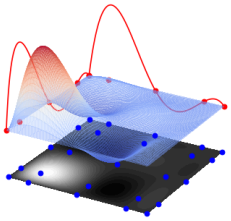
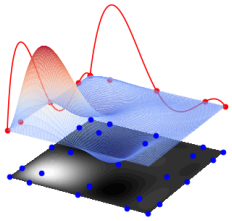


PyApprox



PyApprox: Enabling efficient model analysis

J. D. Jakeman[1]

[1]Optimization and Uncertainty Quantification, Sandia National Laboratories, Albuquerque, NM, 87123

Abstract

PyApprox is a Python-based one-stop-shop for probabilistic analysis of scientific numerical models. Easy to use and extendable tools are provided for constructing surrogates, sensitivity analysis, Bayesian inference, experimental design, and forward uncertainty quantification. The algorithms implemented represent the most popular methods for model analysis developed over the past two decades, including recent advances in multi-fidelity approaches that use multiple model discretizations and/or simplified physics to significantly reduce the computational cost of various types of analyses. Simple interfaces are provided for the most commonly-used algorithms to limit a user's need to tune the various hyper-parameters of each algorithm. However, more advanced work flows that require customization of hyper-parameters is also supported. An extensive set of Benchmarks from the literature is also provided to facilitate the easy comparison of different algorithms for a wide range of model analyses. This paper introduces PyApprox and its various features, and presents results demonstrating the utility of PyApprox on a benchmark problem modeling the advection of a tracer in ground water.

Article Info

Correspondence:
J. D. Jakeman
jdjakem@sandia.gov

SAND Number:
SAND2022-10458

Keywords:
Modeling, uncertainty quantification, sensitivity analysis, Bayesian inference, decision making, surrogate models, experimental design, multi-fidelity

1 Introduction

Numerical models have become an indispensable tool for understanding and predicting the behavior of physical processes across scientific disciplines spanning the environmental sciences to plasma physics. Advances in computational speeds and memory have enabled continuous improvements in the expressivity and predictive accuracy of multi-scale multi-physics phenomena. However, because of their increasing computational cost and high-dimensional parameterizations, it is challenging to use these high-fidelity (high-accuracy) models within model analyses and decision-making processes, such as uncertainty quantification and design, which require repeated evaluation of a model.

PyApprox provides an extensive set of computationally efficient numerical tools for model assessment and can be applied to most numerical simulation models. In this paper we refer to a model as any scalar or vector valued function

$$q = f(z)$$

that maps a set of D input variables $z = [z_1, \ldots, z_D]^\top \in I_z \subset R^D$ to a set of Q quantities of interest (QoI) $q = [q_1, \ldots, q_Q]^\top \subset R^Q$. The variables z can represent various sources of model uncertainty and/or

design variables that can be be used to control/optimize the behavior of a physical system. As a concrete example, consider the following advection-diffusion model used to simulate the flow of a tracer through an $S = 2$-dimensional aquifer

$$\begin{aligned} \frac{\partial c(x,t,z)}{\partial t} &= \nabla \cdot (K(x,z)\nabla c(x,t,z)) - \nabla \cdot (vc) + G(x,t,z), && x \in \Omega \subset R^S, t \in [0,T], z \in I_z \\ c(x,t) &= 0, && x \in \partial\Omega. \end{aligned} \tag{1}$$

Here, K is the hydraulic conductivity, v is the vector valued velocity field that advects the tracer, and G represents any source terms, such as pumps and wells.

The QoI of any model analysis must be tailored to the modeling objective, but are typically formulated as linear or nonlinear functionals of model output, e.g. the solution of (1), for example, $q = f(z) = f(c(z))$. Examples of QoI for our tracer exemplar model (1) include the temporal integral of the integral of concentration within the aquifer in a subdomain $\Gamma \subset \Omega$ or the flux of the tracer concentration across a segment $\partial\Omega_i$ of the boundary at the final time, respectively given by

$$f(z) = \int_0^T \int_\Gamma c(x,t,z)\,\mathrm{d}x\,\mathrm{d}t \quad \text{and} \quad f(z) = \int_{\partial\Omega_i} K(x,z)\nabla c(x,T,z)\cdot n\,\mathrm{d}x. \tag{2}$$

The predictive accuracy of numerical models continues to improve at a rapid pace, nevertheless all models are approximations of the physical systems they represent. Uncertainties are introduced by the mathematical formulation of the model equations (model error), errors introduced by the numerical methods used to solve the model equations e.g. spatial mesh resolution and time-stepping size (discretization error), errors introduced when measuring experimental and observational data (observational error), and uncertainty in model parameters, e.g. changing climate, uncertain initial and boundary conditions, and internal model parameters such as the conductivity field K, which dictate the predictive accuracy of a model (parametric error).

Given a modeling objective and well defined QoI, **PyApprox** comprises a plethora of tools supporting the most common model analyses (analysis algorithms depicted in Figure 1):

1. *sensitivity analysis* for determining the model inputs most influencing predicted quantities of interest (QoI);
2. *Bayesian inference* for using observational and experimental data for model calibration and reducing the prior-based uncertainty in model QoI;
3. *experimental design* for judiciously determining when and where to collect observations in a manner that maximizes information gain and reduction of uncertainty;
4. *multi-fidelity* Monte Carlo quadrature which uses multiple model discretizations or simplified physics to significantly reduce the computational cost of the forward propagation of uncertainties; and
5. *surrogate modeling* (including multi-fidelity algorithms) for reducing costs of sensitivity analysis, model calibration and experimental design.

PyApprox also has an optimization module for design under uncertainty, however this capability is under development, and currently limited, so will not be discussed in this paper.

1.1 Novelty

There are several software tools available for various types of model analyses targeted by **PyApprox** . Some focus on specialized aspects of of model analysis, e.g. SALib [34] for sensitivity analysis, [24], PyGPC [95] for surrogate modeling, and MUQ [70] and HippyLib [93] for Bayesian inference of model parameters. Other packages provide a broader set of tools for model analysis, for example, Dakota [1], UQtk [14] UQPy [68], UQLab [54], UQ-PyL [94] and OpenTURNS[4], whereas other packages, e.g. Equadratures [84] and ChaosPy [19], provide a subset of model analysis tools.

Each of the software packages listed above have their own strengths and weaknesses and their capabilities are continually evolving. So instead of providing a biased summary of these pros and cons, this paper will instead focus on the capabilities and strengths of **PyApprox** . At a high-level these strengths are: 1) easy installation on all major platforms; 2) adoption of state-of the art software quality practices including regular

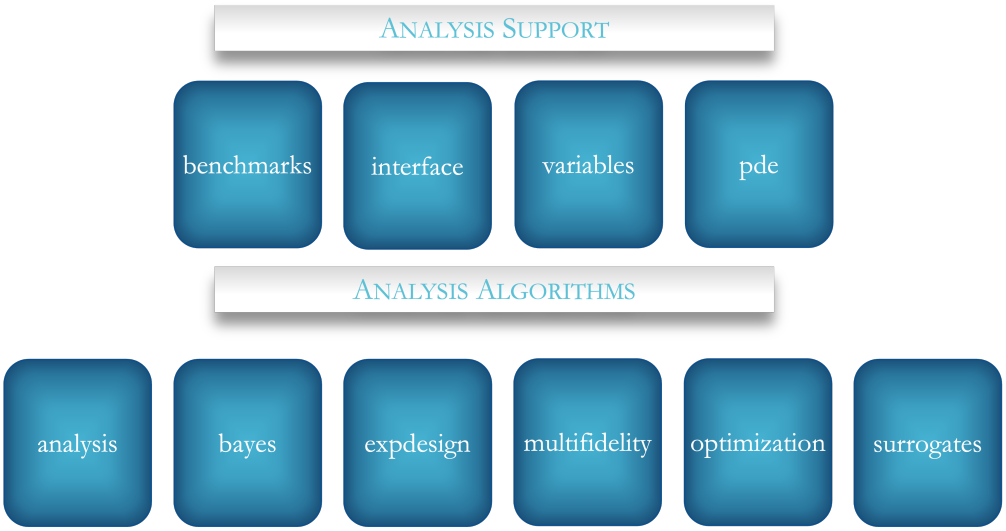


Figure 1: Visual depiction of the main modules in PyApprox .

automated unittesting; 3) multiple complementary algorithms for all major model analyses that allow the user to mitigate the majority of challenges limiting the tractability of model analyses; 4) interfaces for evaluating multiple black-box models written in any programming language; (5) well-chosen default arguments for the most commonly used algorithms, which reduce the user's need to understand the inner-workings, and tune the hyper-parameters, of these algorithm; 6) highly modular components that facilitate the development of new algorithms; and 7) extensive documentation of user and developer features.

While some of the aforementioned software tools provide similar capabilities to PyApprox in a number of areas, PyApprox offers a number of novel features that in combination are not present any of the other packages. Specifically, PyApprox provides:

1. automated (quasi) optimal algorithms for generating training data and refining surrogates in a manner that balances the accuracy of the surrogate in high-probability regions of the model input, with the conditioning of the approximation algorithm, for example of the matrix used in linear least squares to fit polynomials;
2. model-based optimal experimental design for both linear(ized) and nonlinear models that aim to reduce uncertainty in not only model parameters but also model predictions; and
3. multi-fidelity methods that use a (possibly non-hierarchical) ensemble of models of varying cost and accuracy to drastically reduce the computational cost of building surrogates and estimating statistics summarizing uncertainty in model predictions.

1.2 Software overview

PyApprox is highly modular, allowing the user to implement customized end-to-end model analyses with a minimal amount of code. Default function arguments are provided for most commonly-used algorithms that reduce a user's need to understand the inner workings, and tune the various hyper-parameters, of each algorithm. However, the software also supports the needs of advanced users and provides a productive research environment for the development of new model analysis algorithms.

PyApprox is freely available under an MIT license and is managed with a `git` repository `https://github.com/sandialabs/pyapprox/`. The package has extensive capabilities for model analysis, containing over 100,000 lines of code. The software package is implemented entirely with Python-based tools and so is highly portable. Moreover, the automated execution of 550 unit and regression tests ensure the quality of the software. Installation instructions can be found in the online documentation `https://sandialabs.github.io/pyapprox/index.html`. This site also contains user and developer level docu-

mentation of available functions and classes, as well as numerous examples of how to use PyApprox and tutorials on the mathematical foundations of the algorithms implemented.

This paper provides an overview of PyApprox and is organized into two main sections: Section 2 describes the four modules used for supporting model analyses and verifying and validating analysis algorithms; and Section 3 summarizes the five main modules that implement a plethora of algorithms for model analysis. A high-level description of each class of methods implemented is provided along with a demonstration on a advection diffusion benchmark based on (1). Conclusions, including remarks on the future of PyApprox are then presented in Section 4.

2 Analysis support tools

This section summarizes the four PyApprox modules used for setting up model analyses and verifying, validating, and investigating the performance of model analyses tools.

2.1 Variables

All model analyses require defining plausible ranges for the model variables z. Many analyses also necessitate defining prior probability distributions that describe the various sources of model uncertainty. The `variables` module provides various tools to define probability distribution functions (PDFs) that can be used for model analyses. The most frequently-used tools include definitions of multi-variate random variables and their PDFs, variable transformations and sampling routines.

PyApprox supports multivariate random variables comprised of independent univariate variables. Such variables can be defined from a list of any `scipy.stats` variable objects. The following code shows how to create and sample from two independent uniform random variables defined on $[-2, 2]$.

```
>>> nsamples = 1000
>>> univariate_variables = [stats.uniform(-2, 4), stats.uniform(-2, 4)]
>>> variable = IndependentMarginalsVariable(univariate_variables)
>>> samples = variable.rvs(nsamples)
>>> print_statistics(samples)
            z0          z1
count  1000.000000 1000.000000
mean    -0.050373  -0.052408
std      1.135335   1.178004
min     -1.992304  -1.996393
max      1.994013   1.995606
```

Here the `print_statistics` function will print useful information about the samples such as minimum and maximum values, mean and variance.

Custom-dependent random variables can also be constructed by deriving from the abstract `JointVariable` base class. PyApprox only dictates that an `rvs` method be defined to allow sampling from the variable. However, if a PDF function is provided, Nataf [53] and Rosenblatt [76] transformations can be used to convert new custom variables into canonical-independent multivariate uniform and Gaussian distributions.

It is often advantageous to scale the inputs of a model when building surrogates. This can be achieved by using `AffineTransform` which maps all bounded variables to $[-1, 1]$ and maps (semi) unbounded marginal variables supported by `scipy.stats` to canonical 1D variables, e.g. a Gaussian variable $\mathcal{N}(\mu, \sigma^2)$ with arbitrary mean and variance is mapped to the standard normal $\mathcal{N}(0, 1)$ with mean zero and unit variance. If only ranges of a model input have been provided we recommend specifying such variables as uniform on the specified ranges. An example of mapping uniform variables on $[-2, 2]$ to $[-1, 1]$ is shown below.

```
>>> var_trans = AffineTransform(variable)
>>> canonical_samples = var_trans.map_to_canonical(samples)
```

2.2 Interface

PyApprox treats all models simply as python functions that take in a set of variables and return a vector of QoI, and optionally a Jacobian, for each sample. The interface module provides tools to interact with complex numerical simulation software. In this section we provide a brief overview of some useful model interfacing tools.

It is often useful to be able to track the time needed to evaluate a function. We can track this using the `pyapprox.interface.wrappers.TimerModel` and `pyapprox.interface.wrappers.WorkTrackingModel` objects which are designed to work together. The former times each evaluation of a function and appends the time to the quantities of interest returned by the function, i.e returns a 2D `numpy.ndarray` with shape $(N, Q+1)$, where N is the number of evaluations requested. The second extracts the time and removes it from the quantities of interest and returns output with the original shape (N, Q) of the user function while saving the execution times per sample as a member variable.

PyApprox also provides a mechanism to evaluate ensembles of models, which is useful for multi-fidelity modeling (Section 3.5). Specifically, the `pyapprox.interface.ModelEnsemble` object can be used to create a function that takes samples of the input z plus an additional configure variable defining which model to evaluate. Let us use half the samples to evaluate a model with ID $= 0$ and evaluate the second model, with ID $= 1$, at the remaining samples.

```
>>> model_ensemble = ModelEnsemble([pyapprox_fun_1, pyapprox_fun_2])
>>> timer_fun_ensemble = TimerModel(model_ensemble)
>>> worktracking_fun_ensemble = WorkTrackingModel(
    timer_fun_ensemble, num_config_vars=1)
>>> fun_ids = np.ones(nsamples)
>>> fun_ids[:nsamples//2] = 0
>>> ensemble_samples = np.vstack([samples, fun_ids])
>>> values = worktracking_fun_ensemble(ensemble_samples)
>>> query_fun_ids = np.atleast_2d([0, 1])
>>> print(worktracking_fun_ensemble.work_tracker(query_fun_ids))
[0.02505493 0.07497811]
```

By using the `WorkTracking` model we can query the median execution times of each model using the last two lines of code above. The run time of each sample evaluation can be also be obtained if needed.

