SANDIA REPORT

SAND2016-7862
Unlimited Release
Printed August 2016

Implementation of Fast, Emulator-based Code Calibration

Nathaniel Bowman, Matthew R. Denman

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2016-7862
Unlimited Release
Printed August 2016

Implementation of Fast, Emulator-based Code Calibration

Nathaniel Bowman, Matthew R. Denman
Risk & Reliability Analysis
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-MS0748

Abstract

Calibration is the process of using experimental data to gain more precise knowledge of simulator inputs. This process commonly involves the use of Markov-chain Monte Carlo, which requires running a simulator thousands of times. If we can create a faster program, called an emulator, that mimics the outputs of the simulator for an input range of interest, then we can speed up the process enough to make it feasible for expensive simulators. To this end, we implement a Gaussian-process emulator capable of reproducing the behavior of various long-running simulators to within acceptable tolerance. This fast emulator can be used in place of a simulator to run Markov-chain Monte Carlo in order to calibrate simulation parameters to experimental data. As a demonstration, this emulator is used to calibrate the inputs of an actual simulator against two sodium-fire experiments.

Acknowledgments

The authors would like thank Andrew J. Clark (Sandia) and Nathan Andrews (Sandia) for their assistance with the project.

The authors would also like to thank the Nuclear Regulatory Commission for funding the implementation of methods outlined in this report and the Department of Energy: Nuclear Energy's Advanced Reactor Technology department for funding the development of the models used to debug the proposed methodology.

Contents

Appendix

Figures

Nomenclature

1F2	Fukushima Diichi Unit 2
CST	Condensate Storage Tank
eCDF	Empirical Cumulative Distribution Function
DW	Drywell
HMC	Hamiltonian/Hybrid Monte Carlo
GP	Gaussian Process
MCMC	Markov-Chain Monte Carlo
MVN	Multivariate Normal
PDF	Probability Density Function
RPV	Reactor Pressure Vessel
SNL	Sandia National Laboratories
DW	Drywell

1 Introduction

1.1 Problem Description

Codes for simulating large, complex systems often have many input parameters to allow precise specification of the exact initial state of the system under simulation. However, for most phenomena we are interested in tracking, it is impossible to have exact knowledge of the entire initial state. Given the nonlinear nature of many phenomena under simulation, an imprecise knowledge of initial state can lead to unacceptably large uncertainty in simulation outputs. Also, sometimes we have experimental output and are interested in knowing the input as precisely as possible for its own sake.

To narrow down likely distributions of inputs, we can study the results of previous experiments. We back-propagate our knowledge of experimental outputs to narrow down the possible input range. The process of using experimental outputs to infer simulator inputs is known as calibration. This work centered around implementing a fast, principled method for calibration that has previously been used in [1] and others.

1.2 Application Space

The severe accident analysis group at Sandia National Laboratories (SNL) has been engaged in model development, experimental validation, and accident reconstruction efforts for severe accident codes such as MELCOR [2]. The physics models within MELCOR were developed based upon a combination of quantitative evaluations of separate effects test data or a combination of quantitative and qualitative interpretations of integral effects tests data[3, 4]. The representativeness of these models are tested in MELCOR by reproducing the results a range of experiments under a variety of conditions[5]. While model inputs with large uncertainties may be adjusted to provide better agreement with the data, the impacts of these adjustments are difficult to propagate from one experiment to the next. When subsequent uncertainty analyses are conducted, the need for adjustments of default MELCOR parameters can provide invaluable information regarding the shape and correlation structure of input uncertainty distributions if they are collected using a consistent and coherent process.

The technique proposed in this report can also be applied to accident reconstruction efforts. Typically unknown parameters, such as water injection or venting effectiveness, are calibrating one at a time in an attempt to reconstruct the accident. The major drawbacks of this historical approach, that can be avoided using the methodology proposed in this report, include:

- Interference between input calibrations such that new calibrations distort the intended impact of previous calibrations and

- Uncharacterized uncertainty in the degree of sensitivity associated with the calibrated input values.

While the proposed methodology was implimented at SNL with the primary intent of assisting with ongoing accident reconstruction efforts, the complexity of that accident sequence and the state of ongoing model development made initial deployment of the calibration tool on these efforts impractical. As a result the methodology was deployed and debugged on a series of sodium fire experiments conducted at SNL in circa 2010 [6] and modeled with the control volume code Contain-LMR[7].

1.3 Methodology

Our belief about the reasonableness of an input state is a combination of domain knowledge and the degree to which the input leads to an output close to experimental values. Melding these sources of information is done most simply with Bayesian inference [8]. Since Bayesian inference works with distributions, it allows us to produce "error bars" that give a sense of likely input ranges, as opposed to a single, most likely input without accompanying knowledge of uncertainty. This gives us a theoretically sound justification for narrowing our input range.

However, a well-known drawback of Bayesian inference is that all but the simplest posterior distributions are impossible to calculate analytically. Instead, many iterations of a Markov-Chain Monte Carlo (MCMC) algorithm are typically run to draw from the posterior distribution [1, 8]. Each draw of MCMC requires a comparison of the simulator output with the experimental data, which means we could need to run the simulator tens of thousands of times. For simulators that take a few seconds, this is painfully slow. For simulators that take minutes, or even days, running enough trials is infeasible.

To get around this problem, we model the input-output relationship of the simulator via an emulator [1, 9]. An emulator is a routine that produces nearly the same results as the simulator, but which can be evaluated much more quickly than the simulator. One typically wants an emulator to be several orders of magnitude faster than the corresponding simulator. A very popular choice of emulator is the Gaussian process (GP). This type of emulator is straightforward to understand and implement. A GP can be tuned by a user to fit a wide variety of shapes, which allows GPs to emulate many real-world functions effectively.

The process of accelerating code calibration via emulators is as follows. A sample of inputs is drawn from a user-specified range, and the simulator is run on each input to produce training data for the emulator. The emulator then learns the input-output relationship from this training set. Once the emulator has been trained, it is validated against withheld runs of the simulator that were not used to train it. If this step is skipped, we have no way of knowing whether conclusions drawn from the emulator are valid. If the emulator is found to be a good enough match for the simulator, then we proceed to calibration.

The calibration step requires experimental data. This step is a run of MCMC, which requires a prior probability and a likelihood. The prior probability is user-specified and encodes our domain knowledge about probable inputs. The likelihood is a comparison between emulator output and experimental data. The calibration step results in a representative sample from the posterior distribution of the inputs, which can be plotted to determine empirically the shape of the input distribution. This distribution will be optimal in the sense that it uses the experimental data to narrow down the input range as much as possible.

The work related to this report produced `MATLAB` code to perform emulation and validation, along with examples of the calibration process.

1.4 Structure

The remainder of this report is divided in the following way:

Section 2: Overview of Emulators and MCMC

Section 3: Process of Code Calibration with Emulators

Section 4: Demonstration with Sodium-fire Data

Section 5: Future Work - Applications to 1F2

Section 6: Conclusion

Appendix A: Code Dependencies

2 Background

To use the calibration code effectively, it is necessary to have some understanding of Gaussian processes and MCMC.

2.1 Gaussian Processes

A Gaussian process is an infinite-dimensional generalization of the multivariate normal (MVN) distribution [10, 11, 12]. Gaussian processes are fully described by their mean and covariance functions, which serve an analogous purpose to the mean vector and covariance matrix of a MVN distribution. The mean function, $\mu(\boldsymbol{x})$, provides the expected value for any input $\boldsymbol{x}$. The covariance function, $\Sigma(\boldsymbol{x}, \boldsymbol{x'})$, represents the relationship between outputs according to their corresponding inputs $\boldsymbol{x}$ and $\boldsymbol{x'}$. For any finite set of input vectors $\{\boldsymbol{x_i}\}$ with $i = 1 \ldots n$, the output of a Gaussian process is a MVN distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ such that $\mu_i = \mu(\boldsymbol{x}_i)$ and $\Sigma_{ij} = \Sigma(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

