SAND2022-11323R

Development of the uncertainty quantification toolkit's python interface and surrogate construction tutorial

The uncertainty quantification toolkit (UQTk) is a collection of c++ libraries that assess the confidence of numerical models. Surrogate approximations, often polynomial chaos expansions (PCEs), lessen the computational cost of these assessments. I developed a Python interface in UQTk for regression and Bayesian compressive sensing to add to the existing Galerkin projection method. These methods receive an object containing the polynomial basis information and NumPy arrays of sample points, call c++ methods, and return the PCE coefficients in a NumPy array. To demonstrate these methods, I wrote a tutorial in which I use them to construct surrogates for Genz functions and calculate the resulting error. I performed two test cases: creating a 6^{th}-order polynomial surrogate for a 2-dimensional Genz oscillatory function and an 8^{th}-order polynomial surrogate for a 4-dimensional oscillatory Genz function. In both cases, Galerkin projection performed best overall, regression performed best in overdetermined systems, and Bayesian compressive sensing performed best in underdetermined systems.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

I. INTRODUCTION

Uncertainty quantification (UQ) is a process aimed at assessing the confidence of numerical models, which involves surrogate construction to approximate complex models, global sensitivity analysis to determine the most influential parameters, estimation of these parameters, and forward propagation of these parameters through the model to produce probability density functions of the model outputs.[1]

One of the first steps of UQ, surrogate construction, helps a numerically complex model to be run many times by creating a computationally cheap approximation.[2] A surrogate can be a polynomial chaos expansion (PCE), a series of orthogonal polynomials of random variables of increasing order. In the PCE in equation 1, the orthogonal polynomials, ψ_k, are functions of random variable, ξ, multiplied by coefficients, c_k. Terms of increasing order are summed, and K is determined by a truncation rule. The series approximates the model output value, y.

$$y \approx \sum_{k=0}^{K-1} c_k \psi_k(\xi) \tag{1}$$

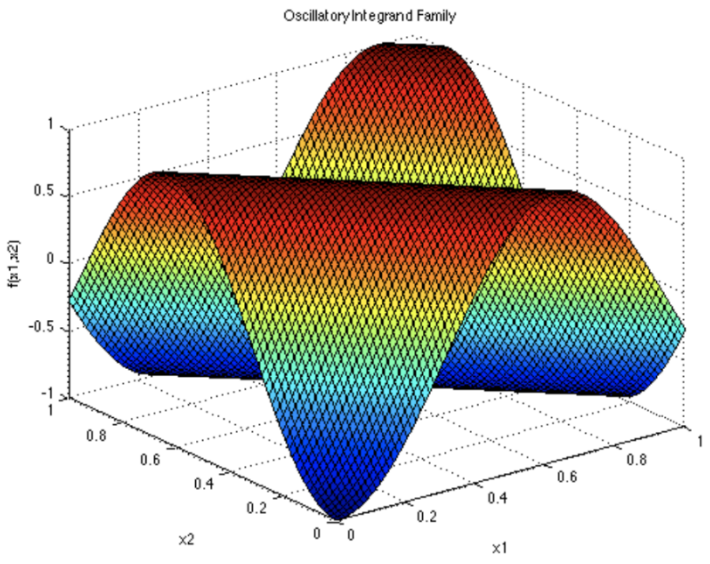
The UQ Toolkit (UQTk) has capabilities to accomplish many UQ tasks, including surrogate construction, through its c++ libraries and Python interface.[3] The methods of UQTk's Python interface were spread across separate scripts and folders, so my project was to gather existing methods relating to surrogate construction into /PyUQTk/PyPCE/pce_tools.py and write new ones, including regression and Bayesian compressive sensing.

I also wrote a tutorial to demonstrate how to use these methods in a surrogate construction workflow for Genz functions (with domains of [-1,1]). Genz functions are often used for algorithm testing in various dimensions; for example, Figure 1 depicts a 2-dimensional Oscillatory Genz function. As I tested my methods and surrogate construction workflow, I used PCEs of orders and dimensions up to eight.