For expensive pure python models it is often useful to be able to evaluate each model concurrently. This can be achieved using `pyapprox.interface.wrappers.PoolModel`. Note this function is not intended for use with distributed memory systems, but rather is intended to use all the threads of a personal computer or compute node. `pyapprox.interface.async_model.AsynchModel` can be used for running multiple simulations in parallel on a distributed memory system and/or interfacing with models that can be invoked via command line executables.

2.3 Benchmarks

PyApprox provides numerous benchmarks that can be used to verify, validate and assess the performance of model analyses algorithms. Only a few attempts have been made at collating such benchmarks. The online repository of benchmarks [88] and the integration test suite [20] are two noteworthy examples. These benchmarks have proved highly popular, however are primarily limited to algebraic functions. PyApprox provides a consistent interface to many of the existing algebraic benchmarks in the aforementioned works, but also to several benchmarks that utilize partial differential equations for modeling engineering and environmental processes.

All benchmarks can be setup using a simple consistent interface that returns a dictionary of benchmark attributes. Each benchmark consists of at least a function and joint random variable. However, in many cases additional attributes are provided whose nature depends on the purpose of the benchmark. The following code shows how to create and list the attributes of a sensitivity analysis benchmark.

```
>>> benchmark = setup_benchmark("ishigami", a=7, b=0.1)
>>> print(benchmarks.keys())
dict_keys(['fun', 'jac', 'hess', 'variable', 'mean', 'variance', 'main_effects',
↪  'total_effects', 'sobol_indices'])
```

The names of available benchmarks can be obtained using `pyapprox.benchmarks.list_benchmarks()`. There are several benchmarks for sensitivity analysis [67, 38, 79, 61], integration [20], surrogates [12, 66, 59, 60], Bayesian inference [1, 55], optimization [1], and multi-fidelity modeling [25, 41]. These benchmarks address one or more of the challenges facing each research area, for example, high-dimensionality, low-regularity of the model response surface and/or intrinsic structure that can be exploited to reduce the computational cost of analyses, for instance, anisotropy, low-rank, or low-dimensional structure. Benchmarks are continually being added to PyApprox to address new challenges posed by model analyses.

2.4 PDE

At the time of writing, PyApprox explicitly supports a number of benchmarks based on solving the advection-diffusion equations in (1). However additional benchmarks will soon be implemented for various other PDEs using PyApprox 's `pde` module. This module uses Chebyshev spectral collocation [9, 91] to solve various PDEs including incompressible Stokes and Navier Stokes equations, Helmholtz equations, Euler-Bernoulli Beam equations, shallow water wave equations, and various PDEs for modeling land-ice evolution including the shallow-ice, shallow-shelf, and first-order Stokes equations.

The `pde` module is not intended to solve practical application problems, but rather provide a purely python model to use as benchmarks. PDEs can be defined on 1D or 2D rectangular meshes. Simple transformations of 2D rectangular meshes are also supported. New PDE solvers can be implemented simply by writing a function that evaluates the residual F of any PDE that can be expressed in the form

$$\frac{\partial u}{\partial t} = F(u),$$

where the solution u can be a scalar or vector valued solution. Given this residual, automatic differentiation (using PyTorch [71]) can then be used to compute the Jacobian $= \frac{\partial F}{\partial u}$ necessary for Newton's method to solve steady state PDEs with $\frac{\partial u}{\partial t} = 0$ or the stage solutions of diagonally implicit Runge Kutta time-integration methods. Jacobians can also be specified manually when available, which can reduce the time in solving a PDE substantially. Gradients of functionals of PDE solutions with respect to model parameters can also be computed for steady-state problems using automatically-constructed discrete adjoint equations [11]. Such gradients are useful for testing gradient-based model analysis tools, e.g. those used for optimization and Bayesian inference. The symbolic mathematical python package Sympy [56] is used to speed up the process of constructing manufactured solutions for each PDE implemented in PyApprox .[1]

3 Model analysis algorithms

In this section we outline the various capabilities of PyApprox and demonstrate their use on the model of the advection and diffusion of a tracer in (1). The benchmark involves determining the the coefficients of a Karhunen Loeve expansion (KLE), with a squared exponential kernel, used to represent the log diffusivity field, i.e

$$K(x,z) = K_0(x)\exp\left(\sum_{d=1}^{D}\sqrt{\lambda_d}\psi_d(x)z_d\right),$$

which best matches noisy synthetically-generated observational data; here $K_0(x) = 1$ and λ and ψ are the eigenvalues and eigenfunctions of the KLE, respectively. The data are obtained assuming that the tracer is

[1] Recent advances in operator inference and discontinuous Galerkin-like methods for multi-scale, multi-physics and domain decomposition will soon be available in PyApprox . These tools are intended to help benchmark recent advances in scientific machine learning for solving dynamical systems.

in equilibrium (steady-state) given a constant addition of the tracer modeled by the forcing function

$$G(x,t,z) = G_{\text{src}}(x) = \frac{s_{\text{src}}}{2\pi h_{\text{src}}^2}\exp\left(-\frac{|x-x_{\text{src}}|^2}{2h_{\text{src}}^2}\right)$$

where $s_{\text{src}} = 100$, $h_{\text{src}} = 0.1$, and $x_{\text{src}} = [1/4, 3/4]^\top$.[2] The following simple code can be used to setup this benchmark.

```
>>>inv_benchmark = setup_benchmark("advection_diffusion_kle_inversion", kle_nvars=3,
    noise_stdev=0.1, nobs=10, kle_length_scale=0.5)
>>> print(inv_benchmark.keys())
dict_keys(['negloglike', 'variable', 'noiseless_obs', 'obs', 'true_sample', 'obs_indices',
    'obs_fun'])
```

First, we will use sensitivity analysis to determine the KLE parameters that most influence the model's ability to fit the observed data. We will assume that our prior distribution for z is the distribution of independent and identically distributed Gaussian variables, with zero mean and unit variance. Second, Bayesian inference will be used to calibrate the KLE parameters. Third, model-based optimal experimental design will be used to determine the spatial locations of the most informative observations, Finally, multi-fidelity modeling will be used to estimate statistics in regard to the uncertainty in predictions of the concentration of the tracer in a subdomain of the physical domain, made using a transient version of the model when a sink is introduced to extract some of the tracer, such that

$$G(x,t,z) = G_{\text{src}}(x) - \frac{s_{\text{sink}}}{2\pi h_{\text{sink}}^2}\exp\left(-\frac{|x-x_{\text{sink}}|^2}{2h_{\text{sink}}^2}\right)$$

where $s_{\text{sink}} = 100$, $h_{\text{sink}} = 0.1$, and $x_{\text{sink}} = [3/4, 3/4]^\top$

In the remainder of the paper we include the major code snippets used for this example. The entire code can be found in the examples folder in the online documentation.

3.1 Surrogate modeling

Many simulation models are extremely computationally expensive such that performing sensitivity analysis, Bayesian inference, optimal experimental design and the forward propagation of uncertainty can be computationally intractable. Various methods have been developed to produce surrogates $f_N(z) \approx f(z)$ of the response surface of the parameter-to-QoI map of a model. Generally speaking, surrogates are built using a "small" number of model simulations N and are then substituted in place of the expensive simulation models in future analysis.

The various surrogate methods supported in **PyApprox** are summarized in Table 1. There is no one best surrogate method. Their utility depends on the type of analysis being performed, the objectives of that analysis and the properties of the model input variables and response surface which maps these inputs to the model QoI. An overview of the strengths and weaknesses of surrogate modeling (including, but not limited to, those implemented in **PyApprox**) can be found in [62]. The surrogates in **PyApprox** were chosen for their ability to estimate the error in the surrogate and/or automatically adapt to intrinsic structure in the response surface. These surrogates can either be trained from a fixed training data set, or be adaptive such that new training data is continually added to improve the surrogate until the surrogate reaches a pre-specified accuracy or a maximum computational budget associated with obtaining the training data is reached. We refer to the first class of surrogates as regression-based and the latter as adaptive and we overview both these classes below.

We remark that deep machine learning methods based upon different types of neural network (NN) structures have gained a lot of popularity in recent years. These methods are not implemented in **PyApprox** for two reasons. First, software tools for constructing neural networks are well developed and supported via tools, and second the author is unaware of any NN method that can simultaneously adapt the structure of the network while judiciously choosing training data to maximize accuracy while minimizing the total amount of training data.

[2] Typically, the velocity field is determined using the equations for Darcy's Flow, however for simplicity we assume that the field is pre-determined and fixed for all random variables.

Table 1: The Surrogate methods implemented in PyApprox ; polynomial chaos expansions (PCE), Gaussian processes (GP), tensor-trains (TT), and sparse grids (SG). The columns labeled regression and adaptive respectively specify whether a method can be used with an arbitrary pre-specified data set or supports adaptive sampling methods. The techniques used for error estimation are listed in the Error Est. column.

Method	Regression	Adaptive	Error Est.	Refs.
PCE	✓	✓	Cross Validation	[42, 44, 43, 65, 57]
GP	✓	✓	Bayesian	[75, 72, 78, 32, 52]
FT	✓	✗	Cross Validation	[26]
SG	✗	✓	Hierarchical Surpluses	[3, 33, 21, 64]

3.1.1 Regression-based surrogates

Regression-based surrogates build a surrogate $f_N(z) \approx f(z)$ from N training samples $\mathcal{Z}_N = z^{(1)}, \ldots, z^{(N)}$ and corresponding values $q = [q_1, \ldots, q_N]^\top$, where $q_i = f(z^{(i)})$. PyApprox supports three different regression-based surrogates – polynomial chaos expansions (PCEs), Gaussian processes (GPs), and tensor-trains (TT) – which we detail now.

All regression-based surrogates in PyApprox can be constructed with error estimates. When using GPs the pointwise standard deviation of its posterior distribution can be used to estimate error as a function of z. In contrast, K-fold cross validation can be used to estimate the error in PCE and TT. The way in which each surrogate estimates error is summarized in the 4th column of Table 1.

Polynomial chaos expansions. PCE [97, 22] represent the model output $f(z)$ as an expansion in orthonormal polynomials,

$$f_{N,K}(z) = \sum_{\lambda \in \Lambda} \alpha_\lambda \phi_\lambda(z), \qquad |\Lambda| = K.$$

where Λ contains multi-indices $\lambda = [\lambda_1 \ldots, \lambda_D]$ specifying the univariate polynomial degrees of each basis term included in the expansion. Often a total degree truncation is used to define Λ, however this requires knowing the best maximum degree of the basis. PyApprox uses cross validation to choose the best degree and more advanced basis adaptation strategies [42] that use cross validation to choose an anisotropic index set; that is, higher-degree polynomials will be used some dimensions relative to other dimensions.

PCEs are popular because the basis functions ϕ are constructed to be orthogonal with respect to the PDF of z, which allows analytical computation of moments and design of well-conditioned sampling schemes. The PCEs in PyApprox can be constructed for independent and dependent random variables z [44]. The recent algorithm in [63] allows PyApprox to compute the recursion coefficients of any continuous 1D variable in `scipy.stats` of the univariate polynomials that are used in the multivariate basis. Unlike most other software packages implementing PCE, PyApprox can also be used when z consists of both continuous and discrete random variables [39].

PyApprox supports several methods for computing the PCE coefficients from a pre-collected training data set. All methods determine the coefficients such that

$$\Phi(\mathcal{Z}_N)\alpha \approx q \qquad \text{with matrix entries} \qquad \Phi_{ij}(\mathcal{Z}_N) = \phi_j(z^{(i)})$$

where there is a one-to-one mapping between the scalar indices j and the multi-indices λ. Discrete linear least squares can be used to minimize the ℓ_2 norm measuring the mismatch between the data and PCE predictions. Least angle regression, its LASSO modification [16, 5] and orthogonal matching pursuit [92] can also be used to build sparse PCEs [42] that attempt to maximize the number of non-zero coefficients in the PCE. PyApprox also implements risk-adapted surrogates[46] that are guaranteed to produce conservative estimates of risk measures such as probability of failure and average value at risk.

Gaussian processes. GPs [78, 75] are an extremely popular tool for approximating multivariate functions from limited data. A GP is a distribution over a set of functions. Given a prior distribution on the class

of admissible functions, an approximation of a deterministic function is obtained by conditioning the GP on available observations of the function. Given a kernel $C(z, z^\star, \theta)$ and mean function, a Gaussian process approximation assumes that the joint prior distribution of f, conditional on kernel hyper-parameters θ (e.g. the kernel variance and length-scales $[\sigma^2, \ell^\top]^\top$), is multivariate normal. For a set of training samples $\mathcal{Z}$ and associated values, the posterior distribution of the GP is

$$f_N(\cdot) \mid \theta, y \sim \mathcal{N}\left(m^\star(\cdot), C^\star(\cdot, \cdot; \theta)\right)$$

where

$$m^\star(z) = t_{\mathcal{Z}}(z)^\top A_{\mathcal{Z}}^{-1} y \quad C^\star(z, z') = C(z, z') - t_{\mathcal{Z}}(z)^\top A_{\mathcal{Z}}^{-1} t_{\mathcal{Z}}(z') \quad t_{\mathcal{Z}}(z) = [C(z, z^{(1)}), \ldots, C(z, z^{(N)})]^\top \tag{3}$$

and $A_{\mathcal{Z}} = C(\mathcal{Z}, \mathcal{Z})$ is the covariance kernel matrix evaluated at the training samples $\mathcal{Z}$.

The GP in PyApprox wraps the basic implementation provided by Scikit-learn [72], which provides well-maintained software for constructing GPs, including using maximum likelihood estimation for optimizing the GP hyper-parameters. The wrapper also provides much needed tools for probabilistic model analysis, such as gradients of the response surface for optimization, analytical estimation of sensitivity indices and moments (Section 3.2), and also extensions that support gradient enhanced GPs [8] and multi-level GPs [49].