To make predictions with a GP, we begin with a set of training inputs and outputs $\{\boldsymbol{x_{tr}}, \boldsymbol{y_{tr}}\}$. We then select a set of test inputs $\boldsymbol{x_*}$ at which to evaluate the model. We assume that our test outputs $\boldsymbol{y_*}$ are drawn from the same distribution as our training outputs, so we can write the joint distribution as

$$\begin{bmatrix} \boldsymbol{y_{tr}} \\ \boldsymbol{y_*} \end{bmatrix} \sim N\left(\begin{bmatrix} \boldsymbol{\mu_{tr}} \\ \boldsymbol{\mu_*} \end{bmatrix}, \begin{bmatrix} \Sigma_{tr} & \Sigma_* \\ \Sigma_*^\top & \Sigma_{**} \end{bmatrix}\right),$$

where

$$\begin{aligned} \boldsymbol{\mu_{tr}} &= \mu(\boldsymbol{x_{tr}}) \\ \boldsymbol{\mu_*} &= \mu(\boldsymbol{x_*}) \\ \Sigma_{tr} &= \Sigma(\boldsymbol{x_{tr}}, \boldsymbol{x_{tr}}) \\ \Sigma_* &= \Sigma(\boldsymbol{x_{tr}}, \boldsymbol{x_*}), \text{ and} \\ \Sigma_{**} &= \Sigma(\boldsymbol{x_*}, \boldsymbol{x_*}). \end{aligned}$$

Because we are working with a normal distribution, it is straightforward to find the distribution of the test outputs conditioned on the known information. This posterior distribution is also MVN and is written

$$\boldsymbol{y_*} | \boldsymbol{y_{tr}} \sim N\left(\boldsymbol{\mu_*} + \Sigma_*^\top \Sigma_{tr}^{-1}(\boldsymbol{y_{tr}} - \boldsymbol{\mu_{tr}}), \Sigma_{**} - \Sigma_*^\top \Sigma_{tr}^{-1} \Sigma_*\right).$$

Although we have an analytical representation of our posterior distribution conditioned on our training data, Gaussian processes have the major drawback of running in $O(n^3)$ time, where n is the number of training inputs. This is because finding the posterior distribution requires a Cholesky factorization of the training covariance matrix, as can be seen by the Σ_{tr}^{-1} term in the equation above.

2.1.1 Hyperparameters

A Gaussian process is considered a non-parametric model because it uses all of its training data to make decisions rather than distilling the data down to a few parameters. However, we can greatly improve the flexibility of a GP by using "hyperparameters". A hyperparameter is a parameter affecting the behavior of the mean function or, more commonly, the covariance function. For example, a very common covariance function used with GPs is the squared-exponential kernel, given by

$$\Sigma(\boldsymbol{x}, \boldsymbol{x'}) = \sigma_y^2 \exp\left(-\frac{(x-x')^2}{2l^2}\right) + \sigma_n^2 \delta_{ii'}.$$

This kernel has three hyperparameters: σ_y, l, and σ_n. These three hyperparameters help the kernel adapt to different signal variance, input length scales, and signal noise, respectively. Varying them can entirely change the sort of function produced by a GP, even with the exact same input data. The usefulness of these hyperparameters is that they can be learned and thereby help the GP to adapt to the training data. Without hyperparameters, covariance functions would be much more limited in the relationships they could capture, and GPs would simply be too inflexible to be useful for most practical problems.

Learning the best hyperparameters from the training data can be done in one of several ways. One way is to set prior distributions over the hyperparameters themselves (hyperpriors) and use MCMC to find the posterior distribution. This has the benefit of giving uncertainty bounds on the hyperparameters. However, it tends to be quite slow. Another way is to find the best hyperparameters via maximum likelihood. Estimation by maximum likelihood is much faster and was the method of choice for this work. However, it does have the drawback of not producing the uncertainty estimates provided by MCMC.

2.1.2 Covariance Functions

Choosing the covariance function is the most important factor in fitting a function with a GP. Different covariance functions lead to completely different shapes of functions predicted by the emulator. Finding a covariance function that leads to a good fit of the data will likely be a process of trial and error, especially for users with little experience working with GP emulators. For this reason, it may be helpful to train and validate on a subset of the data to quickly determine likely candidates for covariance functions. Then, only emulators with the best covariance functions can be trained on and validated against all of the relevant data, and from there the best emulator can be chosen for the actual calibration.

2.2 Markov-Chain Monte Carlo

The Markov-Chain Monte Carlo (MCMC) procedure is used to draw samples from a distribution for which no closed form is available. Only a basic introduction will be given here. For the interested reader, there is a large body of literature available for understanding MCMC, such as [13]. When performing calibration, our desired result is the posterior probability distribution of the input parameters given the experimental data, written $P(\boldsymbol{\theta}|\boldsymbol{y})$, where $\boldsymbol{\theta}$ is the vector of parameters and $\boldsymbol{y}$ is the experimental output. To find this distribution, we first set a prior distribution $P(\boldsymbol{\theta})$ over our parameters based on our domain-specific knowledge of the problem. We then define the likelihood of output $\boldsymbol{y}$ given parameters $\boldsymbol{\theta}$, denoted $P(\boldsymbol{y}|\boldsymbol{\theta})$. In the case of a Gaussian-process emulator, we will assume the likelihood is Gaussian. Finally, we want to use our evidence and corresponding likelihood to update our prior belief according to Bayes' Rule. Studying Bayes' Rule,

$$P(\boldsymbol{\theta}|\boldsymbol{y}) = \frac{P(\boldsymbol{y}|\boldsymbol{\theta})P(\boldsymbol{\theta})}{\int P(\boldsymbol{y}|\boldsymbol{\theta})d\boldsymbol{\theta}},$$

we see that we need to evaluate the integral in the denominator, which in general is not possible. The power of MCMC is that it allows us to draw samples from our posterior using just our prior and likelihood, and to do so without evaluating this infeasible integral.

The output of MCMC is not a closed form for the posterior probability distribution. Instead, it produces a set of samples from the posterior that can be plotted as a histogram or eCDF to determine the shape of the distribution. The set of samples can also be used to find parameters such as the mean or standard deviation of the posterior distribution.

There is a large variety of MCMC algorithms, but most of them follow similar principles. The goal of MCMC is to build a Markov chain that has steady-state density equal to the desired posterior density, denoted $\pi(\boldsymbol{\theta})$. Once such a Markov chain is created, it is run for as many iterations as desired, and the iterates form a sample from the posterior distribution. The most important elements of the Markov chain are the proposal density $q(\boldsymbol{\theta})$ and the decision rule. At each step i of the Markov chain, the function q proposes a value $\hat{\boldsymbol{\theta}}_{i+1}$ for the next iterate. One simple proposal density is a uniform distribution about the current value. Then, the decision rule is applied to determine whether to accept $\hat{\boldsymbol{\theta}}_{i+1}$. A simple decision rule is to accept the iterate if its posterior probability is greater than the current iterate, and otherwise to accept it with a probability equal to the ratio of the posterior probabilities. That is, the new iterate $\hat{\boldsymbol{\theta}}_{i+1}$ is accepted with probability

$$\min\left(\frac{\pi(\hat{\boldsymbol{\theta}}_{i+1})}{\pi(\boldsymbol{\theta}_i)}, 1\right) = \min\left(\frac{P(\boldsymbol{y}|\hat{\boldsymbol{\theta}}_{i+1})P(\hat{\boldsymbol{\theta}}_{i+1})}{P(\boldsymbol{y}|\boldsymbol{\theta}_i)P(\boldsymbol{\theta}_i)}, 1\right).$$

If the new iterate is accepted, then $\boldsymbol{\theta}_{i+1} = \hat{\boldsymbol{\theta}}_{i+1}$. Otherwise, the Markov chain stays in the same spot, so $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i$. This rule allows us to use the unknown posterior density when making decisions because the denominator, which was the only problematic part of the Bayes update, cancels out.

The downside to MCMC is that we can never know with certainty that it has worked correctly. For a variety of reasons, such as a bad choice of q, the convergence of the Markov chain to the true distribution can be slow or even nonexistent. Determining whether a MCMC run has successfully converged requires looking at the iterates to ensure they did not stagnate and they had low autocorrelation, among other things. Diagnosing problems with MCMC is beyond the scope of this report, and the unfamiliar reader should consult related statistics literature.

This work did not involve implementation of MCMC. The user may choose any MCMC library that suits their needs. The library used for this work was an open source ensemble MCMC sampler based on [14] and [15] and available from `https://github.com/grinsted/gwmcmc`. It was chosen for several reasons, including speed, documentation, and a permissive open source license, but the main benefit was that it worked for different problems without requiring tuning.

3 Calibrating with GPEmulator Class

This section will explain the use of the `GPEmulator` class for code calibration. First, we describe the classes used for creating and validating the emulator. Then, we explain the use of these classes as part of a larger code-calibration program.

3.1 GPEmulator

The `GPEmulator` class is constructed with the training input and output for the emulator. It also requires that the user specify which of its built-in mean and covariance functions should be used. Currently, the options for the mean function are quadratic, linear, constant, zero, and exponential. The options for covariance function are squared-exponential and Matern 3/2.

