Implementing Bayesian History Matching

Brian Gu

January 2016

Contents

1 Introduction

Mathematical models in epidemiology have become a key tool in understanding the spread of disease and the effects of various interventions. These models are often referred to as *simulators*, processes which take in demographic and environmental data as input parameters, and stochastically generate predicted values for various outputs we wish to measure. One important problem in epidemiology is the *calibration* of simulators, or the problem of choosing appropriate inputs. Generally, we wish to choosing input parameters such that the outputs

are aligned with the real-world empirical data. Unfortunately, simulators can be computationally expensive to run, so attempts to search over parameter space for appropriate input combinations can quickly become impractical.

This report demonstrates the process of Bayesian History Matching, described by Andrianakis et al., through a simple example problem. We show how to modularize the various components of the history matching process in a way that can be extended to calibrate more complex models, potentially on the scale of IDM's simulators. This existing implementation is also vectorized, to allow for potential extension.

2 Requirements

Running the code requires Python 2.7 and the following packages, which can be installed via Anaconda:

- `scipy`
- `numpy`
- `pyGPs`

3 Usage

To run the program, navigate to the project folder (currently named `toy_gpr`) in the command line. Enter `python main.py` to run the program. First, ensure that all files (`distribution.py`, `emulator.py`, `main.py`, `objfunc.py`, `sampling.py`, and `simulator.py`) are saved in the project folder. The following parameters can be changed via the `main.py` file:

- **simFunc**: The function returning the expected value of simulator outputs. The simulator is based on this function and a noise function available in `simulator.py` (noise is currently sampled from a normal distribution with mean 0). If the simulator is made complex, it should probably be moved to `simulator.py`.
- **threshold**: In the current problem, **threshold** represents the current lower bound for simulator outputs to be deemed "plausible." In other words, the plausible region is meant to be the region of all inputs which can likely result in an output above **threshold**. For more complex versions of the History Matching problem, this should be changed to the target empirical data (and the criteria for the **reject** function in `emulator.py` should be changed to reflect the fact that we wish to find to find inputs near a target vector, rather than over a threshold value.)
- **rejectionDensity**: The density of the probability mass of the emulator's output distribution which we require to be above **threshold** needed to deem a point non-implausible in the current wave (in other words, to not reject the point). For example, if **rejectionDensity** is set to 0.95, then at least 5% of the probability mass of the emulator's predicted simulator distribution at a given input must be over **threshold** to not reject that input. This serves as an implicit measure of implausibility; in the current

implementation, it means that an input's predicted output must be within 2 standard deviations away from the target/threshold not to be rejected. It may be better to write an implausibility measure and to rewrite the **reject** function in `emulator.py` to better reflect the algorithm description from Andrianakis et al.'s paper.

- **emulationPointsPerSamplePoint**: Every wave, we sample points from around each simulated point deemed non-implausible. We emulate at these newly sampled points, rejecting some as implausible. This determines the number of points we sample around each simulated point for emulation.

- **numberOfWaves**: The simulation/emulation process iterates a fixed number of times, given by this parameter. Thus, this currently serves as the termination condition. Future work would need to be done to determine a more appropriate termination condition.

- **simulationsPerWave**: Determines the number of inputs we run the simulator on in each wave.

- **simulationsAtEachPt**: For each input vector, the simulator will be run **simulationsAtEachPt** times on that input. If this value is set to more than 1, this can give us a sense for the variance of outputs on a fixed input. Note that, in total, the simulator function is run **simulationsPerWave** $\times$ **simulationsAtEachPt** times each wave.

The following parameter can be changed via the `simulator.py` file:

- **epsvar**: The variance of the noise function of the toy simulator.

4 Overview of History Matching

This is an overview of the history matching procedure described by Adrianakis et al., with a few slight modifications.