Low-rank tensor-trains. Constructing PCE and GP surrogates of high-dimensional models $f(z)$ with many inputs can be intractable as the number of training points required can grow exponentially with the number of inputs z. PyApprox implements low-rank tensor-trains [69, 28] to mitigate this curse of dimensionality. TTs represent functions as the product of matrix-valued functions, called cores,

$$f_N(z) = F_1(z_1)F_2(z_2)\cdots F_D(z_D) \qquad F_d(z_d) = \begin{bmatrix} f_d^{(1,1)}(z_d) & \cdots & f_d^{(1,r_d)}(z_d) \\ \vdots & \ddots & \vdots \\ f_d^{(r_{d-1},1)}(z_d) & \cdots & f_d^{(d_{d-1},r_d)}(z_d) \end{bmatrix}$$

While tensor-trains can be used with both linear and nonlinear univariate functions, PyApprox only supports the use of univariate PCE. The coefficients of these PCEs are then obtained via maximum likelihood estimation (MLE) using gradients obtained analytically by backwards differentiation [26]. If each univariate PCE has K terms and the rank of each core F_d is R, then the number of unknowns in the tensor-train is DKR^2. Thus, when used with MLE and when the number of training data exceeds, but is proportional to, the number of unknowns, the number of training samples required only grows linearly with dimension. Consequently, tensor-trains can require much less training data than GPs and PCEs in high-dimensions, when the rank of the function is small.[3]

3.1.2 Adaptive surrogates

The surrogates in PyApprox were also chosen because they facilitate experimental design, also referred to as active learning strategies, for improving their accuracy while minimizing the amount of training data required. Most software packages that can be used to build surrogates do not tailor the sampling strategy to the surrogate being constructed; for example "space-filling" Latin-hypercube designs or low-discrepancy sequences are often recommended. However, this can be sub-optimal and in some cases leads to severe ill-conditioning of the optimization procedure, e.g. when least squares is used to build the surrogates. Furthermore most if not all surrogate packages do not consider the distribution of z when building the surrogate. Probabilistic information is then only included when sampling the surrogate to compute moments for example. If this approach is adopted for unbounded variables, such as Gaussian variables, the domain on which the surrogate is constructed must encapsulate the regions of high non-zero probability [40]. Moreover, not taking into account the probability of the random variables when building a surrogate often results in a loss of accuracy when compared to methods that leverage such information [44, 32]. Probability-unaware methods lose efficiency because to build a stable approximation one must often sample in regions of very low probability.

[3] PyApprox provides tools to build GPs and PCEs on low-dimensional active subspaces [13] which can significantly reduce their data requirements in high-dimensions when there is such low-dimensional structure in the model response surface f.

PyApprox implements a number of recent sampling schemes for building surrogates that target accuracy in high-probability regions of the input variables z while maintaining a well conditioned training sample set. Methods are provided for GPs, PCEs, and sparse grids, but currently not TTs.

Sparse grids. Sparse grids [3, 66] represent high-dimensional functions as linear combinations of tensor-product interpolants

$$f(z) \approx \sum_{\beta \in \mathcal{I}} c_\beta f_\beta(z),$$

where $\beta = [\beta_1 \ldots, \beta_D]$ is a multi-index that controls the number of univariate samples comprising the tensor-product grid used to construct the tensor-product interpolants.

While the tensor-product of any univariate basis, e.g. piecewise polynomials, B-splines, can be used to construct f_β PyApprox currently only supports tensor-product Lagrange polynomials, such that

$$\phi_{d,j}(z_d) = \prod_{k=1,k\neq j}^{m_{\beta_d}} \frac{z_d - z_d^{(k)}}{z_d^{(j)} - z_d^{(k)}}, \quad d \in [D], \qquad f_\beta(z) = \sum_{j \leq \beta} f(z^{(j)}) \prod_{d \in [D]} \phi_{d,j_d}(z_d).$$

where m_{β_d} is the number of interpolation points used in each variable dimension and the $\leq$ comparison is applied to multi-indices per-entry. PyApprox uses weighted univariate Leja sequences [64] to construct the tensor-product grids. These Leja sequences minimize the weighted Lebesque constant of the univariate polynomial interpolants, allowing the sparse grid to be highly accurate in high-probability of marginals of independent random variables, while maintaining the conditioning of the interpolant.

A Leja sequence (LS) is a doubly-greedy computation of a determinant maximization procedure. Given an existing set of nodes $\mathcal{Z}_N$, a Leja sequence update chooses a new node $z^{(N+1)}$ by maximizing the determinant of a new Vandermonde-like matrix with an additional row and column; the additional column is formed by adding a single predetermined new basis element, ϕ_{K+1}, and the additional row is defined by the newly added point. Hence an LS is both greedy in the chosen interpolation points, and also assumes some a priori ordering of the basis elements.

In one dimension, a weighted LS can be understood without linear algebra: Let $\mathcal{Z}_N$ be a set of nodes on with cardinality $N \geq 1$. The next point in the Leja sequence is given by

$$z^{(N+1)} = \underset{z \in \Gamma}{\operatorname{argmax}}\, v(z) \prod_{n=1}^{N} |z - z^{(n)}|.$$

A gradient based procedure is used to solve this optimization procedure, using analytical gradients of the orthonormal polynomials computed using their three term recursion.

Traditionally, Leja sequences were developed with $v(z) = 1$. PyApprox supports the use of $v(z) = \sqrt{\rho(z)}$ [64], which is the square-root of the joint probability density of the random variables, and

$$v(z) = \left(\sum_{n=1}^{N} \phi_n^2(z^{(i)}) \right)^{-\frac{1}{2}},$$

which is the square-root of the Christoffel function and is useful when explicit knowledge of the joint PDF is not available, but rather only an orthonormal basis which may have been computed from the moments of z.

PyApprox uses an adaptive algorithm [33, 21] to iteratively build up the index set $\mathcal{I}$ thereby controlling the number of samples (and thus polynomial degree) used to resolve each input dimension. This algorithm is highly effective at identifying and exploiting anisotropy but requires structured samples, which means that code failures when evaluating even one point in the grid have to be dealt with by expert manipulation of the software.

Cross validation cannot be used with sparse grids because they require a structured set of training samples. Although the sparse grid does have some error estimation capabilities based upon hierarchical surpluses which measure the change in the sparse grid as new sets of training samples are added, we have not found them to be practically very useful, although some attempts have been made to improve the accuracy of sparse grid error estimates [17].

Optimal sampling for PCEs. PyApprox uses two different approaches for adaptively constructing a PCE. The first uses multivariate Leja interpolation and the second least-squares or sparse regression.

We use a linear algebra formulation that greedily maximizes the weighted Vandermonde-like determinant to form multivariate Leja sequences, i.e.

$$z^{(N+1)} = \underset{z\in\Gamma}{\operatorname{argmax}} |\det v(z)\Phi(\mathcal{Z}\cup\{z^{(N+1)}\})|$$

The algorithm for generating weighted multivariate Leja sequences using LU factorization is outlined in Algorithm 1. Leja sequences are nested so that additional points can be added with a rank 1-update of the pivoted LU factorization.

Algorithm 1 Multivariate Leja Sequence

1: Choose the index set Λ such that $K \geq N$
2: Specify an ordering of the basis ϕ
3: Generate a set of $S \gg M$ candidate samples $\mathcal{Z}_S$
4: Build Φ, $\Phi_{n,k} = \phi_k(z^{(n)})$, $n \in [S]$, $k \in [N]$
5: Compute preconditioning matrix V, $V_{nn} = v(z^{(n)})$
6: Compute first N pivots of LU factorization, $PLU = LU(V\Phi, M)$
7: Form Leja sequence with candidates corresponding to the first N pivots

Once a Leja sequence has been generated, one can easily generate a polynomial interpolant with two simple steps. Given function evaluations at the samples in the sequence, i.e. $q = f(\mathcal{Z}_\mathcal{N})$, the coefficients of the PCE interpolant can then be computed via

$$\alpha = (LU)^{-1}P^{-1}Vq$$

These coefficients are then used with a sparse grid-like adaptive algorithm to add new important basis functions ϕ. At each iteration the algorithm identifies a number of different sets $\mathcal{J} \subset \Lambda$ of candidate indices λ that when added to the PCE may significantly reduce the surrogate's error. The algorithm then chooses the set $\mathcal{J}$ which does produce the biggest change and uses this set to generate new candidate sets $\mathcal{J}$ for refinement. Here we use the change in variance induced by a set as a proxy for the change in PCE error. This change in variance is simply the sum of the coefficients squared associated with the set, i.e. $\sum_{\lambda\in\mathcal{J}}\alpha_\lambda^2$.

The same steps for adapting the indices of a Leja-based PCE are used to build regression-based PCEs. However the sampling scheme differs. PyApprox supports three types of sampling schemes in this setting, Monte Carlo (MC) sampling of z, induced sampling [57, 63] and equilibrium sampling [43, 65]. Induced sampling has been shown to yield convergence guarantees with a minimum number of samples and equilibrium sampling yields the same guarantees asymptotically. Each time the basis is adapted new samples are added to ensure that condition number of the regression matrix $\Phi(\mathcal{Z}_N)$ is below a user-specified threshold. In general, MC sampling requires the most additional samples, then equilibrium sampling and induced sampling.

Optimal sampling for GPs. PyApprox implements two computationally-efficient algorithms to greedily select training samples that minimize the prediction variance of a GP; pivoted Cholesky [82, 32], also known as power-function sampling, and integrated variance (IVAR) sampling [78]. Pivoted Cholesky sampling minimizes the weighted L_p error of kernel-based approximations for a given number of data. The method uses pivoted Cholesky factorization and iteratively generates nested samples that minimize the error in the GP in high probability regions of densities specified by user.

The unweighted pivoted Cholesky method minimizes the worst case GP prediction variance

$$P_\mathcal{Z}(z) = \sqrt{C(z,z) - t_\mathcal{Z}(z)^\top A_\mathcal{Z}^{-1} t_\mathcal{Z}(z)},$$

also known as the power function [81, 18] in the radial basis approximation community. The procedure greedily selects the best set of samples from a candidate set $\mathcal{Z}_{cand}$ until the desired number of N samples is obtained (Algorithm 2).

Algorithm 2 Pivoted Cholesky Sampling

$z^{(1)} = \text{argmax}_{z \in \mathcal{Z}_{cand}} P_{\{z\}}(z)$
$\mathcal{Z}_1 = \{z^{(1)}\}$
for $j = 2, 3, \ldots, N$ **do**
    $z^{(1)} = \text{argmax}_{z \in \mathcal{Z}_{cand} \setminus \mathcal{Z}_{j-1}} |P_{\mathcal{Z}_{j-1}}(z)|$
    $\mathcal{Z}_j = \mathcal{Z}_{j-1} \cup \{z^{(1)}\}$
end for
return $\mathcal{Z}_N$

PyApprox reduces the computational complexity of this algorithm significantly by leveraging the pivoted Cholesky decomposition to perform the minimization. The first N pivots of the decomposition minimize the worst case prediction variance [32]. Moreover, a simple modification of the way pivots are chosen can be used to create a sample set that is accurate in high-probability regions of z.

PyApprox also supports computing IVAR designs that greedily minimize

$$\int_\Gamma P_{\mathcal{Z}}(z)^2 \rho(z)\, \mathrm{d}z = 1 - \text{Trace}\left[A_{\mathcal{Z}}^{-1} P\right] \qquad P = \int_\Gamma t_{\mathcal{Z}}(z)^\top t_{\mathcal{Z}}(z) \rho(z)\, \mathrm{d}z$$

For smooth kernels, the algorithm performs IVAR sampling and pivoted Cholesky sampling to achieve similar accuracy for the same number of training samples. Moreover, both approaches allow training samples generated in batches which allows training data to be evaluated in parallel. However, the pivoted Cholesky approach is computationally faster, but this often does not matter when the computational cost of collecting training data is large.

3.1.3 Benchmark demonstration

The following code block demonstrates how to use the automated adaptive methods in PyApprox to build a GP surrogate using weighted power function sampling. Defaults are provided for constructing any surrogate, but advanced users can tune those defaults via keyword arguments (**kwargs) which are outlined in the online documentation `https://sandialabs.github.io/pyapprox/index.html`. Each adaptive method allows the user to provide a callback that takes an approximation as its sole argument and interrogates that surrogate to obtain error estimates, number of training samples etc., each time an adaptive step is taken. The example below shows how to use such a callback to plot the decrease of the surrogate error as the number of training samples is increased. Figure 2 depicts the decay in the error in the GP surrogate as the number of samples increases. A very small number of samples is used to highlight the ability to generate error estimates of sensitivity indices computed in the next section.

```
>>> nsamples, errors = [], []
>>>  def callback(approx):
>>>      nsamples.append(approx.sampler.num_training_samples())
>>>      error = np.linalg.norm(approx(validation_samples)-validation_values, axis=0)
>>>      error /= np.linalg.norm(validation_values, axis=0)
>>>      errors.append(error)
>>>
>>> approx_result = adaptive_approximate(inv_benchmark.negloglike, inv_benchmark.variable,
↪  "gaussian_process", {"max_nsamples": 30, "ncandidate_samples": 2e3, "verbose": 0,
↪  "callback": callback, "kernel_variance": 400})
>>> plt.loglog(nsamples, errors, 'o-')
>>> plt.show()
```

As a general rule of the thumb, the efficacy of the adaptive sparse grid, GP and PCE algorithms in PyApprox decreases with dimensionality. If anisotropy is not present then they can be used with $O(1)$ variables but if it is then they can be used with $O(10)$. PCEs and sparse grids exploit anisotropy by increasing the degree of the polynomial basis in the more important directions, whereas GPs exploit anisotropy by optimizing the length-scale of the covariance kernel, upon which they are built, in each input direction.

It should also be remarked that gradients of each of these surrogate types can be obtained analytically using properties unique to each surrogate. This can be used to drastically speed up optimization, for example, used for model calibration. An example of this use is shown in Section 3.3.

.

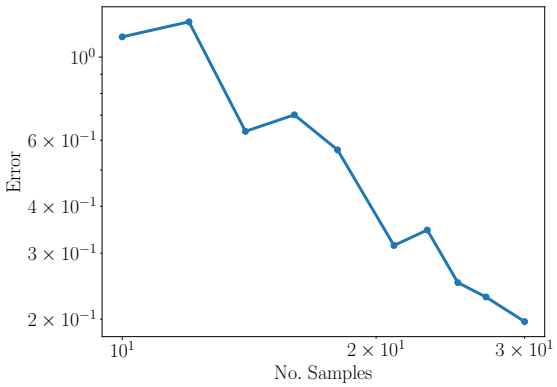


Figure 2: The error in an adaptive GP surrogate obtained using a callback function.