The training input to a `GPEmulator` must be in a specific format that is most easily explained by an example. Assume that we have one control variable, which represents time, and two parameters, θ_1 and θ_2. Our simulator runs for time values from 0 to 2 in one-second increments. We have choose to train on the following pairs of parameters θ: $(6,8)$, $(6,9)$, $(7,8)$. The training inputs for the `GPEmulator` would be:

$$\begin{bmatrix} 0 & 6 & 8 \\ 1 & 6 & 8 \\ 2 & 6 & 8 \\ 0 & 6 & 9 \\ 1 & 6 & 9 \\ 2 & 6 & 9 \\ 0 & 7 & 9 \\ 1 & 7 & 9 \\ 2 & 7 & 9 \end{bmatrix}.$$

That is, for each set of parameters, we must have a row for that set for each time variable. This is one drawback of GP emulators: the input size grows quickly with dimension, and there is an $O(n^3)$ cost for factoring the covariance matrix, where n is the total number of rows in the input. The corresponding training output would be a single column vector containing the stacked simulator output time series for each parameter set. A `MATLAB` function, `cartesian_general`, is provided that takes as input a vector of control variables and a matrix of parameters and produces the appropriate input for a `GPEmulator`. For example, to produce the training input above, a user would call

$$\texttt{cartesian\_general}\left(\begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 6,8 \\ 6,9 \\ 7,8 \end{bmatrix}\right)$$

It is usually good practice to normalize the inputs to a GP so they have zero mean and unit variance. This normalization is done automatically by `GPEmulator` for the training input, and all later inputs used for prediction or likelihood are also automatically normalized using the same mean and standard deviation used to normalize the training inputs. The user can simply pass in the parameters they actually used and need not deal with normalization.

When the `GPEmulator` is created, it automatically trains itself on the training data to infer the hyperparameters. The maximum likelihood hyperparameters are found via optimization. Once hyperparameters have been chosen, the covariance matrix for the GP is created and factored. When the training process is complete, the `GPEmulator` is ready to make predictions or be used in MCMC.

Before the `GPEmulator` can safely be used, however, it should be run through a set of validation tests to ensure it matches the simulator. The validation tests must be performed with a set of data that was not seen by the emulator during training. There is a utility class, `GPValidator`, that accepts a set of validation data and runs a variety of diagnostics on a `GPEmulator`. This class will be described in more depth in the next section. Once a user has performed validation tests on the emulator and is satisfied with its performance, the user can proceed to the calibration process.

A `GPEmulator` has four public functions available: `predict`, `covariance`, `errorbars`, and `likelihood`. Each of `predict`, `covariance`, and `errorbars` takes posterior inputs in the same format as the inputs described above. The `predict` function returns the expected value of the GP at the inputs. The `covariance` function returns the posterior covariance matrix. The `errorbars` function returns half of the width of the ninety-five percent credible interval for the prediction, which is helpful when plotting the uncertainty in the GP prediction. The final function, `likelihood`, accepts two inputs: posterior inputs, as described above, and posterior outputs. This function gives the likelihood of the posterior outputs given the posterior inputs and the GP model trained on the training inputs. This is the function that, given a set of experimental outputs, can be used with MCMC to find the distribution of inputs.

The `GPEmulator` was designed to be similar to the `RegressionGP` class available in `MATLAB`, though `GPEmulator` has fewer options. The `RegressionGP` class, however, could not be used to run MCMC with the emulator because the class did not provide a posterior likelihood function or the factorized training covariance matrix that is used in calculating the posterior likelihood.

3.1.1 GPEmulatorGPML

There is a popular open source library for GPs, GPML, that provides everything needed to perform calibration. Our code provides a wrapper around GPML called `GPEmulatorGPML`. This class has exactly the same interface as `GPEmulator`, so they can easily be swapped for one another in the same code. To maintain this compatibility and ease of use, the wrapper provides access to only a subset of GPML.

The benefit of interfacing with GPML is that is provides access to more options for covariance functions, which allows for more flexibility in the emulator. There are two downsides, however. The first and most important downside is that `GPEmulatorGPML` is noticeably slower than `GPEmulator` for calibration. The second is that GPML does not provide access to the factorized training covariance matrix, which is used to calculate the posterior covariance matrix. This means that the `covariance` function is not available with a `GPEmulatorGPML`. Also, several of the validation tests provided by `GPValidator` require the covariance matrix, and thus cannot be performed. Instead, a `GPEmulatorGPML` is validated with a `GPValidatorGPML`, described briefly in the next section, which runs fewer tests. Due to these drawbacks, we recommend using `GPEmulator` whenever possible and switching to `GPEmulatorGPML` when needed to model more difficult simulators.

3.2 **`GPValidator`**

The `GPValidator` class runs a `GPEmulator` through a series of diagnostics described in [16]. It accepts as inputs a `GPEmulator`, validation inputs in the same format as the inputs to a `GPEmulator`, the control variables from the validation inputs, and the validation outputs corresponding the the validation inputs.

As a concrete example, if we with to validate at times 1 through 3 and parameters $(4,6)$, $(4,7)$, and $(5,7)$, we would use the following call:

$$\texttt{GPValidator}\left(\texttt{emulator,cartesian\_general}\left(\begin{bmatrix}1\\2\\3\end{bmatrix},\begin{bmatrix}4,6\\4,7\\5,7\end{bmatrix}\right),\begin{bmatrix}1\\2\\3\end{bmatrix},\texttt{val\_outputs}\right).$$

When a `GPValidator` is created, it automatically calculates validation errors and decorrelated validation errors. After the errors are calculated, the user can check diagnostics described in [16], as well as simply plotting the emulator output against the validation output against the validation inputs. It is recommended to run the `display_all` routine, which runs all available diagnostics and presents them to the user. Section 3 of [16] explains how to interpret each of the diagnostics.

3.2.1 GPValidatorGPML

To validate a `GPEmulatorGPML` emulator, described in a previous section, the `GPValidatorGPML` class is required. This class does not run any of the diagnostics in [16]. Instead, it simply plots the emulator vs the simulator on the validation data to help the user spot any issues. It is created and run with exactly the same interface as a `GPValidator`.

3.3 Usage for Code Calibration

This subsection will describe the process of calibration using a `GPEmulator` from beginning to end. The first step is to decide on appropriate ranges for each input of the GP. There will usually be at least one control input, which is often time. The range for this variable should be known based on the experimental data. Then, an appropriate distribution is chosen separately for each parameter, and a Latin-hypercube sample is taken over the distributions. `MATLAB` has a built-in routine, `lhsdesign`, that can be used for this. The distributions for the parameters should be chosen to cover all likely values, but a simulator run must be performed for each set of parameters, so there is a tradeoff between runtime and coverage.

Once the parameter combinations are decided on, the simulator is run for each parameter combination. Then, the inputs and outputs must be shaped into a form recognized by the `GPEmulator` class, as described in the `GPEmulator` section. The majority of the code will likely be dedicated to creating the inputs and outputs, since the rest of the process is quite straightforward.