[1] Bert Debusschere, Khachik Sargsyan, Cosmin Safta, and Kenny Chowdhary, "Uncertainty Quantification Toolkit (UQTk)," in *Handbook of Uncertainty Quantification,* (Springer International Publishing Switzerland, 2017), pp. 1807-1827.

[2] Khachik Sargsyan, "Surrogate Models for Uncertainty Propagation and Sensitivity Analysis," in *Handbook of Uncertainty Quantification,* (Springer International Publishing Switzerland, 2017), pp. 673-695.

[3] Khachik Sargsyan, Cosmin Safta, Luke Boll, Katherine Johnston, Mohammad Khalil, Kenny Chowdhary, Prashant Rai, Tiernan Casey, Xiaoshu Zeng, Bert Debusschere, in UQTk Version 3.1.2 User Manual, (Sandia National Laboratories, 2022).



$$f(\mathbf{x}) = \cos\left(2\pi u_1 + \sum_{i=1}^{d} a_i x_i\right)$$

Figure 1: A 2-dimensional oscillatory Genz function with a domain of [0,1].[4]

II. PYTHON INTERFACE

A. UQTkRegression

The first method for PCE surrogate construction that I added to pce_tools.py is UQTkRegression, which interfaces with the c++ code and performs standard regression. This method solves $Ax = b$, where A is a matrix of the evaluated basis of the polynomial expansion, b is a vector of model evaluations, and x is a vector of coefficients. This method accepts an object of the PCSet class (which contains the basis information), a NumPy array of sample points, and a NumPy array of model evaluations at the sample points. It returns a NumPy array of the coefficients for the PCE.