3.2 Sensitivity analysis

Sensitivity analysis (SA) is used to quantify the relative influence of uncertainties in model factors (inputs and parameters) on model outputs. Specifically, SA is often used to determine: the dominant sources of output variability; the nature of interactions between factors; and modifying model structure by removing insensitive model inputs. Broadly speaking, SA methods can be classified as either local or global sensitivity techniques. Local sensitivity methods quantify sensitivity via localized, typically gradient-based, information. Because sensitivities obtained at nominal samples of the inputs are not representative of the entire input space, PyApprox does not utilize local SA methods but rather implements global techniques.

Table 2: The SA methods implemented in PyApprox . The number of parameters that can be handled by each method is summarized in the Dimensionality column. The rate at which the accuracy improves with the number of model evaluations is summarized in the Convergence column. The techniques used for error estimation are listed in the Error Est. column.

Method	Smoothness	Dimensionality	Convergence	Error Est.	Refs.
Sobol PCE	✓	$O(1)$-$O(10)$	Fast	Cross Validation	[87, 15]
Sobol GP	✓	$O(1)$-$O(10)$	Fast	Bayesian	[67]
Sobol SG	✓	$O(1)$-$O(10)$	Fast	None	[10]
Main Effects MC	✗	$O(10^3)$	Slow	Bootstrapping	[7]
Sobol MC	✗	$O(10^2)$	Slow	None	[85]

Sobol sensitivity analysis is arguably the most popular global SA method. Main effect, total effect and Sobol sensitivity indices [85] are used to quantify the relative importance of parameters and their interactions on the variance of a QoI. These SA indices are useful when the variance of a function $\mathbb{V}[f]$ parameterized by D independent random variables with probability distribution $\rho(z) = \prod_{d=1}^{D} \rho(z_d)$ with support $I_z = \bigotimes_{d=1}^{D} I_{z_j}$ can be decomposed into the following finite sum

$$\mathbb{V}[f] = \sum_{i=1}^{D} V_i + \sum_{i,j=1}^{D} V_{i,j} + \cdots + V_{1,\ldots,D}. \tag{4}$$

Here, V_i is the amount of variance explained by an individual variable acting alone and $V_{i,j}$ is the contribution of the pairwise interactions of parameters, and so on. Provided this decomposition holds, the main-effect and total effect sensitivity indices are respectively given by

$$S_d^M = \frac{V_d}{\mathbb{V}[f]} \qquad \text{and} \qquad S_d^T = 1 - \frac{V_{\sim d}}{\mathbb{V}[f]},$$

where $V_{\sim d}$ is the variance explained by all variables except z_d. Some algorithms allow these quantities to be computed without computing all 2^D entries in (4), however PyApprox also supports computation of selected terms in the expansion when one is interested in comparing the strengths of parameter interactions. The so called Sobol indices are given by the terms of the decomposition (4) normalized by the total variance, e.g. $\frac{V_i}{\mathbb{V}[f]}$.

PyApprox supports using MC sampling and Quasi-Monte Carlo (QMC) sampling to compute main-effect, total-effect and Sobol sensitivity indices using the algorithm from [85]. This algorithm is recommended when the response surface of the model under consideration is not smooth, e.g. is only piecewise continuous. However, unlike MC sampling for computing only the variance, this method is not suitable for high-dimensional inputs because of its need to generate large replicate samples, with small modifications, for each input variable. For non-smooth and high-dimensional functions we suggest the MC-based algorithm presented in [7] which can be used for computing main-effect (but not total-effect or Sobol) indices.

The aforementioned MC algorithms can be applied to non-smooth functions, but their estimates of sensitivity converge slowly as the number of model evaluations is increased. Consequently, for smooth functions of moderate dimension, we suggest using surrogates for estimating sensitivity. Surrogates can be used in place of computationally expensive simulation models and interrogated at a fraction of the cost using MC sensitivity methods. However, surrogates often provide a means to compute sensitivity indices analytically. PyApprox supports computing sensitivity indices exactly (to machine precision) using GPs, PCEs, and sparse grids. The variance, main effect indices and total effect indices can be computed from PCE using [87]

$$\mathbb{V}[f] = \sum_{\lambda\in\Lambda} \alpha_\lambda^2 - \alpha_0^2, \qquad S_d^M = \frac{1}{\mathbb{V}[f]} \sum_{\substack{\lambda\in\Lambda,\lambda_d>0,\\ \lambda_i=0,i\neq d}} \alpha_\lambda^2 \qquad S_d^T = \frac{1}{\mathbb{V}[f]} \sum_{\lambda\in\Lambda,\lambda_d>0} \alpha_\lambda^2$$

Cross validation can be used to estimate errors in these quantities as suggested in [15]. SA indices computed using GPs are built using the squared-exponential kernel using the algorithm in [67], which can be used for any set of independent random variables, and provide estimates of the error (in the form of variance) for each index computed. The expressions for these quantities for GP are much more complicated and so omitted here for brevity. SA indices can also be computed for Lagrange polynomial-based sparse grids by transforming them to PCE using [10]. However, due to the nature of the construction of the sparse grid, error estimates cannot be computed.

Finally, for high-dimensional smooth functions, the Morris screening method can also be used to rank model inputs in terms of their impact on model outputs. This method perturbs variables one-at-a-time through the input space and provides sensitivity indices akin to finite difference based derivatives averaged over the input space. This method can quickly identify highly insensitive variables but it often does not provide information that can be easily used to guide other types of probabilistic model analyses.

3.2.1 Benchmark demonstration

The following code snippet shows how to compute and plot the sensitivity indices using the GP surrogate we have already constructed. Figure 3 contains the plots generated by PyApprox . The confidence bands on each indices allows one to determine the confidence with which the sensitivity of each parameter can be ranked. For example, it is clear that the variable z_1 has the largest main-effect, however one cannot confidently rank the other two variables, because thier confidence intervals overlap significantly.

```
>>> sa_result = run_sensitivity_analysis("surrogate_sobol", approx_result.approx,
    inv_benchmark.variable)
>>> plot_sensitivity_indices(sa_result)
```

3.3 Bayesian inference

When observational data are available, that data should be used to inform prior assumptions of model uncertainties. This so-called inverse problem, which seeks to estimate uncertain parameters from measurements

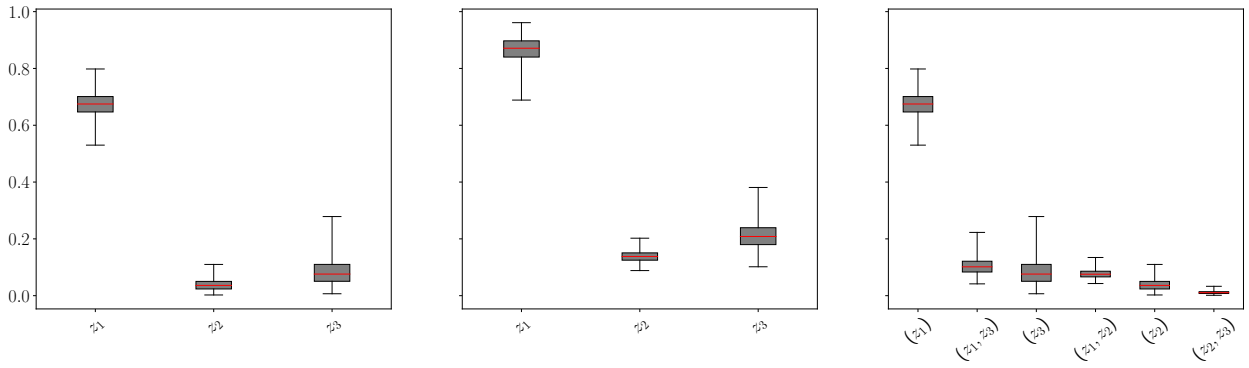


Figure 3: Main effect (left), total effect (middle), and Sobol (right) sensitivity indices obtained from the GP. The red lines in the box plots represent the median value, the bottom and top are the first and third quartiles, and the whiskers cover outliers.

or observations, is usually ill-posed such that the parameters are non-identifiable [29]. Many different realizations of parameter values may be consistent with the data. The lack of a unique solution can be due to the non-convexity of the parameter-to-QoI map, lack of data, and model structure and measurement errors.

Deterministic model calibration is an inverse problem that seeks to find a single parameter set that minimizes the misfit between the measurements and model predictions. A unique solution is found by simultaneously minimizing the misfit and a regularization term which penalizes certain characteristics of the model parameters. In the presence of uncertainty we typically do not want a single optimal solution, but rather a probabilistic description of the extent to which different realizations of parameters are consistent with the observations. Bayesian inference [48] can be used to define a posterior density for the model parameters z given observational data $y = (y_1, \ldots, y_{n_y})$.

Bayes Theorem describes the probability of the parameters conditioned on the data as proportional to the conditional probability of observing the data given the parameters multiplied by the probability of observing the data, that is

$$\pi_(z \mid y) = \frac{\pi(y|z)\pi(z)}{\int_\Gamma \pi(y|z)\pi(z)dz} \tag{5}$$

PyApprox assumes that $y = m(z) + \eta$ where $\eta \sim N(0, \Sigma_{\text{noise}})$ is normally distributed noise so that the likelihood $\pi(d \mid z)$ of observing the data given a realizations of the parameter z is

$$\pi(d|z) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}_{\text{noise}}|}} \exp\left(-\frac{1}{2}(m(z) - y)^{\mathrm{T}} \boldsymbol{\Sigma}_{\text{noise}}^{-1} (m(z) - y)\right)$$

In special limited situations the posterior density can be evaluated analytically, but in general this is difficult and the posterior must be characterized by samples drawn from the posterior. **PyApprox** provides two main classes tools to draw posterior samples. If using a multivariate Gaussian prior and a linear(ized) model, the first class leverages properties of linear-Gaussian inference to approximate very-high-dimensional model parameters. The second class of methods uses Markov Chain Monte Carlo (MCMC) to draw samples from the posterior distribution. The available methods are summarized in Table 3. The first class of methods are not exact when the model is nonlinear whereas the second class will sample from the true posterior (column labeled Exact in Table 3). General rules of thumb about the number of parameters that each method can handle are listed in the Dimensionality column of Table 3.

3.3.1 Large-scale methods for linear models and Gaussian priors

Computing the covariance of the posterior for linear(ized) Gaussian models is conceptually straightforward; it satisfies

$$\Sigma_{\text{post}} = \left(H_{\text{misfit}} + \Sigma_{\text{prior}}^{-1}\right)^{-1}$$

Table 3: Bayesian inference methods in PyApprox . The number of parameters that can be handled by each method is summarized in the Dimensionality column. The column labeled Exact states whether draws samples from the true posterior (exact) or an approximation of it.

Method	Dimensionality	Exact	Multi-Modal	Refs.
Laplace	$O(10^6)$	✗	✗	[74]
Gaussian Bayesian Networks	$O(10^5)$	✗	✗	[74]
SMCMC	$O(10)$	✓	✓	[80, 58]
NUTS	$O(10^2)$	✓	✗	[80, 35]

where $H_{\text{misfit}} = (\nabla m)^\top \Sigma_{\text{noise}}^{-1} \nabla m$ for linear parameter-observable maps (and when neglecting second-order terms for linearized nonlinear maps).[4] However, using this formula is intractable for very high-dimensional maps due to the need to invert very large dense matrices. Following [37], PyApprox makes the construction of the covariance tractable by performing a low-rank update of priors based upon a low rank approximation of the prior-preconditioned Hessian misfit $\Sigma_{\text{prior}} H_{\text{misfit}}$ obtained using randomized singular value decomposition; this algorithm requires Jacobians of the parameter-to-observable map. The resulting Gaussian approximation will be accurate when the parameter-to-observable map is weakly nonlinear over the support of the posterior, and when many parameter directions are weakly informed by the observational data such that the prior dominates in those directions. The latter property can be checked by looking at the eigen-spectrum of the low-rank approximation of $\Sigma_{\text{prior}} H_{\text{misfit}}$. Another implementation of this algorithm can also be found in the Hippylib software package [93].

Gaussian Bayesian networks [50] can also be constructed using PyApprox for high-dimensional linear(ized) Gaussian models. These methods are highly effective when using sparse Gaussian prior covariances (or inverses) that are encoded by conditional independencies.

3.3.2 Markov Chain Monte Carlo methods

For lower-dimensional problems that are not weakly nonlinear it can be advantageous to sample from the posterior using Markov Chain Monte Carlo (MCMC) methods. Although a closed form expression for the posterior density is available (5), using this expression is often intractable due to the need to compute the integral in the denominator. Instead MCMC methods draw samples from the posterior which is often all that is needed for subsequent model analysis, e.g. to compute statistics such as mean and variance of the uncertainty in model predictions.

MCMC methods draw samples from a proposal distribution so that each sample only depends on the previous sample. Samples are either accepted or rejected based on acceptance criteria that compare successive samples with respect to the unnormalized posterior distribution, e.g. the numerator of (5). The efficiency of MCMC depends on the proposal distribution. PyApprox provides a lightweight wrapper of the PyMC3 package [80] that supports various types of MCMC algorithms. The wrapper provides access to the metropolis, NUTS and Sequential Monte Carlo algorithms MCMC samplers in PyMC3. If the user can compute analytical derivatives of the observation model, then NUTS should be used. Sequential Monte Carlo is useful when the posterior is multi-modal. We refer the reader to the PyMC3 documentation for further advice.

3.3.3 Benchmark demonstration

The following code shows how to use PyApprox to draw samples from the posterior distribution of the KLE coefficients of our advection-diffusion model. If one only wants to produce a point estimate of the optimal KLE parameters, the map point, which is also computed, is often considered the most representative point. For linear models and Gaussian priors, the MAP has close ties with the optimal point obtained using Tikhonov regularization to penalize the defiviation of the optimal point from the prior mean [86].

[4] Linearizing nonlinear models is akin to using a quadratic approximation of the negative log-likelihood

```
>>> algorithm, npost_samples, njobs = "nuts", 100, 1
>>> loglike = partial(loglike_from_negloglike, approx)
>>> mcmc_variable = MCMCVariable(inv_benchmark.variable, loglike, algorithm, njobs=njobs,
↪  loglike_grad=True)
>>> post_samples = mcmc_variable.rvs(npost_samples)
>>> map_sample = mcmc_variable.maximum_aposteriori_point()
>>> mcmc_variable.plot_2d_marginals(nsamples_1d=30, plot_samples=[[post_samples, {}],
↪  [map_sample, {"c": "k", "marker": "X", "s": 100}]])
```

Here we use the GP surrogate instead of the spectral collocation model to evaluate the negative log-likelihood, but one can wasily use the numerical model if one is willing to incur a substantial increase in the computational time required to draw the posterior samples. Figure 4 plots the 1D and 2D marginals of the unnormalized posterior overlaid with samples from the posterior obtained using NUTS MCMC. The MAP is depicted with a cross. Plotting the marginals is only computationally tractable when using the GP surrogate.

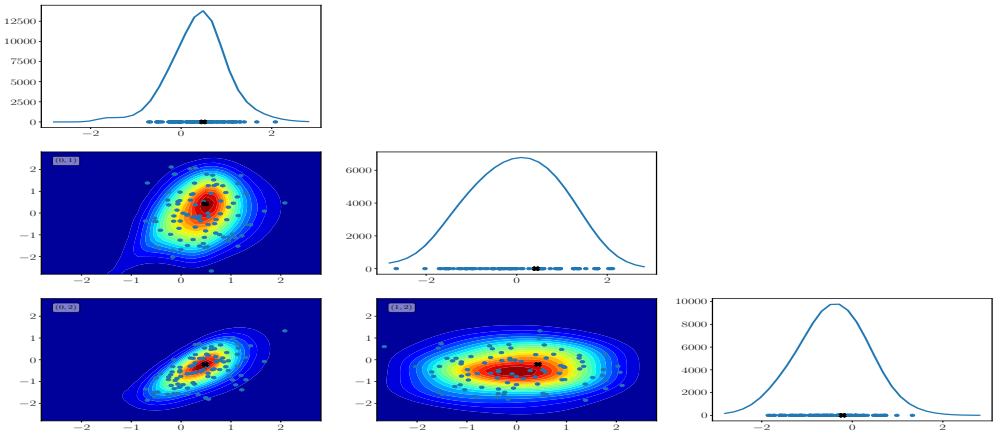


Figure 4: 1D and 2D marginals of the unnormalized posterior overlaid with samples from the posterior obtained using NUTS MCMC. The MAP is depicted with a cross. The tuple in the upper left corner of the contour plots indicates the active dimensions that have not been marginalized. The first index is the active dimension of the 1D marginal in the same column.

3.4 Experimental design

Bayesian inference provides a means to condition the uncertainty in model parameters on observational data. However, not all observational data reduces the uncertainty in the model parameters or predictions equally. Experimental design can be used to determine where and when to collect observations to reduce uncertainty in model parameters and predictions. The locations and timing of observations is referred to as the design.

The simplest experimental design strategies are so called space-filling or low-discrepancy designs that distribute the design locations "evenly" over the design space. PyApprox currently implements two such designs, called Halton sequences [31] and Sobol sequences [47]. While typically better than simply randomly choosing design locations, these algorithms do not take into account the properties of the process generating the observational data. Fortunately, the information content and cost-effectiveness of space-filling designs can often be significantly improved using optimal experimental design (OED) strategies that use numerical models to predict the anticipated information content of different experimental designs. OED methods can be broadly characterized into two main classes. Those that utilize linear(ized) numerical models and those that use nonlinear models. PyApprox implements instances of both classes. The methods available are listed in Table 4.

Table 4: Experimental design methods in PyApprox .. The number of parameters that can be handled by each method is summarized in the Dimensionality column.

Method	Model-informed	Dimensionality	Refs.
Halton Seq.	✗	$O(10^3)$	[31]
Sobol Seq.	✗	$O(10^3)$	[47]
Fisher OED	✓	$O(10)$	[2, 51]
Bayesian OED	✓	$O(10)$	[77, 96]

3.4.1 Linear OED

Let ϵ be mean zero Gaussian noise, with variance σ^2, and x parameterizes how to collect observations, e.g. spatial locations. Then, given noisy data (possibly heteroscedastic) and a model that satisfies

$$y(x) = m(x;\theta) + \eta(x)\epsilon, \tag{6}$$

the first class of methods focus on selecting experiments when the unknown parameters θ of the model are estimated using M-estimation, i.e. by solving

$$\min_\theta \frac{1}{L}\sum_{l=1}^{L} e(y_l - m(x^{(l)};\theta)) \tag{7}$$

using a vector of observations $y \in \mathbb{R}^L$ collected at a finite set of inputs $\mathcal{X}_L = \{x^{(l)}\}_{l=1}^L$ and an error function e. Least squares and quantile regression are both examples of M-estimation. In this setting OED can be very beneficial because the accuracy of the recovered θ depends on the location of the set $\mathcal{X}_L$, which may include multiple repetitions of the same x.

PyApprox implements a candidate-based OED that selects a set of design points from a set of P candidate (support) points $\Xi = \{\xi^{(i)}\}_{i=1}^P$. Letting $r_i, i = 1, \cdots, P$ be an integer which specifies the number of times the ith support point $\xi^{(i)}$ is chosen in an M-point experimental design,PyApprox seeks an experimental design as a (mathematical) measure on the set of support points, i.e.

$$\mu_M = \begin{Bmatrix} \xi^{(1)} & \xi^{(2)} & \cdots & \xi^{(P)} \\ r_1/M & r_2/M & \cdots & r_P/M \end{Bmatrix} \tag{8}$$

where $\sum_{i=1}^P r_i = M$. Generally speaking, the measure μ is chosen to optimize a function $\Psi(\mu)$ which quantifies the optimality of a design for the chosen estimation procedure, that is

$$\min_{\mu\in\Omega} \Psi(\mu), \tag{9}$$

where Ω is the set of admissible design measures.

For linear models, i.e. $m(x;\theta) = F(\mathcal{X})\theta$ (or linearized models $F(\mathcal{X}) = \nabla m(\mathcal{X}),\theta)$) the vast majority of design criteria used to generate OED are based upon the optimality criteria Ψ that are functionals acting on the Fisher information matrix $F(\mathcal{X}(\mu))^T F(\mathcal{X}(\mu))$. PyApprox implements the well known D optimality criterion, given by

$$\Psi(\mu) = \sigma^2 |\left(F(\mathcal{X})^T F(\mathcal{X})\right)^{-1}|$$

for homoscedastic noise, i.e. $\eta(x) = 1$. PyApprox also implements the well known A, C, I and G optimality criteria. The A and D optimality criteria focus on reducing uncertainty in the model parameters while the others focus on reducing uncertainty in predictions made using the updated parameters. For example, for homoscedastic noise, the I-optimal criterion is

$$\Psi(\mu) = \int_{X'} \sigma^2 \nabla g(x',\theta)^T \left(F(\mathcal{X})^T F(\mathcal{X})\right)^{-1} \nabla g(x',\theta)\, \mathrm{d}x',$$

where $g(x',\theta)$ is a model dependent on the parameters θ of the observational model m (but perhaps different from m) that makes predictions at a set of points $x' \in X'$ that are often but not always the same as the candidate design points.

Linear OED methods can be effectively applied with nonlinear models when good initial estimates of the model parameters are available. The resulting designs are considered locally optimal. Even though more globally optimal designs can be found in PyApprox using minimax designs that compute the design minimizing the worst case criteria over a set of discretely chosen parameter values, it is often better to use Bayesian OED strategies if using nonlinear models.

We also remark that Linear OED can only be used to select designs that consist of more points than the number of parameters θ. There is also no good way to place a limit on the number of design points chosen. Once PyApprox computes a design measure, the set of unique points used to estimate the parameters of the linear model is found using $\mathcal{X}_L = \{\xi^{(i)} \in \Xi \mid r_i > \delta,\ i = 1, \dots, P\}$ where δ is a user-defined threshold to limit the impact of numerical noise.

3.4.2 Nonlinear OED

When performing OED with a nonlinear model or when the user wants to impose a limit on the number of observations, PyApprox 's Bayesian nonlinear experimental design [77, 36, 96], should be used. This method uses the numerical model, via the relationship in (6), to predict the observational data that will be seen when an observation is collected. Specifically, given a prior density $p(\theta)$, PyApprox implementation of Bayesian OED maximizes the expected utility

$$\mathcal{X}^\star = \underset{\mathcal{X} \in \mathcal{X}_{\text{cand}}}{\operatorname{argmax}} \mathbb{E}_{\mathcal{Y}} \left[u(\mathcal{X}, y, \theta) \right]$$

to find a design $\mathcal{X}^\star$ for a set of possible locations $\mathcal{X}_{\text{cand}}$. The utility measures some notion of the information gained or reduction in uncertainty achieved by collecting observations. We take an expectation of all possible data because we do not know exactly what data will be collected and want to be 'good' on average.

Various utility functions can be used for OED. The most common is the expected Kullback-Leibler (KL) divergence that quantifies the difference between the prior and posterior for a given realization of the observations. Averaging over all possible realizations yields

$$\mathbb{E}_y \left[u(\mathcal{X}, y, \theta) \right] = \int_{\mathcal{Y}} \int_{\Theta} \log \left[\frac{p(\theta \mid y, \mathcal{X})}{p(\theta)} \right] p(\theta \mid y, \mathcal{X}) p(y \mid \mathcal{X}) \, \mathrm{d}\theta \, \mathrm{d}\theta.$$

Maximizing the KL divergence is equivalent to Bayesian D-optimal design in the case of linear parameter to observable maps. When estimating the expected KL utility, we use the double-loop approach in [77], such that

$$\widehat{U}(\mathcal{X}) = \frac{1}{N} \sum_{n=1}^{N} \log \left[p(y_n \mid \theta_n, \mathcal{X}) \right] - \log \left[\widehat{p}(y_n \mid \mathcal{X}) \right].$$

Here $(\theta_n, y_n), n = 1 \dots, N$ are samples from $p(y, \theta \mid \mathcal{X})$ obtained by first sampling θ_n from $p(\theta)$ and then sampling y_n from $p(y \mid \theta_n, \mathcal{X})$, and $\widehat{p}(y_n \mid \mathcal{X})$ is typically a sample-based estimate of $p(y_n \mid \mathcal{X})$ given by $\sum_{m=1}^{M} p(y_n \mid \theta_{nm}, \mathcal{X})$ where θ_{nm} are M samples that can be generated independently of the pairs (y_n, θ_n).

In many model analyses one is not interested explicitly in estimating the model parameters but rather estimating the impact of their uncertainty on predictions. In this case it is more advantageous to use utility criteria that target the prediction uncertainty, i.e.

$$U(\mathcal{X}) = \int_{\Theta} D(f(\theta)) p(\theta \mid y) \, d\theta$$

where D is a deviation measure that quantifies the uncertainty in the prediction, e.g. standard deviation. In this setting PyApprox also uses a double-loop MC approach for estimating the expected utility. This can require a large number of model evaluations. Thus combining OED with the surrogates in PyApprox is often necessary.

3.4.3 Benchmark demonstration

The following code can be used to greedily select an experimental design using Bayesian OED for nonlinear models.

```
>>> design_candidates = inv_benchmark.mesh.mesh_pts[:, inv_benchmark.obs_indices]
>>> oed = get_bayesian_oed_optimizer("kl_params", design_candidates, inv_benchmark.obs_fun,
    noise_stdev, benchmark.variable)
>>> oed_results = []
>>> ndesign = 3
>>> for step in range(ndesign):
>>>     results_step = oed.update_design()[1]
>>>     oed_results.append(results_step)
```

Figure 5 depicts the best three design locations selected using Bayesian OED with the expected K-L utility from the 10 (red squares) originally-used for inferring the KLE models in our advection diffusion example. In this case OED selected one location above the source and two locations down steam (given the specified advection field) of the source. This choice matches intuition for this example because the other candidate design points are at locations where the concentration of the tracer is typically very small and thus influenced little by changes in the KLE parameters. However, such intuition is often not available or two qualitative to be useful which necessitates OED.

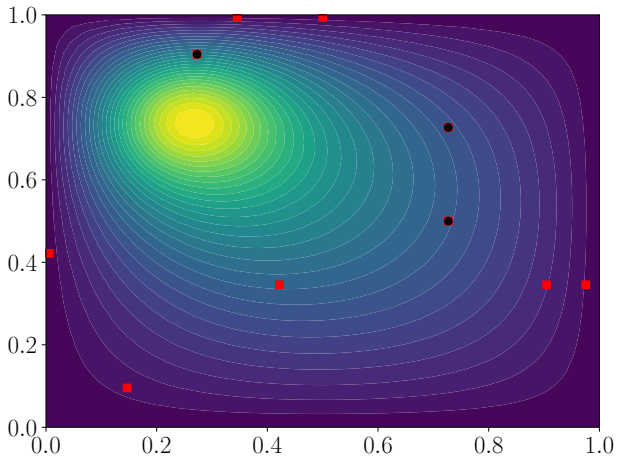


Figure 5: Concentration $c(x)$ for a realization of the diffusivity field overlaid with the candidate design points (red squares) and the design points selected by OED (black circles).

3.5 Multi-fidelity analysis

For many practical applications, a number of viable models of varying cost and accuracy may be available to simulate a physical system of interest. These models may use different simplifying assumptions about the system and/or arise from using different values of hyper-parameters that control the numerical accuracy of the model - e.g., mesh size, time step, convergence tolerance of a nonlinear solver, and so forth - that can be used to simulate the component with varying accuracy and cost. Model analysis methods that leverage only one model or solver setting are referred to as single-fidelity methods, whereas approaches that leverage multiple models and settings are called multi-fidelity methods. When multiple models were available, multi-fidelity methods have been repeatedly used to reduce the cost of single-fidelity model analyses by orders or magnitude.

PyApprox supports two types of multi-fidelity analyses, building surrogates and estimating statistics of prediction uncertainty. The methods implemented can significantly reduce the computational cost of these tasks compared to traditional single-fidelity approaches that just use one model. This reduction in computational cost is achieved by enriching a small number of the highest-fidelity simulations, used to maintain predictive accuracy, with larger numbers of simulations from the lower-fidelity to allow greater exploration of the model input space. The effectiveness of multi-fidelity approaches depends on the ability

to identify and exploit relationships among models within the ensemble. Most existing approaches focus on exploiting a hierarchy of models of increasing fidelity, with varying physics and/or numerical discretizations, such that $\|f_{\alpha'} - f\| \leq \|f_\alpha - f\|$ in some suitable norm if $\alpha' \geq \alpha$ and f is the highest-fidelity model. However PyApprox provides methods that do not require a strict hierarchy. Table 5 list all the methods implemented in PyApprox .

Table 5: Multi-fidelity analysis techniques implemented in PyApprox . The first three rows are used to estimate statistics. The remaining are used to build surrogates. The column labeled Auto Sample Allocation states whether each method can automatically allocated samples to each model fidelity. A cross indicates that the method can only leverage a fixed data set. The Ensemble column list the assumptions on the relationships between models.

Method	Ensemble	Auto Sample Allocation	Refs.
MLMC	1D Hierarchy	✓	[23]
MFMC	Cost/Correlation constrained	✓	[73]
ACV	None	✓	[25, 6]
Multi-level GP	1-D Hierarchy	✗	[49]
Multi-level SG	1-D Hierarchy	✓	[89]
Multi-index SG	N-D Hierarchy	✓	[30, 41, 45]
MFNets	Directed Acyclic Graph	✗	[27]

3.5.1 Multi-fidelity statistical estimation

Often the support of decision making requires estimating uncertainty in model model predictions, for example computing the expected response of model QoI to uncertainty in model parameters. Given the ability to sample from a random variable, either from its prior or from its posterior conditioned on available data using MCMC, Monte Carlo-based multi-fidelity methods can be used compute the expectations of the output of a function (model), i.e.

$$Q_\alpha = \int_\Gamma f_\alpha(z)\rho(z)\,\mathrm{d}z$$

where α is a multi-index used to denote the fidelity of the model being used, e.g. each entry dictates the value of a different discretization hyper-parameter. Here we assume that $\alpha = 0$ denotes the highest-fidelity model. When using a single model we can approximate the integral Q_α using MC quadrature by drawing N random samples of z from ρ and evaluating the function at each of these samples to obtain the data pairs $\{(z^{(n)}, f_\alpha^{(n)})\}_{n=1}^N$, where $f_\alpha^{(n)} = f_\alpha(z^{(n)})$, and computing

$$Q_{\alpha,N} = N^{-1}\sum_{n=1}^{N} f_\alpha^{(n)}$$

The mean squared error (MSE) of this single-fidelity MC estimator can be edecomposed into two terms

$$\mathbb{E}\left[\left(Q_{\alpha,N} - \mathbb{E}\left[Q\right]\right)^2\right] = \underbrace{N^{-1}\mathbb{V}\left[Q_\alpha\right]}_{I} + \underbrace{\left(\mathbb{E}\left[Q_\alpha\right] - \mathbb{E}\left[Q\right]\right)^2}_{II};$$

a so called stochastic variance (I) and a deterministic bias (II). The first term is the variance of the MC estimator which comes from using a finite number of samples. The second term is due to using an approximation of f. These two errors should be balanced, but in the vast majority of all MC analyses a single model f_α is used and ad-hoc choices of α, e.g. mesh resolution, are made a priori to balance the bias and variance. For example often a low-resolution model is used to reduce the stochastic variance at the cost of increasing the deterministic bias.

Multi-fidelity sampling methods can be used to balance the bias and variance of a MC estimator. Various multi-fidelity estimators have been developed, for example Multi-level Monte Carlo (MLMC) [23] and Multi-fidelity Monte Carlo (MFMC) [73], however here we focus on approximate control variate (ACV) estimators which include the aforementioned estimators as special cases.

Consider an ensemble of S models $\{f_s(z)\}_{s=1}^S$ parameterized by the same input variables z. If we assume that the highest-fidelity model is the truth, i.e. has zero bias, then ACV methods can be used to reduce the variance of the estimator and thus its MSE. Given M lower-fidelity models with sample ratios $r_\alpha >= 1$, for $\alpha = 1, \ldots, M$, the ACV estimator of the mean of the high-fidelity model $Q_0 = \mathbb{E}[f_0]$ is

$$Q_0 \approx Q^{\mathrm{ACV}} = Q_{0,\mathcal{Z}_{0,1}} + \sum_{\alpha=1}^{M} \eta_\alpha \left(Q_{\alpha,\mathcal{Z}_{\alpha,1}} - \mu_{\alpha,\mathcal{Z}_{\alpha,2}}\right) = Q_{0,\mathcal{Z}_{0,1}} + \sum_{\alpha=1}^{M} \eta_\alpha \Delta_{\alpha,\mathcal{Z}_{\alpha,1},\mathcal{Z}_{\alpha,2}} = Q_{0,N} + \eta\Delta$$

Here $\Delta = [\Delta_{1,\mathcal{Z}_{1,1},\mathcal{Z}_{1,2}}, \ldots, \Delta_{M,\mathcal{Z}_{M,1},\mathcal{Z}_{M,2}}]^\top$, $\mathcal{Z}_{\alpha,1}$, $\mathcal{Z}_{\alpha,2}$ are sample sets that may or may not be disjoint, and $\mu_{\alpha,\mathcal{Z}_{\alpha,2}}$ are MC estimates of the low-fidelity model means computed using the samples in $\mathcal{Z}_{\alpha,2}$. Specifying the exact nature of these sets $\mathcal{Z}_{\alpha,1}$, $\mathcal{Z}_{\alpha,2}$, including their cardinality, can be used to design different ACV estimators and are the major difference between MLMC, MFMC and other ACV estimators.

The control variate weights $\eta = [\eta_1, \ldots, \eta_M]^\top$ that produce the minimum variance are given by the closed form expression $\eta = -\mathbb{C}\mathrm{ov}[\Delta, \Delta]^{-1}\,\mathbb{C}\mathrm{ov}[\Delta, Q_0]$. Using these weights, the variance of the ACV estimate of the high-fidelity mean is

$$\mathbb{V}\left[Q^{\mathrm{ACV}}\right] = \left(1 - \mathbb{C}\mathrm{ov}[\Delta, Q_0]^T \frac{\mathbb{C}\mathrm{ov}[\Delta, \Delta]^{-1}}{\mathbb{V}[Q_0]} \mathbb{C}\mathrm{ov}[\Delta, Q_0]\right) \frac{\mathbb{V}[f_0]}{N_0}$$

where $N_0 = |\mathcal{Z}_{0,1}|$ is the number of evaluations of the highest-fidelity model. This variance is a function purely of the number of samples allocated to evaluating each model and the how those samples overlap, i.e. the way in which $\mathcal{Z}_{\alpha,1}$, $\mathcal{Z}_{\alpha,2}$ are constructed. **PyApprox** uses this estimate of variance to optimally allocate the number of samples to each model in a manner that minimizes the variance (MSE) for a specified total computational cost that is distributed to the tasks of evaluating all models. The method in [6] is used to iterate over a large set of choices of $\mathcal{Z}_{\alpha,1}$, $\mathcal{Z}_{\alpha,2}$. This avoids the user needing to specify whether to use MFMC, MLMC, or other ACV estimators, instead providing an automated means of selecting the best estimator for given estimates of the covariance between models, which is determined with a small pilot study, e.g. 10 evaluations of each model.

3.5.2 Multi-fidelity surrogates

PyApprox also supports various multi-fidelity methods for constructing surrogates. These methods again balance the bias introduced in the surrogate by using training data from lower-fidelity models with the approximation error introduced by using only a finite amount of training data. Specifically, **PyApprox** implements multi-level GPs [49], multilevel collocation (sparse-grids) [89], and multi-index collocation [30, 41, 45]. These methods are extensions of the single-fidelity surrogates presented in Section 3.1 and assume that lower-fidelity models can be ordered in a 1D or multi-dimensional hierarchy. We refer the reader to the online documentation of **PyApprox** and the citations listed in Table 5 for more details. If the model-ensemble available does not admit the strict hierarchies needed by the aforementioned methods, then the use of the MFNets algorithm [27] in **PyApprox** is recommended.

3.5.3 Multi-level Gaussian processes

Given a hierarchy of models, **PyApprox** constructs multi-level GPs using the algorithm in [49]. Specifically, adopting the GP convention that the models are indexed from lowest to highest fidelity, the s-th model is assumed to satisfy

$$f_s(x) = \rho_{s-1}(z) f_{s-1}(z) + \delta_s(z) \quad s > 0$$

where the discrepancies δ_s are independent GPs and $f_0(x) = \delta_0(z)$. When two models are available this leads to the following redefinition of the covariance matrices and vectors used to compute the single-fidelity GP in (3):

$$A_{\mathcal{Z}} = \begin{bmatrix} C_1(\mathcal{Z}_1, \mathcal{Z}_1) & \rho_1 C_1(\mathcal{Z}_1, \mathcal{Z}_2) \\ \rho_1 C_1(\mathcal{Z}_2, \mathcal{Z}_1) & \rho_1^2 C_1(\mathcal{Z}_2, \mathcal{Z}_2) + C_2(\mathcal{Z}_2, \mathcal{Z}_2) \end{bmatrix} \qquad t(z)^T = \left[\rho_1 C_1(z, \mathcal{Z}_1),\ \rho_1^2 C_1(z, \mathcal{Z}_2) + C_2(z, \mathcal{Z}_2)\right]$$

where $\mathcal{Z} = \{\mathcal{Z}_1, \mathcal{Z}_2\}$ and C_s is the covariance kernel used for the discrepancies at the s-th level. Similar covariance matrices can be constructed when using more than three models, however the computational cost of computing the multi-fidelity GP (after training data has been collected) grows quickly with the number of models and number of samples used per model. If N samples are used for each of the S models then the computational cost of building the GP will be $O(N^3S^3)$.

The implementation of multi-level GPs in PyApprox is an extension of the PyApprox wrapper of Scikit-learn used for single fidelity GPs. Given a definition of a multi-level kernel matrix, as given above for two models, PyApprox uses the same infrastructure to learn the hyper-parameters of the multi-level GP, which now include the scaling coefficients ρ_s, and compute the mean and pointwise variance of the GP. Currently multi-level GPs can only be constructed in a regression context, i.e. from a pre-specified set of samples for each model. An adaptive sample allocation strategy similar to the single fidelity GP algorithms is currently under development.

3.5.4 Multi-level and multi-index collocation

Multi-level collocation and multi-index collocation can respectively be used when one or multiple hyper-parameters control the discretization and thus accuracy of a numerical model. PyApprox constructs such multi-fidelity approximations using an extension of single fidelity sparse grids (Section 3.1.2). Letting f (without a subscript) denote the true model output without discretization error, the multi-index surrogate[5] is given by

$$f(z) \approx \sum_{\alpha,\beta\in\mathcal{I}} c_{\alpha,\beta} f_{\alpha,\beta}(z).$$

Here $f_{\alpha,\beta}$ is a tensor-product interpolant constructed using evaluations of the model f_α at the samples dictated by the multi-index β. An extension of the adaptive algorithm used to build single-fidelity sparse grids can be effectively used to determine the index set $\mathcal{I}$ that assigns evaluations of the different model fidelities f_α in a manner that minimizes error in the surrogate subject to a total computational budget. Provided the deterministic bias of the numerical model decreases as the entries of α increase, this algorithm builds a highly accurate sparse grid using only a small number of high-fidelity model evaluations combined with a much larger number of lower-fidelity evaluations. Even when the number of evaluations of the high-fidelity model is typically smaller, the computational cost of collecting these evaluations typically exceeds the cost required to collect the much more numerous low-fidelity evaluations, highlighting the value of multi-fidelity surrogate methods.

3.5.5 MFNets

The multi-level (multi-index) surrogates above require a (possibly multi-dimensional) hierarchy) of numerical models. However, for some model analyses the available models do not admit such a hierarchy. In this setting, MFNets provides a flexible means to encode and exploit prior knowledge about the relationship between models. MFNets expresses the multi-fidelity surrogate as a directed acyclic graph (DAG) of models of varying fidelities. Nodes represent models and edges the connections between models. Multi-level surrogates are a special instance of MFNets that is obtained when the DAG is a chain of models.

The roots of the graph (the lowest fidelity models) are represented by simple discrepancies $\delta_s(z)$ and the remaining nodes are functions of the low-fidelity nodes (parents) directly connected to them and the inputs z, i.e.

$$f_s(z) = \Delta_s(z, \{f_j(z)\}_{j\in\mathrm{pa(s)}})$$

where pa(s) denotes all indices of the the low-fidelity model directly connected to the s-th model. There are little restrictions theoretically on the form of Δ, however PyApprox currently only supports

$$f_s(z) = \sum_{j\in\mathrm{pa(s)}} \rho_{js}(z) f_j(z) + \delta_s(z)$$

[5] the multi-level surrogate is just a special case when the multi-index α has only one entry

where $\rho(z)$ and $\delta_s(z)$ are multivariate PCEs.[6].

Although each PCE is linear in its coefficients the MFNets surrogate is nonlinear in all its unknowns. Consequently, PyApprox uses gradient-based maximum likelihood estimation of the unknowns, employing analytical gradients of the objective obtained using backwards differentiation to reduce the computational cost of training the surrogate once data has been collected. Like multi-level GPs, MFNets can currently only be used in a regression context. Adaptive sampling methods are currently being developed.

3.5.6 Benchmark demonstration

The following code demonstrates how to setup a pilot study for ACV that will predict the benefit of using ACV, relative to single fidelity MC using only the highest fidelity model, when estimating uncertainty in predictions made by the transient advection diffusion equation (1). Specifically, we set $T = 0.2$ and quantify the expected value (with respect to the posterior distribution of the KLE parameters) of the left QoI in (2) with the subdomain $\Gamma = [3/4, 1] \times [0, 1/4]$. Ten evaluations of eight possible model instances, corresponding to two different discretizations of the mesh in each physical directions and the time step size, are used to compute the correlation between the two models. The `multifidelity.get_best_models_for_acv_estimator()` function is then used to find the best subset of models that minimizes the ACV estimator variance (thus MSE, assuming the highest-fidelity model is unbiased).

```
>>> fwd_benchmark = setup_benchmark("multi_index_advection_diffusion",
↪  kle_nvars=inv_benchmark.variable.num_vars(), kle_length_scale=0.5, time_scenario=True)
>>> model = WorkTrackingModel(TimerModel(fwd_benchmark.model_ensemble), num_config_vars=1)
>>> npilot_samples = 10
>>> cov = multifidelity.estimate_model_ensemble_covariance(npilot_samples, post_samples,
↪  model, fwd_benchmark.model_ensemble.nmodels)[0]
>>> model_costs =
↪  model.work_tracker(np.asarray([np.arange(fwd_benchmark.model_ensemble.nmodels)]))
# make costs in terms of fraction of cost of high-fidelity evaluation
>>> model_costs /= model_costs[0]
>>> best_est, best_model_indices = (multifidelity.get_best_models_for_acv_estimator("acvgmfb",
↪  cov, model_costs, inv_benchmark.variable, 1e2, max_nmodels=3, tree_depth=4))
>>> best_est.allocate_samples(target_cost)
>>> print("Predicted variance", best_est.optimized_variance)
```

The left plot of Figure 6 depicts the relative variance of the MC and ACV estimators, normalized by the variance of the MC estimator using 10 samples, for different target costs measured in fractions of the computational cost of a single evaluation of the highest-fidelity model. The right plot, depicts the number of samples assigned to the two models found to minimize the ACV variance and that were used to generate the left plot. Because of the high-correlation between the high- and low-fidelity models many more evaluations of the less computationally expensive low-fidelity model are used. See the online tutorial for code to reproduce Figure 6.

4 Conclusions

Many model analyses require the repeated evaluation of computationally expensive numerical models. The central mission of PyApprox is to capture an eclectic and comprehensive range of research methods and algorithms for minimizing the number of model evaluations whatever the modeling objective in question. PyApprox consists of nine high-level modules, four dedicated to setting up model analyses and five for algorithms providing analysis tools. The variables and interface modules, respectively, provide tools: for characterizing prior probability distributions that describe the various sources of model uncertainty, and to interface with computationally expensive simulation models. The benchmarks and Partial Differential Equation (PDE) modules provide numerous challenge problems that can be used to verify, validate and assess the performance of model analyses algorithms. The five classes of model analysis tools include sensitivity

[6]Development of other forms of Δ_s are currently in development.

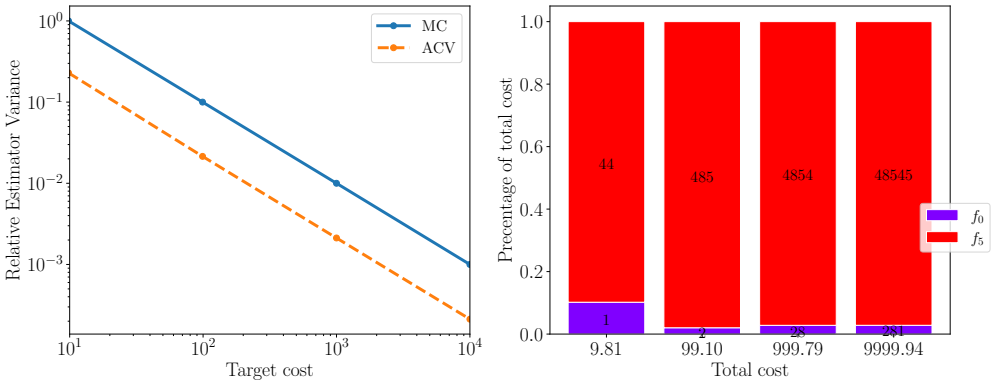


Figure 6: (Left) The variance of an ACV estimate of the mean QOI relative to Monte Carlo estimate. (Right) The computational cost (in fractions of a single high-fidelity evaluation) and the number of samples allocated to each model used by the ACV estimator.

analysis, Bayesian inference, experimental design; multi-fidelity forward propagation of uncertainties; and surrogate modeling (including multi-fidelity algorithms).

PyApprox has been used for a number of high-impact studies. For example, **PyApprox** was used to perform the first ever global sensitivity study of a fully coupled version 1 of the Energy Exascale Earth System Model (E3SM) [90]. **PyApprox** was able to determine the dominant sources of uncertainty in predictions of quantities characterizing changes in the arctic climate over a 75 year period, using only 139 model evaluations that took 10^6 CPU hours to generate. As another major example, multi-fidelity uncertainty quantification was used to compute the expected mass ice loss of Humboldt Glacier in Greenland under a likely climate scenario using 3 different physics models each with their own varying numerical resolutions [83]. **PyApprox** was able to identify a set of three model instances that reduced the computational cost of a single model analysis, which uses only the highest-fidelity model, by approximately an order of magnitude.

Future developments in **PyApprox** will focus on further strengthening its multi-fidelity capabilities, adding further PDE based benchmarks for comparing, verifying and validating methods for model analysis including those focused on hybrid machine learning/ physics-based dynamical models, and utilizing all of **PyApprox** 's existing tools to enable efficient optimization under uncertainty.

5 Acknowledgements

The author would like to thank all his co-authors that contributed to the development of the numerical algorithms in this paper. Special thanks go to Alex Gorodetsky for his work on multi-fidelity methods, Drew Kouri for his work on optimal experimental design, Akil Narayan for his work on surrogate methods, and Tim Wildey for his work on numerical solvers of PDES, who have had the largest impact on the novel algorithms implemented in **PyApprox** . While this work does implement many novel algorithms developed by the author and his co-authors, this work would also not have been possible without building off the amazing work of other researches across the globe.

The development of **PyApprox** was sponsored by the Sandia National Laboratories Laboratory Directed Research Development (LDRD) program, the US Department of Energy's Office of Advanced Scientific Computing Research, the US Department of Energy's Office of Biological and Environmental Research, and the Defense Advanced Research Projects Agency (DARPA).

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] B.M. Adams, L.E. Bauman, W.J. Bohnhoff, K.R. Dalbey, J.P. Eddy, M.S. Ebeida, M.S. Eldred, P.D. Hough, K.T. Hu, J.D. Jakeman, L.P. Swiler, and D.M. Vigil. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.3.1 theory manual. Technical Report SAND2011-9106, Sandia National Laboratories, Albuquerque, NM, 2013. URL: https://dakota.sandia.gov.

[2] A.C. Atkinson and A.N. Donev. *Optimum Experimental Designs*. Oxford University Press, 1992. URL: https://doi.org/10.1007/978-3-642-04898-2_434.

[3] Volker Barthelmann, Erich Novak, and Klaus Ritter. High dimensional polynomial interpolation on sparse grids. *Advances in Computational Mathematics*, 12(4):273–288, 2000. URL: https://doi.org/10.1023/A:1018977404843.

[4] Michael Baudin, Anne Dutfoy, Bertrand Iooss, and Anne-Laure Popelin. Openturns: An industrial software for uncertainty quantification in simulation, 2015. URL: https://openturns.github.io/openturns/1.16/index.html.

[5] Géraud Blatman and Bruno Sudret. Adaptive sparse polynomial chaos expansion based on least angle regression. *Journal of Computational Physics*, 230(6):2345–2367, 2011. doi:10.1016/j.jcp.2010.12.021.

[6] G.F. Bomarito, P.E. Leser, J.E. Warner, and W.P. Leser. On the optimization of approximate control variates with parametrically defined estimators. *Journal of Computational Physics*, 451:110882, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0021999121007774, doi:10.1016/j.jcp.2021.110882.

[7] Emanuele Borgonovo, Gordon B. Hazen, and Elmar Plischke. A common rationale for global sensitivity measures and their estimation. *Risk Analysis*, 36(10):1871–1895, 2016. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/risa.12555, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/risa.12555, doi:https://doi.org/10.1111/risa.12555.

[8] Mohamed Amine Bouhlel, John T. Hwang, Nathalie Bartoli, Rémi Lafage, Joseph Morlier, and Joaquim R.R.A. Martins. A python surrogate modeling framework with derivatives. *Advances in Engineering Software*, 135:102662, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0965997818309360, doi:https://doi.org/10.1016/j.advengsoft.2019.03.005.

[9] John P Boyd. *Chebyshev and Fourier spectral methods*. Courier Corporation, 2001. URL: https://link.springer.com/book/9783540514879.

[10] Gregery T Buzzard. Global sensitivity analysis using sparse grid interpolation and polynomial chaos. *Reliability Engineering and System Safety*, 107:82–89, 2012. URL: http://dx.doi.org/10.1016/j.ress.2011.07.011, doi:10.1016/j.ress.2011.07.011.

[11] Yang Cao, Shengtai Li, Linda Petzold, and Radu Serban. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint dae system and its numerical solution. *SIAM Journal on Scientific Computing*, 24(3):1076–1089, 2003. arXiv:https://doi.org/10.1137/S1064827501380630, doi:10.1137/S1064827501380630.

[12] Patrick R. Conrad and Youssef M. Marzouk. Adaptive smolyak pseudospectral approximations. *SIAM Journal on Scientific Computing*, 35(6):A2643–A2670, 2013. arXiv:https://doi.org/10.1137/120890715, doi:10.1137/120890715.

[13] Paul G Constantine. *Active subspaces: Emerging ideas for dimension reduction in parameter studies*. SIAM, 2015.

[14] Bert Debusschere, Khachik Sargsyan, Cosmin Safta, and Kenny Chowdhary. *Uncertainty Quantification Toolkit (UQTk)*, pages 1807–1827. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-12385-1_56.

[15] S. Dubreuil, M. Berveiller, F. Petitjean, and M. Salaün. Construction of bootstrap confidence intervals on sensitivity indices computed by polynomial chaos expansion. *Reliability Engineering & System Safety*, 121:263–275, 2014. URL: https://www.sciencedirect.com/science/article/pii/S0951832013002688, doi:https://doi.org/10.1016/j.ress.2013.09.011.

[16] Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani. Least angle regression. *The Annals of Statistics*, 32(2):407 – 499, 2004. doi:10.1214/009053604000000067.

[17] Martin Eigel, Oliver G. Ernst, Björn Sprungk, and Lorenzo Tamellini. On the convergence of adaptive stochastic collocation for elliptic partial differential equations with affine diffusion. *SIAM Journal on Numerical Analysis*, 60(2):659–687, 2022. arXiv:https://doi.org/10.1137/20M1364722, doi:10.1137/20M1364722.

[18] Gregory E Fasshauer. Positive definite kernels: past, present and future. *Dolomites Research Notes on Approximation*, 4:21–63, 2011. URL: http://www.math.iit.edu/~fass/PDKernels.pdf.

[19] Jonathan Feinberg and Hans Petter Langtangen. Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal of Computational Science*, 11:46–57, 2015. URL: https://www.sciencedirect.com/science/article/pii/S1877750315300119, doi:https://doi.org/10.1016/j.jocs.2015.08.008.

[20] Alan Genz. Testing multidimensional integration routines. In *Proc. of International Conference on Tools, Methods and Languages for Scientific and Engineering Computation*, page 81–94, USA, 1984. Elsevier North-Holland, Inc. URL: https://doi.org/10.5555/2837.2842.

[21] T. Gerstner and M. Griebel. Dimension-adaptive tensor-product quadrature. *Computing*, 71(1):65–87, SEP 2003. doi:{10.1007/s00607-003-0015-5}.

[22] R.G. Ghanem and P.D. Spanos. *Stochastic Finite Elements: A Spectral Approach*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[23] Michael B. Giles. Multilevel monte carlo methods. *Acta Numerica*, 24:259–328, 2015. URL: https://doi.org/10.1017/S096249291500001X.

[24] Dirk Gorissen, Ivo Couckuyt, Piet Demeester, Tom Dhaene, and Karel Crombecq. A surrogate modeling and adaptive sampling toolbox for computer based design. *Journal of Machine Learning Research*, 11(68):2051–2055, 2010. URL: http://jmlr.org/papers/v11/gorissen10a.html.

[25] A.A. Gorodetsky, G. Geraci, M.S. Eldred, and J.D. Jakeman. A generalized approximate control variate framework for multifidelity uncertainty quantification. *Journal of Computational Physics*, 408:109257, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0021999120300310, doi:10.1016/j.jcp.2020.109257.

[26] A.A. Gorodetsky and J.D. Jakeman. Gradient-based optimization for regression in the functional tensor-train format. *Journal of Computational Physics*, 374:1219 – 1238, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0021999118305321, doi:10.1016/j.jcp.2018.08.010.

[27] A.A. Gorodetsky, J.D. Jakeman, and G. Geraci. MFNets: data efficient all-at-once learning of multifidelity surrogates as directed networks of information sources. *Computational Mechanics*, 68(4):741–758, 2021. doi:10.1007/s00466-021-02042-0.

[28] Alex Gorodetsky, Sertac Karaman, and Youssef Marzouk. A continuous analogue of the tensor-train decomposition. *Computer Methods in Applied Mechanics and Engineering*, 347:59–84, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0045782518306133, doi:https://doi.org/10.1016/j.cma.2018.12.015.

[29] Joseph H.A. Guillaume, John D. Jakeman, Stefano Marsili-Libelli, Michael Asher, Philip Brunner, Barry Croke, Mary C. Hill, Anthony J. Jakeman, Karel J. Keesman, Saman Razavi, and Johannes D. Stigter. Introductory overview of identifiability analysis: A guide to evaluating whether you have

the right type of data for your modeling purpose. *Environmental Modelling & Software*, 119:418 – 432, 2019. URL: http://www.sciencedirect.com/science/article/pii/S1364815218307278, doi:10.1016/j.envsoft.2019.07.007.

[30] A. Haji-Ali, F. Nobile, L. Tamellini, and R. Tempone. Multi-index stochastic collocation for random pdes. *Computer Methods in Applied Mechanics and Engineering*, 306:95 – 122, 2016. URL: http://www.sciencedirect.com/science/article/pii/S0045782516301141, doi:10.1016/j.cma.2016.03.029.

[31] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, dec 1964. doi:10.1145/355588.365104.

[32] H. Harbrecht, J.D. Jakeman, and P. Zaspel. Cholesky-based experimental design for Gaussian process and kernel-based emulation and calibration. *Communications in Computational Physics*, 29(4):1152–1185, 2021. URL: http://global-sci.org/intro/article_detail/cicp/18644.html, doi:10.4208/cicp.OA-2020-0060.

[33] M. Hegland. Adaptive sparse grids. In K. Burrage and Roger B. Sidje, editors, *Proc. of 10th Computational Techniques and Applications Conference CTAC-2001*, volume 44, pages C335–C353, April 2003. URL: http://anziamj.austms.org.au/V44/CTAC2001/Hegl.

[34] Jon Herman and Will Usher. SALib: An open-source python library for sensitivity analysis. *The Journal of Open Source Software*, 2(9), jan 2017. doi:10.21105/joss.00097.

[35] Matthew D. Hoffman and Andrew Gelman. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15(47):1593–1623, 2014. URL: http://jmlr.org/papers/v15/hoffman14a.html.

[36] Xun Huan and Youssef M. Marzouk. Simulation-based optimal bayesian experimental design for nonlinear systems. *Journal of Computational Physics*, 232(1):288–317, 2013. URL: https://www.sciencedirect.com/science/article/pii/S0021999112004597, doi:https://doi.org/10.1016/j.jcp.2012.08.013.

[37] Tobin Isaac, Noemi Petra, Georg Stadler, and Omar Ghattas. Scalable and efficient algorithms for the propagation of uncertainty from data through inference to prediction for large-scale problems, with application to flow of the antarctic ice sheet. *Journal of Computational Physics*, 296:348–368, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0021999115003046, doi:https://doi.org/10.1016/j.jcp.2015.04.047.

[38] T. Ishigami and T. Homma. An importance quantification technique in uncertainty analysis for computer models. In *[1990] Proceedings. First International Symposium on Uncertainty Modeling and Analysis*, pages 398–403, 1990. doi:10.1109/ISUMA.1990.151285.