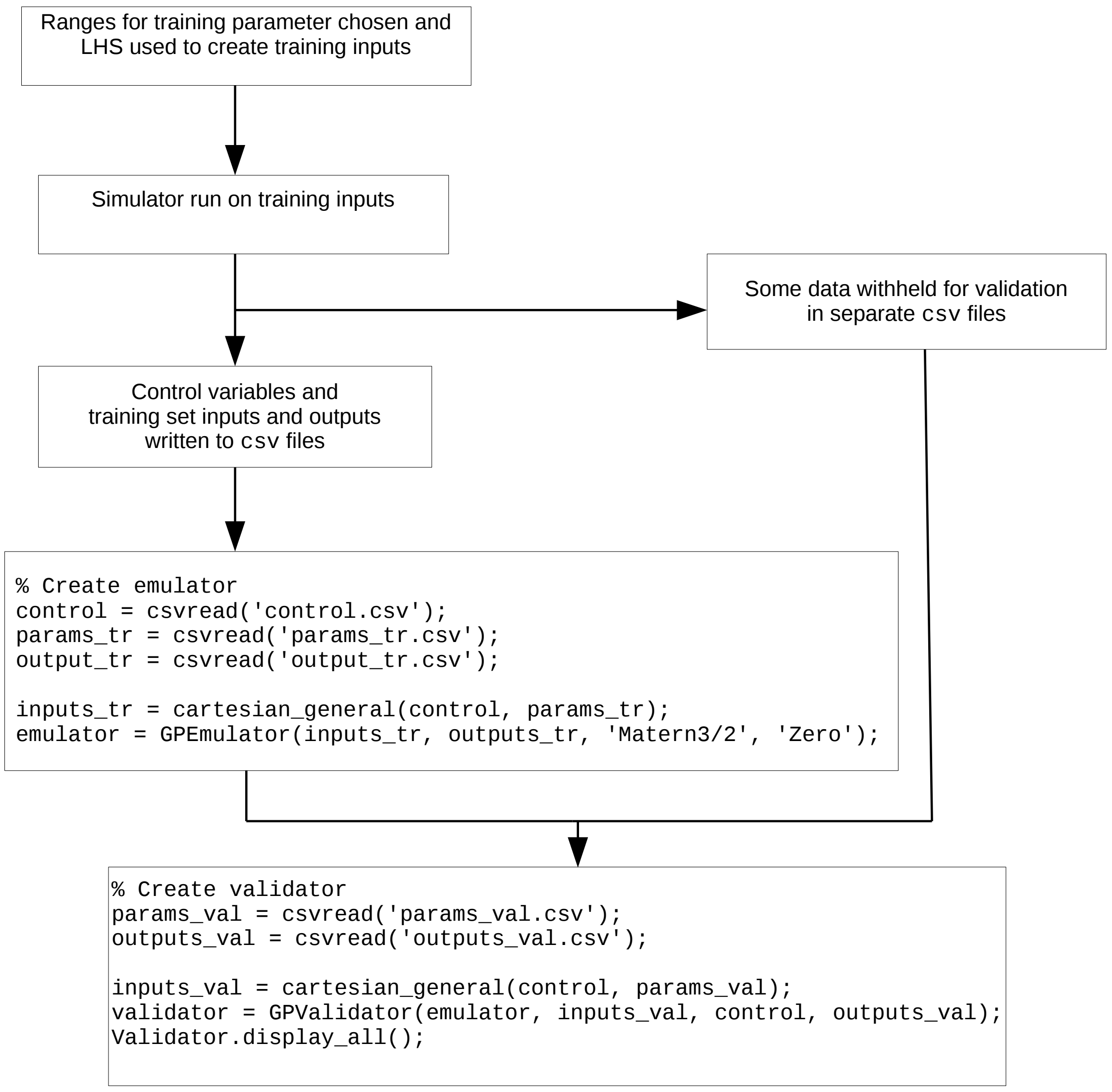
After the `GPEmulator` is created, it must be examined using a `GPValidator`, which requires choosing a set of validation inputs. These inputs should be from within the same general range as the training data, but the parameters must not have been used for training the emulator, although the control variables will likely be the same. The simulator must also be run for each set of validation parameters to produce the validation output that the `GPValidator` compares against. Finally, before being passed to the `GPValidator`, the validation inputs must be put into the same form as training inputs using `cartesian_general`. Once the validation inputs and outputs are available and in the correct form, the `GPValidator` is created and run, and the user can inspect the output to confirm that the `GPEmulator` is working well.

The process of creating and validating an emulator is shown graphically in Figure 1.

After the user has a validated `GPEmulator`, they can begin calibration with MCMC. First, a log prior function is specified. This function should accept only parameters as input, and should not accept the control variables. A log likelihood must also be specified, and must take the same inputs in the same order as the log prior. The log likelihood will generally be an anonymous function that calls the emulator likelihood function with the control variables and the posterior output already specified. The posterior output is the experimental output, which will need to be read in from a file.

Once the prior and likelihood functions are available, the MCMC function is called with a suitable initial guess. This will return the desired distribution of calibrated parameters. As is the case with any MCMC routine, the user should check the output to ensure good mixing and low autocorrelation of the output before trusting the results.

The process of creating the inputs to the MCMC routine is displayed in Figure 2.


Ranges for training parameter chosen and LHS used to create training inputs
Simulator run on training inputs
Some data withheld for validation in separate csv files
Control variables and training set inputs and outputs written to csv files

```
% Create emulator
control = csvread('control.csv');
params_tr = csvread('params_tr.csv');
output_tr = csvread('output_tr.csv');

inputs_tr = cartesian_general(control, params_tr);
emulator = GPEmulator(inputs_tr, outputs_tr, 'Matern3/2', 'Zero');
```

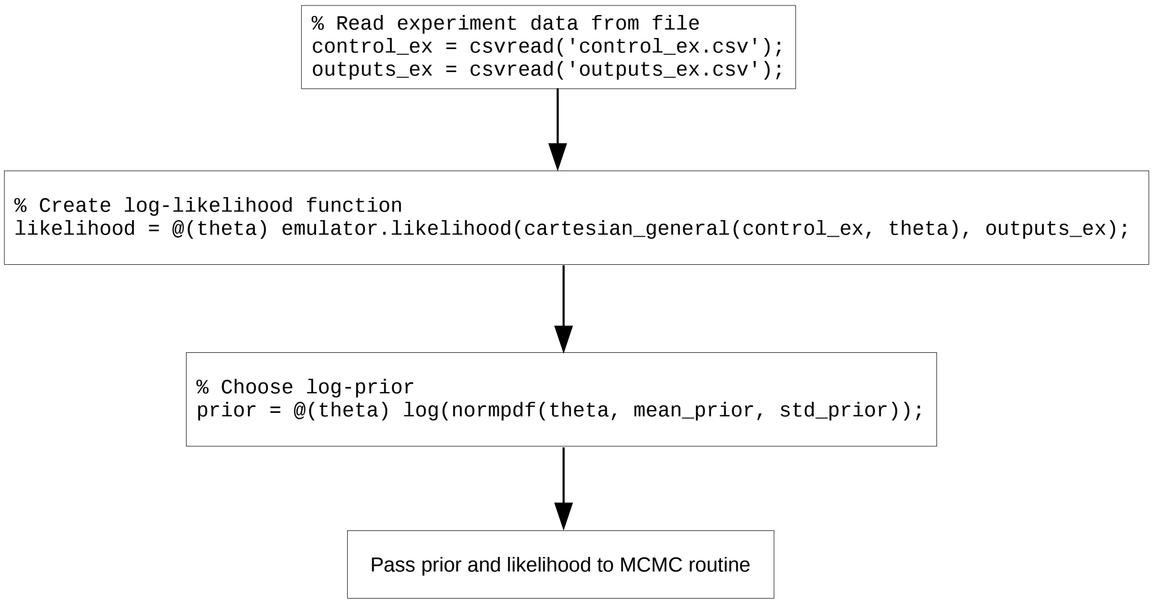
```
% Create validator
params_val = csvread('params_val.csv');
outputs_val = csvread('outputs_val.csv');

inputs_val = cartesian_general(control, params_val);
validator = GPValidator(emulator, inputs_val, control, outputs_val);
Validator.display_all();
```

Figure 1: Creation and Validation

3.3.1 Using Multiple Experiments

A useful feature of the calibration process is that a simulator can be calibrated against more experiments as they become available. The process is theoretically straightforward: once we have calibrated against an experiment, our posterior is our new current belief, so we can take our posterior as the prior for all future experiments. Another way to look at this is that since we have seen the experimental data, all future beliefs are conditioned on that experiment even though we do not usually write this out explicitly.



```
% Read experiment data from file
control_ex = csvread('control_ex.csv');
outputs_ex = csvread('outputs_ex.csv');
```

```
% Create log-likelihood function
likelihood = @(theta) emulator.likelihood(cartesian_general(control_ex, theta), outputs_ex);
```

```
% Choose log-prior
prior = @(theta) log(normpdf(theta, mean_prior, std_prior));
```

Figure 2: Preparing for MCMC

If the posterior parameters appear to follow an analytically representable distribution, such as a normal distribution with some mean and covariance that we can find from the MCMC output, then we can approximate our posterior with this distribution and our calculations become simple. If, however, the distribution does not appear to match a common probability distribution, then we need to represent our probability density function (PDF) empirically. In this work, we did so by first creating an empirical cumulative distribution function (eCDF). We then used this eCDF to create an empirical piecewise-linear PDF. Example `MATLAB` code for obtaining the PDF and evaluating the new prior probability of a parameter is given below:

```
[prior_cdf, prior_x] = ecdf(draws_from_mcmc);
prior_x(1) = prior_x(1) - 1e-10;
prior_distribution = makedist('PiecewiseLinear', 'x', prior_x', 'Fx', prior_cdf');
prior_distribution.pdf(some_param);
```

4 Demonstration with Sodium-fire Data

This section briefly describes the use of our calibration code on a real simulator with actual experimental data. The simulator used was CONTAIN-LMR [17, 7], a tool used for modeling sodium fires. We had two related experiments to calibrate the code against, referred to as the T3 and T4 experiments [6].

