Ball Drop Example, No Bias

Grant Hutchings

2/23/2022

In this example we consider the experiment of dropping different sized balls from a tower and recording the time at certain distances during the fall. We will generate time data at distance traveled d from the equation

$$t(d) = \frac{acosh(\exp\{Cd/R\})}{\sqrt{Cg/R}}$$

where C is the coefficient of drag, R is the radius of the ball, and g is the gravitational constant. We will generate both field observations and computer simulation data from this function. Five sets of field observations will use the values $C = .1$, $g = 9.8$ and $R = \{.05, .1, .15, .2, .25\ (m)\}$, while computer simulations will be evaluated over a space filling design of (R, C, g) tuples in the domain $R \in (.025, .3)$, $C \in (.05, .15)$, $g \in (7.8, 10.8)$. The drag coefficient and the gravitational constant are parameters to be calibrated while R is a controlled input.

During this tutorial we will detail how to do emulation, calibration, and prediction using our library functions.

First, lets visualize our experiments and simulations. We have the five experiments shown in black, where time observations are recorded at four distances, 5, 10, 15, and 20 meters from the drop. White noise is added to the data with $\sigma_y = .1$. Our computer model is evaluated on a much denser grid of distances; every 1.5 seconds from .5 seconds to 24.5. Below we see the height time curves for both simulation and experiment. Our simulations cover the range of experiment, which is desirable so that our emulator does not have to extrapolate.

Step 0: Pre-scale data

The first step for using our package is to pre-scale the data. We scale all inputs to the computer model to lie in the unit interval. We will denote controllable inputs as x and calibration parameters at t. Here we have $x = \{R\}$ and $t = \{C, g\}$. Scaling to the unit interval is done using the function `transform_xt()` which accepts at minimum a matrix `Xsim` which has, as columns, the R (controllable inputs) values passed into the simulator. In an emulation only problem `Xsim` is all we need. Here we are interested in calibration as well so we pass in a matrix `Tsim`, which has as columns the C, g values used to generate our simulated data. In this example we have $m = 25$ simulations so `Xsim` $\in \mathbb{R}^{m \times 1}$ and `Tsim` $\in \mathbb{R}^{25 \times 2}$. We also pass in a matrix `Xobs` $\in \mathbb{R}^{n \times 1}$ which contains the n experimental values of R. Additionally, for testing problems where the true value of the calibration parameters are known, a matrix `Tobs` can be passed to the function. The purpose of this is to have a stored copy of the true parameter values scaled similarly to `Tsim`.

Next we will scale the computer simulation outputs `Ysim` $\in \mathbb{R}^{n_s \times m}$ where n_s is length of the simulation height-time curves (i.e. the number of distances we have time measurements for). We also scale `Yobs` using the scaling determined by `Ysim`. We scale the simulations so that at each distance location, the times are mean zero and unit variance. Responses are scaled using the function `transform_y` which accepts the simulator data `Ysim` as well as a matrix of the locations (distances) where the observations are taken `YindSim`. Similarly for the field observations we pass in `Yobs` $\in \mathbb{R}^{n_y \times n}$, `YindObs` $\in \mathbb{R}^{n_y \times 1}$ where n_y is the number of distances for which times are recorded. The function also accepts arguments which control the type of scaling. `center` and `scale` are flags indicating if the data should be centered to mean zero and scaled to variance one. `scaletype` can be one of 'columnwise' or 'scalar'. Columnwise scales each location to be variance one, and scalar scales