[39] J.D Jakeman and B. Debusschere. Surrogate models for mixed continuous and discrete input variables. *In preparation*, 2022.

[40] J.D. Jakeman, M. Eldred, and D. Xiu. Numerical approach for quantification of epistemic uncertainty. *Journal of Computational Physics*, 229(12):4648–4663, 2010. URL: http://www.sciencedirect.com/science/article/B6WHY-4YM7FRC-1/2/4dd64b16ceaef199ff2c8545106ea6e6, doi:DOI:10.1016/j.jcp.2010.03.003.

[41] J.D. Jakeman, M.S. Eldred, G. Geraci, and A. Gorodetsky. Adaptive multi-index collocation for uncertainty quantification and sensitivity analysis. *International Journal for Numerical Methods in Engineering*, 2019. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.6268, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.6268, doi:10.1002/nme.6268.

[42] J.D. Jakeman, M.S. Eldred, and K. Sargsyan. Enhancing ℓ_1-minimization estimates of polynomial chaos expansions using basis selection. *Journal of Computational Physics*, 289(0):18 – 34, 2015. URL: http://www.sciencedirect.com/science/article/pii/S0021999115000959, doi:http://dx.doi.org/10.1016/j.jcp.2015.02.025.

[43] J.D. Jakeman, A. Narayan, and T. Zhou. A generalized sampling and preconditioning scheme for sparse approximation of polynomial chaos expansions. *SIAM Journal on Scientific Computing*, 39(3):A1114–A1144, 2017. arXiv:https://doi.org/10.1137/16M1063885, doi:10.1137/16M1063885.

[44] John D. Jakeman, Fabian Franzelin, Akil Narayan, Michael Eldred, and Dirk Plfüger. Polynomial chaos expansions for dependent random variables. *Computer Methods in Applied Mechanics and Engineering*, 351:643 – 666, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0045782519301884, doi:10.1016/j.cma.2019.03.049.

[45] John D. Jakeman, Sam Friedman, Michael S. Eldred, Lorenzo Tamellini, Alex A. Gorodetsky, and Doug Allaire. Adaptive experimental design for multi-fidelity surrogate modeling of multi-disciplinary systems. *International Journal for Numerical Methods in Engineering*, 123(12):2760–2790, 2022. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.6958, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.6958, doi:https://doi.org/10.1002/nme.6958.

[46] John D. Jakeman, Drew P. Kouri, and J. Gabriel Huerta. Surrogate modeling for efficiently, accurately and conservatively estimating measures of risk. *Reliability Engineering & System Safety*, page 108280, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0951832021007523, doi:https://doi.org/10.1016/j.ress.2021.108280.

[47] Stephen Joe and Frances Y. Kuo. Remark on algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 29(1):49–57, mar 2003. doi:10.1145/641876.641879.

[48] J. Kaipio and E. Somersalo. *Statistical and Computational Inverse Problems*. Springer, 2005. URL: https://link.springer.com/book/10.1007/b138659.

[49] M. C. Kennedy and A. O'Hagan. Predicting the output from a complex computer code when fast approximations are available. *Biometrika*, 87(1):1–13, 2000. URL: http://www.jstor.org/stable/2673557.

[50] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[51] Drew P. Kouri, John D. Jakeman, and J. Gabriel Huerta. Risk-adapted optimal experimental design. *SIAM/ASA Journal on Uncertainty Quantification*, 10(2):687–716, 2022. arXiv:https://doi.org/10.1137/20M1357615, doi:10.1137/20M1357615.

[52] J. Laurenceau and P. Sagaut. Building efficient response surfaces of aerodynamic functions with kriging and cokriging. *AIAA Journal*, 46(2):498–507, 2008. arXiv:https://doi.org/10.2514/1.32308, doi:10.2514/1.32308.

[53] Pei-Ling Liu and Armen Der Kiureghian. Multivariate distribution models with prescribed marginals and covariances. *Probabilistic Engineering Mechanics*, 1(2):105–112, 1986. URL: https://doi.org/10.1016/0266-8920(86)90033-0, doi:https://doi.org/10.1016/0266-8920(86)90033-0.

[54] Stefano Marelli and Bruno Sudret. *UQLab: A Framework for Uncertainty Quantification in Matlab*, pages 2554–2563. 2014. URL: https://ascelibrary.org/doi/abs/10.1061/9780784413609.257, arXiv:https://ascelibrary.org/doi/pdf/10.1061/9780784413609.257, doi:10.1061/9780784413609.257.

[55] Youssef M. Marzouk, Habib N. Najm, and Larry A. Rahn. Stochastic spectral methods for efficient bayesian solution of inverse problems. *Journal of Computational Physics*, 224(2):560–586, 2007. URL: https://www.sciencedirect.com/science/article/pii/S0021999106004839, doi:https://doi.org/10.1016/j.jcp.2006.10.010.

[56] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando,

Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. doi:10.7717/peerj-cs.103.

[57] Giovanni Migliorati. Adaptive approximation by optimal weighted least-squares methods. *SIAM Journal on Numerical Analysis*, 57(5):2217–2245, 2019. arXiv:https://doi.org/10.1137/18M1198387, doi:10.1137/18M1198387.

[58] S. E. Minson, M. Simons, and J. L. Beck. Bayesian inversion for finite fault earthquake source models I—theory and algorithm. *Geophysical Journal International*, 194(3):1701–1726, 06 2013. arXiv:https://academic.oup.com/gji/article-pdf/194/3/1701/39595731/gji_194_3_1701.pdf, doi:10.1093/gji/ggt180.

[59] Hyejung Moon. *Design and analysis of computer experiments for screening input variables*. PhD thesis, The Ohio State University, 2010. URL: http://rave.ohiolink.edu/etdc/view?acc_num=osu1275422248.

[60] Hyejung Moon, Angela M. Dean, and Thomas J. Santner. Two-stage sensitivity-based group screening in computer experiments. *Technometrics*, 54(4):376–387, 2012. arXiv:https://doi.org/10.1080/00401706.2012.725994, doi:10.1080/00401706.2012.725994.

[61] M.D. Morris. Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33(2):161–174, 1991. doi:http://dx.doi.org/10.2307/1269043.

[62] Z. Morrow, J.D. Jakeman, and B. et. al Van Bloemen Waanders. An overview of machine learning approximation methods for scientific computing. *In preparation*, 2022.

[63] A. Narayan. Computation of induced orthogonal polynomial distributions. *Electronic Transactions on Numerical Anlysis*, 50:71–97, 2018. URL: https://doi.org/10.1553/etna_vol50s71.

[64] A. Narayan and J.D. Jakeman. Adaptive leja sparse grid constructions for stochastic collocation and high-dimensional approximation. *SIAM Journal on Scientific Computing*, 36(6):A2952–A2983, 2014. URL: http://dx.doi.org/10.1137/140966368, arXiv:http://dx.doi.org/10.1137/140966368, doi:10.1137/140966368.

[65] Akil Narayan, John D. Jakeman, and Tao Zhou. A Christoffel function weighted least squares algorithm for collocation approximations. *Mathematics of Computation*, 86:1913–1947, 2017. URL: https://doi.org/10.1090/mcom/3192.

[66] F. Nobile, R. Tempone, and C.G. Webster. A sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, 46(5):2309–2345, 2008. URL: http://link.aip.org/link/?SNA/46/2309/1, doi:10.1137/060663660.

[67] Jeremy E. Oakley and Anthony O'Hagan. Probabilistic sensitivity analysis of complex models: a bayesian approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 66(3):751–769, 2004. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9868.2004.05304.x, arXiv:https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-9868.2004.05304.x, doi:https://doi.org/10.1111/j.1467-9868.2004.05304.x.

[68] Audrey Olivier, Dimitris G. Giovanis, B.S. Aakash, Mohit Chauhan, Lohit Vandanapu, and Michael D. Shields. Uqpy: A general purpose python package and development environment for uncertainty quantification. *Journal of Computational Science*, 47:101204, 2020. URL: https://www.sciencedirect.com/science/article/pii/S1877750320305056, doi:https://doi.org/10.1016/j.jocs.2020.101204.

[69] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011. arXiv:https://doi.org/10.1137/090752286, doi:10.1137/090752286.

[70] M. Parno, A. Davis, and L. Seelinger. MUQ: The MIT uncertainty quantification library. *Journal of Open Source Software*, 6(68):3076, 2021. URL: https://doi.org/10.21105/joss.03076.

[71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL: https://doi.org/10.5555/1953048.2078195.

[73] Benjamin. Peherstorfer, Karen. Willcox, and Max. Gunzburger. Optimal model management for multifidelity monte carlo estimation. *SIAM Journal on Scientific Computing*, 38(5):A3163–A3194, 2016. arXiv:https://doi.org/10.1137/15M1046472, doi:10.1137/15M1046472.

[74] Noemi Petra, James Martin, Georg Stadler, and Omar Ghattas. A computational framework for infinite-dimensional bayesian inverse problems, part ii: Stochastic newton mcmc with application to ice sheet flow inverse problems. *SIAM Journal on Scientific Computing*, 36(4):A1525–A1555, 2014. arXiv:https://doi.org/10.1137/130934805, doi:10.1137/130934805.

[75] Carl Edward Rasmussen. *Gaussian Processes in Machine Learning*, pages 63–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-28650-9_4.

[76] Murray Rosenblatt. Remarks on a multivariate transformation. *The Annals of Mathematical Statistics*, 23(3):470–472, 1952. URL: http://www.jstor.org/stable/2236692.

[77] Kenneth J. Ryan. Estimating expected information gains for experimental designs with application to the random fatigue-limit model. *Journal of Computational and Graphical Statistics*, 12(3):585–603, 2003. URL: http://www.jstor.org/stable/1391040.

[78] Jerome Sacks, William J. Welch, Toby J. Mitchell, and Henry P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4(4):409–423, 1989. URL: http://www.jstor.org/stable/2245858.

[79] Andrea Saltelli, Paola Annoni, Ivano Azzini, Francesca Campolongo, Marco Ratto, and Stefano Tarantola. Variance based sensitivity analysis of model output. design and estimator for the total sensitivity index. *Computer Physics Communications*, 181(2):259–270, 2010. URL: https://www.sciencedirect.com/science/article/pii/S0010465509003087, doi:https://doi.org/10.1016/j.cpc.2009.09.018.

[80] Fonnesbeck C. Salvatier J, Wiecki TV. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016. URL: https://doi.org/10.7717/peerj-cs.55.

[81] R. Schaback. Error estimates and condition numbers for radial basis function interpolation. *Adv Comput Math*, 3:251–264, 1995. URL: https://doi.org/10.1007/BF02432002.

[82] Robert Schaback and Holger Wendland. Kernel techniques: From machine learning to meshless methods. *Acta Numerica*, 15:543–639, 2006. doi:10.1017/S0962492906270016.

[83] J.D. Seidl, T. Jakeman, M. Perego, K. Sockwell, M Hoffman, and S. Price. Using multi-fidelity methods to quantify uncertainty in mass loss projections from humboldt glacier, greenland. *In prepartation*, 2022.

[84] Pranay Seshadri, Chun Yui Wong, Ashley D. Scillitoe, Bryn N. Ubald, Barney Hill, Irene Virdis, and Tiziano Ghisu. *Programming with equadratures: an open-source package for uncertainty quantification, dimension reduction, and optimisation.* 2022. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2022-2108, arXiv:https://arc.aiaa.org/doi/pdf/10.2514/6.2022-2108, doi:10.2514/6.2022-2108.

[85] I.M Sobol. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and Computers in Simulation*, 55(1):271–280, 2001. The Second IMACS Seminar on Monte Carlo Methods. URL: https://www.sciencedirect.com/science/article/pii/S0378475400002706, doi:https://doi.org/10.1016/S0378-4754(00)00270-6.

[86] A. M. Stuart. Inverse problems: A bayesian perspective. *Acta Numerica*, 19:451–559, 2010. doi:10.1017/S0962492910000061.

[87] B. Sudret. Global sensitivity analysis using polynomial chaos expansions. *Reliability Engineering & System Safety*, 93(7):964–979, JUL 2008. doi:{10.1016/i.ress.2007.04.002}.

[88] S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Retrieved July 13, 2022, from http://www.sfu.ca/~ssurjano.

[89] A. Teckentrup, P. Jantsch, C. Webster, and M. Gunzburger. A multilevel stochastic collocation method for partial differential equations with random input data. *SIAM/ASA Journal on Uncertainty Quantification*, 3(1):1046–1074, 2015. doi:10.1137/140969002.

[90] Irina Kalashnikova Tezaur, Kara Peterson, Amy Powell, John Jakeman, and Erika Roesler. Global sensitivity analysis using the ultra-low 1 resolution energy exascale earth system model 2. *Earth and Space Science Open Archive*, page 38, 2021. doi:10.1002/essoar.10508267.1.

[91] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000. URL: https://epubs.siam.org/doi/book/10.1137/1.9780898719598.

[92] Joel A. Tropp and Anna C. Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on Information Theory*, 53(12):4655–4666, 2007. doi:10.1109/TIT.2007.909108.

[93] Umberto Villa, Noemi Petra, and Omar Ghattas. Hippylib: An extensible software framework for large-scale inverse problems governed by pdes: Part i: Deterministic inversion and linearized bayesian inference. *ACM Trans. Math. Softw.*, 47(2), apr 2021. doi:10.1145/3428447.

[94] Chen Wang, Qingyun Duan, Charles H. Tong, Zhenhua Di, and Wei Gong. A gui platform for uncertainty quantification of complex dynamical models. *Environmental Modelling & Software*, 76:1–12, 2016. URL: https://www.sciencedirect.com/science/article/pii/S1364815215300955, doi:https://doi.org/10.1016/j.envsoft.2015.11.004.

[95] Konstantin Weise, Lucas Poßner, Erik Müller, Richard Gast, and Thomas R. Knösche. Pygpc: A sensitivity and uncertainty analysis toolbox for python. *SoftwareX*, 11:100450, 2020. URL: https://www.sciencedirect.com/science/article/pii/S2352711020300078, doi:https://doi.org/10.1016/j.softx.2020.100450.

[96] R. White, J.D. Jakeman, Bart Van Bloemen Waanders, and B. Alexanderian. Bayesian active learning for risk-averse sequential experimental optimal design. *In preparataion*, 2022.

[97] D. Xiu and G.E. Karniadakis. The Wiener-Askey Polynomial Chaos for stochastic differential equations. *SIAM J. Sci. Comput.*, 24(2):619–644, 2002. URL: http://dx.doi.org/10.1137/S1064827501387826, doi:10.1137/S1064827501387826.