Because most of the experimental setup stayed consistent between T3 and T4, we were able to choose inputs that should be the same for both experiments and use the data from both experiments to calibrate these parameters sequentially. This ability to update our distributions whenever new data is available is a powerful feature of Bayesian calibration.

We simultaneously calibrated three parameters of CONTAIN-LMR against the experiments. Each parameter began with a different prior distribution. The calibration process was run first with the T3 data. This updated the distributions of the three parameters to account for the experiment. These updated distributions from the T3 calibration became the priors, representing our additional knowledge of the system learned from T3, and the calibration process was repeated with T4. The final output distributions from the T4 calibration represent our best knowledge of the parameters given all of the available experimental data.

Figure 3 compares the emulator against the simulator on some of the validation data for the T3 experiment. The fit is not perfect, but about 91% of the simulation data points lie within the 95% confidence interval of the emulator, and the emulator was deemed adequate for calibration. The validation results for T4 were similar and are not shown.

The effect of the sequential calibrations on our parameter distributions is shown graphically in Figures 4-6. Since the posterior distribution after the T3 experiment was not a close match for a common distribution, an empirical distribution was created as described in Section 3.3.1 and used as the prior for the T4 calibration. We see from the plots that each experiment helps to narrow the probability down into more specific regions for the three parameters, which is what we would hope to happen.

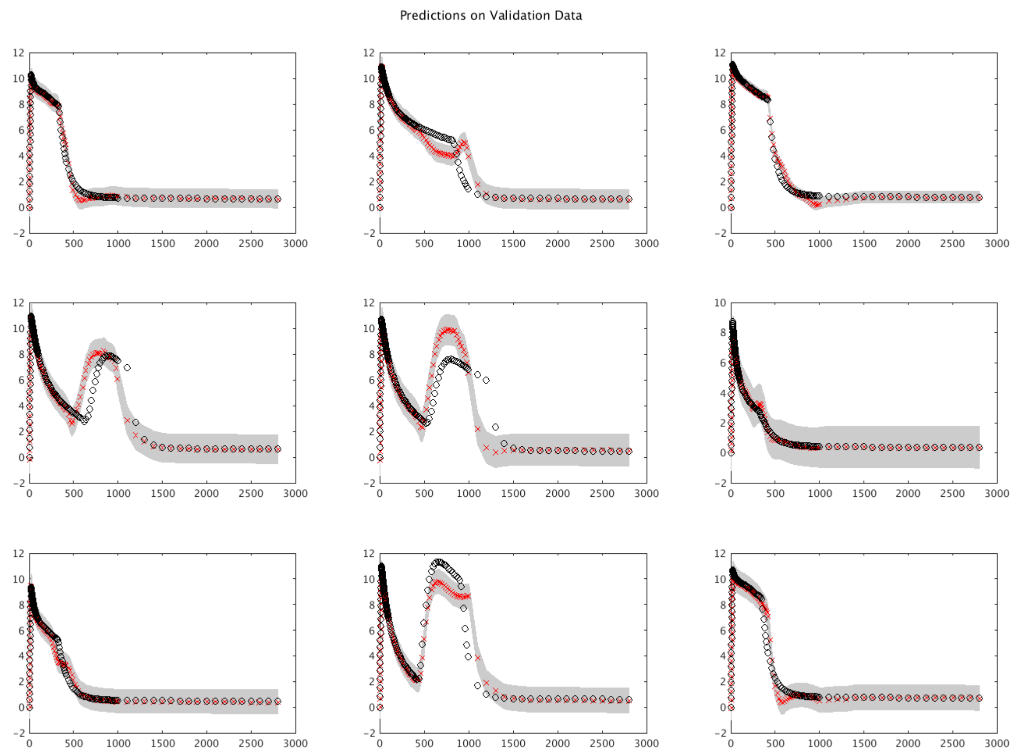


Figure 3: Validation of emulator for T3

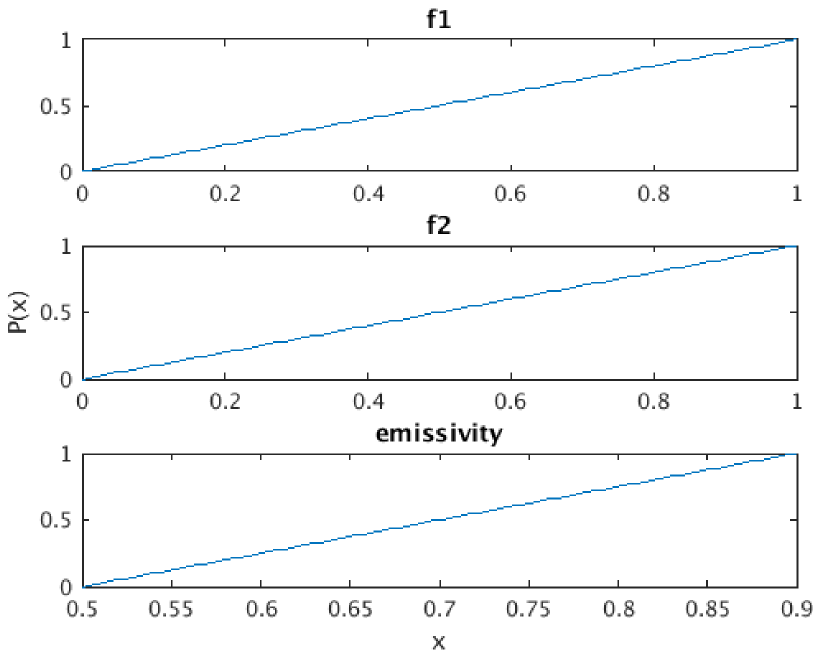


Figure 4: CDFs of flat priors

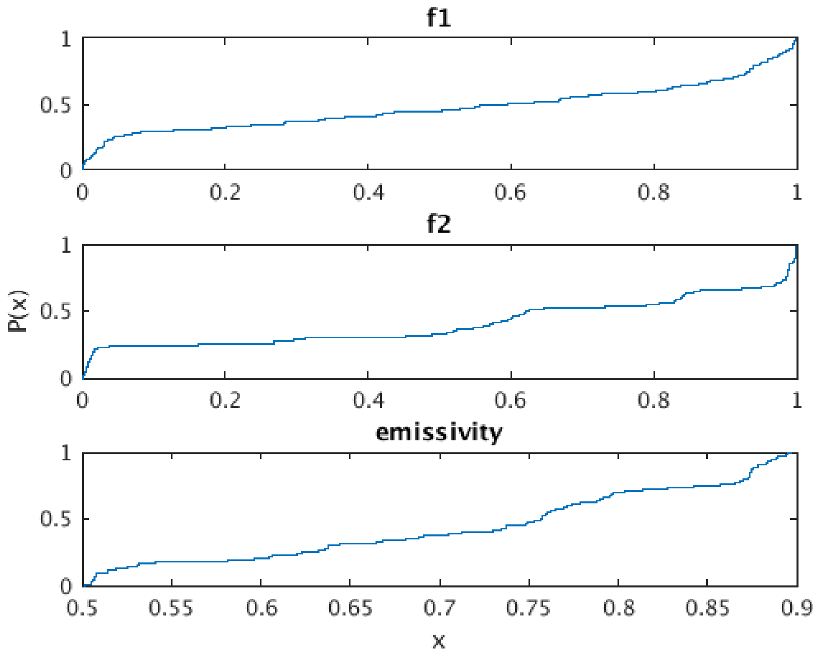


Figure 5: Posterior CDFs after T3

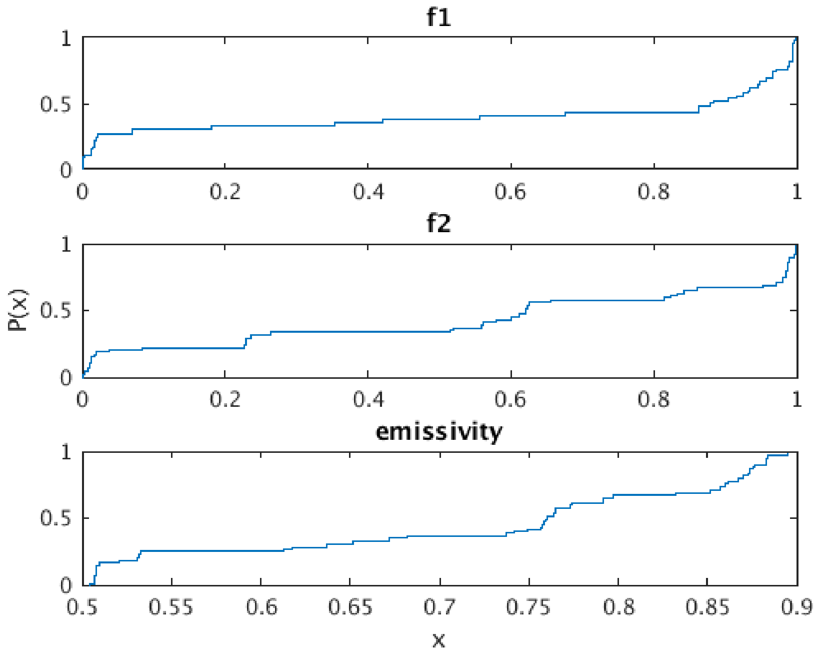


Figure 6: Posterior CDFs after both calibrations

5 Future Work - Applications to 1F2

This section will describe the potential benifits of the Bayesian calibration method described in this report on the 1F2 accident reconstruction analysis. Information on the 1F2 accident, including the offical data-sets, can be found at the fdata website[18]. The 1F2 accident reconstruction efforts are currently ongoing; thus the results presented in this section are not the most current 1F2 reconstructed results. These preliminary results are an illustrative representation of the current approach and the demonstrate opportunity presented by the Bayesian calibration technique.