the entire suite of simulations to be variance one. We recommend using columnwise scaling unless there is a location with no variability, meaning we cannot divide by the standard error at that location.

```
XTdata = transform_xt(Xsim,Tsim,Xobs)
Ydata = transform_y(Ysim,YindSim,Yobs,YindObs,center = T,scale = T, scaletype='columnwise')
```
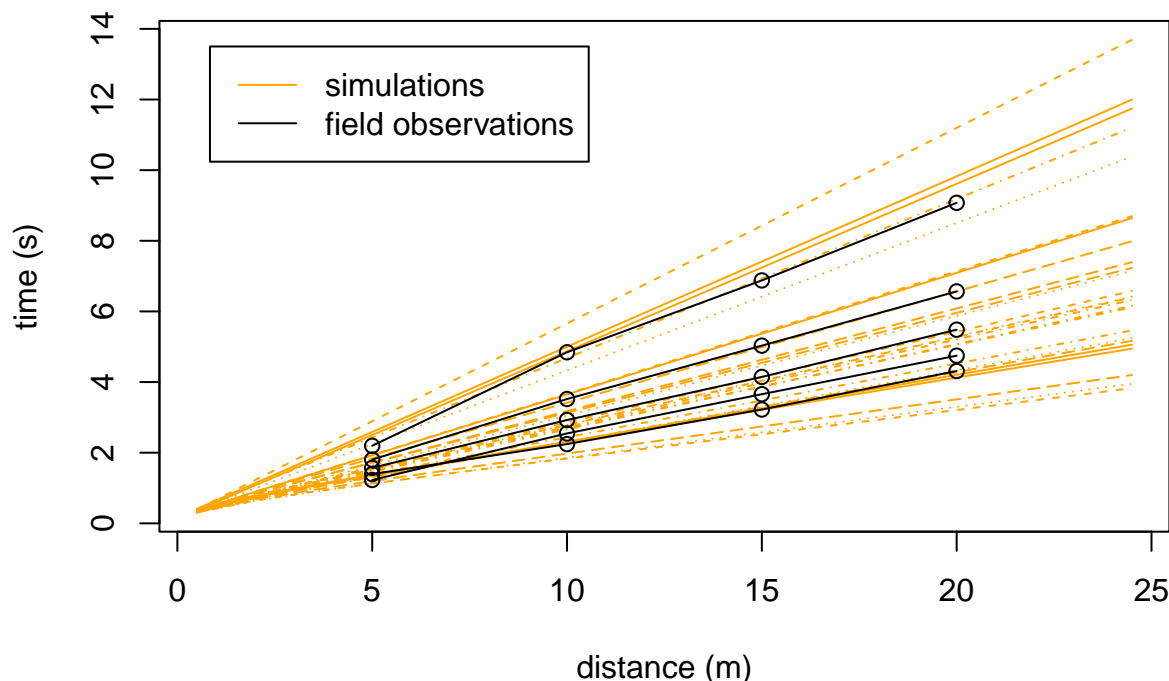

Step 1: get basis

Next we decompose our data using an orthogonal basis decomposition, namely the singular value decomposition. We first create a basis for our simulations using the function `get_basis()` which accepts our simulator responses that have already been scaled by `transform_xt()` as well as nPC, the number of basis vectors to use. If you do not know a-priori how many basis vectors you would like to use, the parameter $\texttt{pctVar} \in (0, 1)$ can be specified indicating that the decomposition should retain enough basis vectors to account for a certain percentage of the variability in the data. We then generate the basis for the field observations, using the basis for the simulations with the function `get_obs_basis`. At this point you will recognize all the inputs to this function except for `sigY`. This value defaults to one, and has no effect if not specified. If one has an idea of the error variance for the field observations this can be passed in here and will be used when producing basis vectors.

```
simBasis = get_basis(Ydata$sim$trans,nPC,pctVar=NULL)
obsBasis = get_obs_basis(simBasis,Ydata$obs$trans,YindSim,YindObs,sigY=ifelse(sd_true>0,sd_true^2,1))
```

It may be useful to visualize the basis weights for both the simulations and the observations. We see that some of the weights from the field observations are not within the cloud of weights from the simulator. This is not ideal as our models will be doing some extrapolation to predict the observed data.

```
plot_wz_pairs(simBasis$Vt,obsBasis$Vt,legend=T)
```
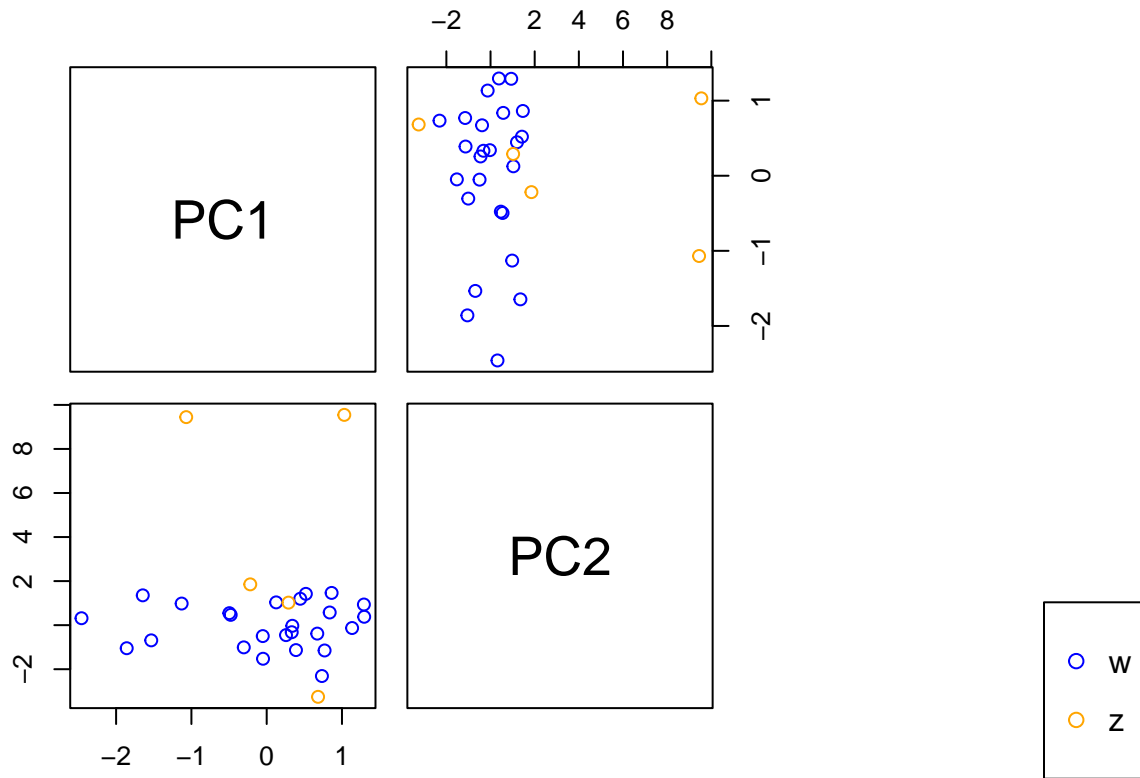
Step 2: get lengthscale estimates and stretch + compress input space

Next we will globally estimate length-scale parameters for the laGP models using a subset of the simulations. This is done using the function `mv_lengthscales()` which accepts the output from `transform_xt()` as well as the simulation basis weights. Lastly you can specify a value for the nugget, we use `g` in accordance with the laGP package. These global length-scale estimates are a key aspect of the model. The inputs are scaled according to these inputs, which eliminates the need for future length-scale estimation during emulation and calibration, greatly increasing speed. We use the function `get_SC_inputs()` to scale the inputs. This function accepts the outputs from `mv_lengthscales()` and `transform_xt()` and returns an object similar to `XTdata`, but containing scaled inputs.

```
estLS = mv_lengthscales(XTdata,simBasis$Vt,g=1e-7)
SCinputs = get_SC_inputs(estLS,XTdata,nPC)
```

Step 3: calibrate over grid of t values

Now we can calibrate over a grid of possible (C, g) pairs. Since our inputs are scaled to the unit hypercube, we will take a simple uniformly space grid over the unit square. In the future we will evaluate our calibration on a small space filling design, then use those results as initial values for a much more sophisticated optimization routine. We do this initial calibration over a user defined set of (C, g) values using the function `mv.calib.nobias()` the parameters to this function have all been discussed above. We show the likelihood over the range of potential calibration parameters as well as the maximum likelihood estimate. In this case, the reason the ML estimate is not equal to the true value is because the data is noisy. If the data was noise-free, and the emulator truly unbiased, as it is here, we should get a ML estimate equal to the true value.
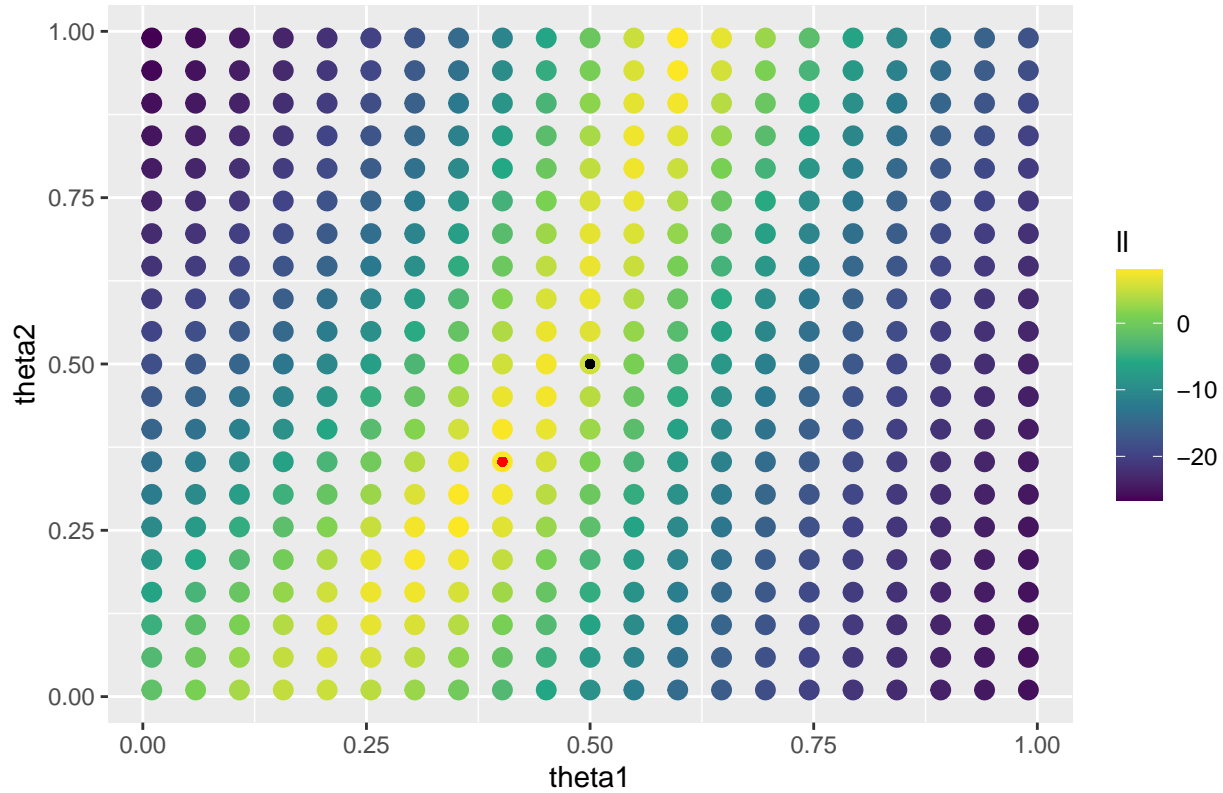
Estimate of error variance is a little high.

```
## True error variance    : 0.01
## Estimated error variance: 0.01450124
```

Prediction at observed locations

We use the function `mv.em.pred()` to make predictions from our emulator at the chosen t^*. Here we are interested in how well we can predict the observed data. This is useful metric since the emulator was not trained on the observed data, the only way the observed data influences these predictions is in the choice of calibration parameters. This function accepts a matrix of inputs for prediction, in this case `Xobs`, the length-scale estimates `estLS`, and the transformed simulation inputs stored in `XTdata` as well as the simulation responses `simBasis$Vt`. We can then pass the emulator Gaussian Process information into the function `yPred.nobias` to predict on the original scale. This function also needs an estimate of the error variance, and the observed basis vectors to case samples of the basis weights to the original scale.

```
emGPs = mv.em.pred(calib$theta.hat.orig,Xobs[1:n,,drop=F],estLS,XTdata,simBasis$Vt)
Ypred = yPred.nobias(emGPs$GPs,calib$ssq.hat,obsBasis$B,BtBinv,nsamples=1000,Ydata$obs,ysamp=F)
```
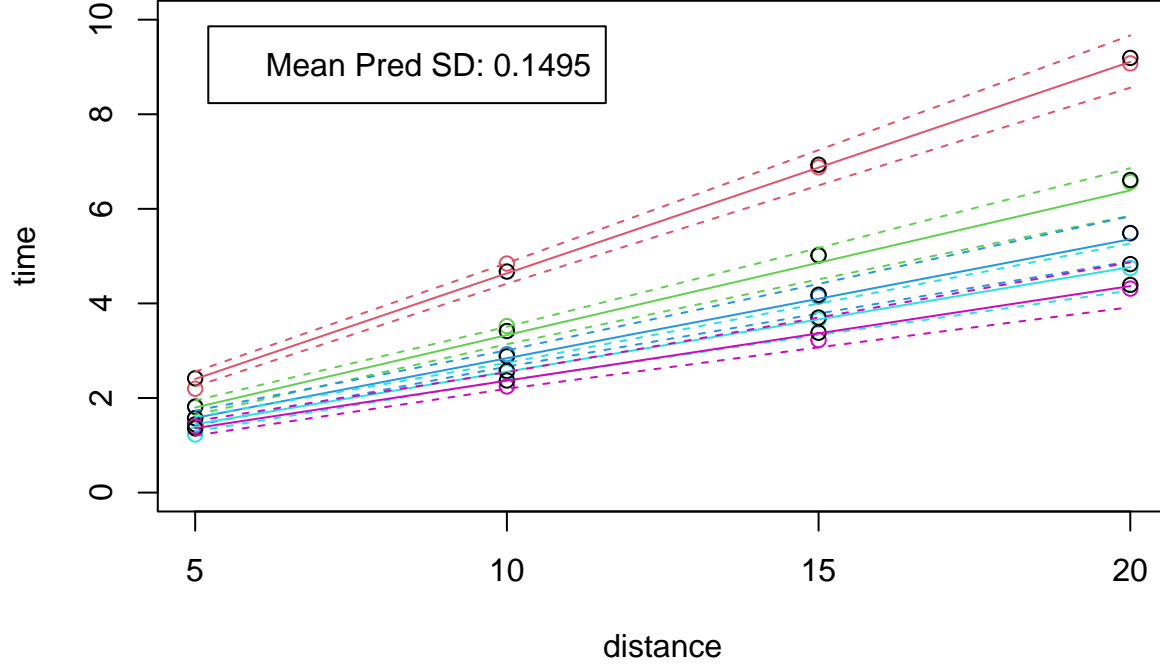
We can similarly predict by directly sampling Y rather than sampling w and transforming through the basis.

Above we used the function `yPred.nobias` with the flag `ysamp=F`. In that case we get samples of basis weights from the sampling distribution

$$z \sim N\big(\mu_w, \hat{\sigma}_y^2 (B^T B)^{-1} + \Sigma_w\big)$$

But, we could also sample from

$$Y \sim N\big(B\mu_w, B\Sigma_w B^T + \hat{\sigma}_y^2 I\big)$$

Which for some reason increases our predictive variability. I believe I understand now why the predictive variances are different, but I'm not sure what the solution is. Consider the following

$$Bz \sim B\mathcal{N}(\mu_w, \Sigma_w) + \mathcal{N}(0, \sigma^2 I) \qquad (1)$$
$$z \sim \mathcal{N}(\mu_w, \Sigma_w) + (B^T B)^{-1} B^T \mathcal{N}(0, \sigma^2 I) \qquad (2)$$
$$Bz \sim B\mathcal{N}(\mu_w, \Sigma_w) + B(B^T B)^{-1} B^T \mathcal{N}(0, \sigma^2 I) \qquad (3)$$

(1) and (3) are equivalent iff $B(B^T B)^{-1} B^T = I$ which we might expect to be true as B came from SVD and is the product of an orthogonal matrix and a diagonal matrix. It turns out that $B(B^T B)^{-1} B^T = I$ if we use all the columns of B, but not if we use only the first nPC columns. For this reason we are getting different predictive variances.

5
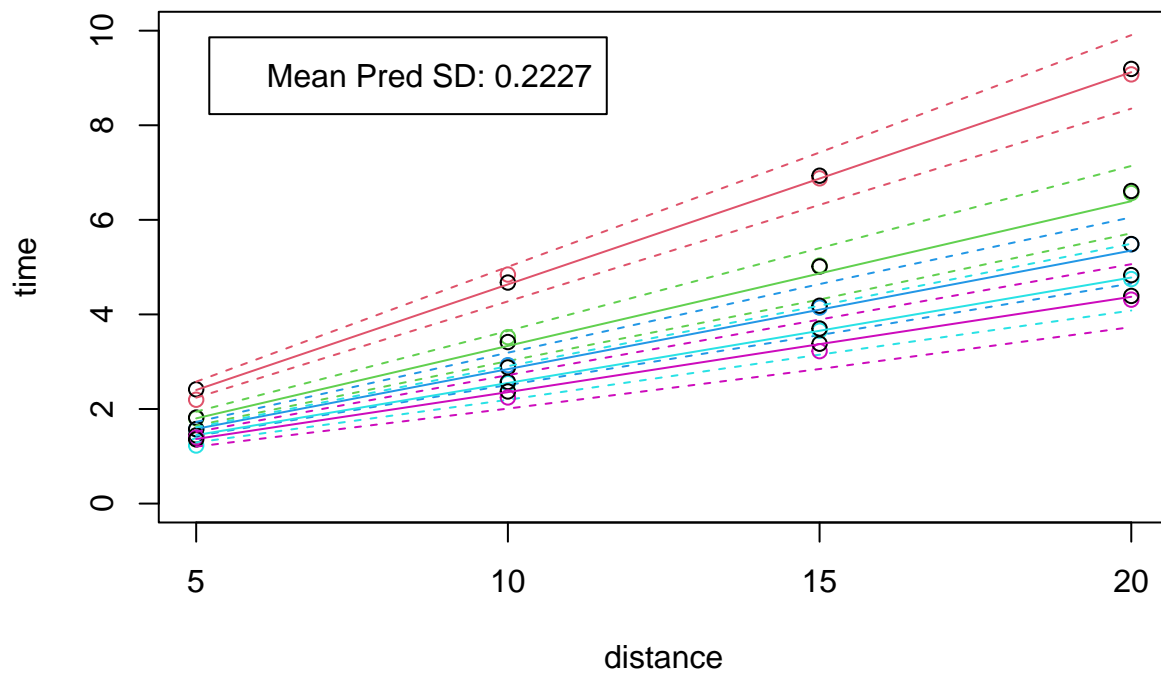
Prediction at new location that was not included for calibration

Mean Pred SD: 0.1413