σ	95% CI	PC – μ	PC – σ
$T_{1,c}$	609.97	(609.87, 610.06)	18.22	(18.18, 18.27)	609.97	18.23
$T_{2,c}$	586.85	(586.77, 586.94)	16.64	(16.60, 16.68)	586.85	16.64
$T_{1,e}$	506.31	(506.25, 506.37)	12.05	(12.03, 12.08)	506.31	12.05
$T_{2,e}$	507.92	(507.85, 507.98)	12.13	(12.10, 12.15)	507.92	12.12
$\delta_{x,1,c}$	4.032E-04	(4.028E-04, 4.037E-04)	8.608E-05	(8.581E-05, 8.636E-05)	4.032E-04	8.625E-05
$\delta_{x,2,c}$	4.996E-04	(4.990E-04, 5.001E-04)	1.078E-04	(1.075E-04, 1.082E-04)	4.996E-04	1.081E-04
$\delta_{x,1,e}$	4.220E-04	(4.213E-04, 4.226E-04)	1.255E-04	(1.252E-04, 1.258E-04)	4.220E-04	1.256E-04
$\delta_{x,2,e}$	4.793E-04	(4.786E-04, 4.800E-04)	1.352E-04	(1.349E-04, 1.356E-04)	4.794E-04	1.354E-04
$\delta_{z,1}$	6.139E-04	(6.131E-04, 6.146E-04)	1.466E-04	(1.462E-04, 1.470E-04)	6.139E-04	1.469E-04
$\delta_{z,2}$	4.995E-04	(4.989E-04, 5.000E-04)	1.067E-04	(1.065E-04, 1.072E-04)	4.995E-04	1.071E-04

Sobol sensitivities, or Sobol indicies, are a metric to compare the global sensitivity a parameter has on a QoI. This examples demonstrates several different types of the sensitivities. The first is total sensitivity, which measure the total sensitivity from a parameter, Figure 15 shows these values for each QoI and parameter. The second is a correlative sensitivity, which measures the sensitivity due to a combination of parameters, Figure 16 show heat maps of these values for several QoIs.



(a) $T_{1,c}$



(b) $\delta_{x,1,c}$

Figure 14: Resulting probability distribution for several QoIs.

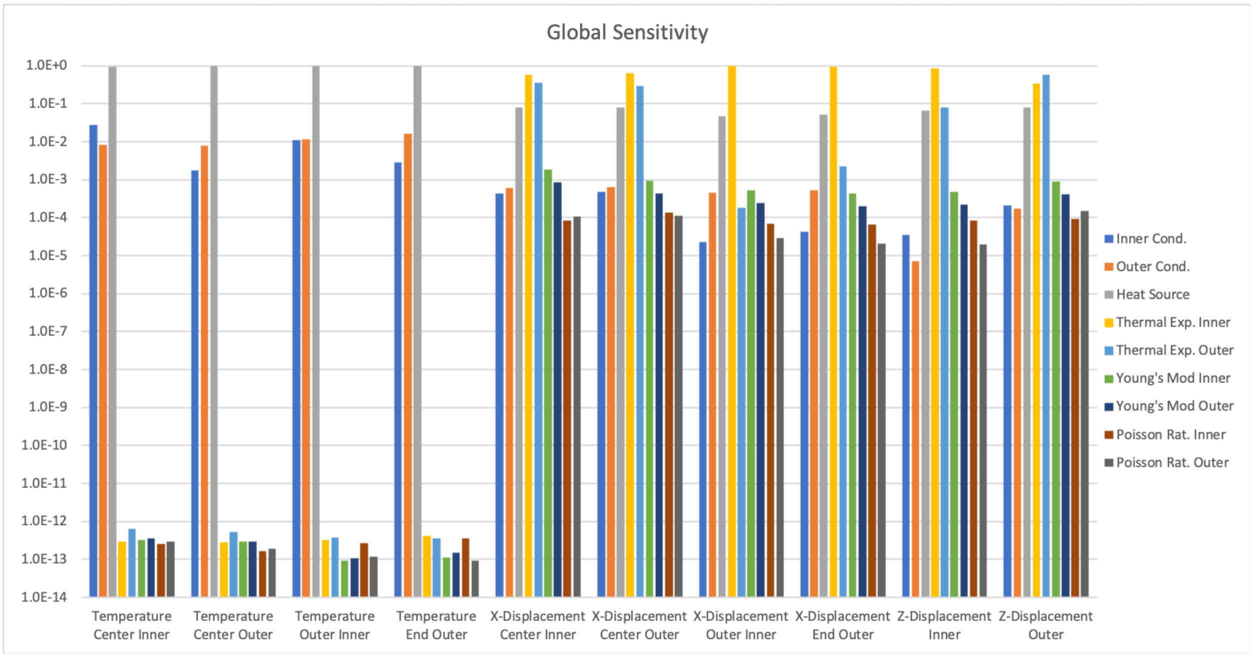
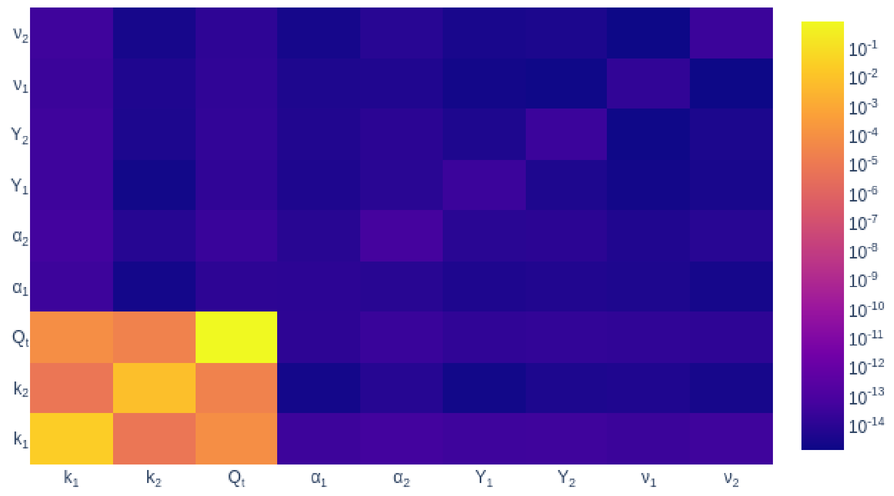
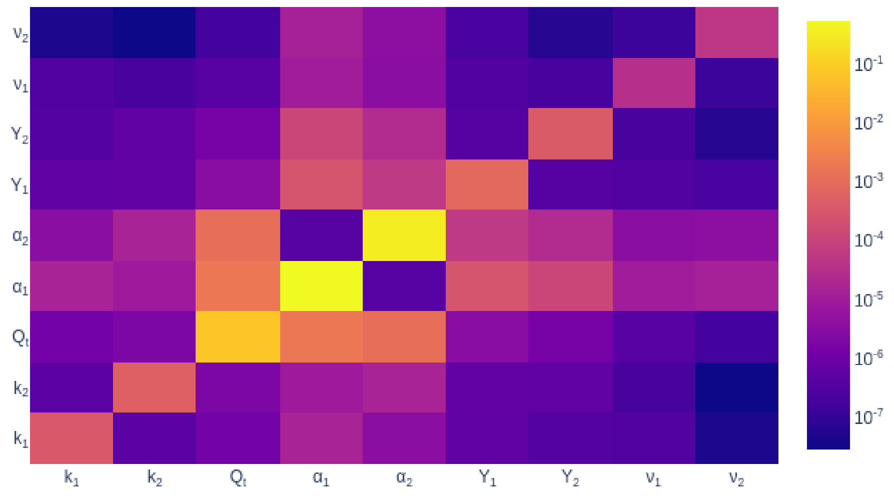


Figure 15: Total Sobol sensitivities



(a) $T_{1,c}$



(b) $\delta_{x,1,c}$

Figure 16: Second order Sobol sensitivities for several QoIs.

3 Inverse Optimization

This section gives an overview of the inverse optimization capabilities developed last year. Section 3.1 describes the theory for PDE constrained constrained optimization. Gradient terms needed for gradient based optimization are derived in Section 3.2 for the adjoint equation and PDE derivatives are given in Section 3.3. Section 3.4 gives an example of PDE constrained optimization applied to the parameterization of boundary conditions and body loads. The over all flow of the optimization algorithm implemented in the MOOSE based app isopod is shown in Figure 17. In this example, the internal heat source, q_v, is being parameterized to match the simulated and experimental steady state temperature fields, $\tilde{T}$ and T, respectively. Step one of the optimization cycle consists of adjusting the internal heat source, q_v. In step two, the physics model is solved with the current q_v to obtain a simulated temperature field, T. In step three, the simulated and experimental temperature fields are compared via the objective function, J. If J is below the user defined threshold, the optimization cycle stops and the best fit parameterization of q_v is found. If J is above the user defined threshold, the optimization algorithm determines a new q_v and the process is repeated. In the next section, methods for determining the next iteration of the parameterized value, in this case q_v, will be presented.