5.1 Current Calibration Approach

In addition to the phenomena related accident progression uncertainties, many accident sequence and timing uncertainties exist in the 1F2 reconstruction analysis. Key questions in the first 80 hours of the accident may include:

- Was the fraction of the Reactor Core Isolation Cooling (RCIC) injection rerouted back to the Condensate Storage Tank (CST):
  - only rerouting RCIC injection during RCIC throttling or
  - continually redirecting RCIC injection to the CST throughout CST suction?
- How much did steam quality reduce RCIC injection performance once the Main Steam Line (MSL) flooded?
- Did the CST isolate when TEPCO states (13.5 hours to 14.2 hours) or did the CST isolate when the Reactor Pressure Vessel (RPV) pressure data starts to increase at approximately 11 hours?
- If the test line was redirecting RCIC injection to the CST during the CST suction, did that test line continue to redirect RCIC injection to the CST continue during WW suction?
- Are the early RPV pressure readings in error due difficulties in re-energizing equipment?
- Was the torus room flooded during the initial portion of the accident? If so:
  - When did the flooding occur?
  - How much flooding occurred?
  - When did the flooding stop?

Currently, simple Monte Carlo analyses are being used to determine the impact of various input parameters on the code output. The input distributions for the current 1F2 calibration approach are bounded not by physical constraints on the input parameters. Instead they are tailored by the analyst to improve batches of 20-50 MELCOR simulation results when compared to instrumentation data. An iterative approach is taken where visual trends in the input parameter's influence on the output horsetails are noted and the input distributions are adjusted accordingly to provide a better fit to the experimental results for the next batch of 20-50 MELCOR simulations. Unfortunately, what constitutes a better fit to the instrumentation data may vary from analyst-to-analyst and/or day-to-day. Additionally, there is little to no official acknowledgment given to the impacts that drift in instrumentation accuracy throughout the transient may have on the definition of a good fit to experimental data-sets.

The current calibration approach samples all uncertain parameter's in unison, but compares the agreement with data in isolation; even though input parameter interactions many impact various portions of the accident progression. Figures 7, 8, 9, and 10 illustrate this point with RPV pressure trends.

1. Figure 7 shows that that larger parameters for the higher void fraction operational range parameter, c, allow for higher RCIC flow rates which monotonically cools off the water in the RPV more quickly than smaller values for c. This monotonic influence on RPV pressure all but disappears upon the occurrence of CST switch-over.
2. Figure 8 shows that hours after CST switch-over, RPV pressure decreases approximately monotonically with later switch-over times.
3. Figure 9 shows that the RCIC pump parameter for the void fraction operational range term, c, nearly monotonically influences RPV pressure trends between 20 and 45 hours.
4. Figure 10 shows that between 40 and 70 hours the torus room flooding timing monotonically influences RPV pressure.

While simple Monte Carlo can show the existence of these trends, the process for manually adjusting the sample ranges and accounting for temporal dependencies can be time-consuming. The current calibration approach requires expert judgment regarding the suitability of overshooting the 1F2 data in some areas and undershooting the 1F2 data in other areas. The calibration method proposed in this report will use GP meta models to help automate and formalize these judgments through a scrutable and efficient process.

5.2 Correlated Unequal Data-sets

One major calibration related question that remains unsolved is: How should multiple instrumentation data-sets which are inherently correlated be processed though the calibration model?

Examine the correlations evident in the Wetwell flooding distribution's impact on RPV pressure in Figure 10 and compare those results to the Wetwell pressure in Figure 11 and the Drywell pressure in Figure 12. All three volumes are in communication with each other during the first 70 hours of the accident; thus their pressure trends should, and do, appear correlated. If one ignores this correlation, the prototypical Bayesian calibration approach would be to:

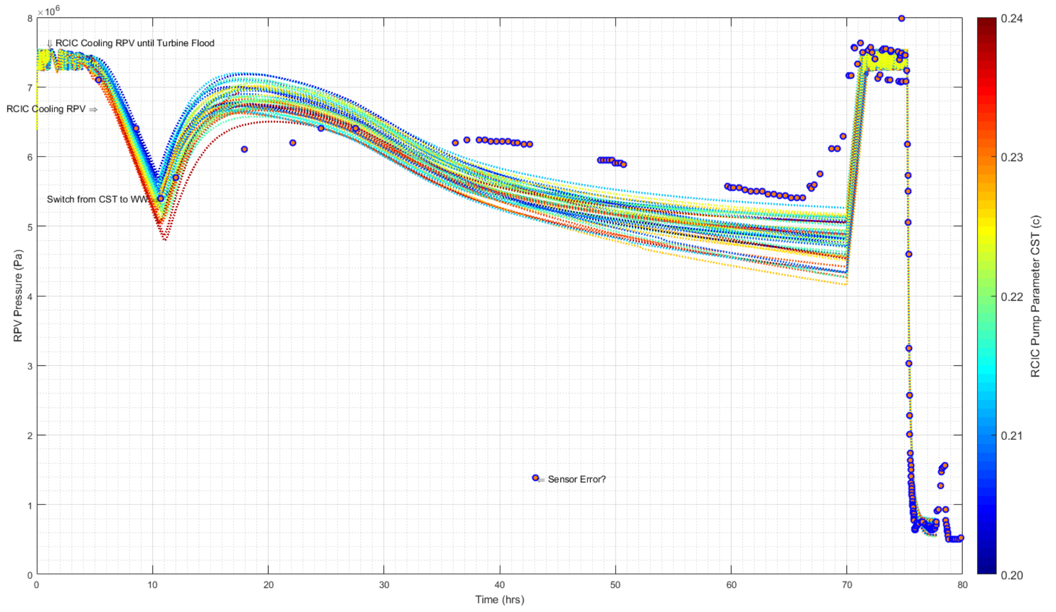


Figure 7: 1F2 Reactor Pressure Vessel (RPV) Pressure Horse Tails Color Coded by Reactor Core Isolation Cooling (RCIC) Pump Void Fraction Operating Range with Suction from the Condensate Storage Tank(CST)

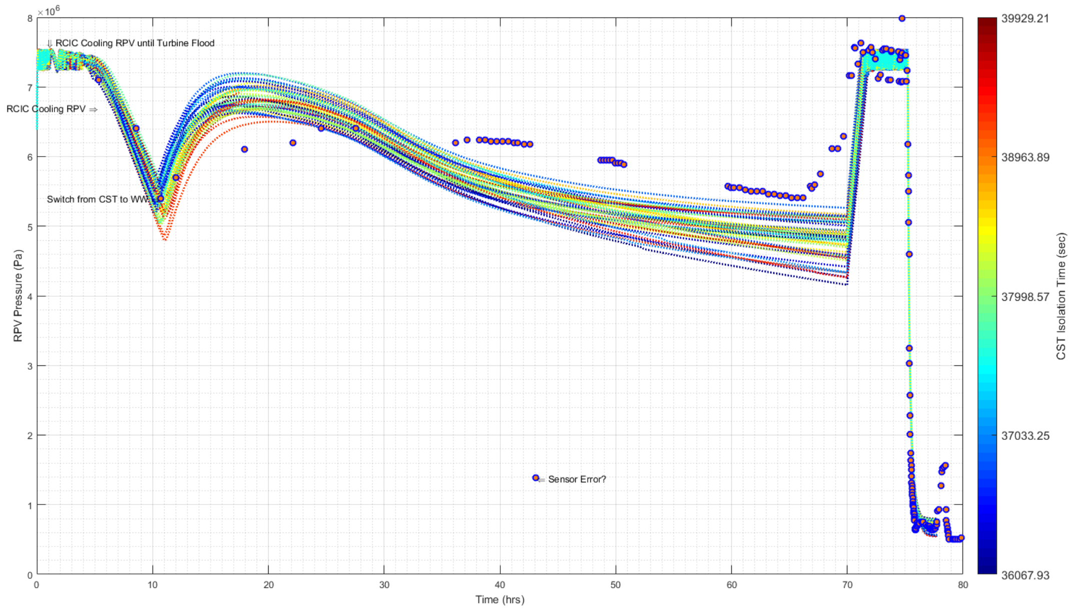


Figure 8: 1F2 Reactor Pressure Vessel (RPV) Pressure Horse Tails Color Coded by Condensate Storage Tank Isolation Timing

