The CheKIPEUQ normal flow process for a Bayesian Parameter Estimation is as follows:

1. The user creates a new directory for each analysis. The user creates a python runfile that they can name as they would like. Inside the runfile, first the CheKiPEUQ software package is imported since it does the work (the parameter estimation).
2. To setup what is needed to do the analysis, everything is passed into or defined in the CheKiPEUQ UserInput namespace (that is, the UserInput module).
   1. The user provided observed data and uncertainties
   2. The user provided prior value distributions, including uncertainties. (This can be regarded as the “initial guesses” based on prior knowledge)
   3. User choices for when the parameter estimation is formed. The UserInput namespace is initially populated with some defaults, which the user can then change. *Users are encouraged to open the package’s UserInput.py file to see the comments in there, which will be more up to date than any of the online documentation.*
      1. Although it would not be the standard usage, advanced users can make their own UserInput python files in the analysis directory
   4. The order which the user enters their choices into the UserInput name space does not matter.
3. This UserInput serves as a module, a namespace, and an object, which is fed as an argument into a ParameterEstimation object.
   1. Various parameter estimation routines can then be called from the parameter estimation object.
4. After the parameter estimation is done, the user can call commands to create graphs and can also access the results of the parameter estimation.

Below, a few specific examples and the features of the CheKiPEUQ package are explained.

**#TODO: Change it so that at least example 1a has a more simple example of importing data from a file. Should just use numpy getfromcsv and then specify the column number where the data is for errors etc.**

**Example 1a:** This example shows users how to do a Bayesian Parameter estimation using Markov Chain Monte Carlo (basically, using statistical sampling of the parameter space). For this example, new users do not need to understand MCMC nor understand the example’s model since the purpose is to learn how to use CheKiPEUQ. Users should open the runfile to look inside. All users need to understand is that there are six parameters. Ea1, Ea2, A1, A2, g1, g2. Notice that the prior values and their uncertainties are specified in runfile\_Example1a. Notice that every variable made is passed into the UserInput. One thing that is important to recognize is that a *model* must be provided to the runfile in the form of a python function given to UserInput.model['simulateByInputParametersOnlyFunction']. This function **must** take **exactly** the same parameters in exactly the same order. Users may optionally provide UserInput.model['simulationOutputProcessingFunction'] which can parse the output from the simulation function in order to match the format of the fs\_observed.

Note: The 'simulateByInputParametersOnlyFunction' must return ONLY a vector (or set of vectors) with the same length as responses\_observed. If for example 'simulateByInputParametersOnlyFunction' returns both an abscissa and a set of simulated responses, then a 'simulationOutputProcessingFunction' must be provided which will extract or produce something with the same size as responses\_observed.

After the UserInput is defined, it is passed as an argument to create a parameter estimation object which we call PE\_object (an arbitrary choice).

Next, we use the command PE\_object.doMetropolisHastings()

This will run the MCMC using the default choices that were given for mcmc.

Finally, we use the command PE\_object.createAllPlots() to create the default plots for whatever type of choices we have made. In this case, looks focus on just the final plot. We see several things come out:

1. The observed data from experiments
2. The simulation from mu\_guess which is the initial guess (which is taken as the prior if no other initial guess was provided)
3. The MAP result, which is the output using the parameter set which has the highest posterior probability. This is the BPE equivalent of a “best fit”
4. The mu\_AP result, which is the highest expected value. In practice, we ideally want the mu\_AP result and MAP result to be approximately the same.
   1. Details beyond the scope of this tutorial: When they are not the same, it usually means one of a few things: a) that more mcmc sampling must be done to get a better exploration of the response surface, b) that the response surface is not well behaved and is hard to sample (this issue is described a bit in the next tutorial), c) that the mcmc step length(s) are not a good match to the curvature of the response surface so it is not being sampled well, d) that the model is very sensitive to parameter choices, e) that the posterior probability is asymmetric (it may even be multi-modal).

For those curious about the model, this is an example with two states. Each state has a “base” activation energy, a pre-exponential and a coefficient (gamma value) to describe a coverage dependent activation energy. A temperature programmed reaction consisting of only desorption is simulated.

**Example 1b:** This is the same as example 1a, except that now we use PE\_object.doOptimizeNegLogP(method="BFGS", printOptimum=True, verbose=True)

This set of choices will ignore the mcmc settings, and will not include a mu\_AP in the output.

**Example 1c:** This is the same as example 1a, except that now we use

PE\_object.doGridSearch('getLogP', verbose = False)

This set of choices will ignore the mcmc settings, and will not include a mu\_AP in the output (at this time).