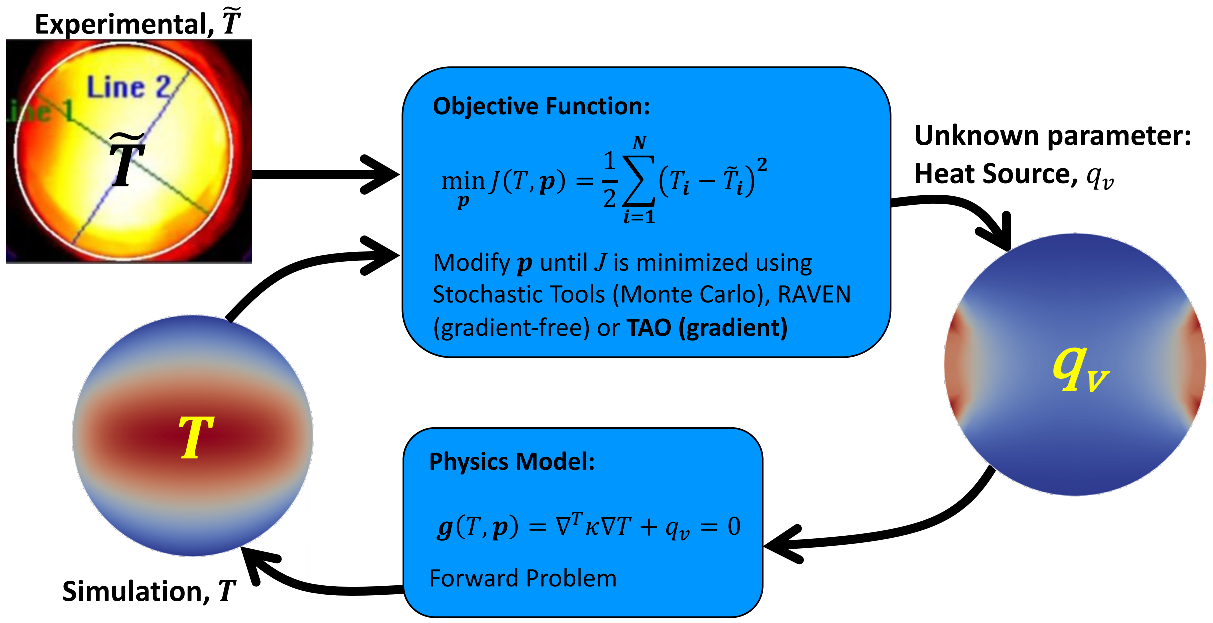

Experimental, $\tilde{T}$
Line 2
ine 1
$\tilde{T}$
Objective Function:
$\min_{\boldsymbol{p}} J(T,\boldsymbol{p}) = \frac{1}{2}\sum_{i=1}^{N}(T_i - \tilde{T}_i)^2$
Modify $\boldsymbol{p}$ until J is minimized using Stochastic Tools (Monte Carlo), RAVEN (gradient-free) or TAO (gradient)
Unknown parameter: Heat Source, q_v
q_v
T
Physics Model:
$\boldsymbol{g}(T,\boldsymbol{p}) = \nabla^T \kappa \nabla T + q_v = 0$
Forward Problem
Simulation, T

Figure 17: Optimization cycle example for parameterizaing an internal heat source distribution q_v to match the simulated and experimental temperature field, T and $\tilde{T}$, respectively.

3.1 PDE Constrained Inverse Optimization

Inverse optimization is a mathematical framework to infer model parameters by minimizing the misfit between the experimental and simulation observables. In this work, our model is a PDE describing the physics of the experiment. We solve our physics PDE using the finite element method as implemented in MOOSE. The physics of our problem constrains our optimization algorithm. A PDE-constrained inverse optimization framework is formulated as an abstract optimization problem

[11]:

$$\min_{\boldsymbol{p}} J\left(\boldsymbol{u},\boldsymbol{p}\right); \quad \text{subject to } \boldsymbol{g}\left(\boldsymbol{u},\boldsymbol{p}\right)=\boldsymbol{0}, \tag{66}$$

where $J(\boldsymbol{u},\boldsymbol{p})$ is our objective function, which is a scalar measure of the misfit between experimental and simulated responses, along with any regularization [12]. The constraint, $\boldsymbol{g}\left(\boldsymbol{u},\boldsymbol{p}\right)=\boldsymbol{0}$, consists of the PDEs governing the multiphysics phenomena simulated by MOOSE (e.g. coupled heat and elasticity equations), $\boldsymbol{p}$ contains model parameters (e.g. material properties or loads) and $\boldsymbol{u}$ contains simulated responses (e.g. temperature and displacement fields). The equations in Eq. (66) appear simple on the outset but are extremely difficult to solve. The solution space can span millions of degrees of freedom and the parameter space can also be very large. Finally, the PDEs can be highly nonlinear, time-dependent and tightly coupling complex phenomena across multiple physics.

Optimization problems can be solved using either global (gradient-free) or local (gradient-based) approaches [13]. Global approaches require a large number of iterations compared to gradient-based approaches (e.g. conjugate gradient or Newton-type methods), making the latter more suitable to problems with a large parameter space and computationally expensive models. The PETSc TAO optimization library [14] is used to solve Eq. (66). Optimization libraries like TAO require access to functions for computing the objective (J), gradient ($\mathrm{d}J/\mathrm{d}\boldsymbol{p}$) and Hessian $\left(\mathrm{d}^2J/\mathrm{d}\boldsymbol{p}^2\right)$ or a function to take the action of the Hessian on a vector. An objective function measuring the misfit or distance between the simulation and experimental data usually has the form

$$J(\boldsymbol{u},\boldsymbol{p})=\frac{1}{2}\sum_i\left(\boldsymbol{u}_i-\bar{\boldsymbol{u}}_i\right)^2+\frac{\rho}{2}\sum_i\boldsymbol{p}^2, \tag{67}$$

where the first integral is an L_2 norm or euclidean distance between the experimental solution, $\bar{\boldsymbol{u}}$, and the simulated solution, $\boldsymbol{u}$. The second integral provides Tikhonov regularization on the parameters, $\boldsymbol{p}$, for ill-posed problems where ρ controls the amount of regularization. Other types of regularization may also be used.

Gradient-free optimization solvers only require a function to solve for the objective given in Eq. (67). Solving for the objective only requires solving a forward problem to determine $\boldsymbol{u}$ and then plugging that into Eq. (67) to determine J. The forward problem is defined as the FEM model of the experiment which the analyst should have already made before attempting to perform optimization. The parameters that go into the forward problem (e.g. pressure distributions on sidesets or material properties) are adjusted by the optimization solver and the forward problem is recomputed. This process continues until J is below some user defined threshold. The basic gradient-free solver available in TAO is the simplex or Nelder-Mead method. Gradient-free optimization solvers are robust and straight-forward to use. Unfortunately, their computational cost scales exponentially with the number of parameters. When the forward model is a computationally expensive FEM model, gradient-free approaches quickly become computationally expensive.

Gradient-based optimization algorithms require fewer iterations but require functions to solve for the gradient vector and sometimes Hessians matrix. TAO has `petsc_options` to evaluate finite difference based gradients and Hessians by solving the objective function multiple times with perturbed parameters, which also requires multiple solves of the forward problem. Finite difference gradients and Hessians are good for testing an optimization approach but become computationally expensive for realistic problems.

Given the large parameter space, we resort to the adjoint method for gradient computation; unlike finite difference approaches, the computational cost of adjoint methods is independent of the number of parameters (ref18). In the adjoint method, the gradient, i.e. the total derivative $\mathrm{d}J/\mathrm{d}\boldsymbol{p}$, is computed as,

$$\frac{\mathrm{d}J}{\mathrm{d}\boldsymbol{p}}=\frac{\partial J}{\partial\boldsymbol{p}}+\boldsymbol{\lambda}\frac{\partial\boldsymbol{g}}{\partial\boldsymbol{p}}, \tag{68}$$

where $\partial J/\partial \boldsymbol{p}$ accounts for the regularization in Eq. (67) and $\boldsymbol{\lambda}$ is the adjoint variable solved for from the adjoint equation

$$\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{\top} \boldsymbol{\lambda} = \left(\frac{\partial J}{\partial \boldsymbol{u}}\right)^{\top}, \tag{69}$$

where $(\partial \boldsymbol{g}/\partial \boldsymbol{u})^{\top}$ is the adjoint of the Jacobian for the original forward problem, $\boldsymbol{g}$, and $(\partial J/\partial \boldsymbol{u})^{\top}$ is a body force like term that accounts for the misfit between the computed and experimental data. Thus, the solution to Eq. (69) has the same complexity as the solution to the forward problem.

The remaining step for evaluating the derivative of the PDE in Eq. (68) is to compute $\partial \boldsymbol{g}/\partial \boldsymbol{p}$, the derivative of the PDE with respect to the parameter vector. The form this term takes is dependent on the physics (e.g. mechanics or heat conduction) and the parameter being optimized (e.g. force inversion versus material inversion). In what follows, we will derive the adjoint equation for steady state heat conduction and the gradient term for both force and material inversion.

3.2 Adjoint Problem for Steady State Heat Conduction

In this section, we are interested in solving the following PDE-constrained optimization problem from Eq. (66) for steady state heat conduction:

$$\begin{aligned} \min_{\boldsymbol{p}} \qquad & J\left(T, \boldsymbol{p}\right) = \frac{1}{2} \sum_{i=1}^{N} \left(T_i - \bar{T}_i\right)^2, \\ \text{subject to} \quad & g\left(T, \boldsymbol{p}\right) = \nabla \cdot \kappa \nabla T + f_b = 0, \quad \text{in } \Omega, \end{aligned} \tag{70}$$

where J is the objective function from Eq. (67) without regularization, f_b is the distributed heat flux, T is the experimental temperature field being compared to our simulation temperature at discrete locations, $\bar{T}_i$. Other forms for the objective function are possible such as different norms or different types of measurements that may require integration over a line or volume.

We also have the following boundary conditions for our PDE,

$$\begin{aligned} T &= T_D, & &\text{on } \Gamma_D, \\ (\kappa \nabla T) \cdot \boldsymbol{n} &= G\left(T\right), & &\text{on } \Gamma_R, \end{aligned} \tag{71}$$

where $\boldsymbol{n}$ is the normal vector, Γ_D is the Dirichlet boundary, and Γ_R is the Robin or mixed boundary. Common cases for $G\left(T\right)$ are:

$$\begin{aligned} &\text{Neumann:} \quad G\left(T\right) = G = f_n, \\ &\text{Convection:} \quad G\left(T\right) = h(T - T_\infty), \end{aligned} \tag{72}$$

where h is the heat transfer coefficient and f_n is independent of T. The objective function can also be expressed in a integral form for the case of N point measurements as follows

$$J = \frac{1}{2} \sum_{i=1}^{N} \int \delta(x - x_i) \left(T - \bar{T}_i\right)^2 \mathrm{d}\Omega. \tag{73}$$

We take the equivalent, variational approach to derive the adjoint. Thus, the Lagrangian of this problem is

$$\begin{aligned} \mathcal{L}(T, \boldsymbol{p}, \lambda) &= J + \int \left(\nabla \cdot \kappa \nabla T + f_b\right) \lambda \ \mathrm{d}\Omega \\ &= J + \int \left(f_b \lambda\right) \ \mathrm{d}\Omega + \int \left(\nabla \cdot \kappa \nabla T\right) \lambda \ \mathrm{d}\Omega. \end{aligned} \tag{74}$$