1. Assume a prior distribution for calibration parameters.

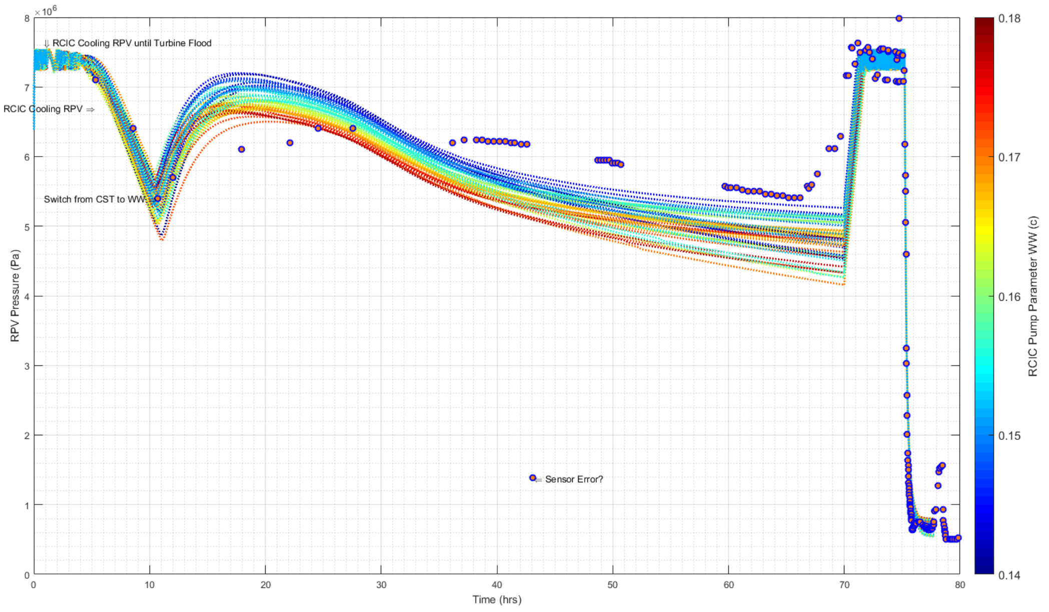


Figure 9: 1F2 Reactor Pressure Vessel (RPV) Pressure Horse Tails Color Coded by Reactor Core Isolation Cooling (RCIC) Pump Void Fraction Operating Range with Suction from the Wetwell (WW)

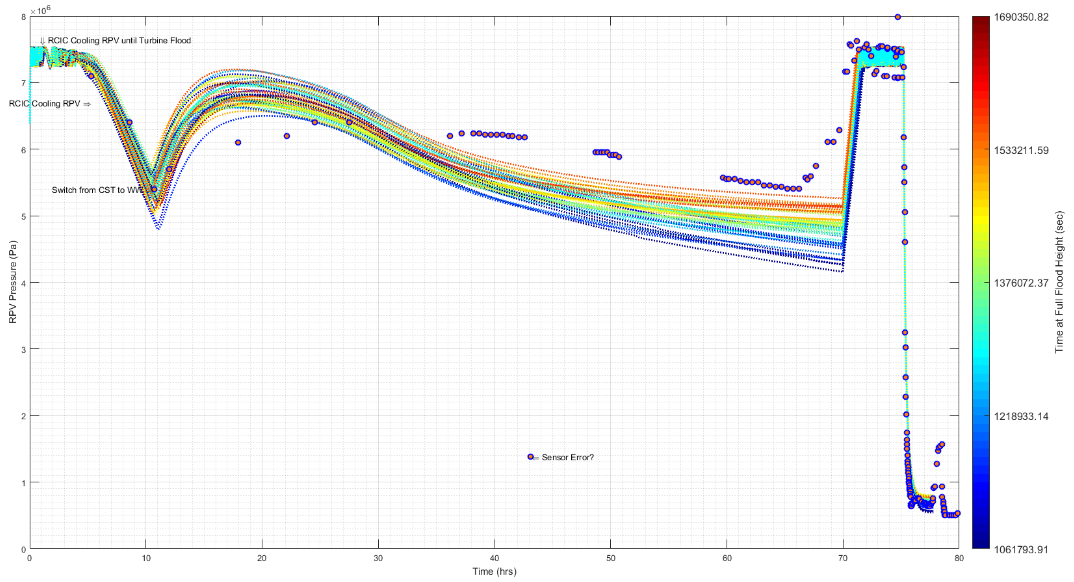


Figure 10: 1F2 Reactor Pressure Vessel (RPV) Pressure Horse Tails Color Coded by Flood Timing

2. Take the assumed prior distribution and apply the calibration approach for RPV pressure data.

3. Take the RPV pressure data informed posterior distribution and using it as the prior distribution for Wetwell Pressure data.

4. Take the Wetwell pressure data informed posterior distribution and using it as the prior for Drywell pressure data.

5. Continue daisy-chaining experimental data-sets until one is left with a final posterior distributions on the calibration parameters.

6. Running Monte Carlo sampling on this posterior distribution should give the analyst the best possible understanding of the accident progression for that set of input parameters.

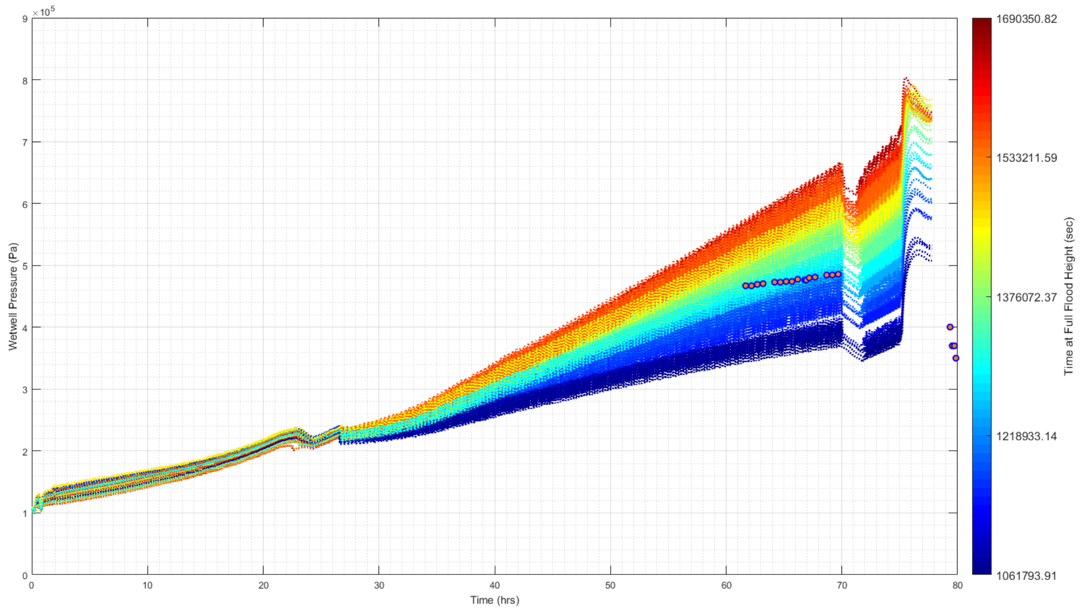


Figure 11: 1F2 Wetwell (WW) Pressure Horse Tails Color Coded by Flood Timing

This approach likely gives too much weight to the overall experimental data-sets due to the unaccounted for correlation structure in the the data-sets. This known-unknown error may be acceptable given the large uncertainties and unformalized calibration approach currently employed on the accident reconstruction effort.