Given a simulator $f: \mathbb{R}^n \to \mathbb{R}^m$ and a target output $\mathbf{y} \in \mathbb{R}^m$ (often set from empirical data), we wish to find appropriate sets of input parameters $\mathbf{x} \in \mathbb{R}^n$ calibrated to the target output. Because running the simulator can be computationally expensive, we rely heavily on a more efficient process called an *emulator*. An emulator, given information on previous simulator runs, can approximate simulator outputs at any other point in parameter space. It does so by performing a Gaussian process regression on previous simulator runs to find the expectation and variance of simulator outputs for new input combinations, under the typical assumption of GPR that outputs at nearby points are correlated.

The goal of the History Matching algorithm presented here is to find plausible input combinations by rejecting large parts of parameter space as *implausible* (very unlikely to match the empirical data), using the emulator. Every time a *wave* (iteration) of the algorithm is completed, the non-implausible region should shrink. A more rigorous definition of *implausible* will be given later.

First, an initial set of n points A_0 in parameter space is chosen for initial simulator runs. These points are sampled through the Latin Hypercube Sampling method. Once the

simulator is run at these points, we choose a set of points B_0 to emulate at, using the data from the simulations at points in A_0. We construct B_0 by sampling in the spaces near non-implausible points from A_0: for each point $\mathbf{a} \in A_0$ that is not implausible, we center a multivariate Gaussian at $\mathbf{a}$ and draw k samples from this Gaussian. Ideally, the variance of the Gaussian should be large enough so that the proportion of non-implausible points in B_0 is not high; this way, we can ensure that we do not end up directing all of our attention to small plausible regions that form only a fraction of the truly plausible regions in parameter space.

Once we have determined and emulated at the set of points B_0, we choose n non-implausible points from B_0 to form A_1, the set of points to run the next iteration of simulations on. There are several different strategies for choosing this next set of n points: we can draw randomly from the non-implausible points in B_0, we can weight the probability of drawing a given $\mathbf{b} \in B_0$ by its predicted implausibility (so that points that are likely to implausible can be quickly ruled out), or we can adopt a number of other strategies. We then run the simulator at all points in A_1, giving the emulator more information, and iterate. By our choices for n, k, and the variance of the Gaussian, the number of simulator runs should be far fewer than the number of the emulator runs. This allows us to quickly narrow down the non-implausible region while minimizing simulator runs in favor of more computationally inexpensive emulator runs.

The *implausibility* of a point $\mathbf{x}$ in parameter space is a metric for the likelihood that $f(\mathbf{x})$ can take on values near the target output. If $g\colon \mathbf{x} \to (\mathbf{E}, \mathbf{V})$ represents the emulator (where $\mathbf{E}$ is the emulator's prediction for the expected value of $f(\mathbf{x})$ and $\mathbf{V}$ is the predicted variance), then one measure of the implausibility of point $\mathbf{x}$ is simply

$$\frac{|\mathbf{y} - \mathbf{E}[f(\mathbf{x})]|}{(\mathbf{V}_\mathbf{x})^{1/2}}$$

This roughly corresponds to the number of standard deviations away that the expected value of $f(\mathbf{x})$ is from the target value. This implausibility measure can also be adjusted to account for the simulator's variance, and for the observation uncertainty in the empirical data, by adding terms to the denominator for each respective uncertainty. We call a point implausible if its implausibility is above a certain upper bound for plausibility.

We can also select one of several termination conditions for the algorithm. We can choose to simply terminate the algorithm after a fixed number of waves, when the emulator variance is lower than the inherent uncertainties (simulator variance, observation uncertainty, etc.), or when emulated/simulated values are sufficiently close to the target empirical data. These termination conditions are detailed in the history matching paper.

5 Toy Problem Specification

To confirm that our own framework for modularization of the history matching algorithm is correct and feasible to implement, we test a simple implementation on the following toy problem.

Given a one-dimensional simulator (one parameter and one output), we wish to find a range of appropriate inputs to match with a particular set of outputs. For the sake of ease of

visualization, suppose our target outputs are "real values above a certain threshold," rather than a specific target value.