The divergence theorem is applied to the last term in Eq. (74) giving

$$\begin{aligned}\int (\nabla\cdot\kappa\nabla T)\,\lambda\ \mathrm{d}\Omega &= \int \lambda\,(\kappa\nabla T)\cdot\boldsymbol{n}\ \mathrm{d}\Gamma - \int (\kappa\nabla\lambda)\cdot\nabla T\ \mathrm{d}\Omega \\ &= \int \left[\lambda\,(\kappa\nabla T)\cdot\boldsymbol{n} - T\,(\kappa\nabla\lambda)\cdot\boldsymbol{n}\right]\ \mathrm{d}\Gamma + \int (\nabla\cdot\kappa\nabla\lambda)\,T\ \mathrm{d}\Omega.\end{aligned} \tag{75}$$

By substituting the above and Eq. (73) into Eq. (74), we have

$$\begin{aligned}\mathcal{L}(T,\boldsymbol{p},\lambda) =& \mathcal{A}(T,\boldsymbol{p},\lambda) + \mathcal{B}(T,\boldsymbol{p},\lambda),\\ \mathcal{A}(T,\boldsymbol{p},\lambda) =& \frac{1}{2}\sum_{i=1}^{N}\int \delta(x-x_i)\left(T-\bar{T}\right)^2\mathrm{d}\Omega + \int (f_b\lambda)\ \mathrm{d}\Omega + \int (\nabla\cdot\kappa\nabla\lambda)\,T\ \mathrm{d}\Omega,\\ \mathcal{B}(T,\boldsymbol{p},\lambda) =& \int \left[\lambda\,(\kappa\nabla T)\cdot\boldsymbol{n} - T\,(\kappa\nabla\lambda)\cdot\boldsymbol{n}\right]\ \mathrm{d}\Gamma,\end{aligned} \tag{76}$$

where λ is the Lagrange multiplier field known as the adjoint state or costate variable. The Lagrangian has been broken up into terms integrated over the body, $\mathcal{A}$, and boundary terms, $\mathcal{B}$. In order to determine the boundary conditions for the adjoint equation, the boundary integral terms, $\mathcal{B}$, in Eq. (76) are further broken up into their separate domains, Γ_D and Γ_R, given in Eq. (71) resulting in

$$\begin{aligned}\mathcal{B} &= \int \left[\lambda\,(\kappa\nabla T)\cdot\boldsymbol{n} - T\,(\kappa\nabla\lambda)\cdot\boldsymbol{n}\right]\ \mathrm{d}\Gamma \\ &= \int_{\Gamma_R} \lambda G(T)\ \mathrm{d}\Gamma + \int_{\Gamma_D} \lambda\kappa\nabla T\cdot\boldsymbol{n}\ \mathrm{d}\Gamma - \int_{\Gamma_R} T\kappa\nabla\lambda\cdot\boldsymbol{n}\ \mathrm{d}\Gamma - \int_{\Gamma_D} T_o\kappa\nabla\lambda\cdot\boldsymbol{n}\ \mathrm{d}\Gamma,\end{aligned} \tag{77}$$

where T_o is the prescribed temperature on the Dirichlet boundary, Γ_D. The variation of $\mathcal{L}$ with respect to T is then given by $\delta\mathcal{L} = \delta\mathcal{A} + \delta\mathcal{B}$ where the variation of the body terms with respect to T are given by

$$\begin{aligned}\delta\mathcal{A} =& \sum_{i=1}^{N}\int \delta(x-x_i)\left(T-\bar{T}\right)\delta T\mathrm{d}\Omega + \int (\nabla\cdot\kappa\nabla\lambda)\,\delta T\ \mathrm{d}\Omega \\ =& \int \left((\nabla\cdot\kappa\nabla\lambda) + \sum_{i=1}^{N}\delta(x-x_i)\left(T-\bar{T}\right)\right)\delta T\ \mathrm{d}\Omega,\end{aligned} \tag{78}$$

and the variation of $\mathcal{B}$ with respect to T is given as

$$\begin{aligned}\delta\mathcal{B} &= \int_{\Gamma_R} \lambda\frac{\mathrm{d}G(T)}{\mathrm{d}T}\delta T\mathrm{d}\Gamma - \int_{\Gamma_R} (\kappa\nabla\lambda\cdot\boldsymbol{n})\,\delta T\mathrm{d}\Gamma + \int_{\Gamma_D} \lambda\,(\kappa\nabla\delta T\cdot\boldsymbol{n})\ \mathrm{d}\Gamma \\ &= \int_{\Gamma_R} (\lambda h - (\kappa\nabla\lambda\cdot\boldsymbol{n}))\,\delta T\mathrm{d}\Gamma + \int_{\Gamma_D} \lambda\,(\kappa\nabla\delta T\cdot\boldsymbol{n})\ \mathrm{d}\Gamma,\end{aligned} \tag{79}$$

where $\mathrm{d}G(T)/\mathrm{d}T = h$ from Eq. (71) was used. Combining Eq. (78) and Eq. (79) to get $\delta\mathcal{L}$ results in

$$\begin{aligned}\delta\mathcal{L} =& \delta\mathcal{A} + \delta\mathcal{B} \\ =& \int \left((\nabla\cdot\kappa\nabla\lambda) + \sum_{i=1}^{N}\delta(x-x_i)\left(T-\bar{T}\right)\right)\delta T\ \mathrm{d}\Omega \\ &+ \int_{\Gamma_R} (\lambda h - (\kappa\nabla\lambda\cdot\boldsymbol{n}))\,\delta T\mathrm{d}\Gamma + \int_{\Gamma_D} \lambda\,(\kappa\nabla\delta T\cdot\boldsymbol{n})\ \mathrm{d}\Gamma.\end{aligned} \tag{80}$$

Stationarity of $\mathcal{L}$ would require $\delta\mathcal{L} = 0$ for all admissible δT. Setting each of the integrals in Eq. (80) results in the adjoint problem and its boundary conditions

$$\boxed{\begin{aligned} \nabla \cdot \kappa \nabla \lambda + \sum_{i=1}^{N} \delta(x - x_i)(T - \bar{T}) = 0, \text{ in } \Omega, \\ \lambda = 0, \text{ on } \Gamma_D, \\ \kappa \nabla \lambda \cdot \boldsymbol{n} = \lambda h, \text{ on } \Gamma_R. \end{aligned}} \tag{81}$$

Solving Eq. (81) comes down to adjusting the boundary conditions and load vector from the forward problem and re-solving.

3.3 PDE Derivatives for Inversion

In this section we will present derivatives for steady state heat conduction Eq. (70) with respect to the force or material parameters. For all of these examples, measurement data is taken at specific locations where the objective function can be represented by Eq. (73). We will present the discrete forms of our PDE and its derivative which most closely matches the implementation that will be used in MOOSE. The discrete form of the PDE constraint for steady state heat conduction in Eq. (70), $\hat{g}$, is given as

$$\hat{\mathrm{g}} = \mathbf{K}\hat{\mathrm{T}} - \hat{\mathrm{f}} = 0, \tag{82}$$

where $\mathbf{K}$ is the Jacobian matrix, $\hat{\mathrm{T}}$ and $\hat{\mathrm{f}}$ are the discretized temperature and residual vectors. Element-wise definitions of the terms are

$$\begin{aligned} \mathbf{K}^{\alpha\beta} &= \sum_e \int \nabla^\top \mathrm{N}^\alpha \cdot \kappa \cdot \nabla \mathrm{N}^\beta \, \mathrm{d}\Omega, \\ \hat{\mathrm{f}}^\alpha &= \sum_e \int N^\alpha f_b \mathrm{d}\Omega + \sum_e \int N^\alpha G(T) \, \mathrm{d}\Gamma_R, \end{aligned} \tag{83}$$

where N^α denotes the finite element shape function at node α. The solution can be expressed as $T(x) \approx \sum_\alpha \mathrm{N}^\alpha \hat{\mathrm{T}}^\alpha = N\hat{\mathrm{T}}$. Note here the $\hat{\mathrm{f}}$ includes the contribution from both the body load term (f_b in Eq. (70)) and boundary conditions (see Eq. (71)). We are assuming a Galerkin form for our discretized PDE by making our test and trial functions identical (both are N^α). Note in the rest of this document, we omit the superscript of the shape function (i.e., $(\cdot)^\alpha$) and the summation over all the elements (i.e., $\sum_e$) for simplicity.

To compute the derivative of PDE with respect to the design parameters, i.e., $\partial g / \partial \boldsymbol{p}$, it can be seen from Eq. (83) that the derivatives can come from either $\mathbf{K}$ or $\hat{\mathrm{f}}$. For problems that have design parameters that are embedded in f_b and/or the Neumann boundary condition in $G(T)$, we call them **force inversion** problems, since the derivative only depends on the body load. For design parameters that are embedded in $\mathbf{K}$ and/or the convection boundary condition in $G(T)$, indicating dependence on the material property, we call them **material inversion** problems. Derivative calculation of different force inversion and material inversion problems are included in Subsections 3.3.1 - 3.3.3.

3.3.1 PDE Derivatives for Force Inversion

We will first consider the simple case of load parameterization for body loads, f_b (in Eq. (70)), where the gradient is given as

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} &= \frac{\partial \hat{g}}{\partial \hat{f}}\frac{\partial \hat{f}}{\partial f_b}\frac{\partial f_b}{\partial \boldsymbol{p}} \\ &= \frac{\partial \hat{f}}{\partial f_b}\frac{\partial f_b}{\partial \boldsymbol{p}} \\ &= \int \mathrm{N}\cdot\frac{\partial f_b(x)}{\partial \boldsymbol{p}}\ \mathrm{d}\Omega,\end{aligned} \tag{84}$$

by taking the chain rule from Eq. (82) and Eq. (83). The gradient term requires the derivative of f_b to be integrated over the volume Ω.

