A detailed example of using the GPM/SA code

Kary Myers, Dave Higdon, Jim Gattiker, Bill Rust

December 2008; Revised January 2012

1 Introduction

GPM/SA code is a MATLAB program that can be used with real or vectored output. It can be used to perform multiple analyses including those listed below.

- Predict the output from a computer code based on an emulator constucted from a fully Bayesian Gaussian spatial process (GaSP) model
- Compute main effect, joint effect, and total effect sensitivity indices for the output from a computer code based on an emulator constucted from a fully Bayesian GaSP model
- Estimate the posterior distribution of the calibration parameters of a computer model based on computer code output and field data output using a Bayesian implementation of the Kennedy and O'Hagan (2001) model as described in Higdon et al. (2008).
- Predict the output of a "true" input/output relationship based on computer code output and field data output.

This document walks through a specific application, showing how to set up, run, and visualize the results using the GPM/SA code. Specifically, this tutorial shows

- How to set up a problem using the GPM/SA code,
- how to calibrate model parameters using physical observations,
- make predictions for the physical system at new input settings, and
- assess the accuracy of the Gaussian process emulator used in the modeling.

2 An Illustrative Problem

Suppose we drop balls of a given density from a tower at initial heights 5, 10, 15, and 20 meters. We wish to predict the time it takes them to fall to the ground. We can develop a predictor by conducting a *field experiment* in which we drop balls of different radii R (and the desired density) from different heights. Each dropped ball produces vector output in the form of a height-time curve, i.e., a curve showing the time as function of the current height $t = t(h)$ of the ball at a set of measured heights h.

Suppose the following equation (in terms of the ball's acceleration) describes the true height-time curve[1]:

$$\frac{d^2h}{dt^2} = g - \frac{C}{R}\left(\frac{dh}{dt}\right)^2. \tag{1}$$

where g is the acceleration due to gravity, C is the coefficient of drag, and, as introduced above, R is the radius of the ball.

To make this toy problem realistic, suppose we know the acceleration due to gravity g but don't know the coefficient of drag C. In addition, our only insight into the physical process comes from the field data we collect at our known initial tower heights h_0, and that we don't know the true physical process described by (1).

In addition to the field data, we have computer code (called a *simulator*) that takes as input a radius R and a coefficient of drag C and produces a height-time curve giving the computed times t at some set of tower heights h (they don't need to be the same heights used in the field experiments). Our simulator does this in a deterministic way, so that the same inputs (R, C) will always produce the same height-time curve. In this illustration, let's suppose that the simulator produces output according to this equation for acceleration:

$$\frac{d^2h}{dt^2} = g - \frac{C}{R}\frac{dh}{dt}. \tag{2}$$

Note that while (1) has a squared velocity term (dh/dt), (2) includes velocity as a linear term only. Thus our computer model is systematically "off" from reality, and indeed the units of (2) are no longer acceleration units. Complicating matters, imagine that our simulator is very expensive to run, so we can't exhaustively try all possible input pairs (R, C). This is a typical situation in fields like climate modeling.

Since we can't run the simulator at all possible locations in (R, C, h)-space, we use a simplistic Latin hypercube (LH) design to choose locations at which to run the code. More serious applications would be better served by enforcing some form of space filling in the LH design. We would like to model the resulting simulator output so that we can (inexpensively) predict what the

[1] Equations (1) and (2) are differential equations that can be solved for height h as a function of time t and ball radius R. We require the inverse of this: our field data have time as the output recorded as a function of height and radius. We use an optimizer to perform this inversion in MATLAB.

output would be at untried (R, C, h) inputs. In addition, we want to use our field data to adjust the simulator output so that it matches reality as best as possible. That is, we want to adjust the unknown parameters (C in this example) until our simulator output fits the field data, even with the inadequate model in (2).

As described in the sections below, the Gaussian process model allows us to do both these things. We first use a stationary Gaussian process to model the simulator's output surface, at the same time providing a measure of the uncertainty in our model at different locations. We call the Gaussian process model in this context an *emulator*, as it is "emulating" the output of the simulator. We then use another stationary Gaussian process to model the *discrepancy* between the simulator's output and the field data in order to calibrate the simulator.

Here's a summary of the setup of our tower example, where n denotes the size of our experimental data set, m denotes the number of runs of the simulator, and n_η denotes the number of heights h_s used by the simulator:

- We have data from $n = 3$ field experiments, one each for balls of radius $R \in \{0.1, 0.2, 0.4\}$ meters. Each experiment produces a curve of drop times made of three or four height-time pairs. For the two smallest balls, the experimental heights are $h_e \in \{5, 10, 15, 20\}$ meters. We'll imagine that the largest ball is too heavy to carry to the tallest tower, so its drop time is only measured from experimental heights $h_e \in \{5, 10, 15\}$ meters.
- We used a scaled Latin hypercube design to select the $m = 25$ (R, C) pairs, shown in Figure 1, at which to run our simulator. For each (R, C) pair in the design, the simulator produces a curve of $n_\eta = 16$ height-time pairs, where the simulation heights h_s are evenly spaced in $[1.5, 24]$ meters.
- Our simulator uses a model (2) that is inadequate to describe the true physical process (1).

3 How we use the Gaussian process model

The Gaussian process model considers two kinds of inputs:

1. $\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$ denotes inputs that are under the control of (or are observable by) the experimenter in both the field experiments and the simulator runs. In our example we have $p = 1$ input of this type: $\boldsymbol{x} = x = R$, the radius of the ball being dropped.
2. $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_q)$ denotes inputs to the simulator that we need to estimate using the experimental data. These $\boldsymbol{\theta}$ could correspond to real physical quantities or could be parameters of the simulator code. In our example we have $q = 1$ input of this type: $\boldsymbol{\theta} = \theta = C$, the coefficient of drag.

m	R $(p=1)$	C $(q=1)$
1	0.0996	0.2105
2	0.3995	0.1795
3	0.2956	0.1167
4	0.4033	0.1457
5	0.4478	0.0610
6	0.0971	0.2376
7	0.1222	0.0982
8	0.3155	0.1072
9	0.1676	0.0742
10	0.1480	0.0979
11	0.2331	0.1639
12	0.2251	0.2275
13	0.0795	0.1977
14	0.3272	0.2237
15	0.2460	0.1914
16	0.2881	0.2466
17	0.3502	0.0702
18	0.3613	0.1345
19	0.3833	0.1564
20	0.2095	0.1762
21	0.1308	0.1498
22	0.1898	0.1236
23	0.4264	0.2087
24	0.0579	0.0544
25	0.2676	0.0885

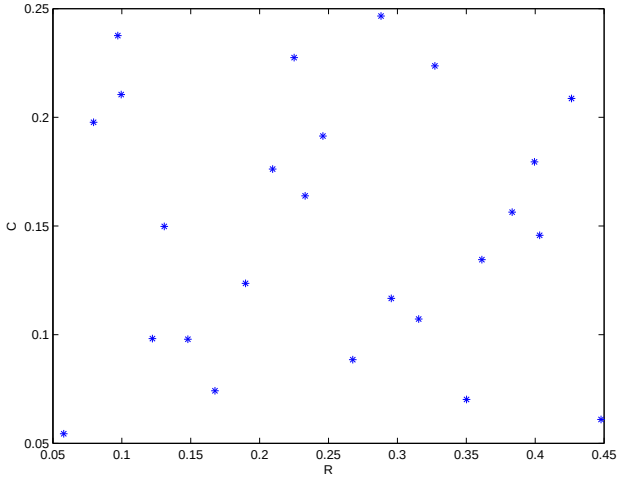

Figure 1: *Left:* Scaled Latin hypercube design with $m = 25$ rows of (R, C) pairs. *Right:* A plot of the design.

Note that h is not considered an input to the simulator since the simulator produces a 16-vector of drop times for an evenly spaced set of heights between $h = 1$ and $h = 24$ meters.

The model also makes use of two kinds of outputs:

1. $y_{\text{obs}}(\boldsymbol{x})$, the output of the field experiments. For each experiment, y_{obs} can be a scalar, or, as in our tower example, it can be a curve. For the tower experiments, $y_{\text{obs}} = \boldsymbol{t}_{\text{obs}}$, the vector of times, one time t for each tower height h.

   Note that not all experiments need produce output of the same size. For instance in our tower experiment we have 4 recorded times for the two smaller balls but only 3 for the largest ball.

2. $y_{\text{sim}}(\boldsymbol{x}, \boldsymbol{\theta})$, the output of the simulation runs. As with the observed data, y_{sim} can be a scalar or a curve. For our tower example, we'll assume that the simulator uses an evenly spaced grid of $n_\eta = 16$ heights h_s and computes a time for each height on the grid, so $y_{\text{sim}} = \boldsymbol{t}_{\text{sim}}$.