In our particular example, our toy simulator f is given by the following expression:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} + \epsilon$$

where ϵ is a random variable representing noise, distributed normally with mean zero and variance 0.001. Note that $f(x)$ can be any function–the fact that it is a normal distribution density function is not significant here. ϵ also does not need any particular distribution; the only restriction is that it must have expectation of zero.

We set the threshold at 0.2, and we implicitly set a threshold for implausibility at approximately 2 standard deviations. Points are deemed implausible if less than 5% of the probability mass of the normal distribution with mean and variance given by the emulator is above the 0.2 threshold.

Our implementation of the History Matching algorithm quickly finds the plausible set of points for this toy problem–points between approximately -1.2 and 1.2.

6 File Overview

The main file is `main.py`. At the beginning of each wave, we must first chooes a set of input points for simulation; the sampling algorithms (**lhcSample** [called only before the first wave] and **sampleAround** [called at the end of every wave to determine candidate points for simulation in the next wave]) are available in `sampling.py`. During each wave, the program calls the toy simulator function **sim** in `simulator.py` to run simulations with the given inputs. The file then calls **sampleAround** from `sampling.py` to obtain a set of candidate points for future iterations, rejecting or accepting these points based on the results of the **emulate** and **reject** functions in `emulator.py`.

`distribution.py` and `objfunc.py` provide structures to represent the simulator function expectation, and to represent probability distributions.

6.1 Known Issues

Currently, the **sample** function in `distribution.py` relies on the **invcdf** function, which relies on `scipy.integrate`. However, this sometimes does not return expected results because `scipy.integrate` gives mathematically incorrect results when integrating functions over input ranges where the function is almost always 0. Additionally, **sample** only currently works for one-dimensional (vectorized!) inputs.

7 File Descriptions

7.1 `distribution.py`

`distribution.py` contains the implementation of the `Distribution` class (with simple pdf, cdf, and sample functions available for one dimensional vectorized distributions), as well as

several commonly-used distributions (one dimensional, n-dimensional Gaussian multivariate distribution pdfs).

function `normpdf(mean, var)`

Returns a one-dimensional normal pdf.

Parameters

- **mean**: The mean of the distribution, in a 1-element list.
- **var**: The variance of the distribution, in a 1-element list.

Returns

- **normpdfFunc**: A normal pdf function with the specified mean and variance.

function `normpdfNd(mean, var)`

Returns an n-dimensional Gaussian pdf, where all variables are pairwise independent. Can easily be modified to allow for nonzero covariances.

Parameters

- **mean**: The mean of the distribution, in an n-element list.
- **var**: The variance of the distribution, in an n-element list.

Returns

- **normpdfFunc**: A multivariate Gaussian pdf function with the specified mean and variance.

7.2 `emulator.py`

This subsection contains the functions necessary to run emulations. Currently emulations are performed with pyGPs. However, there are a number of limitations to pyGPs (can't handle multi-dimensional inputs, gives the user almost no control over plotting). It is advised that any future GPR emulators are written using the GPy package.

function `emulate(training, test, wantToPlot = True, k = pyGPs.cov.RBF())`

Returns a prediction for test values given training values, using GPR from the pyGPs package. Also plots the simulator and emulator results.

Parameters

- **training**: The training data, a list of (x, y) pairs (x and y should be vectors represented as lists).
- **test**: The values to run the emulator at. A list of vectors, where vectors are represented as lists.
- **wantToPlot**: Either `True` or `False`.
- **k**: The kernel for the pyGPs model.

Returns

- **prediction**: The pyGPs prediction. `prediction[0][i][0]` is the expectation for the value of the simulator at `test[i]`, and `prediction[1][i][0]` is the variance.

function `reject(predictedMean, predictedVariance, threshold, rejectionDensity)`

Returns true or false, depending on whether or not the output (captured in `predictedMean` and `predictedVariance`) is implausible (has less than 5% probability mass over the threshold).