For the first case, we will consider a volumetric heat source that varies linearly across the body, which is given by

$$f_b(x) = p_0 + p_1 x, \tag{85}$$

with p_0 and p_1 the design parameters. Therefore, the derivative can be calculated as

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} &= \int \mathrm{N}\cdot\frac{\partial f_b(x)}{\partial \boldsymbol{p}}\ \mathrm{d}\Omega \\ &= \left[\int \mathrm{N}\cdot\frac{\partial f_b}{\partial p_0}\ \mathrm{d}\Omega, \int \mathrm{N}\cdot\frac{\partial f_b}{\partial p_1}\ \mathrm{d}\Omega\right]^\top \\ &= \left[\int \mathrm{N}\cdot(1)\ \mathrm{d}\Omega, \int \mathrm{N}\cdot(x)\ \mathrm{d}\Omega\right]^\top.\end{aligned} \tag{86}$$

In the next force inversion case, we parameterize the intensity of np point heat sources, where the heat source takes the form

$$f_b(x) = \sum_{i=1}^{np} \delta\left(x - x_i\right) p_i \quad \text{for } i = 1\ldots np. \tag{87}$$

The corresponding gradient term is given by

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial p_i} &= \int \mathrm{N}\cdot\frac{\partial f_b(x)}{\partial p_i}\ \mathrm{d}\Omega \\ &= \int \mathrm{N}\cdot\delta\left(x - x_i\right) d\Omega \quad \text{for } i = 1\ldots np,\end{aligned} \tag{88}$$

which makes the gradient equal to one at the locations of the point loads.

Next, we will use force inversion to parameterize a Neumann boundary condition with the heat flux on the boundary being a function of the coordinates, $G(x)$, given by

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} &= \frac{\partial \hat{g}}{\partial \hat{f}}\frac{\partial \hat{f}}{\partial G(x)}\frac{\partial G(x)}{\partial \boldsymbol{p}} \\ &= \frac{\partial \hat{f}}{\partial G(x)}\frac{\partial G(x)}{\partial \boldsymbol{p}} \\ &= \int_{\Gamma_R} \mathrm{N}\cdot\frac{\partial G(x)}{\partial \boldsymbol{p}}\ \mathrm{d}\Gamma,\end{aligned} \tag{89}$$

where the derivative of $G(x)$ is now integrated over the boundary Γ_R. For instance, if we have a linearly varying heat flux

$$G(x) = p_0 + p_1 x, \tag{90}$$

with p_0 and p_1 the design parameters, then

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} &= \left[\frac{\partial \hat{\mathrm{g}}}{\partial p_0}, \frac{\partial \hat{\mathrm{g}}}{\partial p_1}\right]^\top \\ &= \left[\int_{\Gamma_R} \mathrm{N} \cdot \frac{\partial G(x)}{\partial p_0}\,\mathrm{d}\Gamma, \int_{\Gamma_R} \mathrm{N} \cdot \frac{\partial G(x)}{\partial p_1}\,\mathrm{d}\Gamma\right]^\top \\ &= \left[\int_{\Gamma_R} \mathrm{N}\,\mathrm{d}\Gamma, \int_{\Gamma_R} \mathrm{N}x\,\mathrm{d}\Gamma\right]^\top .\end{aligned} \tag{91}$$

The above force inversion examples, Eqs. (84)-(91), are all linear optimization problems where the parameter being optimized does not show up in the derivative term. Linear optimization problems are not overly sensitive to the location of measurement points or the initial guesses for the parameter being optimized, making them easy to solve. In the following we parameterize a Gaussian body force given by

$$f_b(x) = a \cdot \exp\left(-\frac{(x-b)^2}{2c^2}\right), \tag{92}$$

where a is the height or intensity of the Gaussian curve, b is the location of the peak of the curve, and c is the standard deviation of the curve or its width. Parameterizing a Gaussian curve can result in a linear or nonlinear optimization problem depending on which parameter is being optimized. Parameterizing this function for the height, a, results in the following linear optimization problem derivative

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} = \frac{\partial \hat{\mathrm{g}}}{\partial a} &= \int \mathrm{N} \cdot \frac{\partial f_b(x)}{\partial a}\,\mathrm{d}\Omega \\ &= \int \mathrm{N} \cdot \exp\left(-\frac{(x-b)^2}{2c^2}\right)\,\mathrm{d}\Omega,\end{aligned} \tag{93}$$

where the parameter a being optimized does not show up in the derivative term. However, if we try to parameterize for the location of the peak of the Gaussian curve, b, we get the following nonlinear optimization problem derivative

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} = \frac{\partial \hat{\mathrm{g}}}{\partial b} &= \int \mathrm{N} \cdot \frac{\partial f_b(x)}{\partial b}\,\mathrm{d}\Omega \\ &= \int \mathrm{N} \cdot \frac{a(x-b)}{c^2} \exp\left(-\frac{(x-b)^2}{2c^2}\right)\,\mathrm{d}\Omega,\end{aligned} \tag{94}$$

where the parameter b remains in the derivative term. Optimizing for the peak location of a Gaussian heat source, Eq. (94), will be a much more difficult problem to solve than Eq. (93) and convergence will be dependent on the initial guesses given for b and the locations where measurement data is taken.

3.3.2 PDE Derivative for Convective Boundary Conditions

In this section, the convective heat transfer coefficient, h, is considered as a parameter for the convection boundary condition given in Eq. (72). This is a material inversion problem with integral limited to the boundary, Γ_R. The boundary condition is given by $G(T) = h\left(T - T_\infty\right)$ on Γ_R. The PDE derivative term is given by

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} = \frac{\partial \hat{\mathrm{g}}}{\partial h} &= \int_{\Gamma_R} \mathrm{N} \cdot \frac{\partial G(x)}{\partial h}\ \mathrm{d}\Gamma \\ &= \int_{\Gamma_R} \mathrm{N} \cdot \left(T - T_\infty\right)\ \mathrm{d}\Gamma.\end{aligned} \tag{95}$$

This derivative requires the solution from the forward problem T to be included in the integral over the boundary Γ_R which again results in a nonlinear optimization problem.

3.3.3 PDE Derivative for Material Inversion

In Sections 3.3.1 and 3.3.2, the design parameters only exist in the load terms. Therefore, the derivatives of $g(x)$ with respect to the design parameters are only related to the form of the load function, not the Jacobian. In this section, we consider cases where the design parameters exist in the Jacobian term, which make the derivative calculation more convoluted. One such example is the material inversion, where we identify the thermal conductivity (κ) through experimentally measured temperature data points. Here, the derivative of $g(x)$ is taken with respect to κ. This requires the derivative of $\mathbf{K}$ in Eq. (82) leading to

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} &= \int \nabla \mathrm{N}^\top \cdot \frac{\partial \kappa}{\partial \boldsymbol{p}} \cdot \nabla \mathrm{N}\ \mathrm{d}\Omega \cdot \hat{\mathrm{T}} \\ &= \int \nabla \mathrm{N}^\top \cdot \frac{\partial \kappa}{\partial \boldsymbol{p}} \cdot \nabla T\ \mathrm{d}\Omega,\end{aligned} \tag{96}$$

where $\nabla \mathrm{N}\hat{T} = \nabla T$ was used in the last line. Material inversion is also a nonlinear optimization problem since T shows up in the derivative, making the derivative dependent on the solution to the forward problem.

This also works for temperature dependent thermal conductivity

$$\kappa\left(T\right) = \alpha T^2 + \beta T, \tag{97}$$

where α and β are the design parameters. The resultant derivative is then

$$\begin{aligned}\frac{\partial \hat{g}}{\partial \boldsymbol{p}} = \left[\frac{\partial \hat{g}}{\partial \alpha}, \frac{\partial \hat{g}}{\partial \beta}\right]^\top &= \left[\int \nabla \mathrm{N}^\top \cdot \frac{\partial \kappa}{\partial \alpha} \cdot \nabla T\ \mathrm{d}\Omega, \int \nabla \mathrm{N}^\top \cdot \frac{\partial \kappa}{\partial \beta} \cdot \nabla T\ \mathrm{d}\Omega\right]^\top \\ &= \left[\int \nabla \mathrm{N}^\top \cdot \left(T^2\right) \cdot \nabla T\ \mathrm{d}\Omega, \int \nabla \mathrm{N}^\top \cdot \left(T\right) \cdot \nabla T\ \mathrm{d}\Omega\right]^\top.\end{aligned} \tag{98}$$

3.4 Force Inversion Example

The framework for solving PDE constrained optimization problems presented in Sections 3.1-3.3 has been implemented in the MOOSE based app isopod for the steady state heat equation. Isopod is a research application containing the classes needed to perform optimization. Most of these optimization classes will be merged into STM. In this section, isopod examples will be given for the parameterization of boundary conditions and body forces given by Eqs. (84) and (89).

3.4.1 Neumann Boundary Condition Force Inversion Example

In the first example, a Neumann boundary condition is optimized on the left side of a rectangular domain, shown in Figure 18, in order to match four temperature measurements taken at the locations marked by the $\times$ symbols. The temperature measurements are taken from synthetic data created by solving the forward problem with $a = 113$ and $b = 288$. The Neumann BC in this problem is a combination of the two distributed loads shown in the figure given by the functions

$$\begin{aligned} \nabla T \cdot n &= a, \\ \nabla T \cdot n &= by, \end{aligned} \tag{99}$$

where a and b are the parameters being optimized and y is the y-coordinate location. The other boundary conditions on the top, bottom and right sides are fixed during the optimization.

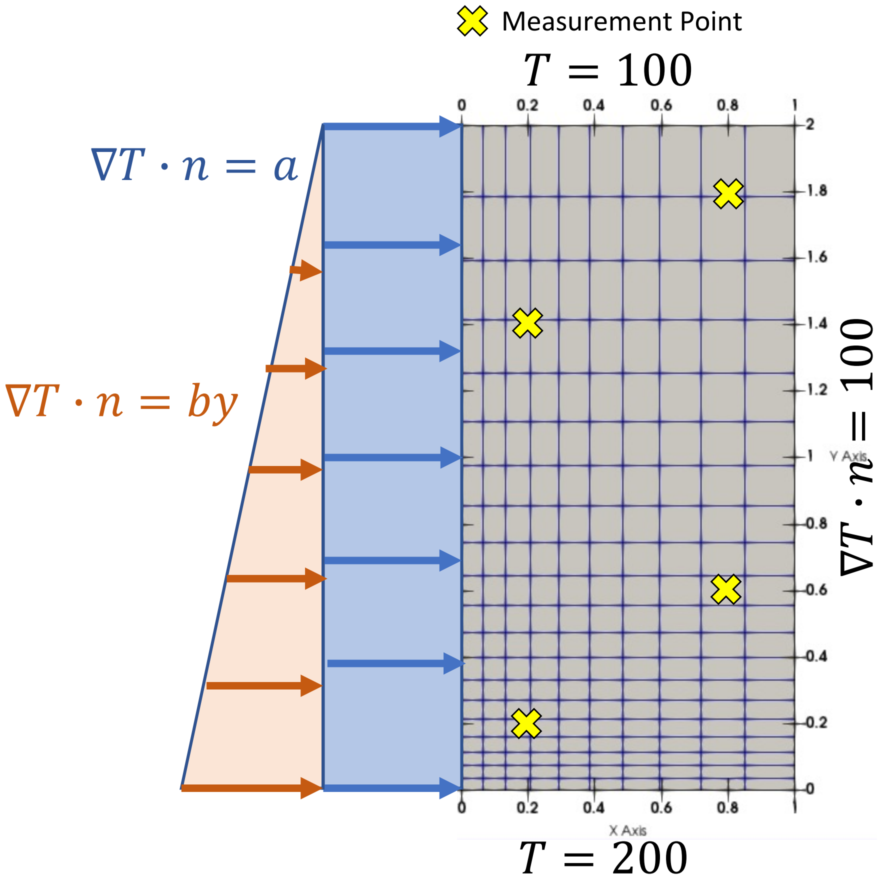


Figure 18: Force inversion setup for the parameterization of Neumann BCs on the rectangular domain shown in grey. The finite element mesh used in this example is shown by the blue lines. In this example the loading on the left boundary is described by the superposition of a constant and linearly increasing load parameterized by a and b, indicated by the orange and blue arrows. These parameters are being optimized to match the temperature field at the measurement points shown by the $\times$'s in the body. A known constant heat flux is applied to the right boundary and the top and bottom boundaries are at a fixed temperature.