   Unlike the observed data, the simulator output will always have the same size, i.e., the same number of computed times. The grid of heights will be the same from run to run.

Here's the idea: We use the simulator output y_{sim} to construct a Gaussian process model that *emulates* the simulator's behavior at arbitrary (R, C) pairs. We use the field data y_{obs} to *calibrate* the coefficient of drag C so that the simulator output best matches the field data. And we use another Gaussian process model to capture the *discrepancy* δ between the simulator output and the field data.

4 Some details of Gaussian processes

Gaussian processes (GP's) are random functions that have proven useful for modeling output from computer code, as well as other spatial phenomena. Figure 2 shows how a a Gaussian process model $\eta(x)$ defined on a one-dimensional space $\mathcal{X} = [0, 1]$, can be used to estimate a smooth, deterministic function.

Any Gaussian process is determined by its mean function $\mu(x)$ (defined on $\mathcal{X}$) and its covariance function $\text{Cov}(x, x')$ (defined on $\mathcal{X} \times \mathcal{X}$.). In the case of the GP used in Figure 2, the mean function is 0, and the covariance function is given by

$$\text{Cov}(x, x') = \exp\{-[(x - x')/.3]^2\}.$$

Note that while the mean function can be rather general, the covariance function must be positive definite.

Given the five function evaluations shown in Figure 2, the resulting Gaussian process fit is assured to be smooth, and to interpolate the function evaluations given by the black dots. The fitted GP that interpolates the data is now described by a posterior distribution with updated mean and covariance functions which depend on the given function evaluations. Draws from this *posterior* GP are given by the cyan lines. The mean function is given by the dashed red line. This basic modeling approach can be extended to higher dimensional support $\mathcal{X}$ by generalizing the mean and covariance functions. In practice, parameters controlling the covariance function are also estimated from the computer model output. See Higdon et al. (2008) for more details about the model.

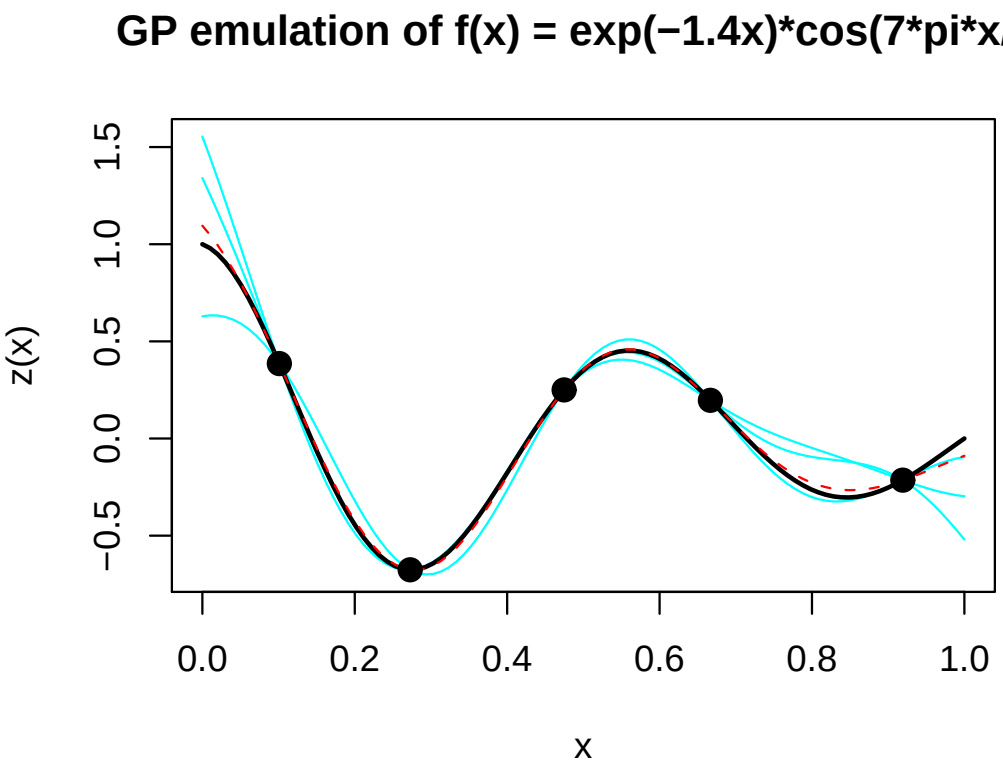

Figure 2: A Gaussian process emulator for the function $f(x) = \exp\{-1.4x\}\cos(3.5\pi x)$ over $x \in [0, 1]$ using the five observations of the function given by the black dots. The black line gives the true function $f(x)$, the dashed red line gives the posterior mean GP estimate, and the cyan lines give plausible realizations of the GP emulator given the five observed function evaluations. The GP emulator exploits the smoothness in $f(x)$ and the fact that the five evaluations are noiseless.

5 Preparing the data for use by the GPM/SA code

We use the GPM/SA code to calculate the model and to make predictions from the model. Before we can use the code, we need to read in our data; transform the inputs and outputs; compute basis functions for transforming the standardized outputs and the discrepancy term; and package the data into MATLAB structures that we can pass to the GPM/SA code.

We'll walk through this for our tower example. Remember that

- $x = R$, the radii of the balls;
- $\theta = C$, the coefficient of drag; and
- $y = \boldsymbol{t}$, the vector of times (for both the field experiments and the simulator runs).

The heights $\boldsymbol{h}$ serve as indices to the time vectors.

All the code in this section can be found in the MATLAB file `readdata.m`.

5.1 Reading the data

For our example we have three data files for the field experiments and three for the simulator runs: One with the inputs (R_{obs} for the field experiments, R_{sim} and C_{sim} for the simulator runs), one with the outputs ($\boldsymbol{t}_{\text{obs}}$ and $\boldsymbol{t}_{\text{sim}}$), and one with the heights ($\boldsymbol{h}_{\text{obs}}$ and $\boldsymbol{h}_{\text{sim}}$). The set of simulator inputs (the particular values of R_{sim} and C_{sim}) is called the *design* to make the analogy with experimental design.

```
% read in the field (observed) data
>> Robs = textread('field.radii'); % radii R
>> hobs = textread('field.height'); % heights h
>> tobs = textread([dirstr 'field.dat'])'; % times t
>> tstd = textread([dirstr 'field.sd'])'; % sd of measured times

Robs =

    0.1000    0.2000    0.4000

% hobs has all the tower heights;
% we only use the lowest 3 for the largest ball
hobs =

     5    10    15    20

% tobs has one column per experiment, one row per tower height.
% The NaN in experiment 3 indicates we didn't drop the largest
% ball from the highest tower.
```

```
tobs =

    1.2180    1.1129    1.0611
    2.0126    1.7225    1.5740
    2.7942    2.2898    2.0186
    3.5747    2.8462       NaN

tstd =

    0.1000    0.1000    0.1000
    0.1000    0.1000    0.1000
    0.1000    0.1000    0.1000
    0.1000    0.1000       NaN
```

```
% read in the design and the simulator output
>> [Rsim Csim] = textread('sim.design'); % design (R and C)
>> tsim = textread('sim.dat'); % times t
>> hsim = textread('sim.height'); % heights h

>> n = size(tobs, 2); % number of experiments
>> m = size(tsim, 2); % number of simulation runs
```

5.2 Transforming x and θ

The GPM/SA code requires that the inputs x and θ lie in the interval $[0, 1]^{p+q}$. Here we accomplish this by shifting and scaling the original values of x and θ, but in other settings another approach could be appropriate.

We first transform the inputs to the simulator (R_{sim} and C_{sim}) so they lie in $[0, 1]$, then we use the minimum and range of R_{sim} to transform the input to the field experiments (R_{obs}) onto the same scale.

```
% transform the simulator inputs so each dimension lies in [0, 1]
>> Rsmin = min(Rsim);
>> Rsrange = range(Rsim);
>> Rsim01 = (Rsim - Rsmin) / Rsrange; % transformed R

>> Csmin = min(Csim);
>> Csrange = range(Csim);
>> Csim01 = (Csim - Csmin) / Csrange; % transformed C

% transform the field experiment input the same way
>> Robs01 = (Robs - Rsmin) / Rsrange; % transformed R
```

5.3 Transforming y_{sim} and y_{obs}

The GPM/SA code requires that the outputs y have mean zero and variance one. As above, we first transform the output from the simulator ($\boldsymbol{t}_{\text{sim}}$) and then use those values to transform the output from the field experiment ($\boldsymbol{t}_{\text{obs}}$). Here we want the simulator output to have mean zero at each height h and an overall variance of one.

```
% standardize the simulator output
>> tsimmean = repmat(mean(tsim, 2), [1 m]); % the mean simulator run
>> tsimStd = tsim - tsimmean; % make mean at each height zero
>> tsimsd = std(tsimStd(:)); % standard deviation of ALL elements of tsimStd
>> tsimStd = tsimStd / tsimsd; % make overall variance one
```

Now we transform the field data. We want to use the value of the mean simulator run (`tsimmean` above) at each experimental height to do this, but the height grid of the simulator doesn't match the experimental heights; i.e., we don't know the value of the mean simulator run at all the experimental heights. Instead we'll interpolate `tsimmean` in order to find its (interpolated) value at each experimental height. We'll use this interpolated mean and the overall standard deviation of all elements of the simulator runs, `tsimsd`, to transform the field data.