Parameters

- **predictedMean**: The expected value of the output predicted by the emulator.
- **predictedVariance**: The variance predicted by the emulator run.
- **threshold**: The target threshold for simulation runs.
- **rejectionDensity**: The minimum probability mass needed to be above the threshold to not be rejected.

Returns

- `True` if the mean and variance are implausible and should be rejected, `False` otherwise.

7.3 `main.py`

The main file. The user can set the minimum and maximum input parameter values, the threshold (only relevant to the modified toy problem), the minimum probability density needed for points not to be rejected, the expectation of the simulator, k (as described in the Overview section), the number of waves, the simulations per wave, and the simulations performed at each point. Running this file runs the history matching algorithm. There are two helper functions used to make the code more readable.

function `runSimulations(inputs, waveNumber)`

Given a set of x-values, returns a list of the x values with associated simulator outputs.

Parameters

- **inputs**: A list of 1-dimensional vectors we wish to simulate at.
- **waveNumber**: The current wave.

Returns

- **newSimPoints**: A list of (points in parameter space, associated simulator outputs) after running simulations at each point given. This is a list of length 2 lists of vectors.

function `getEmulationPoints(inputs, prediction)`

Given a set of x-values and the emulator predictions for simulator outputs at each value, returns a list of non-implausible points to sample future simulator input points from.

Parameters

- **inputs**: A list of 1-dimensional vectors, which form the basis for the next round of samples.
- **prediction**: The pyGPs emulator predictions at all of the points in **inputs**. `prediction[0][i][0]` is the expectation for the value of the simulator at `inputs[i]`, and `prediction[1][i][0]` is the variance.

Returns

- **test**: A list of vectors which were emulated at and determined to be non-implausible, suitable for future simulations.

7.4 `sampling.py`

This file contains functions to carry out the Latin Hypercube Sampling procedure (the first sampling method), and also to carry out subsequent sampling procedures in later waves.

function `lhcSampleSimp(dim, div = 10)`

Samples n points on a lattice of div^{dim} points with the LHS method. Only used as a helper function for `lhcSample`.

Parameters

- **dim**: The dimension of the hypercube we wish to sample from.

- **div**: The number of segments each edge of the hypercube is segmented into.

Returns

- **arr**: A list of regions in the hypercube. For example, if $div = 3$ and $dim = 2$, the region $(0, 2)$ would refer to the region in the 0th row and 2nd column.

function `lhcSample(dim, minx = 0, maxx = 1, div = 10)`

Uses Latin Hypercube Sampling method to sample *div* points from a hypercube of dimension *dim*, with specified min and max corners.

Parameters

- **dim**: The dimension of the hypercube we wish to sample from.
- **minx**: A list of the minimum values of each coordinate for the space sampled from.
- **maxx**: A list of the maximum values of each coordinate for the space sampled from.
- **div**: The number of segments each edge of the hypercube is segmented into.

Returns

- **points**: A list of points (vectors represented by lists) sampled from the given hypercube with the LHS method.

function `sampleAround(k, x, dist = distribution.Distribution(normpdf([0.], [0.6])))`

Samples k points from a distribution with variance specified in *dist*, centered at x.

Parameters

- **k**: The number of samples to take.
- **x**: The center of the distribution to sample from.
- **dist**: The distribution to sample from (first recentered at x).

Returns

- **newPoints**: A list of points (vectors represented by lists) sampled from the distribution centered at x.