Gradient and non-gradient based optimization algorithms from TAO are used to parameterize a and b. For non-gradient optimization, the Nelder-Mead (`taonm`) algorithm is used which only requires the solution of the forward problem. Gradient based optimization using the conjugate gradient (`taocg`) method requires the solution of both the forward problem and adjoint problem given by Eq. (81) along with determination of the PDE derivative. For this problem the PDE derivative is given by Eq. (91). The Limited Memory Variable Metric method (`taolmvm`) is also used and is a quasi-Newton optimization solver that uses the gradient to form an approximate Hessian. Isopod currently does not have the ability to construct Hessians which excludes the use of full Newton optimization solvers.

Isopod uses the MOOSE multiapp system for optimization where the forward and adjoint solves are done on separate sub-apps. The main-app contains the optimization executioner and a reporter is used to communicate data between the optimization solver and the transfers that send data down to the sub-apps. Once the sub-app completes its solve, the data is transferred back to the optimization executioner on the main app where the objective is evaluated and a new parameters are determined for the next solve. This process is repeated until the either the objective or its gradient driven below some threshold.

The adjoint and forward solution fields for the gradient based optimization algorithms are shown in Figure 19. The adjoint variable is shown for the first optimization iteration where the misfit between the measured temperature and forward solve temperatures is the largest. This misfit is applied in the adjoint solve using the `DiracKernels`. The contributions for the gradient in Eq. (68) are computed in the adjoint sub-app using a modified `FunctionSideIntegral` postprocessor that will integrate the adjoint variable together with the PDE derivative from Eq. (91) along the boundary. The gradient terms are passed back to the optimization algorithm on the main app.

A threshold of 1e-4 is set for the absolute value of the gradient terms for the `taolmvm` solver which results in an objective value of $J = 0.05$. The final temperature field is shown in the right figure of Figure 19. A `LineValueSampler` is taken along the pink lines in Figure 19 and is plotted in Figure 20 for each `taolmvm` iteration. The solution quickly converges after a single optimization iteration.

The `taonm`, `taocg` and `taolmvm` optimization convergence results are shown in Figure 21 with the threshold $J = 0.05$. `taolmvm` is by far the best performing algorithm for force inversion, reaching an optimal solution in 4 optimization iterations. The `taonm` and `taocg` algorithms find an optimal solution for a and b in a similar number of optimization iterations. However, since `taocg` requires an additional adjoint solve to compute the gradient, `taonm` requires 4x fewer total finite element solves to reach an optimial solve. Default parameters were used for with `taocg` and better performance could probably be achieved with a better understanding of the TAO conjugate gradient implementation and line search.

3.4.2 Body Load Force Inversion Example

In the second example, the quadratically varying body load shown on the left of Figure 22 will be parameterized to optimally fit the temperature measurement data points shown on the right side of Figure 22. The quadratically varying temperature field and its PDE derivative from Eq. (84) is given by

$$f_b(x) = ax^2 + bx + c, \tag{100}$$

$$\begin{aligned}\frac{\partial \hat{\mathrm{g}}}{\partial \boldsymbol{p}} &= \left[\int \mathrm{N} \cdot \frac{\partial f_b}{\partial a}\ \mathrm{d}\Omega, \int \mathrm{N} \cdot \frac{\partial f_b}{\partial b}\ \mathrm{d}\Omega, \int \mathrm{N} \cdot \frac{\partial f_b}{\partial c}\ \mathrm{d}\Omega\right]^\top \\ &= \left[\int \mathrm{N} \cdot \left(x^2\right)\ \mathrm{d}\Omega, \int \mathrm{N} \cdot (x)\ \mathrm{d}\Omega, \int \mathrm{N} \cdot (1)\ \mathrm{d}\Omega\right]^\top,\end{aligned} \tag{101}$$

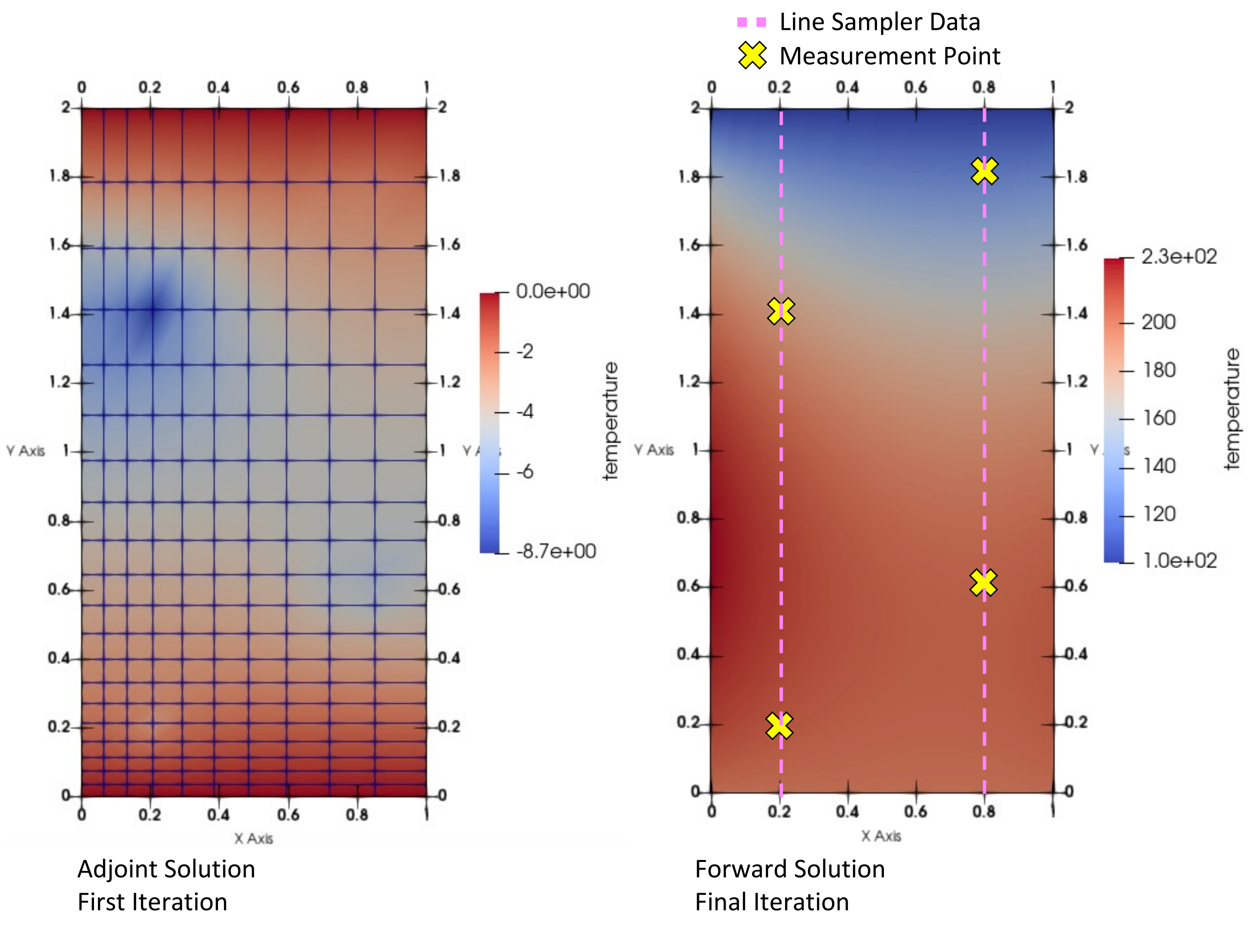


Figure 19: Left: Adjoint variable from the first iteration. Right: Temperature field from the forward problem for the final optimization iteration.

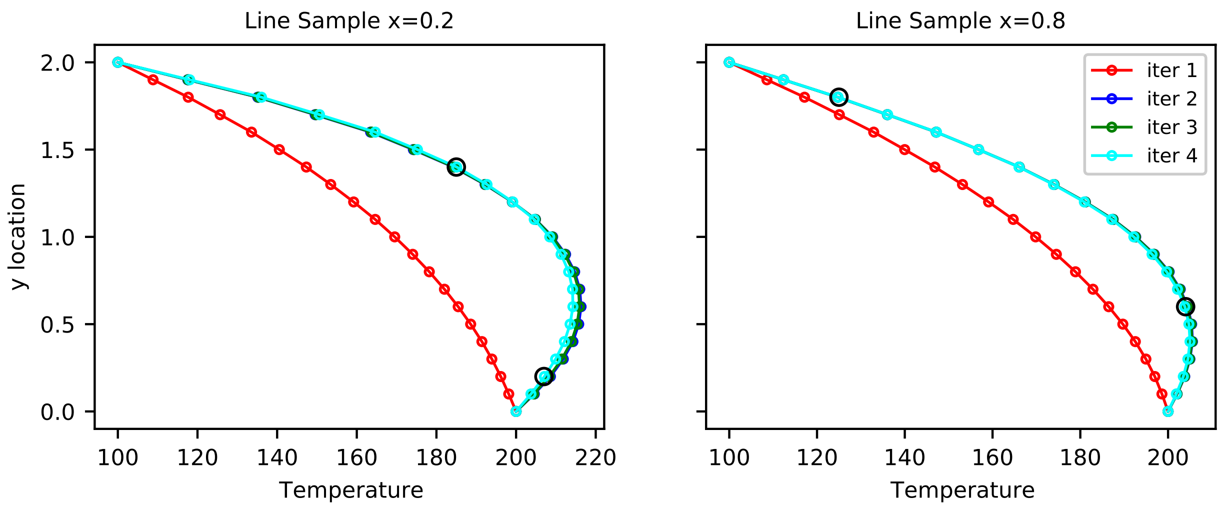


Figure 20: Per optimization iteration `LineValueSampler` results from the forward problem taken along the pink lines in Figure 19. Measurement points and values shown by the black circles.