Since each experiment could have a different size (different number of heights at which the ball was dropped), we'll loop over the n experiments and record the results in a structure array called `yobs`.

```
>> for ii = 1:n
    % number of heights with measurements for experiment ii
    numhts = sum(~isnan(tobs(:, ii)));

    % do the interpolation and get the interpolated values at the experimental heights
    yobs(ii).tobsmean = ...
             interp1(hsim, tsimmean(:,1), hobs(1:numhts), 'linear', 'extrap');

    % do the standardization
    yobs(ii).tobsStd = (tobs(1:numhts, ii) - yobs(ii).tobsmean') / tsimsd;

    % for convenience, record some extra information in yobs
    yobs(ii).hobs = hobs(1:numhts); % the heights where measurements were taken
    yobs(ii).tobs = tobs(1:numhts, ii); % the untransformed output

    % now record the observation covariance matrix for the measured times
    yobs(ii).Sigy = diag(tstd(1:numhts,ii).^2);
    % now the observation covariance for the standardized observations
    yobs(ii).SigyStd = yobs(ii).Sigy./(tsimsd.^2);
end
```

Note that `yobs(ii).Sigy` holds the covariance matrix for the observations of experiment `ii`; one can change the prior specification for the measurement precision to ensure that it stays close to this specified prior value.

5.4 Computing the K basis for transforming y_{sim} and y_{obs}

We want to capture the variation in the height-time curves across simulation runs; that is, we want an efficient representation of how the simulator output varies at different locations in $(x, \theta) = (R, C)$-space. We will do this by computing the singular value decomposition of the simulator output to get a set of basis functions (called the K *basis*). While we typically use SVD, any linear transformation will work. For a compact representation, we use $p_u < m$ basis functions that capture most of the variation in the simulation runs. (Note that p_u, the number of basis elements, shouldn't be confused with p, the dimension of the input x.)

```
>> pu = 2; % number of basis components to keep
>> [U, S, V] = svd(tsimStd, 0); % compute the SVD
>> Ksim = U(:, 1:pu) * S(1:pu, 1:pu) ./ sqrt(m); % get the pu basis components
```

This K_{sim} matrix of basis elements has $n_\eta = 16$ rows (one for each height in the grid used by the simulator) and $p_u = 2$ columns. We now compute a corresponding basis matrix K_{obs} for each experiment in the field data. These will have three or four rows (one for each experimental height used) and again p_u columns.

To get these matrices we interpolate between height grids (like we did to transform y_{obs} above) and again store the results in the structure array `yobs`.

```
>> for ii = 1:n
    yobs(ii).Kobs = zeros(length(yobs(ii).tobsStd), pu); % allocate space

    % compute each basis component
    for jj = 1:pu
        % do the interpolation and get the interpolated values at the experimental heights
        yobs(ii).Kobs(:, jj) = ...
                 interp1(hsim, Ksim(:, jj), yobs(ii).hobs, 'linear', 'extrap');
    end

end
```

5.5 Specifying the D basis for modeling the discrepancy term

The discrepancy term $\delta(x)$ models a systematic bias between the simulator (at the best setting for the calibration parameter θ). We expect that the estimate for $\delta(x)$ will be similar for nearby values of x. To this end, $\delta(x)$ is modeled as a GP with a correlation structure across the x space.

In the ball dropping example, for a given ball radius x, $\delta(x)$ is a function over the possible drop heights $0 \le h \le 24$. Over $h \in [0, 24]$, $\delta(x)$ is represented as a linear combination of basis functions

$$\delta(x) = \sum_{i=1}^{p_v} d_i(x) v_i$$

where $d_i(\cdot)$ is a normal density centered at ω_i, with a standard deviation of 2, and the v_i's are modeled as iid $N(0, \lambda_v)$. This model is depicted in Figure 3. Generally, the user specifies the basis representation for the discrepancy term in the GPM/SA codesetup file, which is named `readdata.m` for this example. Here is how this discrepancy basis is specified.

Specifying the discrepancy basis requires that the user determine the form and location of the basis elements $d_i(\cdot)$. For this example, we take the $d_i(\cdot)$'s to be normal kernels with an sd of 2. The kernels are centered at a grid of 13 heights equally spaced between 0 and 24. For each experiment, we need to construct the matrix `Ddat` whose rows correspond to the number of observations in the experiment, and whose columns correspond to the $p_v = 13$ basis elements. For plotting purposes, we also construct the matrix `Dsim` which has rows corresponding to a dense grid over the h-space:

```
% -- D basis --
% JG: lay it out, and record decomposition on sim and data grids
% JG: Kernel centers and widths
>> Dgrid = 0:2:max(hsim); % locations on which the kernels are centered
>> Dwidth = 2; % width of each kernel


pv = length(Dgrid); % number of kernels

% Compute the kernel function map, for each kernel
% Designate space for the Dsim matrix,
% one row per simulated height, one column per kernel
% (consider making the grid of heights much denser for plotting)
>> Dsim = zeros(length(hsim), pv);

% designate space for the Dobs matrix for each experiment,
% one row per experimental height, one column per kernel
>> for ii = 1:n
    yobs(ii).Dobs = zeros(length(yobs(ii).tobsStd), pv);
end

% create each kernel
>> for jj = 1:pv
    % first create kernel jj for each experiment
    for ii = 1:n
        % normpdf computes the value of a Gaussian with mean
        % Dgrid(jj) and variance Dwidth at each element of hobs
        yobs(ii).Dobs(:, jj) = normpdf(yobs(ii).hobs, Dgrid(jj), Dwidth);
    end
    % now create kernel jj for the simulations
    Dsim(:, jj) = normpdf(hsim, Dgrid(jj), Dwidth);
end

% normalize the basis elements of D so that the marginal variance of delta is about 1
>> Dmax = max(max(Dsim * Dsim'));
>> Dsim = Dsim / sqrt(Dmax);
>> for ii = 1:n
    yobs(ii).Dobs = yobs(ii).Dobs / sqrt(Dmax);
end
```

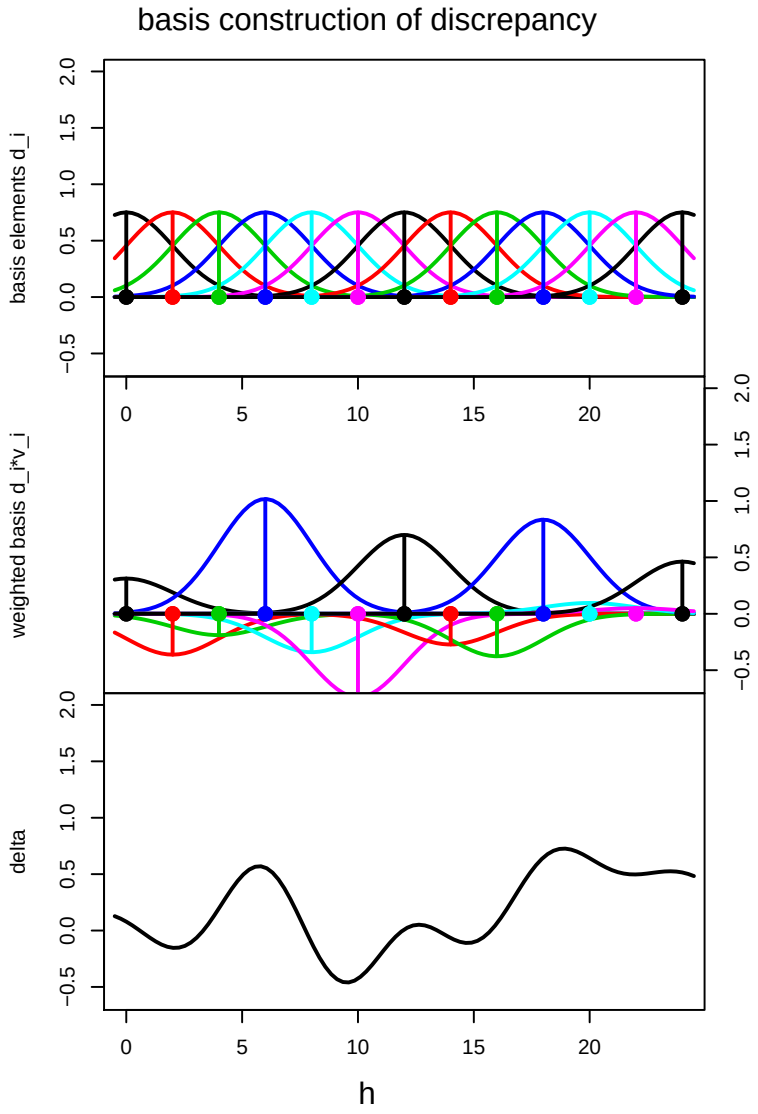

Figure 3: Basis construction of $\delta(x)$ for the ball dropping example. Here a model for $\delta(x)$ – the discrepancy between the calibrated simulator and experimental observations at x – is modeled by a linear combinations of normal kernels. Top: 13 normal kernels with sd=2 are placed at heights $h = 0, 2, \ldots, 24$. Each of the 13 columns in D corresponds to one of these basis kernels. Middle: each basis kernel is multiplied by a random normal variate v_i which is estimated in the GPM/SA code using the simulation output and experimental data. Bottom: the discrepancy is set to the sum of these weighted kernels. In vector form, this is given by $Dv(x)$, where $v(x)$ is the 13 vector of weights corresponding to input condition x.

For modeling purposes, only the `Dobs` matrix in required for each `yobs(ii)`. The `Dsim` matrix is useful for plotting the estimated discrepancy over a denser set of heights. The `D` matricies are normalized so that the prior marginal variance for $\delta(x)$ is approximately one when $\lambda_v = 1$.

For normal basis kernels, the spacing needs to be no more that one standard deviation between adjacent kernels to ensure that no sparsity effects appear. The width of the kernels controls the spatial dependence in $\delta(x)$ – wider kernels will give δ fewer "wiggles" over the support h. The properties desired for the discrepancy term will necessarily depend on the application being considered. The choice of an sd of 2m is ok for this ball dropping application. See other applications for different examples of discrepancy basis construction. For more details regarding the use of kernels to create GP models see Higdon (2002).

5.6 Package all the pieces

Next we make a structure that contains all the information we've computed here. This structure, here called `data`, will contain a field for the simulated data (`simData`) and another for the field data (`obsData`). For both fields, we'll include information that's required by the model as well as extra information (stored in a subfield called `orig`) that will later make it easier for us to return the output to the original scale and to do plots.

Since the simulated data have the same size for each run (unlike the observed data), packaging this information is straightforward.

```
% required fields
>> simData.x = [Rsim01 Csim01]; % our design (standardized)
>> simData.yStd = tsimStd; % output, standardized
>> simData.Ksim = Ksim;

% extra fields: original data and transform stuff
>> simData.orig.y = tsim;
>> simData.orig.ymean = tsimmean;
>> simData.orig.ysd = tsimsd;
>> simData.orig.Dsim = Dsim;
>> simData.orig.t = hsim;
>> simData.orig.xorig = [Rsim Csim]; % original scale for simulated R, C
```

For the observed data we need to package each experiment separately since each could have a different length.

```
% loop over experiments
>> for ii = 1:n
    % required fields
    obsData(ii).x = Robs01(ii);
    obsData(ii).yStd = yobs(ii).tobsStd;
    obsData(ii).Kobs = yobs(ii).Kobs;
    obsData(ii).Dobs = yobs(ii).Dobs;
    obsData(ii).Sigy = yobs(ii).Sigy./(tsimsd.^2);

    % extra fields: original data
    obsData(ii).orig.y = yobs(ii).tobs;
    obsData(ii).orig.ymean = yobs(ii).tobsmean;
    obsData(ii).orig.t = yobs(ii).hobs;
end
```

Now we'll put `simData` and `obsData` in a structure called `data` that we can pass to the GPM/SA code.

```
>> data.simData = simData;
>> data.obsData = obsData;
```

6 Model initialization and MCMC

Now that we have code to package the data appropriately, we can initialize the model, use the data to compute the posterior distribution of the parameters, and then sample from this distribution via Markov chain Monte Carlo (MCMC). MCMC is a general recipe for producing a partial realization of a Markov chain whose stationary distribution is the posterior distribution. This Markov chain realization is then treated as a (dependent) sample from the posterior distribution from which one can estimate posterior moments or probabilities. For details regarding MCMC, see (Robert and Casella, 1999).

The code in this section is in the MATLAB file `runmcmc.m`.

1. First we'll call `readdata.m` from Section 5 in order to get the `data` structure created there; we'll store it in a variable called `towerdat`.

```
>> towerdat = readdata()

towerdat =

    simData: [1x1 struct]
    obsData: [1x3 struct]
```

2. Now we can do the initial setup of the model using the GPM/SA code function `setupModel()`. The function `setupModel()` takes the `obsData` and `simData` fields from `towerdat`, makes all the structures we need to do MCMC, and returns a structure which we'll call `pout` for "**p**arameter **out**put".

```
>> pout = setupModel(towerdat.obsData, towerdat.simData)
SetupModel: Determined data sizes as follows:
SetupModel: n=  3  (number of observed data)
SetupModel: m= 25  (number of simulated data)
SetupModel: p=  1  (number of parameters known for observations)
SetupModel: q=  1  (number of additional simulation inputs (to calibrate))
SetupModel: pu=  2 (response dimension (transformed))
SetupModel: pv= 13 (discrepancy dimension (transformed))

pout =

       data: [1x1 struct]
      model: [1x1 struct]
     priors: [1x1 struct
       mcmc: [1x1 struct]
    obsData: [1x3 struct]
    simData: [1x1 struct]
      pvals: []
```

   We'll describe the gory details of the fields of `pout` in Section 9. Meanwhile it's enough to know that they include the simulated and observed data transformed by the K matrix (`data`);

initial values for the parameters of the Gaussian process model (`model`); priors on the model parameters which we'll be estimating via MCMC (`priors`); details (like step sizes) of the MCMC routine for getting draws from the posterior distribution of the parameters (`mcmc`); and the `obsData` and `simData` structures that we gave it in the call to `setupModel()`. It also includes a placeholder for the `pvals` field which will hold the MCMC draws.

3. We now have the mathematical form of the posterior distribution of the model parameters, and we want to use MCMC to get draws from it via the GPM/SA code function `gpmmcmc()`. These draws will be added to the `pvals` field of the `pout` structure created above.

   (a) First we'd like to update the default MCMC settings in `pout.mcmc` by using the GPM/SA code function `stepsize()` to do burn-in of the MCMC chains and adjust the step size. Here are the default setting values before we manipulate the step size; these values were chosen to provide reasonable performance for a variety of problems:

```
>> pout.mcmc

ans =

     thetawidth: 0.2000
      rhoUwidth: [0.1000 0.1000 0.1000 0.1000]
      rhoVwidth: 0.1000
     lamVzwidth: 10
     lamUzwidth: [5 5]
     lamWswidth: [100 100]
    lamWOswidth: 100
     lamOswidth: 499.9995
          pvars: {1x11 cell}
          svars: {'theta'  'betaV'  'betaU'  'lamVz'  'lamUz'  'lamWs'
                  'lamWOs'  'lamOs'}
       svarSize: [1 1 4 1 2 2 1 1]
          wvars: {1x8 cell}
```

   The draws used for burn-in will be added to the pvals field as seen below. Using 13 levels will give the step size estimation process a good chance to find a near optimal step size.

```
>> nburn = 100;  % number of draws to discard as "burn in"

>> nlev = 13; % number of candidate levels used for step size estimation

>> pout=stepsize(pout,nburn,nlev)
Setting up structures for stepsize statistics collect ...
Collecting stepsize acceptance stats ...
Drawing 100 samples (nBurn) over 13 levels (nLev)
Started timed counter, vals 1 -> 1300
        963..20.29sec
Computing optimal step sizes ...
Step size assignment complete.

pout =
```

```
       data: [1x1 struct]
      model: [1x1 struct]
     priors: [1x1 struct]
       mcmc: [1x1 struct]
    obsData: [1x3 struct]
    simData: [1x1 struct]
      pvals: [1x1300 struct]

% look at the pvals field
>> pout.pvals

ans =

1x1300 struct array with fields:
    theta
    betaV
    betaU
    lamVz
    lamUz
    lamWs
    lamWOs
    lamOs
    logLik
    logPrior
    logPost
    thetaAcc
    betaVAcc
    betaUAcc
    lamVzAcc
    lamUzAcc
    lamWsAcc
    lamWOsAcc
    lamOsAcc
```

The `pvals` object holds the result of the MCMC. Here it records the 1300 draws from the posterior distribution for each parameter produced by the MCMC updates carried out so far. Subsequent calls to `gpmcmc()` will augment the draws recorded in `pout`. In addition to the parameter values at each of the 1300 MCMC steps, the corresponding values for the log likelihood, the log prior, and the log posterior are also recorded for each of the 1300 steps. Here are the updated MCMC settings:

```
>> pout.mcmc

ans =
     thetawidth: 0.2668
      rhoUwidth: [0.5341 0.4523 2.6462 1.7655]
      rhoVwidth: 0.4656
     lamVzwidth: 433.1763
     lamUzwidth: [0.8726 1.9799]
     lamWswidth: [1.6396e+03 4.0254e+03]
    lamWOswidth: 2.0908e+04
     lamOswidth: 3.1539e+04
```

```
          pvars: {1x11 cell}
          svars: {'theta'  'betaV'  'betaU'  'lamVz'  'lamUz'  'lamWs'
                  'lamWOs'  'lamOs'}
        svarSize: [1 1 4 1 2 2 1 1]
          wvars: {1x8 cell}
```

Again, we'll explain the fields of `pout.mcmc` in Section 9.

(b) Now we can get some new MCMC draws (realizations) that we'll use for estimating the model parameters. These will be added to the `pvals` field of `pout`.

```
>> nmcmc = 10000; % number of draws we want
>> pout = gpmmcmc(pout, nmcmc, 'step', 1)
Started timed counter, vals 1 -> 10000
        787..1577..2363..3158..3923..4675..5409..6152..6877..7614..  1.7 min,
        0.5 min remain
        8344..9077..9817..2min:12.57sec
```

The `'step', 1` flag tells `gpmmcmc` to use the proposal widths currently in the `pout` object.

```
pout =

       data: [1x1 struct]
      model: [1x1 struct]
     priors: [1x1 struct]
       mcmc: [1x1 struct]
    obsData: [1x3 struct]
    simData: [1x1 struct]
      pvals: [1x11300 struct]
```

Note that there are now 10,000 additional values recorded for each parameter in the `pout` object. These were produced by the 10,000 MCMC iterations carried out by the last call to `gpmmcmc()`.

4. At this point we have everything we need to make predictions, including the principal components, the discrepancy basis, and the dimensions of the model. We can do a `save pout` to record everything we've computed so far.

7 Some diagnostic plots

In this section we will use (x, θ) and (R, C) interchangeably; the GPM/SA code is written in terms of (x, θ), while the code for our tower example is written in terms of (R, C). Recall that $x = R$ and $\theta = C$ in this example.

We'd like to look at the MCMC draws and the resulting parameter estimates. Since working with all 10,000 draws in `pout.pvals` can be cumbersome, in some cases we'll take a smaller, evenly spaced sample of the draws and examine those instead.

```
from = 2000; % start getting realizations at this index
to = length(pout.pvals); % continue to the last realization
thismany = 500; % grab this many evenly spaced realizations
ip = round(linspace(from, to, thismany)); % indices of the pvals to use
```

With the exception of the `showPvals()` function below, the plotting functions in this section are not part of the GPM/SA code package. The `.m` files used are available separately on the web page.

7.1 Traces of the MCMC draws

The GPM/SA code function `showPvals()` will produce traces of the MCMC draws for the parameters in the model as shown in Figure 4.

```
showPvals(pout.pvals);
Processing pval struct from index 1 to 11300
     theta:   mean           s.d.
         1:        0.372      0.03914
     betaV:   mean           s.d.
         1:       0.5771       0.8957
     betaU:   mean           s.d.
         1:        8.428        2.372
         2:       0.6872       0.3639
         3:        4.281        2.547
         4:        3.329        2.626
     lamVz:   mean           s.d.
         1:        101.4        79.35
     lamUz:   mean           s.d.
         1:       0.4686       0.1506
         2:       0.8496       0.3154
     lamWs:   mean           s.d.
         1:          457        296.1
         2:        968.7        574.3
    lamWOs:   mean           s.d.
         1:    2.851e+04         2010
     lamOs:   mean           s.d.
         1:    1.564e+04         3599
    logLik:   mean           s.d.
         1:        37.55        29.62
  logPrior:   mean           s.d.
         1:         1663        4.191
   logPost:   mean           s.d.
         1:         1700        28.75
```

Note that we're calling `showPvals()` with *all* the draws in `pout.pvals`, not just the ones specified by the index `ip` defined above. This includes the draws used for burn in and step size estimation. The resulting plot is shown in Figure 4. The figure shows that the draws of all the parameters have a stationary distribution after 2000 draws, indicating convergence of the chains. Note that a

rerunning of this MCMC chain will give slightly different answers due to variation in the random numbers generated.

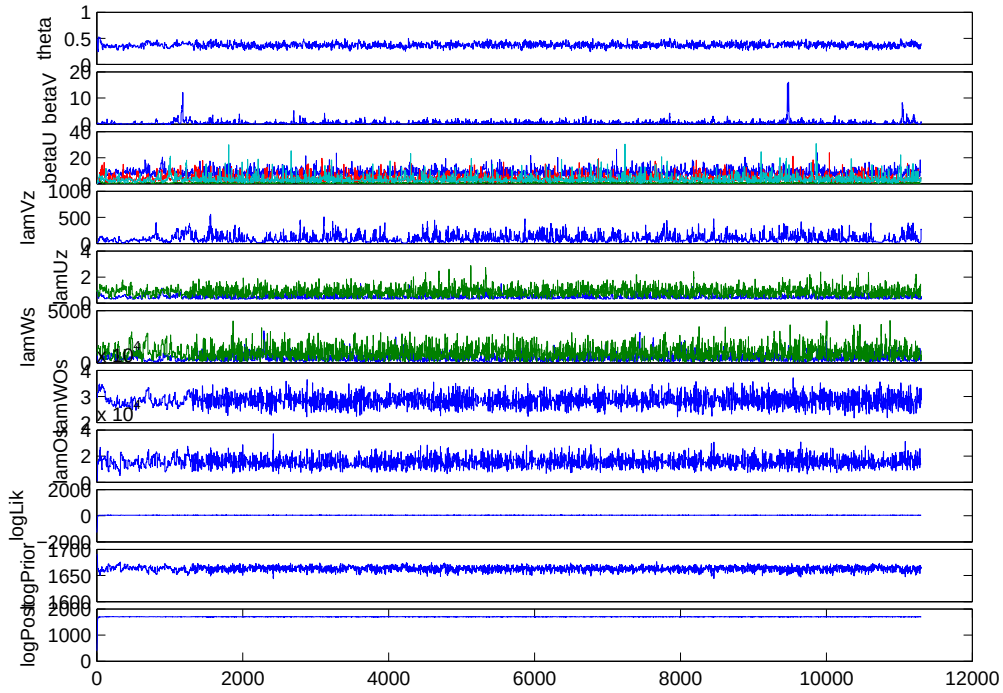

Figure 4: Traces of the MCMC draws of the parameters in `pout.pvals` as generated by the `showPvals()` function.

7.2 Posterior distribution of $\theta = C$

Figure 5 shows the histogram of the MCMC draws from the posterior distribution of θ. These values come from the top row of Figure 4, excluding the burn-in draws. This plot was made with the code in `thetaposthist.m` using just the `thismany` realizations specified by the index `ip` above:

```
thetaposthist(pout, ip);
```

7.3 Principal components

Figure 6 shows the $p_u = 2$ principal components used in this example. Note that the vertical scale for the second principal component is much smaller than that of the first; this confirms that most of

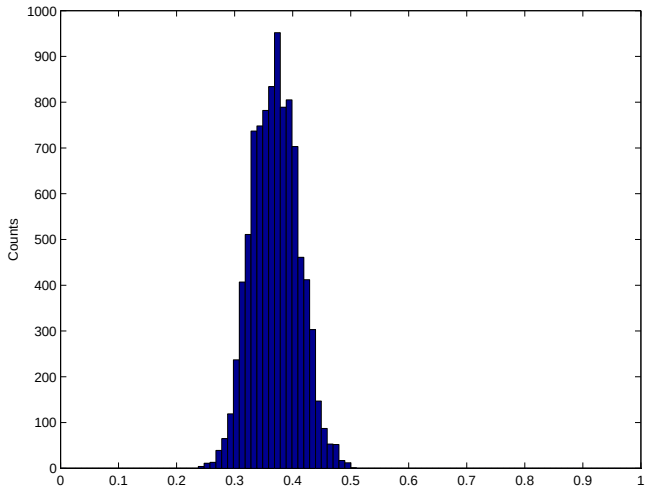

Figure 5: Histogram of draws from the posterior distribution of C, on the $[0, 1]$ scale.

the variation in the data is being captured by the first principal component. We make this plot as follows:

```
npc = size(pout.simData.Ksim, 2); % number of principal components
for ii = 1:npc
    subplot(npc, 1, ii);
    plot(pout.simData.Ksim(:, ii))
    title(['PC ', num2str(ii)])
end
```

Figure 7 shows the posterior mean of the Gaussian process of the weight functions for the two principal component at any given (x, θ) pair. The weight $w(x, \theta)$ at each (x, θ) pair is used to make the predictions from the model. The code `PCresponsesurf.m` calls the function `gPred()` to make predictions at each grid point and then generate the plot:

```
PCresponsesurf(pout, ip);
```

7.4 Correlation parameters in the Gaussian process fit

Using the MCMC draws of the spatial dependence parameters β, we can compute $\rho = \exp\{-\beta/4\}$. The value of ρ gives us information about the dependence of the simulation output on each input parameter x and θ. Figure 8 shows boxplots of the posterior draws for the ρs for each x and θ, and for each principal component. As above, the figure was generated using a subset of the realizations:

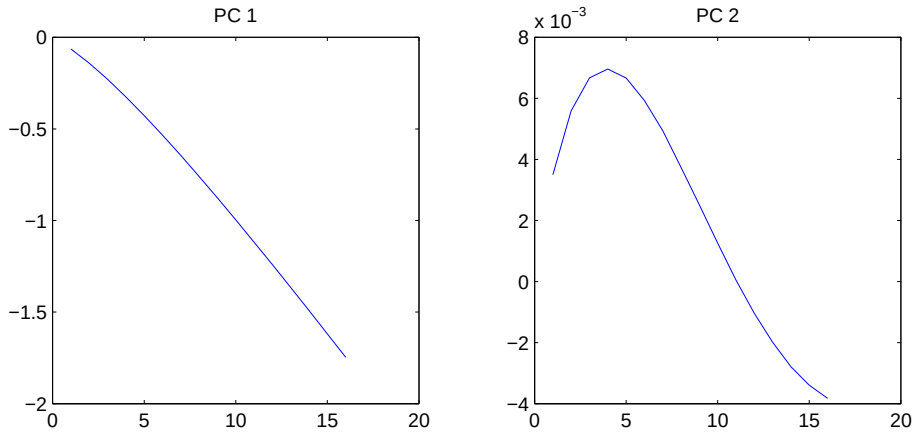

Figure 6: The $p_u = 2$ principal components used in modeling this example. Each is a 16-vector using the $n_\eta = 16$ heights in the simulator grid. Note the different scales on the two y-axes; PC 2 captures much smaller variations in the data than does PC 1.

```
rhoboxplots(pout, ip);
```

When $\rho = 1$ for a particular x or θ and principal component, it means that particular component of the simulator output is constant along that dimension. Then the simulation is not sensitive to that input; i.e., knowing the value of the input gives no information about the value of the output. As ρ goes smaller than 1, this indicates activity associated with that input. The outputs will vary smoothly with the inputs, with smaller ρs indicating less smoothness.

As ρ approaches 0 for a particular dimension, the modeled response is increasingly flexible; the model eventually fits the data as noise, i.e., it doesn't find any smooth trend. This suggests predictions from the model are suspect. Thus if any of the boxplots in Figure 8 show values that are all close to zero, more diagnostics (e.g., cross-validation) should be considered before accepting predictions from the model.

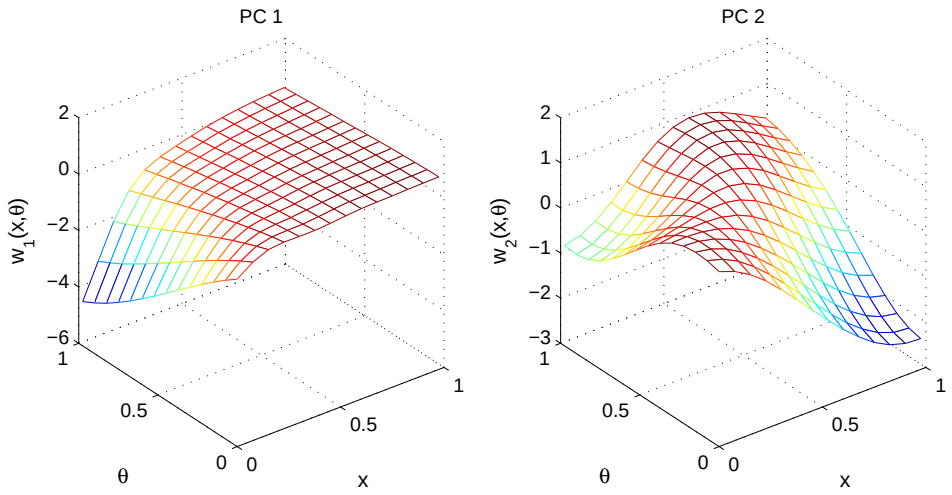

Figure 7: Posterior mean of the Gaussian processes of the weight functions for the two principal components. The weights $w(x, \theta)$ are used to make the predictions.

7.5 Discrepancy estimation

The GPM/SA code calibrates the input parameters to match the field data as well as possible. It then uses another Gaussian process to estimate the discrepancy between the resulting simulation (using the calibrated parameter values) and the field data. Figure 9 shows how this can work. The left column shows the calibrated simulations; the center column shows the discrepancy between the field data (circles) and the calibrated simulations; and the right column shows the calibrated predictions made after adding the discrepancy term to the calibrated simulation.

The code `etasdeltas.m` computes predictions for each experiment, computes the discrepancies, and plots them along with the calibrated predictions:

```
etasdeltas(pout, ip);
```

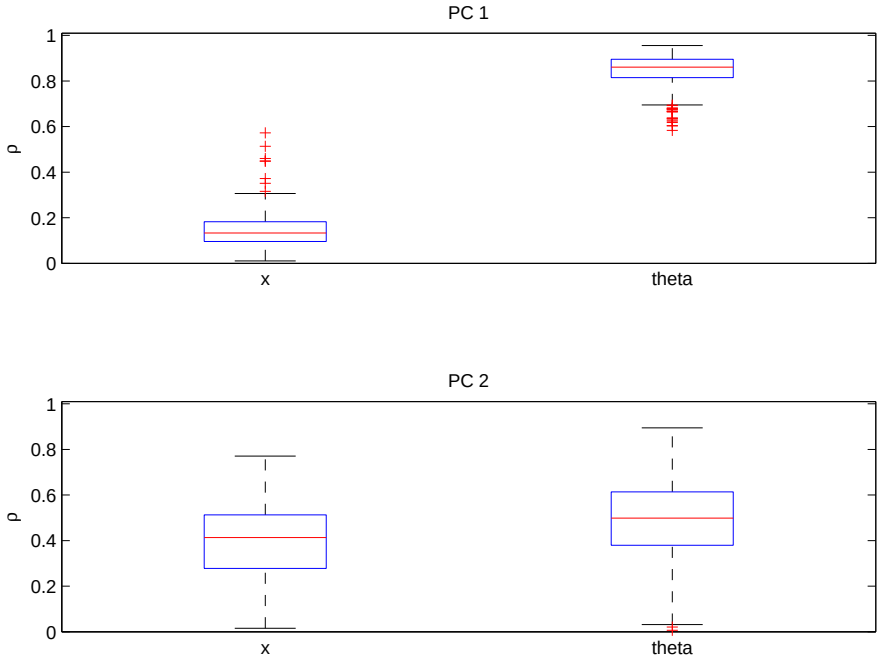

Figure 8: Boxplots of $\rho = \exp\{-\beta/4\}$ for the draws of β associated with x and θ for each principal component.

7.6 Predictions

Predictions. For each run i of the simulator, we construct a model using the other runs (excluding run i), then predict run i based on the resulting model. This allows us to look for trends in the quality of our predictions as a function of the inputs R_{sim} and C_{sim}. Figure 10 shows the results sorted by R_{sim}, while Figure 11 shows them sorted by C_{sim}.

We can also look at the residuals for the predictions on each of these held out runs as shown in Figure 12. Each curve gives the residual for one run at each height. The plot shows how the predictions are better at lower heights, which makes sense since our simulator model's error in using a linear velocity term rather than a squared term will be exacerbated at higher heights.

Predictions at particular input values. Figures 10 and 11 are a little overwhelming with so many plots to consider. A coarser way to explore trends of this sort is to do a similar plot using some selected "high," "middle," and "low" values of R_{sim} and C_{sim}. This is shown in Figure 13.

Figure 14 gives sensitivity plots to show changes in the simulation output values as the values of x (left) and θ are adjusted.

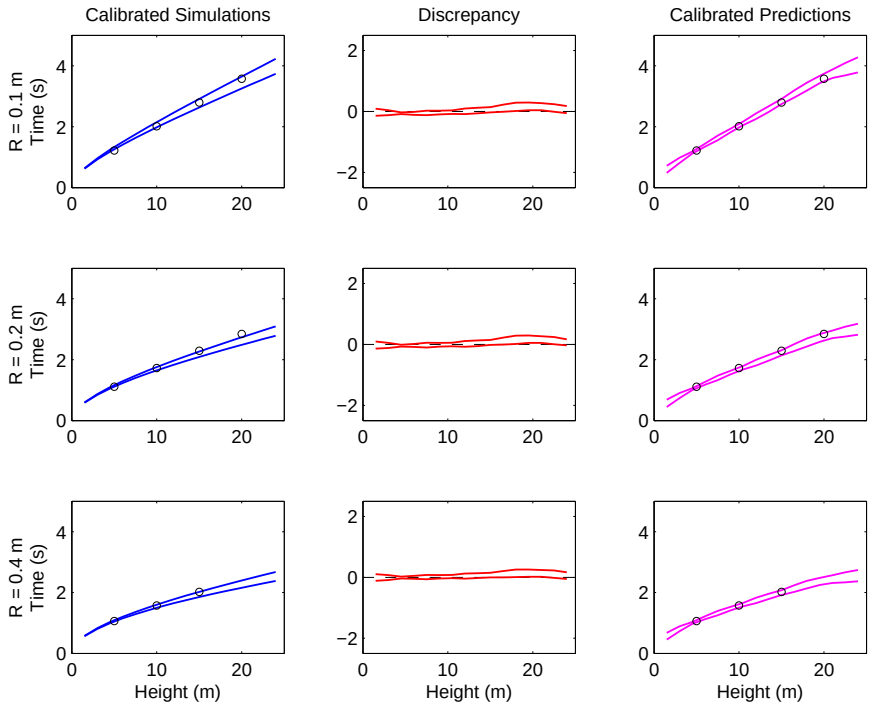

Figure 9: Circles show the field data, and colored lines indicate the 5th and 95th percentiles. Each row is a different ball size. *Left column:* Calibrated simulations. *Center:* Discrepancy term (dashed line shows where zero discrepancy would be). *Right:* Calibrated predictions = calibrated simulations + the discrepancy term.

8 What if I don't have any experimental data?

The GPM/SA code can be used for a sensitivity analysis even in the absence of experimental data. If only simulations are available, set all the components of the `obsData` structure to be `[]`. In this case, there will be no parameters θ to calibrate to field data. Hence the model will construct an emulator for the simulation code.

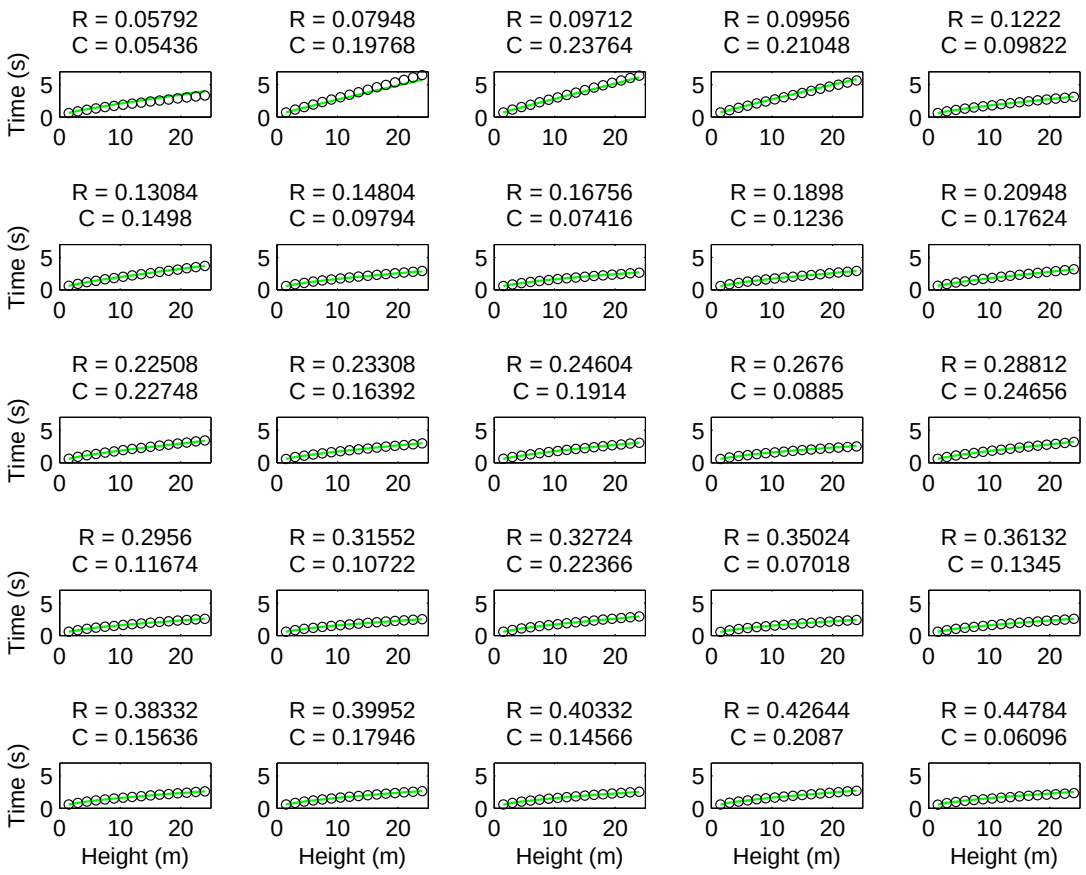

Figure 10: Hold-out predictions, sorted by the value of R_{sim}.

9 The `pout` object

The ball dropping example has produced `pout` which holds a variety of objects. The preprocessing function `readdata()` constructs the `obsData` and `simData` objects. The `obsData` object holds information regarding the physical observation data, while the `simData` object holds information regarding the simulation output, including the basis representations for the multivariate simulation output and the discrepancy basis.

The function `setupModel()` attaches four additional objects to `pout`: `data`, `model`, `priors`, and `mcmc`. It also creates an empty object `pvals`, which will later hold the MCMC output produced by `gpmcmc()`. Hence the posterior samples for the various parameters will be kept in the `pvals` object. The `data` object holds transformations of the simulation and observed data that are required for likelihood evaluations used in the MCMC algorithm. There should be no need to modify this data. The `model` object holds the all of the additional objects required to evaluate the likelihood and prior. The `priors` object holds the prior specification for each of the

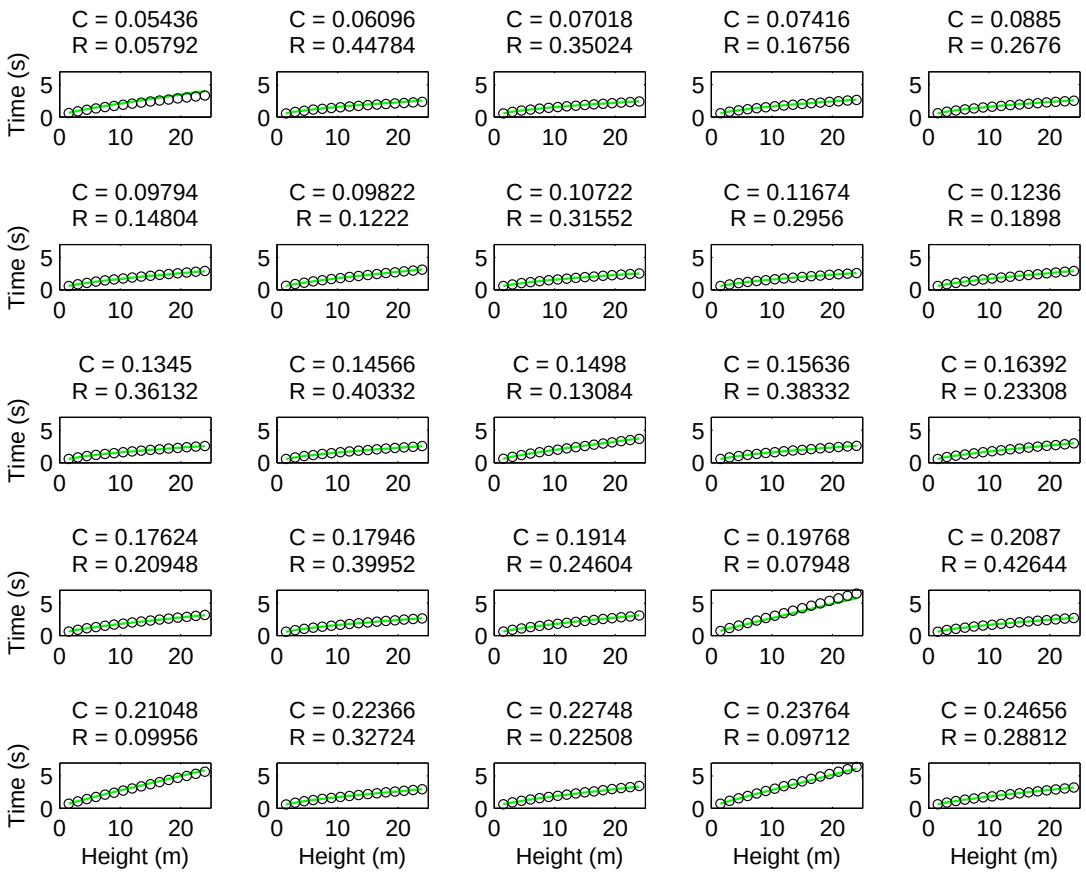

Figure 11: Hold-out predictions, sorted by the value of C_{sim}.

model parameters. This includes upper and lower bounds for each parameter. Finally, the `mcmc` object holds information required to carry out the MCMC sampling. In particular, step sizes used in the Metropolis and Hastings updates for each parameter. This object is modified when the stepsize estimation is carried out in `gpmcmc()` is called.

Detailed descriptions of each of these fields is provided in the reference manual.

References

D. Higdon, J. Gattiker, B. Williams, and M. Rightley. Computer model calibration using high-dimensional output. *Journal of the American Statistical Association*, 103(482):570–583, 2008.

Dave Higdon. Space and space-time modeling using process convolutions. In C. Anderson,

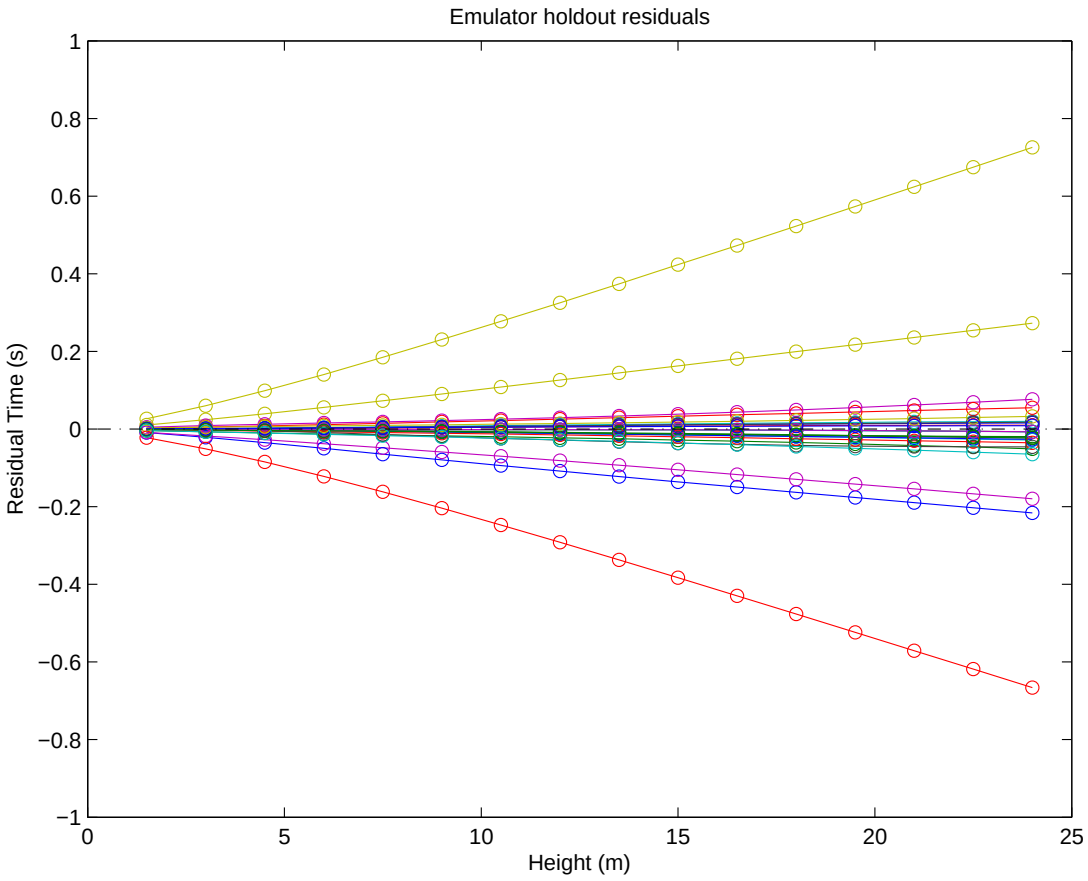

Figure 12: Residuals for each run of the simulator (one curve per run).

V. Barnett, P. C. Chatwin, and A. H. El-Shaarawi, editors, *Quantitative Methods for Current Environmental Issues*, pages 37–56, London, 2002. Springer Verlag.

M.C. Kennedy and A. O'Hagan. Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(3):425–464, 2001.

Christian P. Robert and George Casella. *Monte Carlo statistical methods*. Springer-Verlag Inc, 1999.

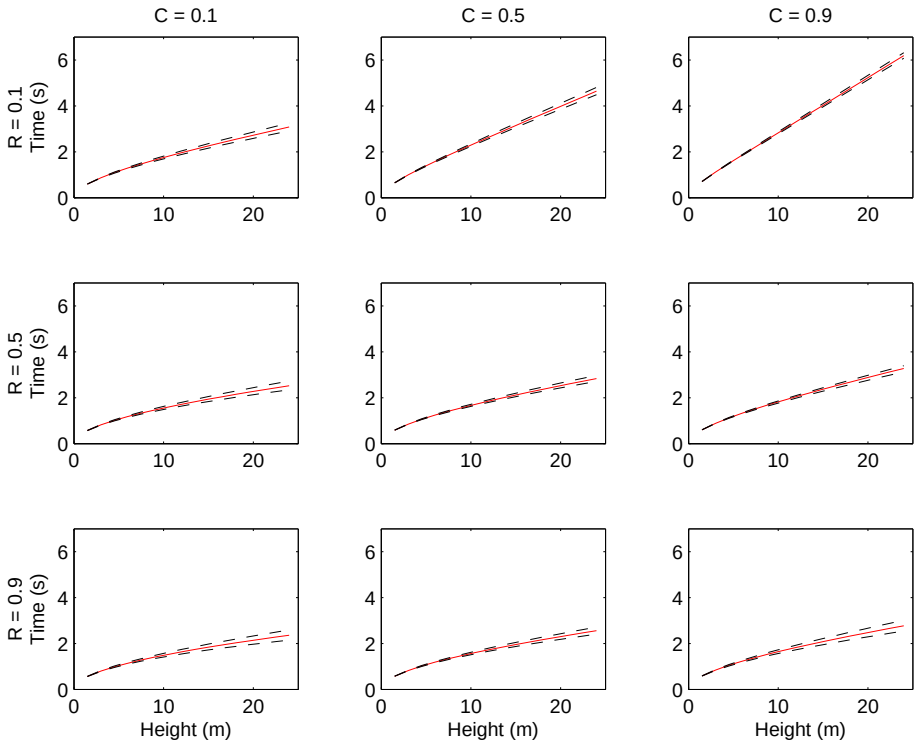

Figure 13: Predictions for a selection of values of R_{sim} and C_{sim} to get a look at any main effects and interactions. The red lines are the medians; the dashed black lines are the 5th and 95th percentiles.

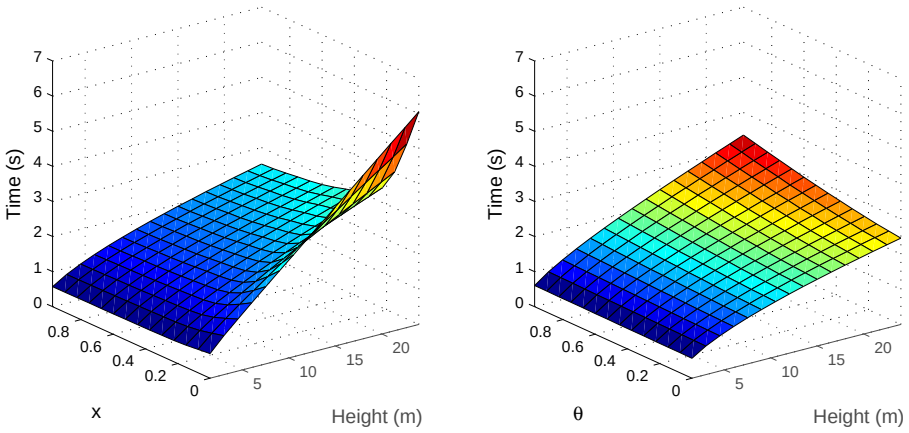

Figure 14: Surface plots showing sensitivity to X (left) and to θ (right).