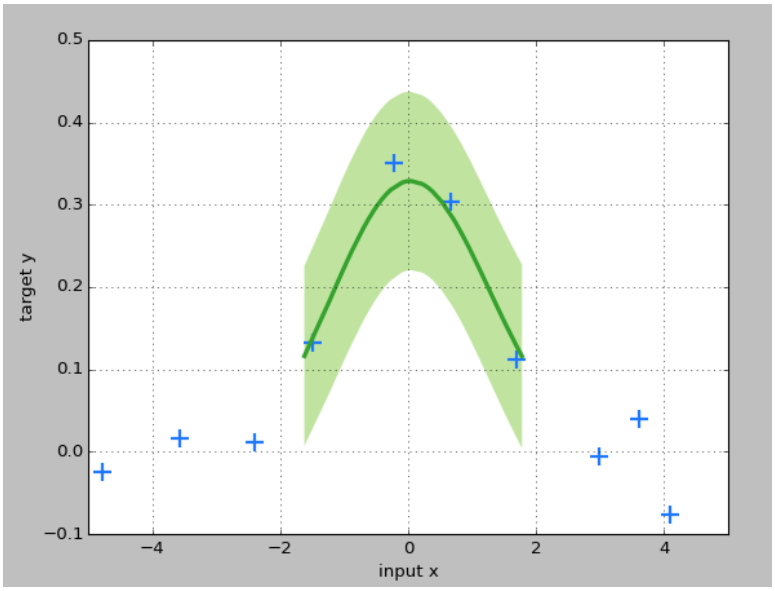

Figure 1: The green segments represent the current non-implausible space. Note that this it is at first much wider than the "true" non-implausible space, which spans from approximately -1.2 to 1.2.

7.5 `simulator.py`

This file contains the functions needed to run simulations. Contains a parameter `epsvar` which controls the noise in simulations. The noise function can be altered from within the `sim` function.

function `sim(x, func)`

Runs a simulation with input parameters x, returning the result of the simulation.

Parameters

- **x**: The number of samples to take.
- **func**: The expectation of the simulation function.

Returns

- **newPoints**: A vector representing the result of the simulation.

8 Toy Problem Results

The solution was successful in quickly identifying the non-implausible region. Four simulator runs are shown in the four figures.

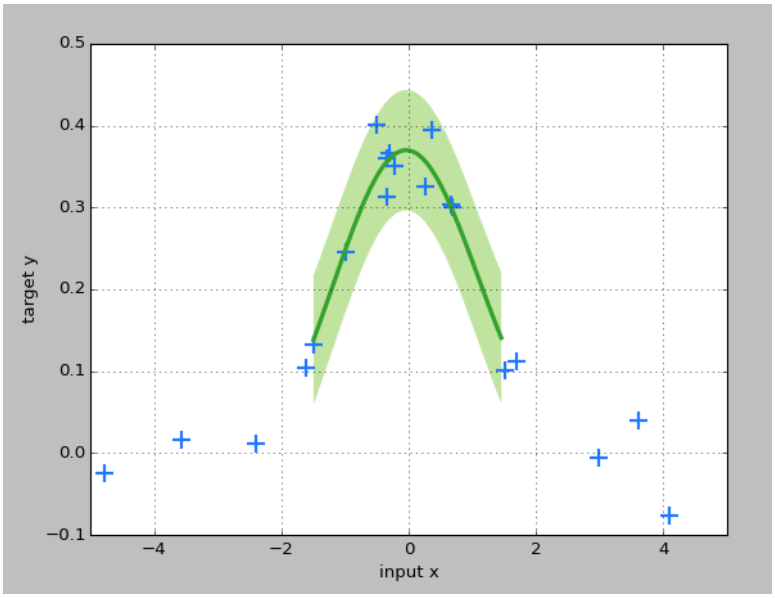

Figure 2: In the next wave, more simulations are performed at locations that were tentatively marked as non-implausible in wave 1. With the extra information, the emulator is able to reject more points and tighten the non-implausible region.