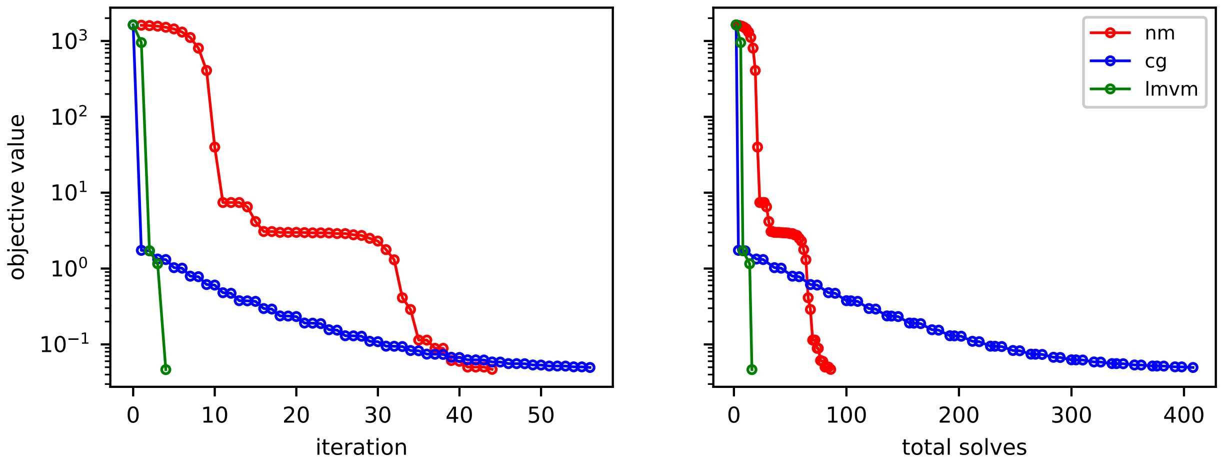


Figure 21: Convergence of the objective function plotted versus optimization iterations (left) and total solves (right)

where a, b and c are the parameters being optimized and x is the x-coordinate location. Dirichlet boundary conditions are placed on the top and bottom surfaces with temperatures shown in Figure 22. The right and left boundaries are insulated with $\nabla T \cdot n = 0$. This example will highlight the need for good initial conditions and bounds on the parameters.

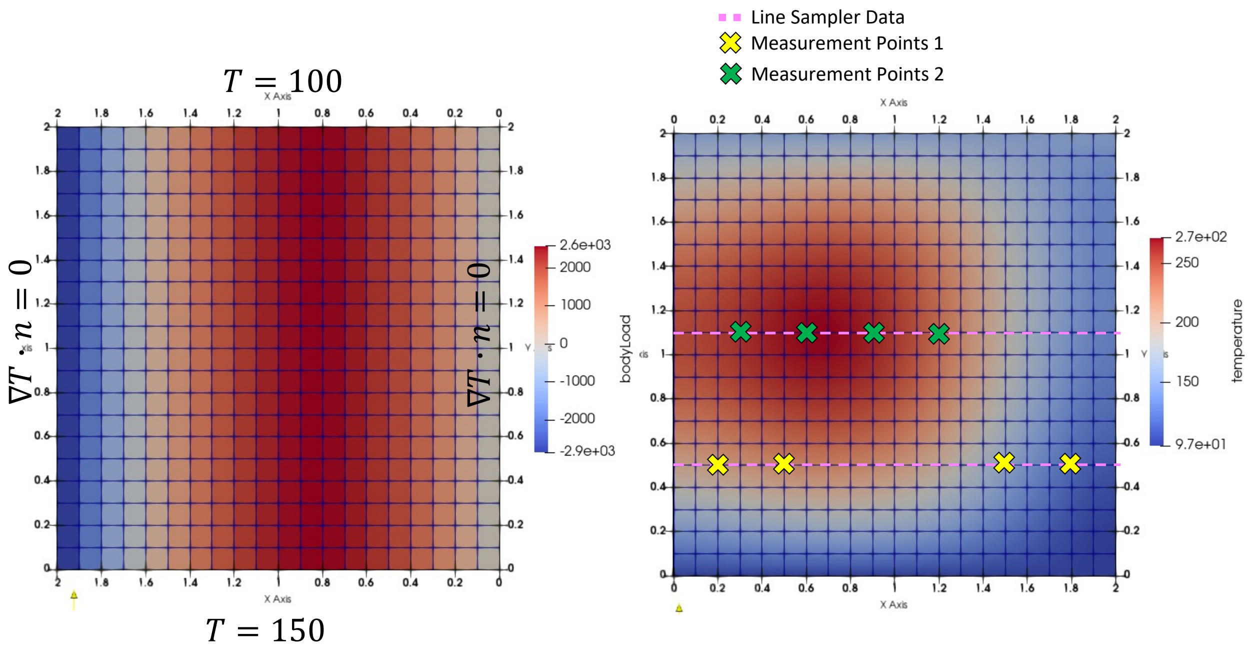


Figure 22: Left: Quadratically varying body load being parameterized. Right: Synthetic temperature field with two sets of measurements points shown by the yellow and green $\times$.

A threshold of 1e-4 is set for the absolute value of the gradient terms for the `taolmvm` solver which results in an objective value of $J = 4$. A `LineValueSampler` is taken along the pink lines in Figure 22 and is plotted in Figure 23 for the `taolmvm` iteration indicated in the legend. All eight measurement points are used in the optimized solution shown in Figure 23. Convergence is much slower in case where the initial guess is much further off.

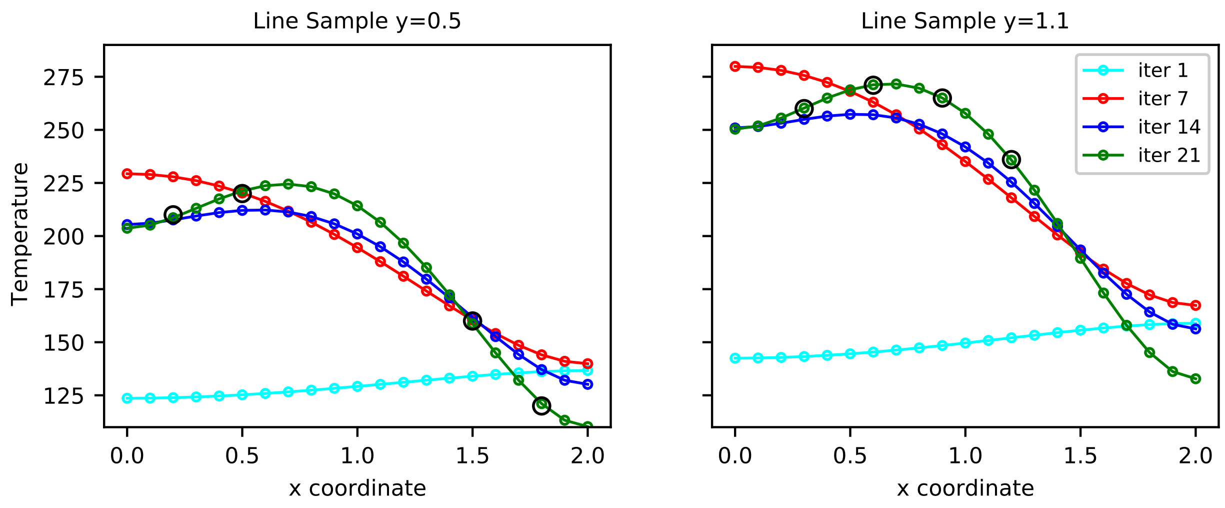


Figure 23: Per optimization iteration `LineValueSampler` results from the forward problem taken along the pink lines in Figure 22. Measurement points and values shown by the black circles.

Figure 24 shows the convergence rate for the `taonm`, `taocg` and `taolmvm` optimization solvers as a function of total finite element solves. `taocg` solver converges to the wrong local minimum. This highlights the need for enabling the bounded optimization algorithms in TAO such as bounded conjugate gradient `taobncg` and `taoblmvm` for bounded LMVM. Again, `taolmvm` provides the least computationally expensive solution. Overall, the convergence for all of the solvers is much slower for this problem than the force boundary condition parameterization of a linear function given in the previous section. This is partly due to a poor initial guess for the parameters. Figure 24 also shows the effect of measurement points on the convergence rates for each solver. The suffix `pts 1` uses the yellow $\times$ points in Figure 22, the suffix `pts 2` uses the green $\times$ points and the suffix `pts 1&2` uses both sets of points. The convergence rate is not shown to be overly sensitive to the set of measurement points chosen. Increasing the number of measurements points should increase the convergence rate. The `pts 2` points also appear to be in a region with a larger gradient of the temperature field than those in `pts 1` which should make `pts 2` more sensitive to the parameters.