It is anticipated that in the future the gridsearch will provide a mu\_AP value.

**Example 1d:** This is like Example 1b, only it uses the transient kinetics feature that converts things to an integral form in an intermediate step.

**Example 2a:** Example 2 is similar to Example 1. The only major different is that example 2 shows that it is possible to perform a simulation using Cantera. It is important to note that the simulation function is called cantera\_simulation\_wrapper\_example2 and is located inside of model\_functions\_example2.py . If one looks inside model\_functions\_example2, one can see that it relies upon using some functions from the CheKiPEUQ package to interface with cantera. Specifically, the canteraSimulate function.

**Example 3a:** is a variation of Example 1 where the integral is used instead of the rate for mcmc

**Example 3b:** is a variation of Example 1 where the integral is used instead of the rate for optimization

**Example 3c:** is a variation of Example 1 where the integral is used instead of the rate for optimization, AND the reduced parameter space feature is used, where *only* the indices that are noted are allowed to change. This reducedParameterSpace feature works with mcmc, grid search, and optimization.

#TODO: make it so that people can put the variable names into the reducedParameterSpace feature, as strings, rather than making them put in the indices.

**Example 3d:** is a variation of Example 1 where the integral is used instead of the rate for grid search optimization

**Example 4: Some kind of Cantera Example**

**Example 5: Some kind of Cantera Example**

**Example 6:** Cantera Example with coverage dependent activation energy offsets. The basic flow of this example is like this:

CheKiPEUQ imports integrated\_cantera\_simulation\_wrapper\_example6 so it can call integrated\_cantera\_simulation\_wrapper\_example6 which calls cantera\_simulation\_wrapper\_example6

In this example, Ea2, A2, and gamma2 are not used.

Note:The times / x values are globals and are implied arguments for the function which does the integration (integrated\_... ). In this specific example, the times and observed\_x values are exactly equal to each other and came from the experimental data.

cantera\_simulation\_wrapper\_example6 then takes the reaction parameters array ***as well as*** some implied arguments/globals: the piecewise\_coverage\_intervals, and everything that is needed to make the canteraPhases object and model (including the heating\_rate).

An important step is that cantera\_simulation\_wrapper\_example6 calls canteraKineticsParametersParser.populatePiecewiseCoverageDependence with the argument of “kineticParameterName” (among others) to change the particular parameter with the modifiers noted.

Then, the file create\_cti\_and\_SimulatePFRorTPRwithCantera does the actual simulations during the first iteration, and modify\_reactions\_and\_SimulatePFRorTPRwithCantera does the actual simulations during later iterations (by calling simulatePFRorTPRwithCantera after modifying the reactions). At present (April 2020), even the modify case requires making a new cantera model each time because the function modify\_reaction does not yet work for surface reactions within cantera.

It then calls simulatePFRorTPRwithCantera

#TODO: put all of the model files in this directory so that we don’t need to use “model\_location = "..\\..\\CheKiPEUQ\\simulationDriver\\" and can change it to “model\_location = “.\” or something.

The file runfile\_Example6\_optimize9.py uses a reduced parameter space to enable an outcome within perhaps 10 minutes.

The file runfile\_Example6\_optimize10.py does not use a restricted parameter space and may run in on the order of 1 hour.

The file runfile\_Example6\_optimize11.py uses the output from optimize10 as a starting point.

Final results from doOptimizeNegLogP: [ 1.48560697e+01 9.99999994e+09 1.46808762e+01 1.05356746e-02

3.00010055e-01 -1.27355612e-05 -2.92310229e-03 -1.81001813e-02

-4.16084666e-03 -2.12184284e-03 2.57320593e+01 -1.20251933e+01

-1.18731155e+01 -6.19418984e+01 -6.19196041e+01 -6.19848662e+01

-6.19572036e+01 -5.97358340e+02 2.94213676e+03 -2.30826349e+03]

Optimize 12 shows that putting this in as a starting point gives little change.

This is probably not a realistic result, and demonstrates that even with BPE when there is parameter estimation for large parameter spaces, it is possible to arrive at “good fits” that are not physically realistic.