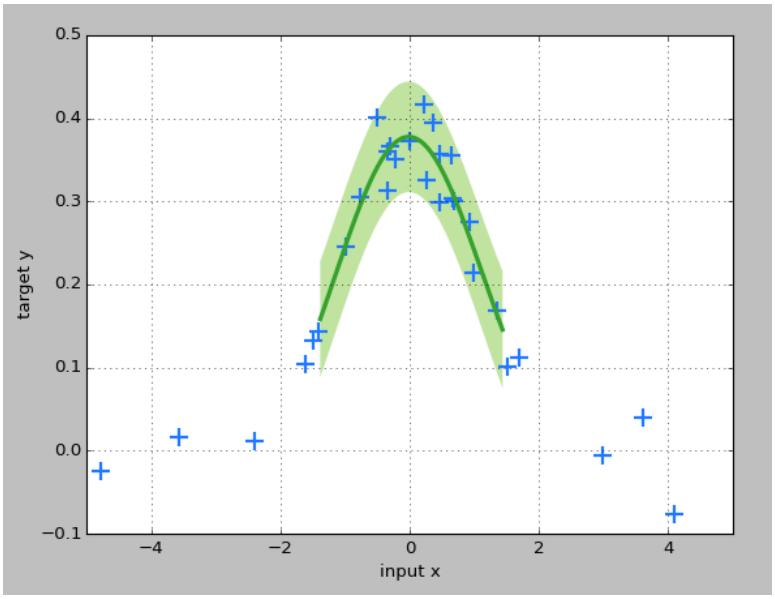

Figure 3: By wave 3, the non-implausible and implausible regions have become fairly clearly defined.

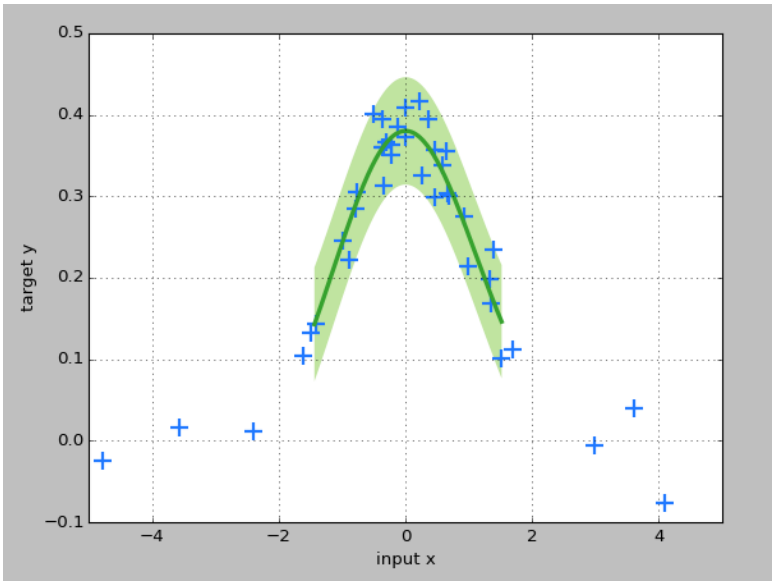

Figure 4: At this point, there are sufficiently many simulator runs around and in the non-implausible region that the non-implausible region is no longer changing. x values between -1.2 and 1.2 are consistently marked as non-implausible. The light green region, which represents the area within 2 standard deviations from the mean, demonstrates that points need at least 5% of their probability mass above the threshold of 0.2 in order not to be rejected.

9 Future Directions

This initial implementation suggests that Bayesian History Matching is a viable and promising method for calibrating epidemiology models. The modularization demonstrated allows for robust future implementations with more complex simulators, emulators, and sampling methods. However, several issues still remain:

1. The emulator clearly needs to be able to handle multi-dimensional output. Additionally, for more effective visualization future work should be done with a package that allows for more control over plotting. We recommend moving away from pyGPs and using the GPy package instead, which allows for greater freedom.

2. It is not clear that the algorithm will still be computationally feasible in its current formulation once parameter space reaches 10 to 20 or more dimensions, and once thousands of simulator runs (and potentially millions of emulator runs) are needed.

3. A consistent termination condition still needs to be developed.

4. Sampling could potentially be made more effective: it is likely that biasing the sampling algorithms to simulate more frequently at points where implausibility/non-implausibility is least certain would be more informative than uniformly sampling over a set of non-implausible points.