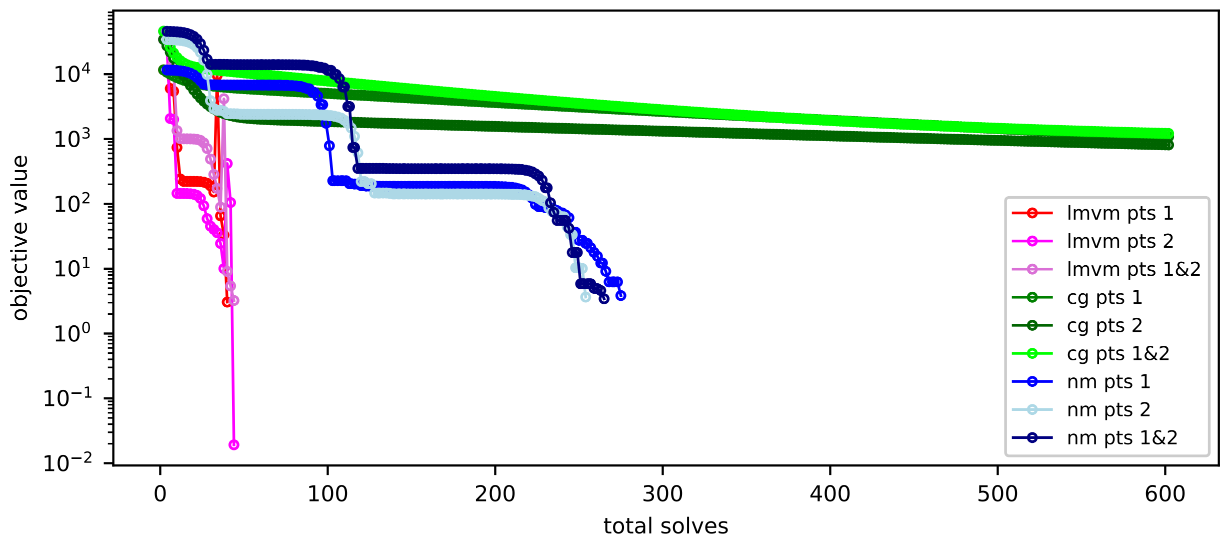


Figure 24: Convergence of the objective function plotted versus total solves. Three sets of optimizations were performed using different sets of measurement points shown in Figure 22, as indicated in the legend.

4 Conclusion

This report details the improvements and added capabilities to the MOOSE framework to support rapid mechanical property evaluation using data analytics. These updates include development in the STM for improved stochastic simulations of complex multiphysics models and inclusion of reduced-order modeling capabilities. Additionally, an inverse optimization framework has been developed to help utilize experimental data for material property and source identification in MOOSE models.

Section 2 gives an overview of the STM and details the updates to the STM. To summarize, a new input parameter in the stochastic *MultiApps* has been added to allow for more control on how stochastic simulations are parallelized. This control shows that memory performance can be vastly improved when dealing with large multiphysics models. Secondly, the surrogate model training architecture in the STM has generalized, which paves the way for more intricate ROM techniques and eases the development of new surrogate models. Finally, the section provides theory on the currently available surrogate models in the STM and includes examples on their usage. The available models include polynomial chaos expansion, polynomial regression, Gaussian processing, and reduced-basis proper orthogonal decomposition. The examples compare the three non-intrusive methods for simple heat-conduction model, which provides some verfication on their implementation. There are also a couple more complex examples using POD and PC to show how these ROMs can significantly improve run-times. The primary focus for future work in the STM involves more realistic applications. Applying the capabilities in the module to high-impact problems might show additional development needed and ideas for new capabilities.

An overview of the optimization algorithms implemented in Isopod are given in Section 3. The source identification examples using Isopod provide a basic overview of the *MultiApps* implementation of Isopod and the MOOSE-based features required for solving gradient based inverse optimization problems. The examples highlight several deficiencies in Isopod that motivate the following future tasks:

- Cleaning up and documenting Isopod optimization algorithms and merging it with STM
- Expanding the optimization algorithms to additional PDE's like elasticity and time dependent problems
- Explore methods to improve convergence of optimization algorithms including regularization and the implementation of a Hessian for use with Newton based optimization algorithms.

5 Acknowledgments

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract DE-AC07-05ID14517. Accordingly, the U.S. Government retains a non-exclusive, royalty free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

This research made use of the resources of the High Performance Computing Center at Idaho National Laboratory, which is supported by the Office of Nuclear Energy of the U.S. Department of Energy and the Nuclear Science User Facilities under Contract No. DE-AC07-05ID14517.

6 References

1. Andrew Slaughter and Zachary Prince. Stochastic tools. `https://mooseframework.inl.gov/modules/stochastic_tools/index.html`. (Accessed: 06.07.2021).

2. Brian M Adams, William J Bohnhoff, KR Dalbey, JP Eddy, MS Eldred, DM Gay, K Haskell, Patricia D Hough, and Laura P Swiler. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: version 5.0 user's manual. *Sandia National Laboratories, Tech. Rep. SAND2010-2183*, 2009.

3. Cody J. Permann, Derek R. Gaston, David Andrš, Robert W. Carlsen, Fande Kong, Alexander D. Lindsay, Jason M. Miller, John W. Peterson, Andrew E. Slaughter, Roy H. Stogner, and Richard C. Martineau. MOOSE: Enabling massively parallel multiphysics simulation. *SoftwareX*, 11:100430, 2020.

4. Michael Eldred. Recent advances in non-intrusive polynomial chaos and stochastic collocation methods for uncertainty analysis and design. In *50th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. American Institute of Aeronautics and Astronautics, may 2009.

5. Thomas Gerstner and Michael Griebel. Numerical integration using sparse grids. *Numerical Algorithms*, 18(3):209–232, January 1998.

6. Eva Ostertagová. Modelling using polynomial regression. *Procedia Engineering*, 48:500–506, 2012. Modelling of Mechanical and Mechatronics Systems.

7. Arthur E. Hoerl and Robert W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

8. Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2005.

9. René Pinnau. *Model Reduction via Proper Orthogonal Decomposition*, chapter 5, pages 95–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

10. Zachary M Prince and Jean C Ragusa. Parametric uncertainty quantification using proper generalized decomposition applied to neutron diffusion. *International Journal for Numerical Methods in Engineering*, 119(9):899–921, 2019.

11. Lorenz T Biegler, Omar Ghattas, Matthias Heinkenschloss, and Bart van Bloemen Waanders. *Large-scale PDE-constrained optimization: an introduction*, pages 3–13. Springer, 2003.

12. Arnold Neumaier. Solving ill-conditioned and singular linear systems: A tutorial on regularization. *SIAM review*, 40(3):636–666, 1998.

13. Richard C Aster, Brian Borchers, and Clifford H Thurber. *Parameter estimation and inverse problems*. Elsevier, 2018.

14. Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. *PETSc users manual*, 2019.

A Example Input File Listings

Listing 1: Simple heat conduction input file

```
[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 100
  xmax = 1
  elem_type = EDGE3
[]

[Variables]
  [T]
    order = SECOND
    family = LAGRANGE
  []
[]

[Kernels]
  [diffusion]
    type = MatDiffusion
    variable = T
    diffusivity = k
  []
  [source]
    type = BodyForce
    variable = T
    value = 1.0
  []
[]

[Materials]
  [conductivity]
    type = GenericConstantMaterial
    prop_names = k
    prop_values = 2.0
  []
[]

[BCs]
  [right]
    type = DirichletBC
    variable = T
    boundary = right
    value = 300
  []
[]

[Executioner]
  type = Steady
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Postprocessors]
  [avg]
```

```
    type = AverageNodalVariableValue
    variable = T
  []
  [max]
    type = NodalExtremeValue
    variable = T
    value_type = max
  []
[]
```

Listing 2: Heat conduction example training input file.

```
[StochasticTools]
[]

[Distributions]
  [k_dist]
    type = Uniform
    lower_bound = 1
    upper_bound = 10
  []
  [q_dist]
    type = Uniform
    lower_bound = 9000
    upper_bound = 11000
  []
  [L_dist]
    type = Uniform
    lower_bound = 0.01
    upper_bound = 0.05
  []
  [Tinf_dist]
    type = Uniform
    lower_bound = 290
    upper_bound = 310
  []
[]

[Samplers]
  [sample]
    type = MonteCarlo
    num_rows = 1000
    distributions = 'k_dist q_dist L_dist Tinf_dist'
    execute_on = PRE_MULTIAPP_SETUP
  []
[]

[MultiApps]
  [sub]
    type = SamplerFullSolveMultiApp
    input_files = sub.i
    sampler = sample
  []
[]

[Controls]
  [cmdline]
    type = MultiAppCommandLineControl
    multi_app = sub
    sampler = sample
    param_names = 'Materials/conductivity/prop_values
                   Kernels/source/value Mesh/xmax
                   BCs/right/value'
  []
[]

[Transfers]
  [data]
    type = SamplerReporterTransfer
    multi_app = sub
    sampler = sample
    stochastic_reporter = results
    from_reporter = 'avg/value'
  []
[]

[Reporters]
  [results]
    type = StochasticReporter
  []
[]

[Trainers]
  [poly_chaos]
    type = PolynomialChaosTrainer
    execute_on = timestep_end
    order = 4
    distributions = 'k_dist q_dist L_dist Tinf_dist'
    sampler = sample
    response = results/data:avg:value
  []
  [poly_reg]
    type = PolynomialRegressionTrainer
    execute_on = timestep_end
    max_degree = 4
    sampler = sample
    response = results/data:avg:value
    regression_type = "ols"
  []
  [gauss_process]
    type = GaussianProcessTrainer
    execute_on = timestep_end
    covariance_function = 'rbf'
    standardize_params = 'true'
    standardize_data = 'true'
    sampler = sampler
    response = results/data:avg:value
    tao_options = '-tao_bncg_type gd'
    tune_parameters = ' signal_variance length_factor'
    tuning_min = ' 1e-9 1e-3'
    tuning_max = ' 100  100'
  []
[]

[Covariance]
  [rbf]
    type = SquaredExponentialCovariance
    noise_variance = 1e-3
    signal_variance = 1
    length_factor = '0.038971 0.038971 0.038971 0.038971'
  []
[]

[Outputs]
  file_base = training
  [out]
    type = SurrogateTrainerOutput
    trainers = 'poly_chaos poly_reg gauss_process'
    execute_on = FINAL
  []
[]
```

