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.

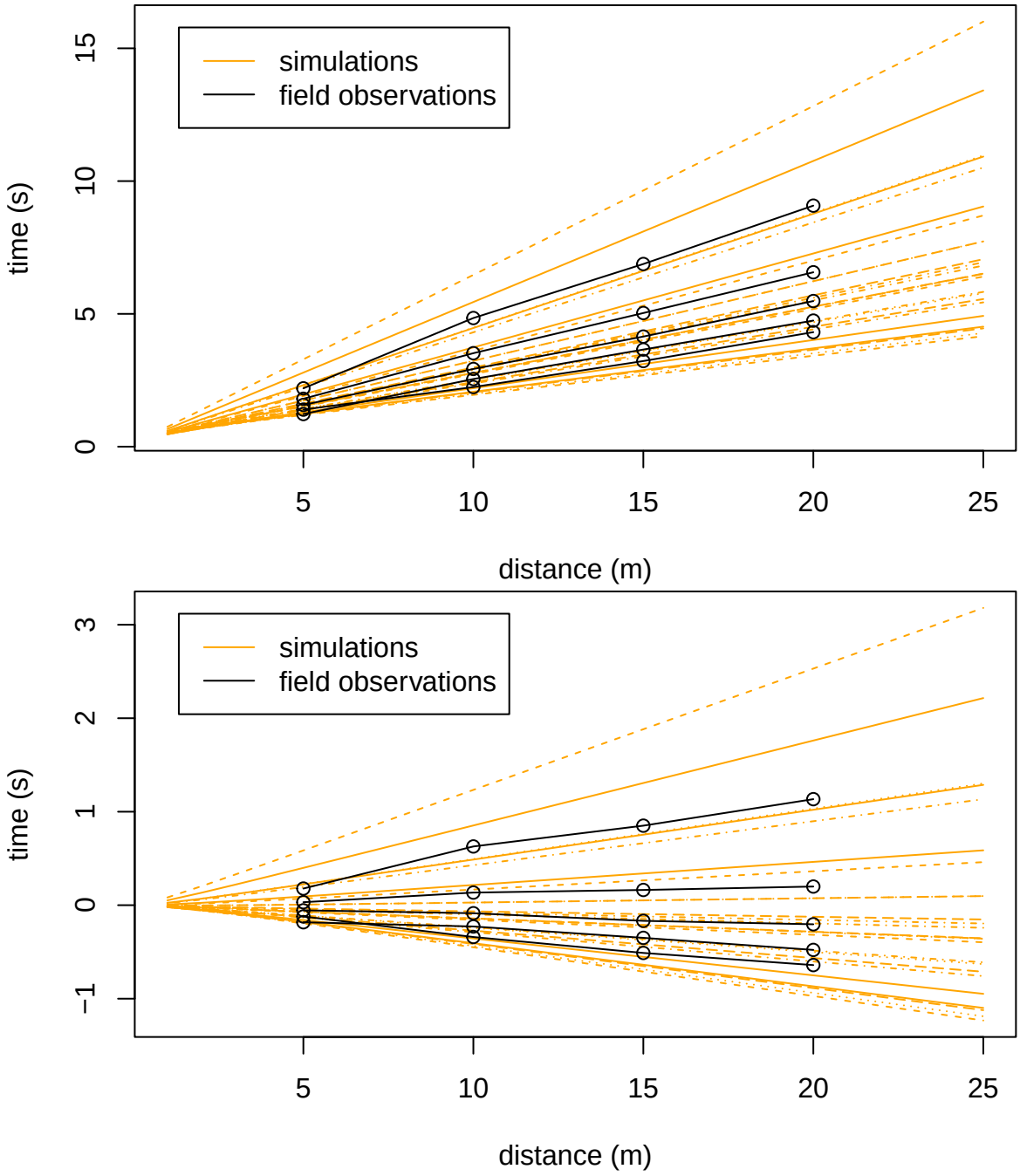
Step 0: Pre-scale data

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 'rowwise' or 'scalar'. rowwise scales each location to be variance one, and scalar scales the

entire suite of simulations to be variance one. We recommend using rowwise 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,Tobs)
Ydata = transform_y(Ysim,YindSim,Yobs,YindObs,center = T,scale = T, scaletype='scalar')
```



Step 1: get basis

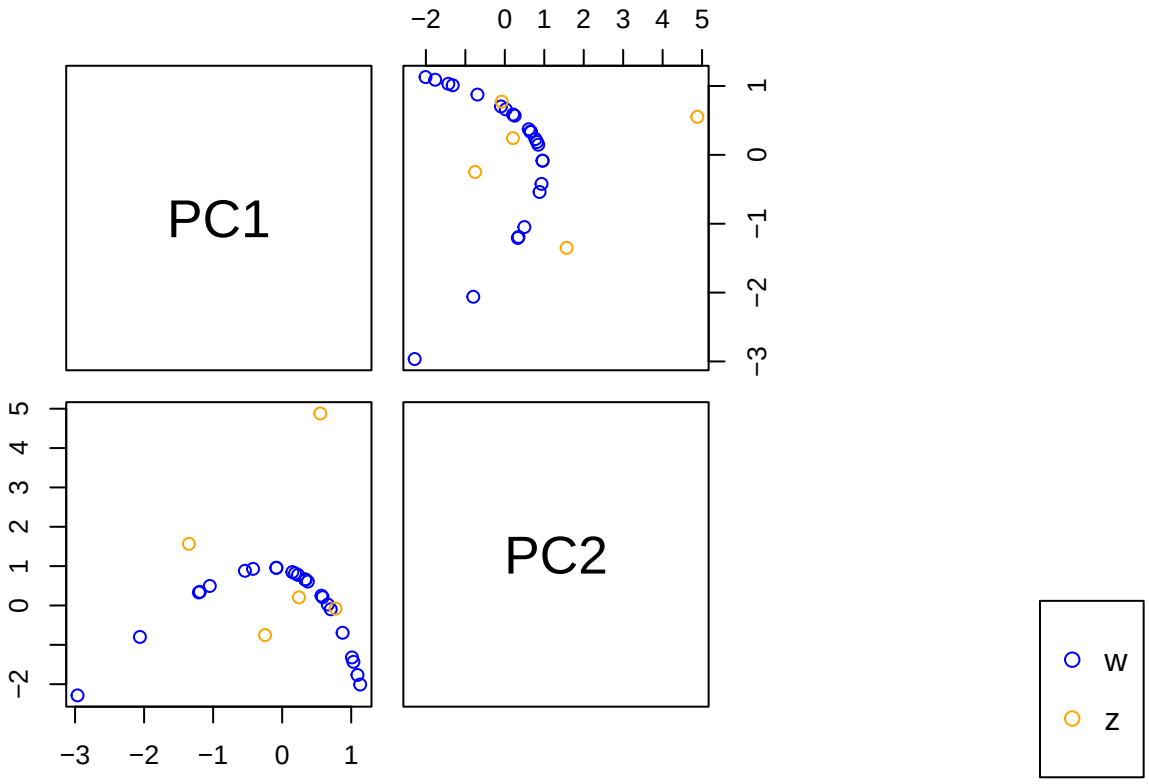
Next we decompose our data using an orthogonal basis decomposition, namely the singular value decomposition. We first create a basis for our simulations using the function `get_basis()` which accepts our simulator responses that have already been scaled by `transform_xt()` as well as `nPC`, the number of basis vectors to

use. If you do not know a-priori how many basis vectors you would like to use, the parameter `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)
obsBasis = get_obs_basis(simBasis,Ydata$obs$trans,YindSim,YindObs)
#obsBasis$B%*%obsBasis$Vt - Ydata$obs$trans
```

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.

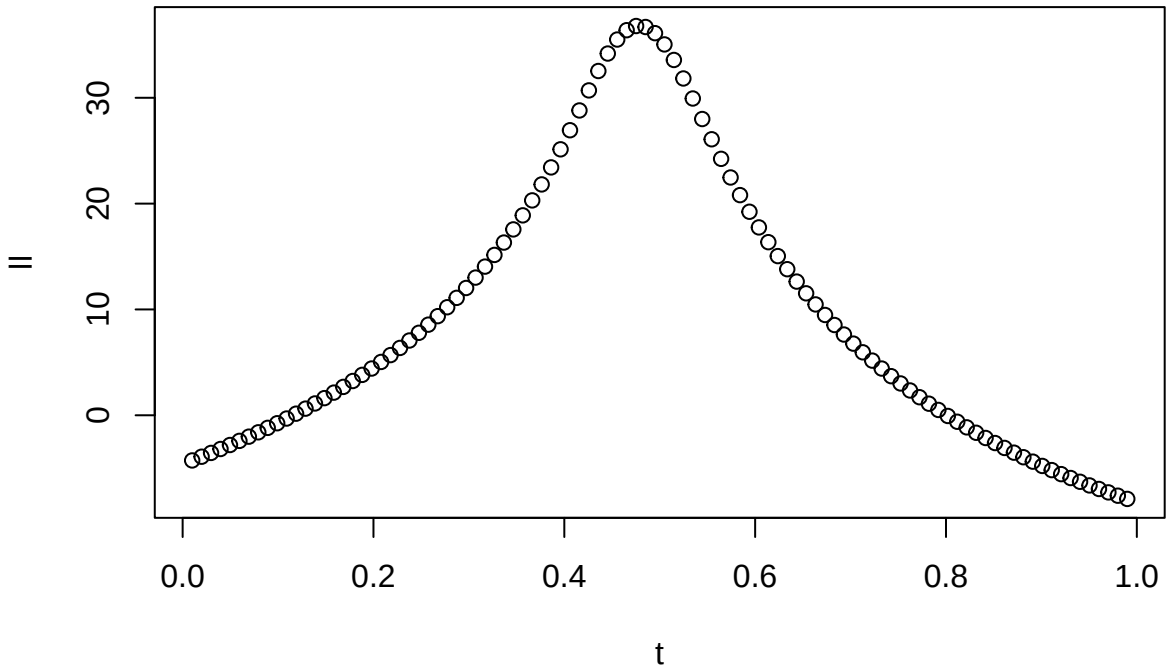
```
estLS = mv_lengthscales(XTdata,simBasis$Vt,g=1e-7)
SCinputs = get_SC_inputs(estLS,XTdata,nPC)
```

```
mvcData = list(XTdata=XTdata,
               Ydata=Ydata,
               simBasis=simBasis,
               obsBasis=obsBasis,
               estLS=estLS,
               SCinputs=SCinputs,
               YindObs=YindObs,
               YindSim=YindSim)
py_save_object(mvcData,'bd_nobias_C_data.pkl')
```

Step 3: calibrate over grid of t values

Now we can calibrate over a range of possible C values. First suppose we are only interested in a point estimate of t^*. In this case, we begin by evaluating the likelihood over a sparse grid of candidate values using the function `calib.init` with `tCalib` a matrix of candidates. Here we specify `sample=F` indicating that we will use the emulator mean rather than sampling from the emulator. This is important as sampling from the emulator will introduce uncertainty in our estimate of t^*. We plot the results of the sparse grid search and see that we have a very well behaved likelihood. By passing the sorted list of likelihood evaluations in `init` to the calibration function `mv.calib` we initialize our optimization algorithm at good starting locations.

```
nT = 100
tCalib = matrix(seq(.01,.99,length.out=nT))
init = calib.init(tCalib,mvcData,bias=F,sample=F)
plot(init$theta1,init$ll,ylab='ll',xlab='t')
```




```
calib = mv.calib(mvcData,init,nrestarts=10,optimize=T,bias=F,sample=F)
cat('t*:', calib$theta.hat,'\n')
```

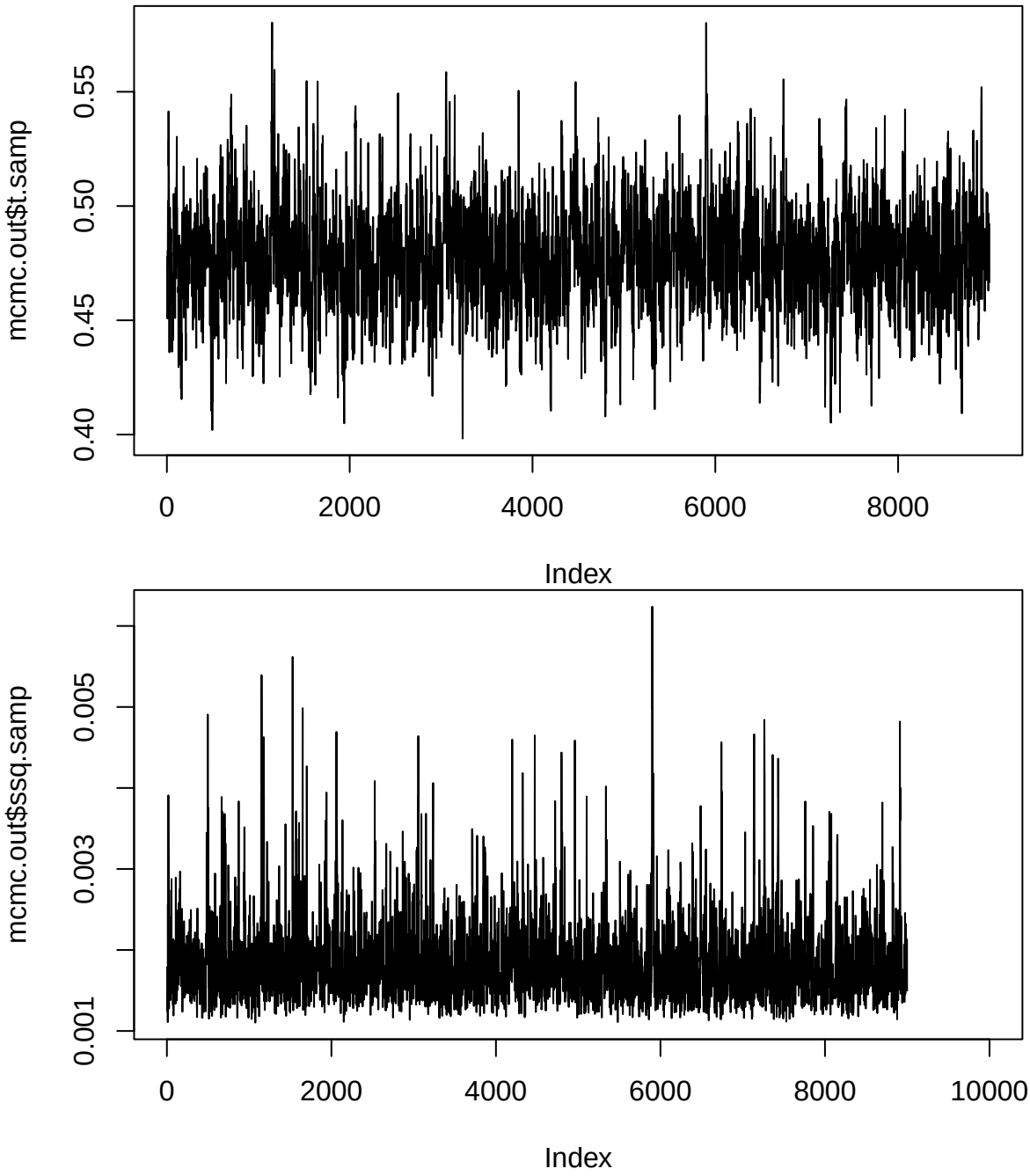
```
## t*: 0.4783175
```

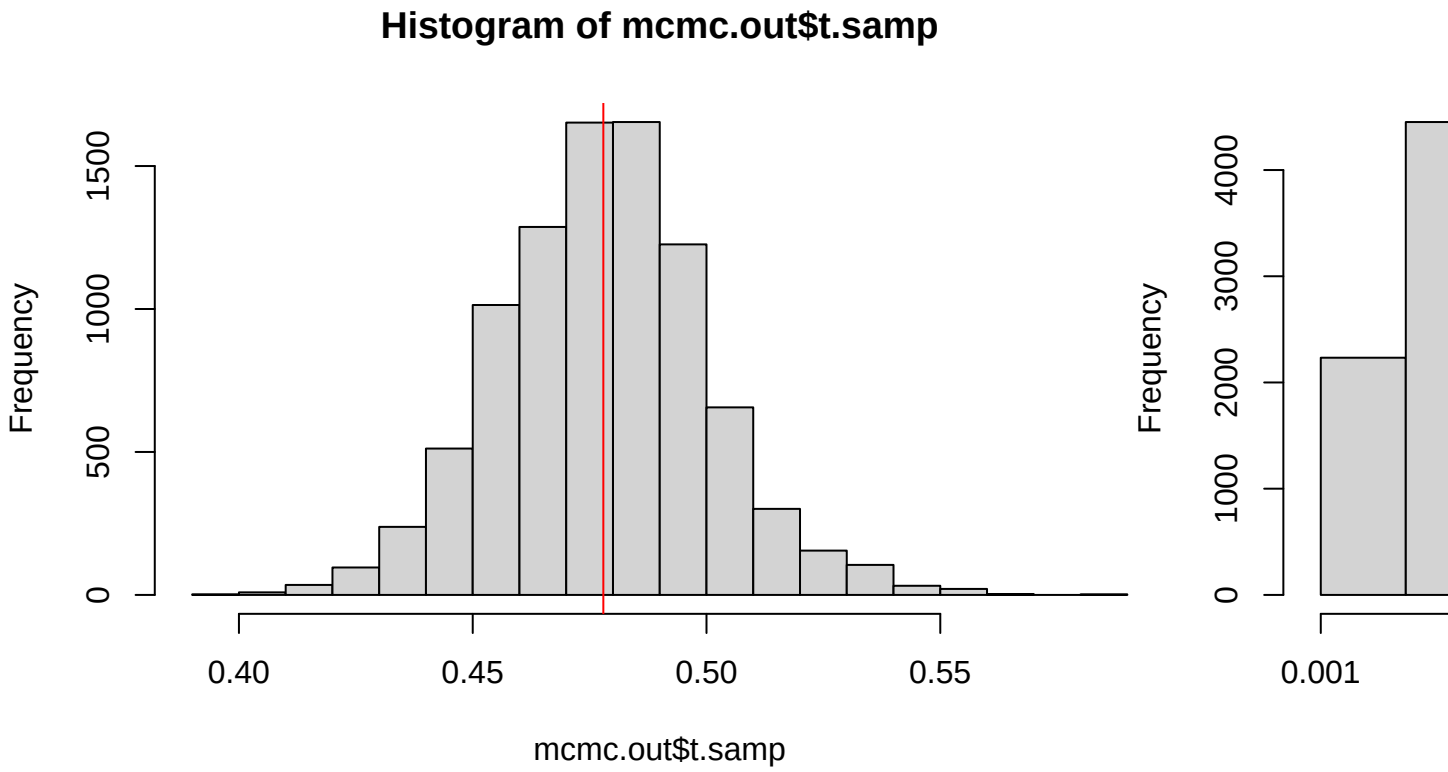
```
cat('estimated error variance:', calib$ssq.hat.orig,'\n')
```

```
## estimated error variance: 0.01027698
```

The more likely scenario is that we are interested in uncertainty. In this case we use a MH-MCMC algorithm to sample from the posterior distribution of t^*.

```
## 10 % 20 % 30 % 40 % 50 % 60 % 70 % 80 % 90 % 100 %
```



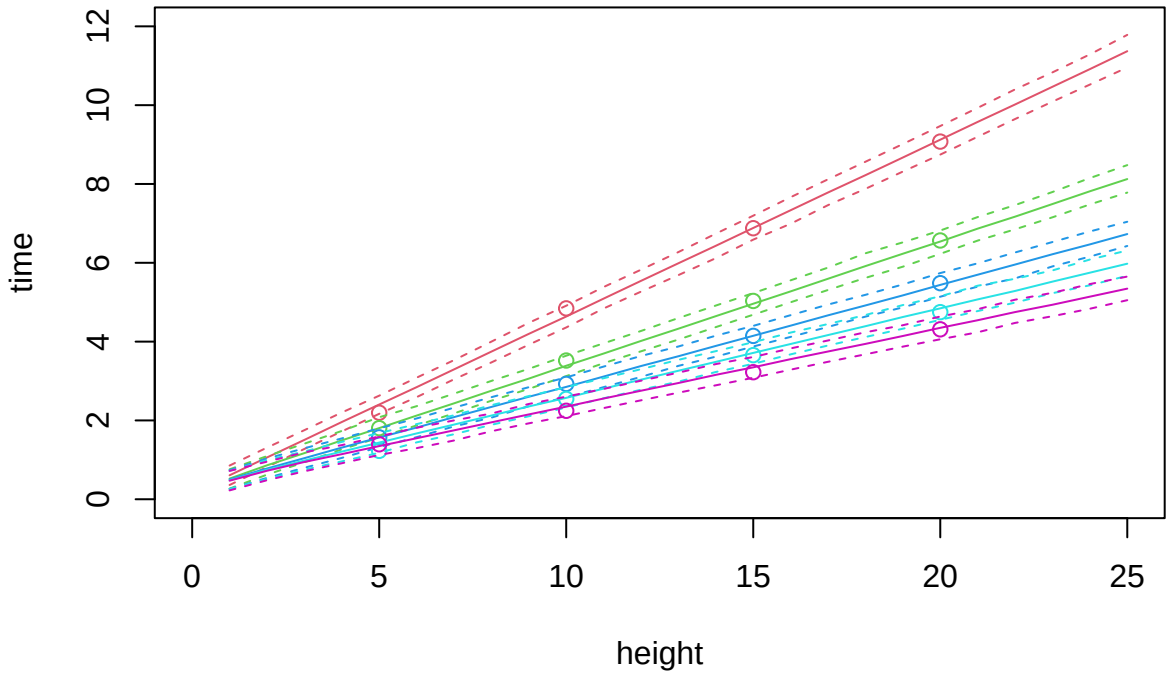


Prediction at observed locations

We now show how to predict from our emulator at both observed and new input settings. Since our emulator is not trained on the observed data, predictive assessment at `Xobs` is of interest. Here we find that prediction using the point estimate and mcmc samples are nearly identical in both mean and uncertainty.

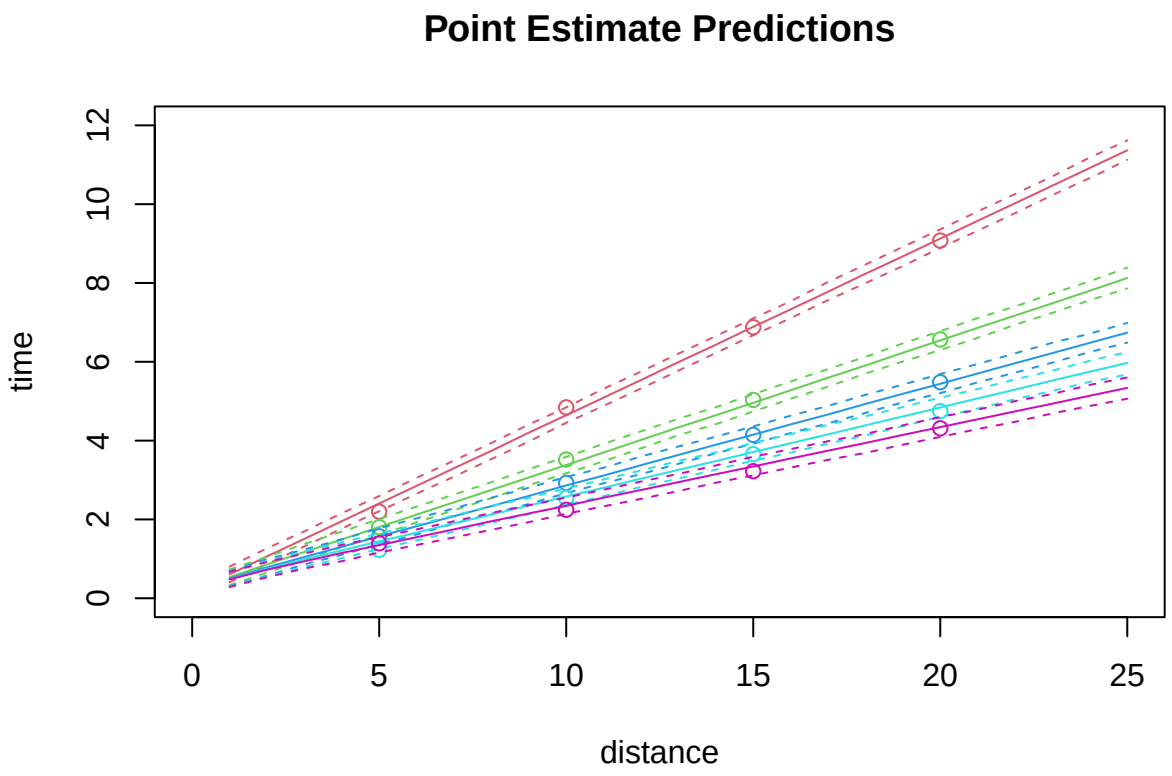
Prediction using mcmc results is done with the function `ypred_mcmc()` passing in a matrix of samples of t^* to use for prediction. We also specify that we would like to predict on the full support of heights from $1m$ to $25m$ by setting `support='sim'`. `support='obs'` will return predictions only at the points `YindObs`.

```
## removing gpsep 0
```



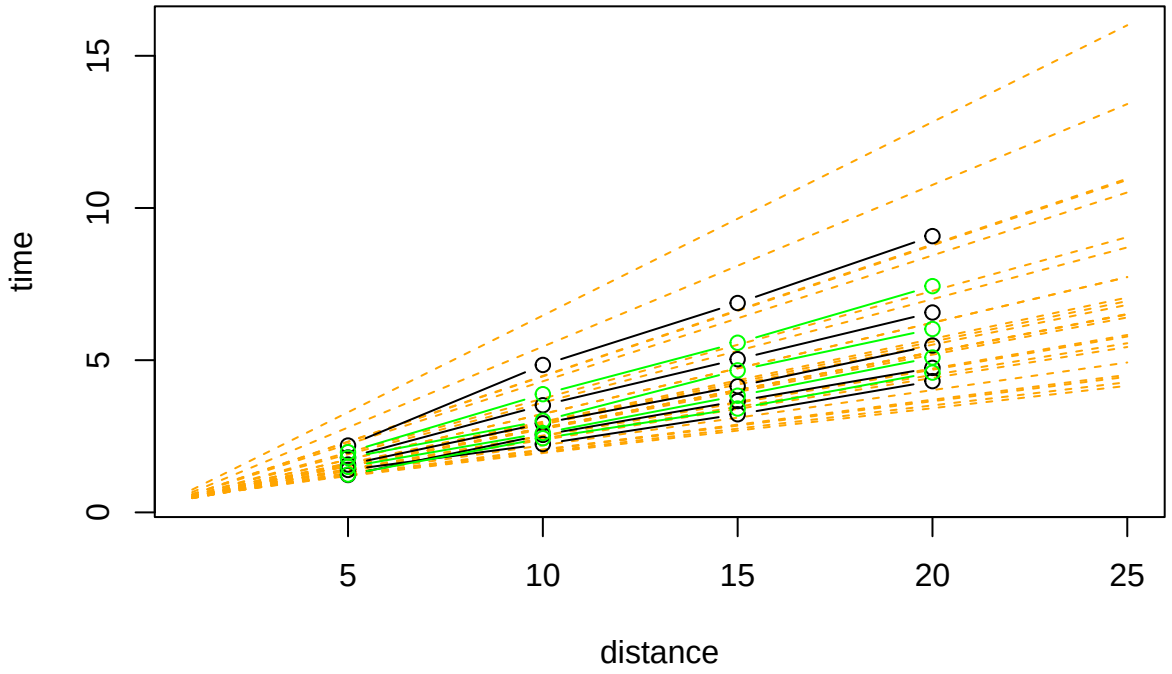
We use the function `ypred_mle()` to make predictions from our emulator at a point estimate of t^* use the function `ypred_mle()` with the `calib` object which contains the MLE of t^*. We find that our mcmc predictions produce a slightly wider confidence band, but the mean predictions are indistinguishable from each other.

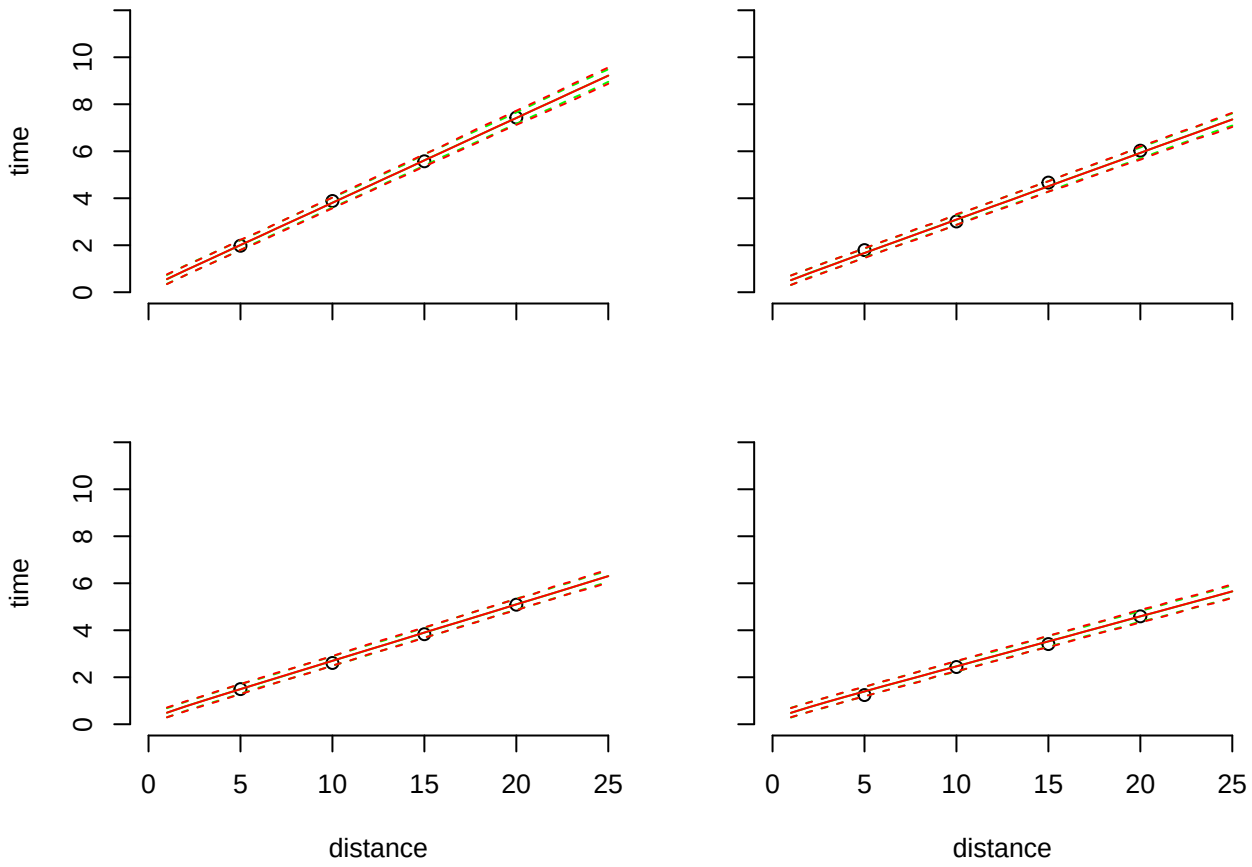
```
support = 'sim'
if(support=='obs'){
  predInd = YindObs[,1]
} else if(support=='sim'){
  predInd = YindSim[,1]
}
yPred = ypred_mle(Xobs[1:n,,drop=F],mvcData,calib,1000,bias=F,support=support)
obs_ids = mclapply(1:n, function(i) seq(i,dim(Ydata$obs$orig)[2],n))
obs_means = mclapply(1:n, function(i) apply(Ydata$obs$orig[,obs_ids[[i]]],drop=F],1,mean))
for(i in 1:n){
  matplot(YindObs[,1],Ydata$obs$orig[,obs_ids[[i]]],drop=F],pch=1,col=i+1,add=ifelse(i==1,F,T),xlim=c(0,1
          xlab='distance',ylab='time',main='Point Estimate Predictions')
  #points(YindObs[,1],Ytrue[,i],col='black')
  lines(predInd,yPred$yMean[,i],col=i+1)
  lines(predInd,yPred$yInt[[i]][,1],type='l',col=i+1,lty=2)
  lines(predInd,yPred$yInt[[i]][,2],type='l',col=i+1,lty=2)
}
```



Prediction at new location that was not included for calibration

We can also compare these two methods for inputs that were not used in calibration. Below we plot four new experiments in green. In the second plot, predictions using mcmc are shown in red and point estimate predictions in green. Again there is almost no difference between them for this simple problem.




time
0
2
4
6
8
10
0
5
10
15
20
25
distance