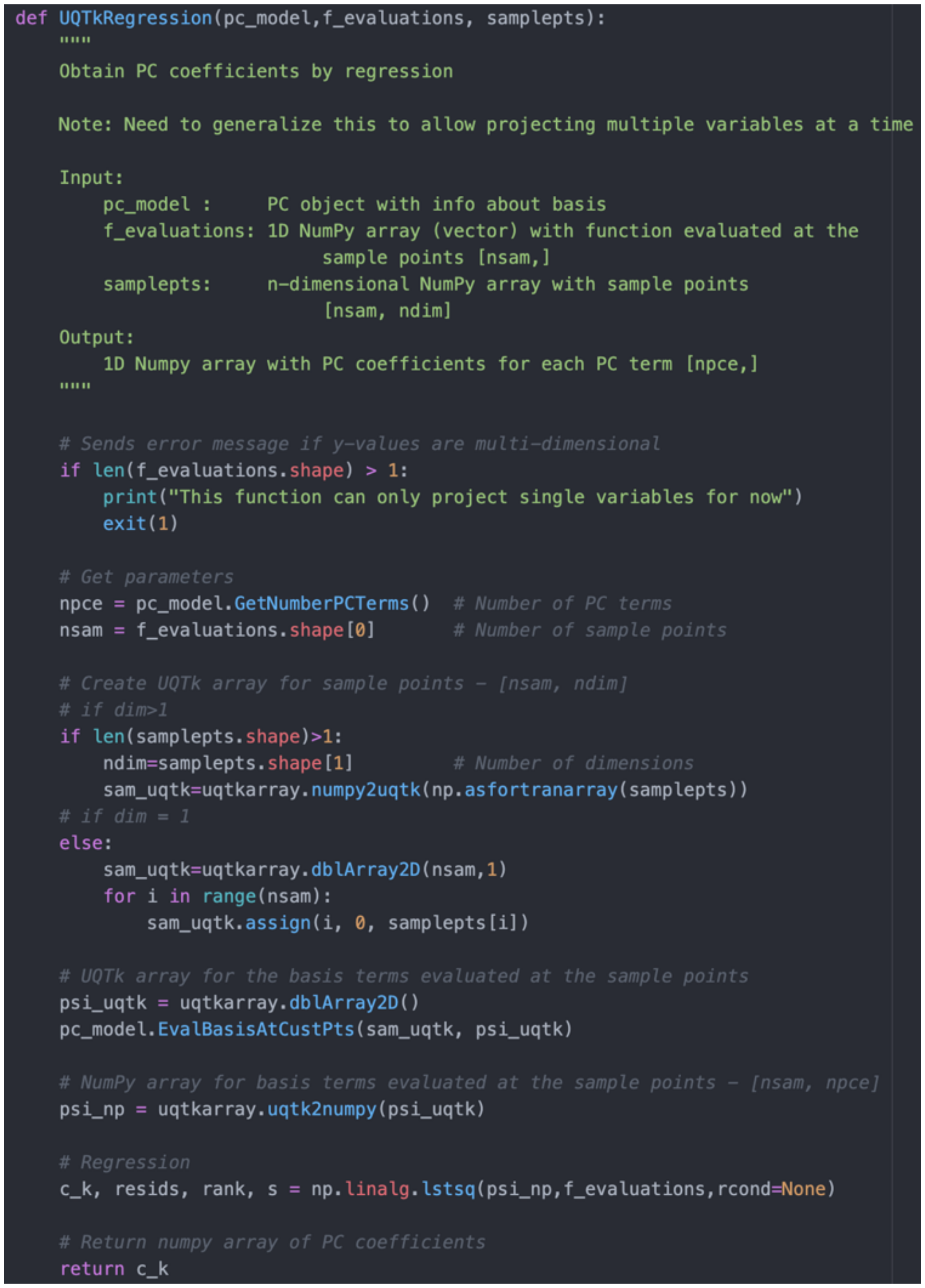
First, UQTkRegression converts the sample points into an UQTk array, which is a data type that the UQTk c++ libraries read. Then, it calls EvalBasisAtCustomPoints using the basis information from the PCSet object and the sample points. This produces a UQTk array of the evaluated basis, which is then converted into a NumPy array. The NumPy arrays of the sample points, the model evaluations, and the evaluated basis are given to linalg.lstsq, a NumPy regression method. The produced coefficients are then returned. It can be cumbersome to convert data types and evaluate the basis matrix for the regression task, and UQTkRegression streamlines this process. The code for UQTkRegression is shown in Figure 2.

To assess the performance of this method, I ran tests using Genz functions for different numbers of input sample points and different numbers of dimensions. Figure 3 depicts the root mean square error of a 4th order Gaussian Genz function, showing that the error decreases when

[4] Simon Surjanovic and Derek Bingham, "Oscillatory Integrand Family," in *Virtual Library of Simulation Experiments: Test Functions and Datasets*, (Simon Fraser University, 2013), https://www.sfu.ca/~ssurjano/oscil.html.

the system becomes overdetermined until it flattens out around the number of sample points equal to 120% of the basis terms.