Listing 3: Heat conduction example evaluation input file.

```
[StochasticTools]
[]

[Distributions]
  [k_dist]
    type = Uniform
    lower_bound = 1
    upper_bound = 10
  []
  [q_dist]
    type = Uniform
    lower_bound = 9000
    upper_bound = 11000
  []
  [L_dist]
    type = Uniform
    lower_bound = 0.01
    upper_bound = 0.05
  []
  [Tinf_dist]
    type = Uniform
    lower_bound = 290
    upper_bound = 310
  []
[]

[Samplers]
  [sample]
    type = MonteCarlo
    num_rows = 100000
    distributions = 'k_dist q_dist L_dist Tinf_dist'
    execute_on = initial
  []
[]

[Surrogates]
  [poly_chaos]
    type = PolynomialChaos
    filename = 'training_poly_chaos.rd'
  []
  [poly_reg]
    type = PolynomialRegressionSurrogate
    filename = 'training_poly_reg.rd'
  []
  [gauss_process]
    type = GaussianProcess
    filename = 'training_gauss_process.rd'
  []
[]

# Computing statistics
[VectorPostprocessors]
  [evaluate]
    type = EvaluateSurrogate
    model = 'poly_chaos poly_reg guass_process'
    sampler = sample
  []
[]

[Reporters]
  [stats]
    type = StatisticsReporter
    vectorpostprocessors = evaluate
    compute = 'mean stddev'
  []
[]

[Outputs]
  csv = true
[]
```

Listing 4: POD example FOM input.

```
halfa = 10
fulla = 20

[Problem]
  type = FEProblem
  extra_tag_vectors = 'diff0 diff1 diff2 diff3 abs0 abs1 abs2 abs3 src0 src1 src2'
[]

[Mesh]
  [msh]
    type = CartesianMeshGenerator
    dim = 2
    dx = '10 20 20 20 20 20 20 20 20'
    dy = '10 20 20 20 20 20 20 20 20'
    ix = '10 20 20 20 20 20 20 20 20'
    iy = '10 20 20 20 20 20 20 20 20'
    subdomain_id = '1 0 0 0 1 0 0 2 3
                    0 0 0 0 0 0 0 2 3
                    0 0 1 0 0 0 2 2 3
                    0 0 0 0 0 0 2 3 3
                    1 0 0 0 1 2 2 3 3
                    0 0 0 0 2 2 3 3 3
                    0 0 2 2 2 3 3 3 3
                    2 2 2 3 3 3 3 3 3
                    3 3 3 3 3 3 3 3 3'
  []
[]

[Variables]
```

```
  [psi]
  []
[]

[Kernels]
  [diff0]
    type = MatDiffusion
    variable = psi
    diffusivity = D0
    extra_vector_tags = 'diff0'
    block = 0
  []
  [diff1]
    type = MatDiffusion
    variable = psi
    diffusivity = D1
    extra_vector_tags = 'diff1'
    block = 1
  []
  [diff2]
    type = MatDiffusion
    variable = psi
    diffusivity = D2
    extra_vector_tags = 'diff2'
    block = 2
  []
  [diff3]
    type = MatDiffusion
    variable = psi
    diffusivity = D3
    extra_vector_tags = 'diff3'
    block = 3
  []
  [abs0]
    type = MaterialReaction
    variable = psi
    coefficient = absxs0
    extra_vector_tags = 'abs0'
    block = 0
  []
  [abs1]
    type = MaterialReaction
    variable = psi
    coefficient = absxs1
    extra_vector_tags = 'abs1'
    block = 1
  []
  [abs2]
    type = MaterialReaction
    variable = psi
    coefficient = absxs2
    extra_vector_tags = 'abs2'
    block = 2
  []
  [abs3]
    type = MaterialReaction
    variable = psi
    coefficient = absxs3
    extra_vector_tags = 'abs3'
    block = 3
  []
  [src0]
    type = BodyForce
    variable = psi
    value = 1
    extra_vector_tags = 'src0'
    block = 0
  []
  [src1]
    type = BodyForce
    variable = psi
    value = 1
    extra_vector_tags = 'src1'
    block = 1
  []
  [src2]
    type = BodyForce
    variable = psi
    value = 1
    extra_vector_tags = 'src2'
    block = 2
  []
[]

[Materials]
  [D0]
    type = GenericConstantMaterial
    prop_names = D0
    prop_values = 1
    block = 0
  []
  [D1]
    type = GenericConstantMaterial
    prop_names = D1
    prop_values = 1
    block = 1
  []
  [D2]
    type = GenericConstantMaterial
    prop_names = D2
    prop_values = 1
    block = 2
  []
  [D3]
    type = GenericConstantMaterial
    prop_names = D3
    prop_values = 1
    block = 3
  []
  [absxs0]
    type = GenericConstantMaterial
    prop_names = absxs0
    prop_values = 1
    block = 0
  []
  [absxs1]
    type = GenericConstantMaterial
    prop_names = absxs1
    prop_values = 1
    block = 1
  []
  [absxs2]
    type = GenericConstantMaterial
    prop_names = absxs2
    prop_values = 1
    block = 2
  []
  [absxs3]
    type = GenericConstantMaterial
    prop_names = absxs3
    prop_values = 1
    block = 3
  []
[]

[BCs]
  [left]
    type = NeumannBC
    variable = psi
    boundary = left
    value = 0
  []
  [bottom]
    type = NeumannBC
    variable = psi
    boundary = bottom
    value = 0
  []
  [top]
    type = DirichletBC
    variable = psi
    boundary = top
    value = 0
  []
  [right]
    type = DirichletBC
    variable = psi
    boundary = right
    value = 0
  []
[]

[Executioner]
  type = Steady
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Controls]
  [stochastic]
    type = SamplerReceiver
  []
[]
```

Listing 5: POD example training input

```
[StochasticTools]
```

```
[]

[Distributions]
  [D012_dist]
    type = Uniform
    lower_bound = 0.2
    upper_bound = 0.8
  []
  [D1_dist]
    type = Uniform
    lower_bound = 0.2
    upper_bound = 0.8
  []
  [D2_dist]
    type = Uniform
    lower_bound = 0.2
    upper_bound = 0.8
  []
  [D3_dist]
    type = Uniform
    lower_bound = 0.15
    upper_bound = 0.6
  []
  [absxs0_dist]
    type = Uniform
    lower_bound = 0.0425
    upper_bound = 0.17
  []
  [absxs1_dist]
    type = Uniform
    lower_bound = 0.065
    upper_bound = 0.26
  []
  [absxs2_dist]
    type = Uniform
    lower_bound = 0.04
    upper_bound = 0.16
  []
  [absxs3_dist]
    type = Uniform
    lower_bound = 0.005
    upper_bound = 0.02
  []
  [src_dist]
    type = Uniform
    lower_bound = 5
    upper_bound = 20
  []
[]

[Samplers]
  [sample]
    type = LatinHypercube
    distributions = 'D012_dist D012_dist D012_dist D3_dist
                     absxs0_dist absxs1_dist absxs2_dist absxs3_dist
                     src_dist src_dist src_dist'
    num_rows = 100
    execute_on = PRE_MULTIAPP_SETUP
  []
```

```
[]

[MultiApps]
  [sub]
    type = PODFullSolveMultiApp
    input_files = sub.i
    sampler = sample
    trainer_name = 'pod_rb'
    execute_on = 'timestep_begin final'
  []
[]

[Transfers]
  [param]
    type = SamplerParameterTransfer
    multi_app = sub
    sampler = sample
    parameters = 'Materials/D0/prop_values
                  Materials/D1/prop_values
                  Materials/D2/prop_values
                  Materials/D3/prop_values
                  Materials/absxs0/prop_values
                  Materials/absxs1/prop_values
                  Materials/absxs2/prop_values
                  Materials/absxs3/prop_values
                  Kernels/src0/value
                  Kernels/src1/value
                  Kernels/src2/value'
    to_control = 'stochastic'
    execute_on = 'timestep_begin'
    check_multiapp_execute_on = false
  []
  [data]
    type = PODSamplerSolutionTransfer
    multi_app = sub
    sampler = sample
    trainer_name = 'pod_rb'
    direction = 'from_multiapp'
    execute_on = 'timestep_begin'
    check_multiapp_execute_on = false
  []
  [mode]
    type = PODSamplerSolutionTransfer
    multi_app = sub
    sampler = sample
    trainer_name = 'pod_rb'
    direction = 'to_multiapp'
    execute_on = 'final'
    check_multiapp_execute_on = false
  []
  [res]
    type = PODResidualTransfer
    multi_app = sub
    sampler = sample
    trainer_name = 'pod_rb'
    execute_on = 'final'
    check_multiapp_execute_on = false
  []
[]
```

```
[Trainers]
  [pod_rb]
    type = PODReducedBasisTrainer
    var_names = 'psi'
    error_res = '1e-9'
    tag_names = 'diff0 diff1 diff2 diff3 abs0 abs1 abs2 abs3 src0 src1 src2'
    tag_types = 'op op op op op op op op src src src'
    execute_on = 'timestep_begin final'
  []
[]

[Outputs]
  [out]
    type = SurrogateTrainerOutput
    trainers = 'pod_rb'
    execute_on = FINAL
  []
[]
```

Listing 6: POD example evaluation input

```
[StochasticTools]
[]

[Distributions]
  [D012_dist]
    type = Uniform
    lower_bound = 0.2
    upper_bound = 0.8
  []
  [D1_dist]
    type = Uniform
    lower_bound = 0.2
    upper_bound = 0.8
  []
  [D2_dist]
    type = Uniform
    lower_bound = 0.2
    upper_bound = 0.8
  []
  [D3_dist]
    type = Uniform
    lower_bound = 0.15
    upper_bound = 0.6
  []
  [absxs0_dist]
    type = Uniform
    lower_bound = 0.0425
    upper_bound = 0.17
  []
  [absxs1_dist]
    type = Uniform
    lower_bound = 0.065
    upper_bound = 0.26
  []
  [absxs2_dist]
    type = Uniform
    lower_bound = 0.04
    upper_bound = 0.16
  []
  [absxs3_dist]
    type = Uniform
    lower_bound = 0.005
    upper_bound = 0.02
  []
  [src_dist]
    type = Uniform
    lower_bound = 5
    upper_bound = 20
  []
[]

[Samplers]
  [sample]
    type = LatinHypercube
    distributions = 'D012_dist D012_dist D012_dist D3_dist
                     absxs0_dist absxs1_dist absxs2_dist absxs3_dist
                     src_dist src_dist src_dist'
    num_rows = 1000
    execute_on = PRE_MULTIAPP_SETUP
  []
[]

[Surrogates]
  [rbpod]
    type = PODReducedBasisSurrogate
    filename = 'trainer_out_pod_rb.rd'
    change_rank = 'psi'
    new_ranks = '40'
  []
[]

[VectorPostprocessors]
  [res]
    type = PODSurrogateTester
    model = rbpod
    sampler = sample
    variable_name = 'psi'
    to_compute = nodal_l2
  []
[]

[Outputs]
  csv = true
[]
```