**Example 7a:** This is a two response example. All the previous examples showed a single dimension for the response. In this example, there are two response dimensions such that in the runfile we have a nested object for the response: UserInput.responses['responses\_observed'] = np.array([[2], [3]])

**Example 7b:** This is the same as 7a, but shows that we can input a full covariance matrix for UserInput.model['InputParametersPriorValuesUncertainties'] = np.array([[1,0.0], [0.0,1]])

**Example 7c:** This is builtoff of 7b, but shows that we can input a full covariance matrix that is asymmetric: UserInput.model['InputParametersPriorValuesUncertainties'] = np.array([[1,0.2], [0.3,1]])

**Example 7d:** This is a custom log likelihood example. In this simple example, we take Example 7b and then double the variance of the responses. We see that the posterior is widened. This is just a test case. Doubling the variance would not require this feature. However, if one wanted to change the likelihood to be asymmetric (for example setting it to be zero for a certain range of the response) one could do so using this function.

**Example 7e:** This is built off of 7d: now we take the log likelihood and set it to a probability of 0 if the sum of the responses is greater than 6. Note that a probability of 0 requires a log likelihood of negative infinity, and that we set the likelihood accordingly. We see from the scatter plots and other plots that an asymmetric output results in the 2D posterior since having a high value for parameter 1 now requires a low value for parameter 2 and vice versa. Making the inequality “>3” rather than “>6” will make the posterior more narrow and ridge-like.

**Example 8:** In this example, a python function with an activation energy that has gamma depending on coverage and mcmc.

**Example 9:** In this example, a python function with a gamma depending on coverage, and gridsearch.

**Example 10:** This example is the same two response example as Example 7, but now it uses the “reducedResponseSpace” feature which allows the posterior to be based on only the response indices selected by the user (neglecting the uncertainties of any other responses, and intended for cases where certain responses cannot be measured or for performing test cases).

**Example 11:** This example is for Design of Experiments. The runfile iterates across several temperatures and shows that the posterior distribution changes with temperature.

**Example 14:** This example uses the various states of CO adsorption on Fe3O4 and the redhead peak maximum method to estimate the DeltaH ads of States 1 through 3. For state 4, bounds on the residence time are used, and then the peak maximum is still calculated at the end based on the posterior. Both model and experimental uncertainties are included.

**Example 15**: This example is based on Example 3 (two states), but cleans up the model files and also adds in a parameter for the ratio between the two states.

**Example 16a\_BPE\_grid:** 1,361,367 grid points.This example takes on the order of 5 hours to run. uses a grid search on negLogP with BEP and data\_overcategory set as transient\_kinetics **,** which enables the data to be transformed during exploring the posterior. This example also uses InputParameterPriorValues\_upperBounds and InputParameterPriorValues\_lowerBounds to restrict the parameter space to what is physically reasonable. As of May 2020, the bounds feature is only used with UserInput.parameter\_estimation\_settings['scaling\_uncertainties\_type'] = "off" (it may also work with other choices like ‘std’, but it has not been checked).

🡪 in general, the options for data\_category are blank (‘ ‘), ‘transient\_kinetics’, and ‘steady\_state\_kinetics’ but right now only ‘transient\_kinetics’ does anything.

🡪 response\_types can have values of ‘I’ for intermediate, ‘P’ for product, ‘R’ for reactant, and ‘O for other.

🡪 response\_data\_type can have ‘c’ for concentration or ‘r’ for rate, or ‘o’ for other.

**Example 16a\_BPE\_grid\_opt: with Nelder-Mead and up to 5000 simulation iterations per point.**

**Example 16a\_BPE\_grid\_mcmc: 243 points with no burn in and 1000 samplings each.**

**Example 16a\_CPE\_grid:** 1,361,367 grid points. On the order of 5 hours to run.

**Example 16a\_CPE\_grid\_opt: with Nelder-Mead and up to 5000 simulation iterations per point.**

**Example 16a\_CPE\_grid\_mcmc: 243 points with no burn in and 1000 samplings each.**

**Example 16b:** Does a Nelder-Mead optimization on the logP from 16a.

**Example 16c:** Does an MCMC on the final result from 16b.

1000 samplings takes 10 seconds. 100 samplings takes 1 second.

243 points at 100 samplings would be 5 minutes. 1000 samplings would be 50 minutes.

**Example 16d:** Like 16a, only using custom log prior that is always 1. Still has physical limits on the likelihood. Takes on the order of 5 hours to run.

**Example 16e:** Does a Nelder-Mead optimization on the logP from 16d.

**Example 16f:** Does an MCMC on the final result from 16e.

**Example 17 directory is like Example 16, only the coverage dependence (with gamma coefficients) is based upon total coverage rather than that species’ coverage.**

runfile\_Example17a\_BPE\_singlePoint.py uses the feature for evaluating (and then plotting the simulation results) of a single point versus the observed data. This is useful when testing a function. It’s probably a good practice to include it in the runfile before an mcmc etc to make sure things at least work for the initial point all the way to plotting.