I also wrote a script that creates a PCE approximation for a Legendre polynomial with pre-determined series of coefficients. Regression determines the coefficients, and the script compares the given coefficients to the regression-determined coefficients. If identical, the method is functional. This test is run with other UQTk compilation tests.



```python
def UQTkRegression(pc_model,f_evaluations, samplepts):
    """
    Obtain PC coefficients by regression

    Note: Need to generalize this to allow projecting multiple variables at a time

    Input:
        pc_model :     PC object with info about basis
        f_evaluations: 1D NumPy array (vector) with function evaluated at the
                          sample points [nsam,]
        samplepts:     n-dimensional NumPy array with sample points
                          [nsam, ndim]
    Output:
        1D Numpy array with PC coefficients for each PC term [npce,]
    """

    # Sends error message if y-values are multi-dimensional
    if len(f_evaluations.shape) > 1:
        print("This function can only project single variables for now")
        exit(1)

    # Get parameters
    npce = pc_model.GetNumberPCTerms()  # Number of PC terms
    nsam = f_evaluations.shape[0]       # Number of sample points

    # Create UQTk array for sample points - [nsam, ndim]
    # if dim>1
    if len(samplepts.shape)>1:
        ndim=samplepts.shape[1]         # Number of dimensions
        sam_uqtk=uqtkarray.numpy2uqtk(np.asfortranarray(samplepts))
    # if dim = 1
    else:
        sam_uqtk=uqtkarray.dblArray2D(nsam,1)
        for i in range(nsam):
            sam_uqtk.assign(i, 0, samplepts[i])

    # UQTk array for the basis terms evaluated at the sample points
    psi_uqtk = uqtkarray.dblArray2D()
    pc_model.EvalBasisAtCustPts(sam_uqtk, psi_uqtk)

    # NumPy array for basis terms evaluated at the sample points - [nsam, npce]
    psi_np = uqtkarray.uqtk2numpy(psi_uqtk)

    # Regression
    c_k, resids, rank, s = np.linalg.lstsq(psi_np,f_evaluations,rcond=None)

    # Return numpy array of PC coefficients
    return c_k
```

Figure 2: The code for UQTkRegression in pce_tools.py

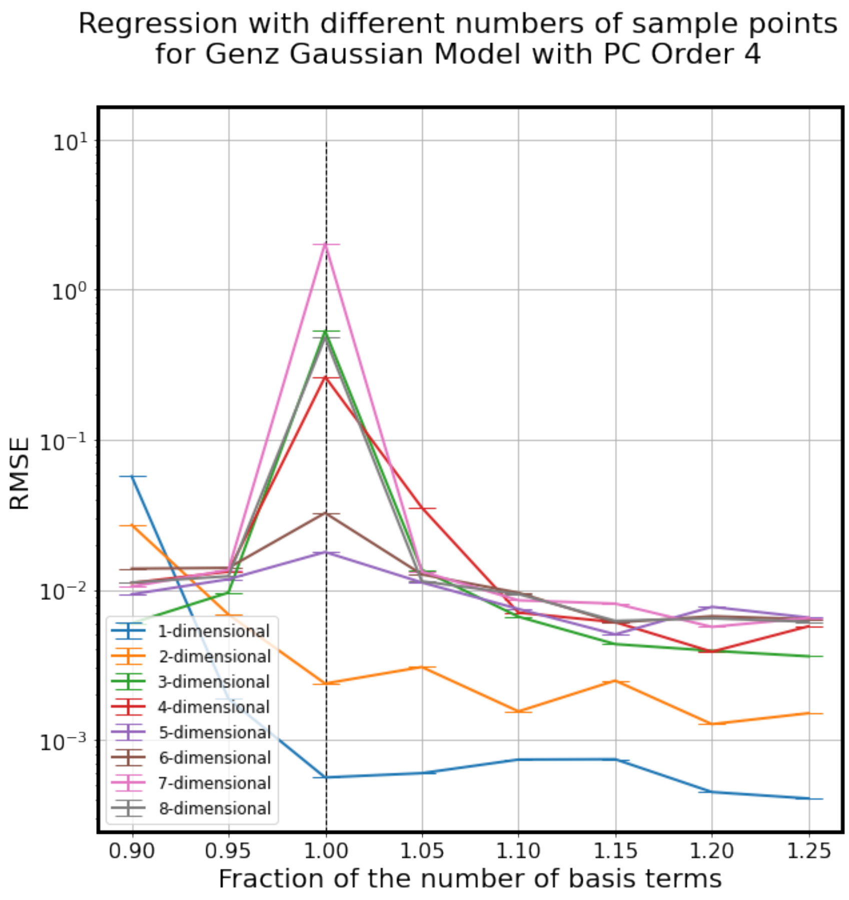


Figure 3: Root mean square error in the regression-constructed 4th-order surrogate for Gaussian Genz functions with one to eight dimensions and varying sample points. The x-axis is the percentage of the number of basis terms to use as sample points. The graph shows the error flattening out around 120% of the number of basis terms, so this would be the ideal number of sample points to use.

B. UQTkBCS

UQTkBCS implements Bayesian compressive sensing, which uses regression and Bayesian inference to determine coefficients while encouraging sparsity.[5] This method accepts an object of the PCSet class, a NumPy array of sample points, a NumPy array of model evaluations at the sample points, a NumPy array of the data noise, the stopping threshold (as a float, list, or NumPy array), a flag to indicate if the conservative growth method is used, the number of up-iterations, the number of folds for stopping threshold cross-validation, and a flag for print statements. By default, a non-conservative growth process, no up-iterations, and 5 folds for cross-validation are used, and print statements are silenced. UQTkBCS returns a NumPy array of coefficients and PCSet object with new basis.

UQTkBCS first chooses whether to optimize the stopping threshold, eta. If eta is an array or list, then the eta that produces the lowest error is selected by UQTkOptimizeEta, which

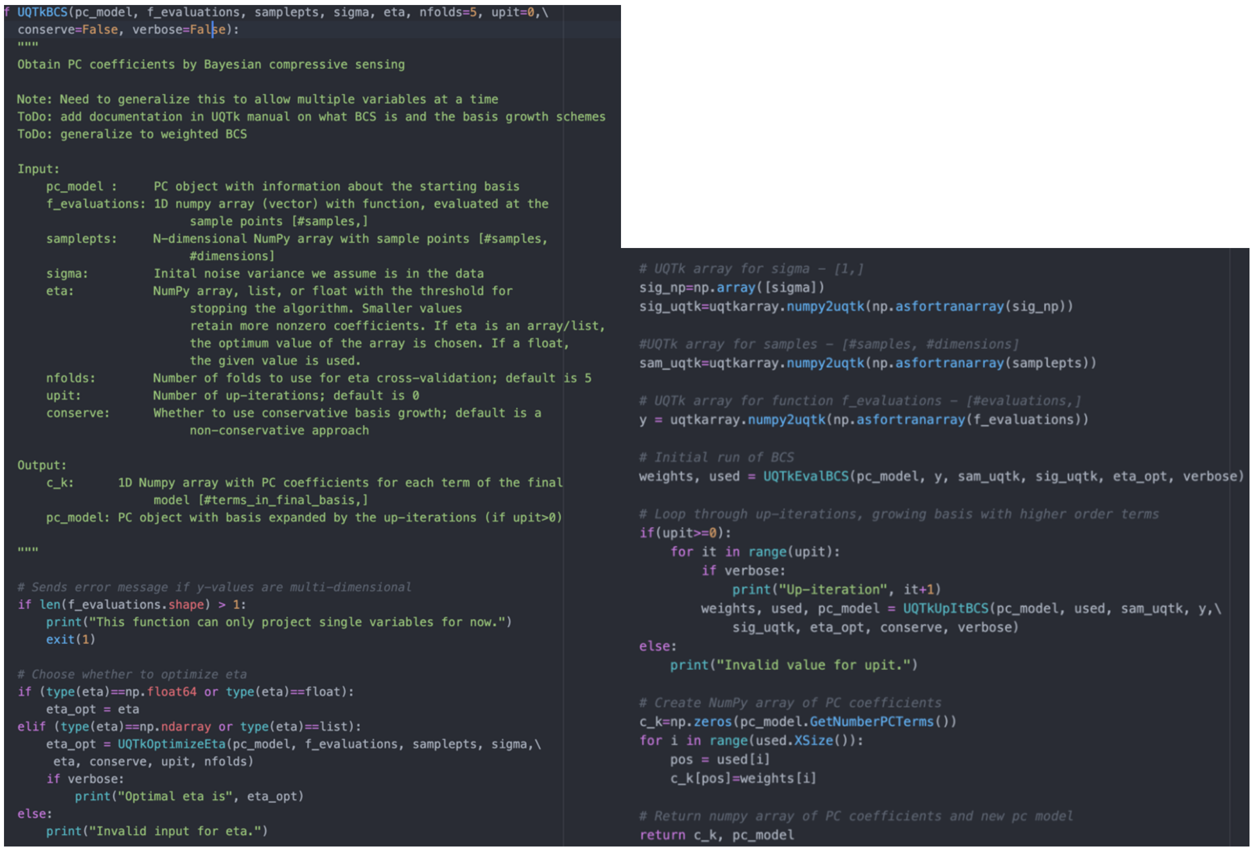
[5] Khachik Sargsyan, Cosmin Safta, Habib N. Najm, Bert J. Debusschere, Daniel Ricciuto, Peter Thornton, “Dimensionality Reduction for Complex Models via Bayesian Compressive Sensing,” International Journal for Uncertainty Quantification 4 (1), 63–93 (2014), https://doi.org/10.1615/Int.J.UncertaintyQuantification.2013006821.

performs cross-validation with the specified number of folds. If the input for eta is a single float, then optimization is skipped, and the given value is used.

Next, the remaining input arrays are converted to UQTk arrays, and another Python method UQTkEvalBCS is called. This method calls the c++ implementation of BCS without adaptive functioning.

Next, if up-iterations are used, the script shrinks the basis to only the coefficients that the first iterations of BCS selected. Then, higher order terms are added to the basis terms in those selected areas with either a conservative or non-conservative approach. This process is repeated for each up-iteration. The final basis (as a PCSet object) and the coefficients (as a NumPy array) are returned. UQTkBCS allows for a user to easily call multiple iterations of BCS. The code is shown in Figure 4.

I again ran tests using Genz functions to assess UQTkBCS's performance. Figure 5 depicts the root mean square error of a 3rd-order exponential Genz function over various dimensions and sample inputs—with the error flattening around the number of sample points equal to 150% of the basis terms.



```python
f UQTkBCS(pc_model, f_evaluations, samplepts, sigma, eta, nfolds=5, upit=0,\
  conserve=False, verbose=False):
  """
  Obtain PC coefficients by Bayesian compressive sensing

  Note: Need to generalize this to allow multiple variables at a time
  ToDo: add documentation in UQTk manual on what BCS is and the basis growth schemes
  ToDo: generalize to weighted BCS

  Input:
      pc_model :     PC object with information about the starting basis
      f_evaluations: 1D numpy array (vector) with function, evaluated at the
                          sample points [#samples,]
      samplepts:     N-dimensional NumPy array with sample points [#samples,
                          #dimensions]
      sigma:         Inital noise variance we assume is in the data
      eta:           NumPy array, list, or float with the threshold for
                          stopping the algorithm. Smaller values
                          retain more nonzero coefficients. If eta is an array/list,
                          the optimum value of the array is chosen. If a float,
                          the given value is used.
      nfolds:        Number of folds to use for eta cross-validation; default is 5
      upit:          Number of up-iterations; default is 0
      conserve:      Whether to use conservative basis growth; default is a
                          non-conservative approach

  Output:
      c_k:      1D Numpy array with PC coefficients for each term of the final
                     model [#terms_in_final_basis,]
      pc_model: PC object with basis expanded by the up-iterations (if upit>0)

  """

  # Sends error message if y-values are multi-dimensional
  if len(f_evaluations.shape) > 1:
      print("This function can only project single variables for now.")
      exit(1)

  # Choose whether to optimize eta
  if (type(eta)==np.float64 or type(eta)==float):
      eta_opt = eta
  elif (type(eta)==np.ndarray or type(eta)==list):
      eta_opt = UQTkOptimizeEta(pc_model, f_evaluations, samplepts, sigma,\
       eta, conserve, upit, nfolds)
      if verbose:
          print("Optimal eta is", eta_opt)
  else:
      print("Invalid input for eta.")

  # UQTk array for sigma - [1,]
  sig_np=np.array([sigma])
  sig_uqtk=uqtkarray.numpy2uqtk(np.asfortranarray(sig_np))

  #UQTk array for samples - [#samples, #dimensions]
  sam_uqtk=uqtkarray.numpy2uqtk(np.asfortranarray(samplepts))

  # UQTk array for function f_evaluations - [#evaluations,]
  y = uqtkarray.numpy2uqtk(np.asfortranarray(f_evaluations))

  # Initial run of BCS
  weights, used = UQTkEvalBCS(pc_model, y, sam_uqtk, sig_uqtk, eta_opt, verbose)

  # Loop through up-iterations, growing basis with higher order terms
  if(upit>=0):
      for it in range(upit):
          if verbose:
              print("Up-iteration", it+1)
          weights, used, pc_model = UQTkUpItBCS(pc_model, used, sam_uqtk, y,\
              sig_uqtk, eta_opt, conserve, verbose)
  else:
      print("Invalid value for upit.")

  # Create NumPy array of PC coefficients
  c_k=np.zeros(pc_model.GetNumberPCTerms())
  for i in range(used.XSize()):
      pos = used[i]
      c_k[pos]=weights[i]

  # Return numpy array of PC coefficients and new pc model
  return c_k, pc_model
```

Figure 4: The code for UQTkBCS in pce_tools.py

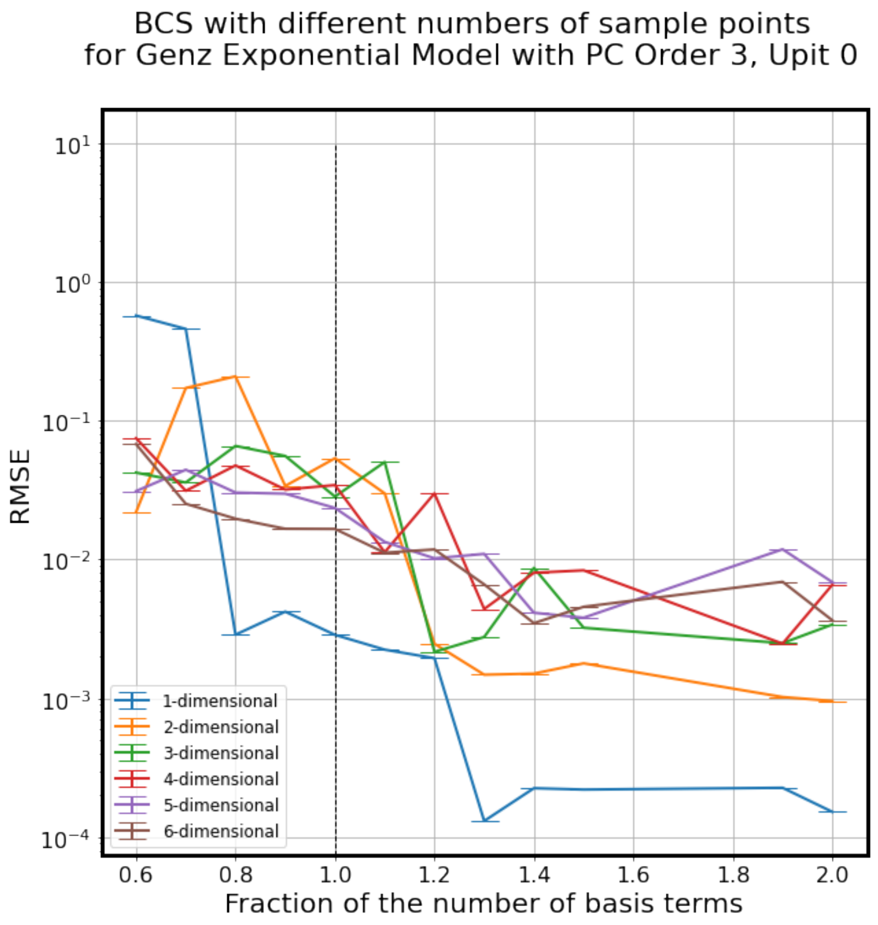


Figure 5: Root mean square error in the BCS-constructed 3rd-order surrogate for exponential Genz functions with one to six dimensions and varying sample points. Eta was given as a range from 1e-1 to 1e-15, and sigma was given as 1e-8. No up-iterations were used. The x-axis is the percentage of the number of basis terms to use as sample points. The graph shows the error flattening out around 150% of the number of basis terms, so this would be the ideal number of sample points to use.

III. SURROGATE CONSTRUCTION TUTORIAL

To show how to use these methods, I created Jupyter notebooks that walk through the process of surrogate construction for a Genz function with regression, with BCS, and with Galerkin Projection. In these tutorial notebooks, I first define the type of polynomial to use (Legendre, Hermite, etc.), the order of the polynomial, the dimension, and other surrogate options. Then, I instantiate an object in the PCSet class that contains the basis information. I generate random training and testing data between -1 and 1. (Quadrature points are calculated instead of random training samples if using Galerkin projection). Then, I perform Galerkin projection, regression, or BCS to determine the coefficients of the PC expansion. With the coefficients, I evaluate the PCE at the testing points, calculating the normalized root mean square error. These tutorials are in UQTk in /examples/surrogate_genz/.

I used the tutorial notebooks to construct a 6th-order Legendre PCEs for a 2-dimensional Genz oscillatory function using regression, BCS, and Galerkin projection. Galerkin projection used full quadrature, and regression and BCS used two collections of random samples (at 200%

and 75% the number of basis terms—overdetermined and undetermined systems, respectively.) For BCS, I also set sigma to 1e-8 and eta to be an array from 1e-1 to 1e-15 and used conservative basis growth and no up-iterations. Of the methods, Galerkin Projection performed best overall with a normalized root mean square error of 1.41e-8, regression performed best in an overdetermined system with an error of 3.21e-8, and BCS performed best in an underdetermined system with an error of 1.52e-5.

I repeated this process with an 8th-order Legendre surrogate for a 4-dimensional oscillatory Genz function. Again, Galerkin projection performed best overall with an error of 8.45e-9, regression performed best in an overdetermined system with an error of 2.58e-8, and BCS performed best in an underdetermined system with an error of 3.95e-6. The normalized root mean square error and the number of points used for each method are summarized in Table 1.

Order	**6**		**8**	
Dimension	**2**		**4**	
Basis Size	28		495	
Galerkin Projection	1.41e-8	49 points	8.45e-9	6561 points
Regression (underdetermined)	4.83e-4	21 points	2.69e-4	371 points
Regression (overdetermined)	3.21e-8	56 points	2.58e-8	990
BCS (underdetermined)	1.52e-5	21 points	3.95e-6	371
BCS (overdetermined)	4.36e-7	56 points	1.04e-6	990

Table 1: Normalized root mean square errors and number of training points for a polynomial chaos expansions of 6th order and 2 dimensions as well as 8th order and 4 dimensions, using Galerkin projection, regression, and Bayesian compressive sensing.

IV. CONCLUSION

The Python methods of UQTkRegression and UQTkBCS interface with the c++ functions of UQTk, helping Python users more easily access them, and the tutorials go through the process of surrogate construction with these methods, explaining the methods in context. These additions to UQTk will be made available to the broader scientific community when the UQTk 3.1.3 update is pushed to GitHub in August 2022. This updated is expected to reduce the UQTk learning curve for end-users.

V. ACKNOWLEDGEMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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-NA-0003525.

This work was funded by the DOE Office of Workforce Development for Teachers and Scientists (WDTS) and the DOE Office of Science, Advanced Scientific Computing Research (ASCR).

I would like to acknowledge my research mentor Bert Debusschere who guided me through this project.