5.3 Additional 1F2 Information Will Increase the Accuracy of this Technique

A key input-set to the Bayesian calibration approach is the uncertainty in the experimental data-set used for the correlation. Any initial calibration trials will use assumed uncertainty distributions for this instrumentation error. The following information will be extremely valuable for improving the accuracy of future input parameter uncertainty calibrations.

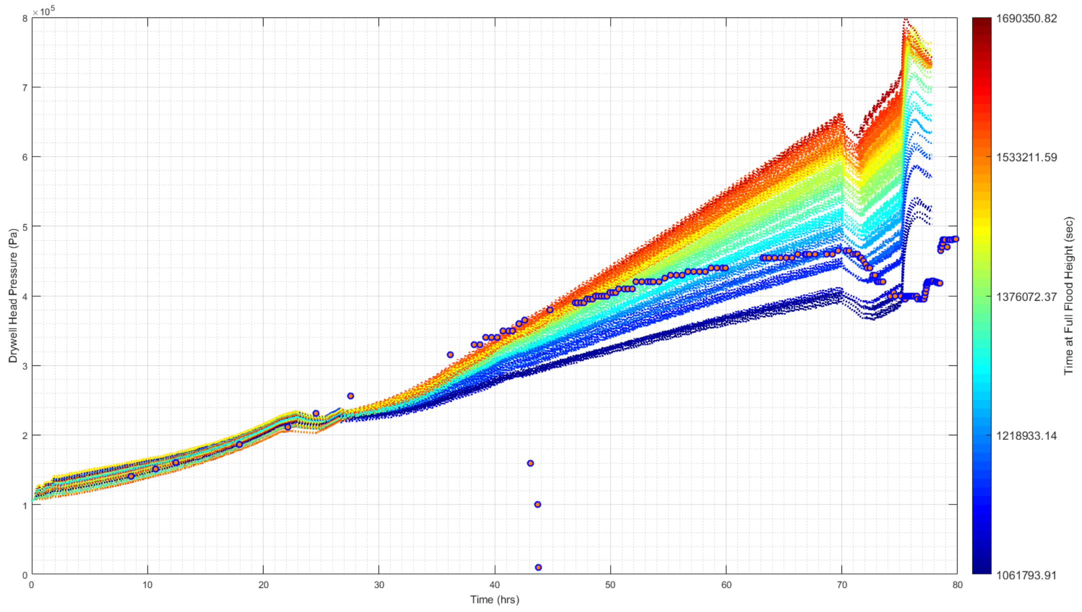


Figure 12: 1F2 Drywell (DW)Pressure Horse Tails Color Coded by Flood Timing

- Initial uncertainties in 1F2 instrumentation,
- Calibration ranges of 1F2 instrumentation,
- Degradation of instrumentation calibration, and thus increase in instrumentation uncertainties, as a function of time and environment, and
- Uncertainties in the timing of data collection efforts, especially in the early and sporadic phase of data acquisition.

5.4 Summary and Future Work

The Bayesian calibration approach can potentially formalize, streamline, and accelerate the overall calibration process for the 1F2 accident reconstruction. The Bayesian calibration process has not yet been tested with the 1F2 model because the 1F2 model is slower, more complicated. There are also larger uncertainties associated with the 1F2 instrument data than there were with the SNL sodium fire experiments. Now that the Bayesian Calibration development process is in a semi-finalized state, it is expected that the implementation of the Bayesian calibration approach on 1F2 accident sequence input parameters will begin in the Fall of 2016.

6 Conclusion

We have implemented fast Bayesian calibration of complex computer codes using a Gaussian process emulator. Our research produced fast, easy-to-use code for GP emulation and validation. Along with an open source MCMC library, our emulator was able to calibrate the inputs of a sodium-fire simulator to related experimental data and significantly decrease uncertainty in the input parameters. In addition to our fast emulator, we wrote a compatible wrapper to a powerful open source GP library that provides the flexibility needed for emulating even more complicated simulators at the cost of some speed.

By following the process laid out in this report with the help of our code, we hope that researchers will be able to give a principled justification of their code parameters based on validation experiments and to decrease the output uncertainty of their simulations by improving their knowledge of inputs.

References

[1] D. Higdon, M. Kennedy, J. C. Cavendish, J. A. Cafeo, and R. D. Ryne, "Combining field data and computer simulations for calibration and prediction," *SIAM Journal on Scientific Computing*, vol. 26, no. 2, pp. 448–466, 2004.

[2] L. L. Humphries, R. K. Cole, V. G. Louie, D.L.and Figueroa, and M. F. Young, "Melcor computer code manuals, vol. 1: Primer and users guide, version 2.1.6840," Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND 2015-6691 R, 2015.

[3] L. L. Humphries, R. K. Cole, D. L. Louie, V. G. Figueroa, and M. F. Young, "Melcor computer code manuals, vol. 2: Reference manual, version 2.1.6840," Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND 2015-6692 R, 2015.

[4] M. R. Denman, "Development of the sharkfin distribution for fuel lifetime estimates in severe accident codes," in *American Nuclear Society Winter Meeting*, Nov. 2016.

[5] L. Humphries, D. L. Louie, V. G. Figueroa, M. F. Young, S. Weber, K. Ross, J. Phillips, and R. J. Jun, "Melcor computer code manuals, vol. 3: Melcor assessment problems, version 2.1.7347," Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND 2015-6693 R, 2015.

[6] T. J. Olivier *et al.*, "Metal fires and their implications for advance reactors part 3: Experimental and modeling results," Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND2010-7113, 2010.

[7] K. K. Murata *et al.*, "CONTAIN LMR/1B-Mod. 1: A computer code for containment analysis of accidents in liquid-metal-cooled nuclear reactors," Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND91-1490, 1993.

[8] M. E. Tipping, "Bayesian inference: an introduction to principles and practice in machine learning," in *Advanced Lectures on Machine Learning*, O. Bousquet, U. von Luxburg, and G. Ratsch, Eds. Springer, 2003, pp. 41–62.

[9] J. P. Yurko, J. Buongiorno, and R. Youngblood, "Demonstration of emulator-based Bayesian calibration of safety analysis codes: Theory and formulation," *Science and Technology of Nuclear Installations*, 2015.

[10] C. Rasmussen, "Gaussian processes in machine learning," in *Advanced Lectures on Machine Learning*, O. Bousquet, U. von Luxburg, and G. Ratsch, Eds. Springer, 2003, pp. 63–71.

[11] C. K. I. Williams, "Prediction with Gaussian processes: from linear regression to linear prediction and beyond," Birmingham, United Kingdom, Tech. Rep., 1997.

[12] S. Roberts, M. Osborne, M. Ebden, S. Reece, N. Gibson, and S. Aigrain, "Gaussian processes for time-series modelling," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1984, 2013.

[13] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, “An introduction to MCMC for machine learning,” *Machine Learning*, vol. 50, no. 1, pp. 5–43, 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1020281327116

[14] J. Goodman and J. Weare, “Ensemble samplers with affine invariance,” *Communications in Applied Mathematics and Computational Science*, vol. 5, no. 1, pp. 65–80, 2010.

[15] D. Foreman-Mackey, D. W. Hogg, D. Lang, and J. Goodman, “emcee: The MCMC hammer,” *Publications of the Astronomical Society of the Pacific*, vol. 125, no. 925, p. 306, 2013.

[16] L. S. Bastos and A. OHagan, “Diagnostics for Gaussian process emulators,” *Technometrics*, vol. 51, no. 4, pp. 425–438, 2009.

[17] K. K. Murata *et al.*, “User’s manual for CONTAIN 1.1: A computer code for severe nuclear reactor accident containment analysis,” Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND87-2309, 1989.

[18] TEPCO. (2016) Fdata. [Online]. Available: https://member.fdada.info/projects/bsaf

A Appendix A: Code Dependencies

The Bayesian Calibration code used in this report has some external dependencies that need to be linked to in MATLAB in order for the code to execute. These dependencies are:

- **GPML** The optional GPEmulatorGPML needs files from the following link - `http://www.gaussianprocess.org/gpml/code/matlab/doc/`
- **GWMCMC** One of many acceptable MCMC packages can be found at the following link - `https://www.mathworks.com/matlabcentral/fileexchange/49820-ensemble-mcmc-sampler`
- **MATLAB** Statistics and Machine Learning Toolbox

DISTRIBUTION:

1	MS	0748	Matthew Denman, 6231
1	MS	0784	Mitch McCrory, 6231
1	MS	0784	Randy Gauntt, 6232
1	MS	0784	Nathan Andrews, 6232
1	MS	0784	Jeff Cardoni, 6232
1	MS	0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories