Comprehensive uncertainty quantification with BayesianTools

David Cameron

2019-09-11

Introduction

This document is the basis of a tutorial on "Comprehensive uncertainty quantification with BayesianTools". The aims are as follows:

1. To consider what can go wrong in Bayesian Calibration when we don't consider all the uncertainties present in the problem.
2. Provide a 'hands-on' opportunity for you calibrate a simple process-based vegetation model using the Bayesian methodology already introduced in the course.
3. To introduce you to a useful R package for Bayesian Calibration - "BayesianTools".

A Very Simple Ecosystem Model (VSEM)

The model that we are going to calibrate in this tutorial is already provided as part of the BayesianTools package.

The Very Simple Ecosystem Model (VSEM) is a 'toy' model designed to be very simple but yet bear some resemblance to deterministic processed based ecosystem models (PBMs) that are commonly used in terrestrial vegetation modelling.

The model determines the accumulation of carbon in the plant and soil from the growth of the plant via photosynthesis and senescence to the soil which respires carbon back to the atmosphere.

The model calculates Gross Primary Productivity (GPP) using a very simple light-use efficiency (LUE) formulation multiplied by light interception. Light interception is calculated via Beer's law with a constant light extinction coefficient operating on Leaf Area Index (LAI).

A parameter (GAMMA) determines the fraction of GPP that is autotrophic respiration. The Net Primary Productivity (NPP) is then allocated to above and below-ground vegetation via a fixed allocation fraction. Carbon is lost from the plant pools to a single soil pool via fixed turnover rates. Heterotropic respiration in the soil is determined via a soil turnover rate.

The model time-step is daily.

The model equations are:

Photosynthesis

$$LAI = LAR \times Cv \tag{1}$$

$$GPP = PAR \times LUE \times (1 - \exp^{(-KEXT \times LAI)}) \tag{2}$$

$$NPP = (1 - GAMMA) \times GPP \tag{3}$$

Carbon pool state equations

$$\frac{dC_v}{dt} = A_v \times NPP - \frac{C_v}{\tau_v} \quad (4)$$

$$\frac{dC_r}{dt} = (1.0 - A_v) \times NPP - \frac{C_r}{\tau_r} \quad (5)$$

$$\frac{dC_s}{dt} = \frac{C_r}{\tau_r} + \frac{C_v}{\tau_v} - \frac{C_s}{\tau_s} \quad (6)$$

- C_v, C_r and C_s : Carbon in vegetation, root and soil pools

VSEM inputs

PAR Photosynthetically active radiation (PAR) $MJm^{-2}day^{-1}$

VSEM parameters

The model has eight uncertain parameters.

KEXT Light extinction coefficient m^2 ground area / m^2 leaf area

LAR Leaf area ratio m^2 leaf area kg^{-1} aboveground vegetation

LUE Light-Use Efficiency ($kgCMJ^{-1}PAR$)

GAMMA Autotrophic respiration as a fraction of GPP

tauV Longevity of aboveground vegetation (days)

tauR Longevity of belowground vegetation (days)

tauS Residence time of soil organic matter (days)

Av Fraction of NPP allocated to aboveground vegetation. The remainder to allocated to the roots. (-)

VSEM states:

The initial states of the carbon pool are also uncertain and so can also be calibrated.

Cv Above-ground vegetation pool ($kgCm^{-2}$)

Cr Root pool ($kgCm^{-2}$)

Cs Carbon in soil organic matter ($kgCm^{-2}$)

VSEM fluxes:

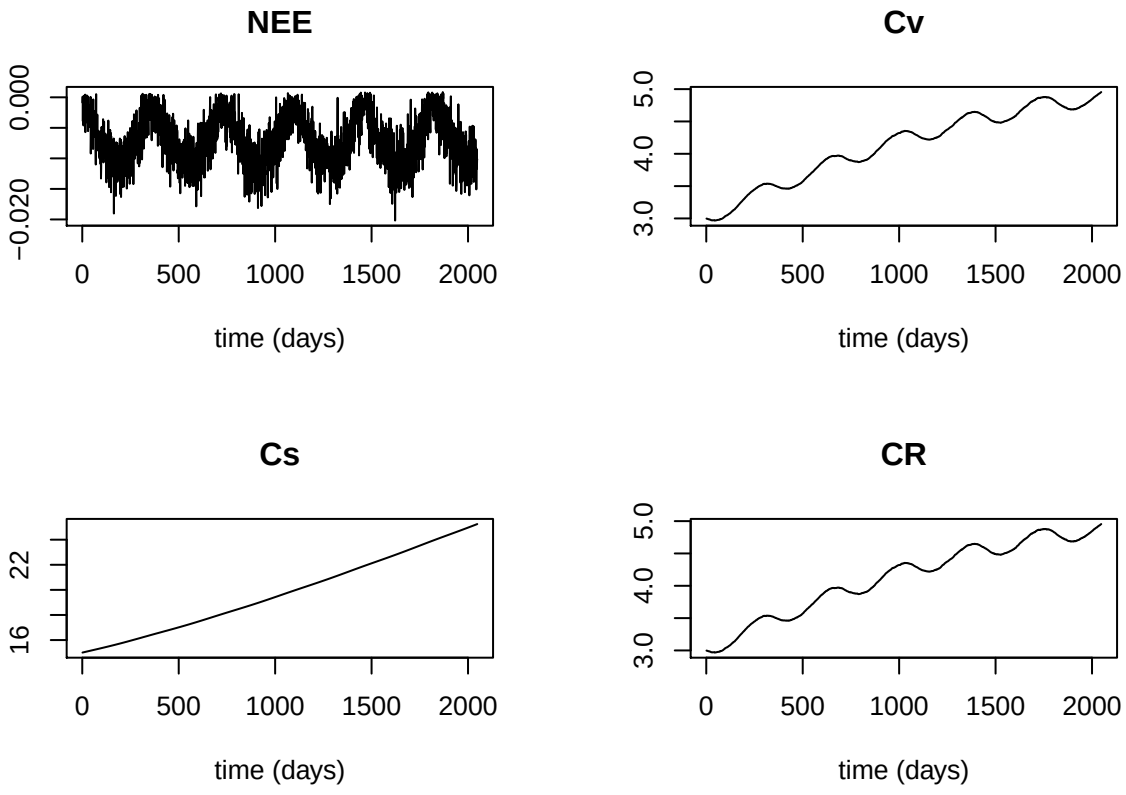
GPP Gross Primary Productivity $kgCm^{-2}day^{-1}$

NPP Net Primary Productivity $kgCm^{-2}day^{-1}$

NEE Net Ecosystem Exchange $kgCm^{-2}day^{-1}$ (C_s/tau_S - NPP)

Calling VSEM in R and plotting output

```
library(BayesianTools)
library(knitr)

set.seed(123)

refPars       <- VSEMgetDefaults()
ndays         = 2048
PAR           <- VSEMcreatePAR(1:ndays)
VSEMoutput    <- VSEM(refPars$best, PAR) # model predictions with reference parameters

par(mfrow = c(2,2))
for (i in 1:4){
 plot(VSEMoutput[,i], main = colnames(VSEMoutput)[i],type='l',ylab="",xlab="time (days)")
}
```



As you can see the VSEM model has been written so that it can accept parameters from R and returns output which we have plotted. This kind of functionality is crucial for calibrating a model.

Setting up the calibration in BayesianTools

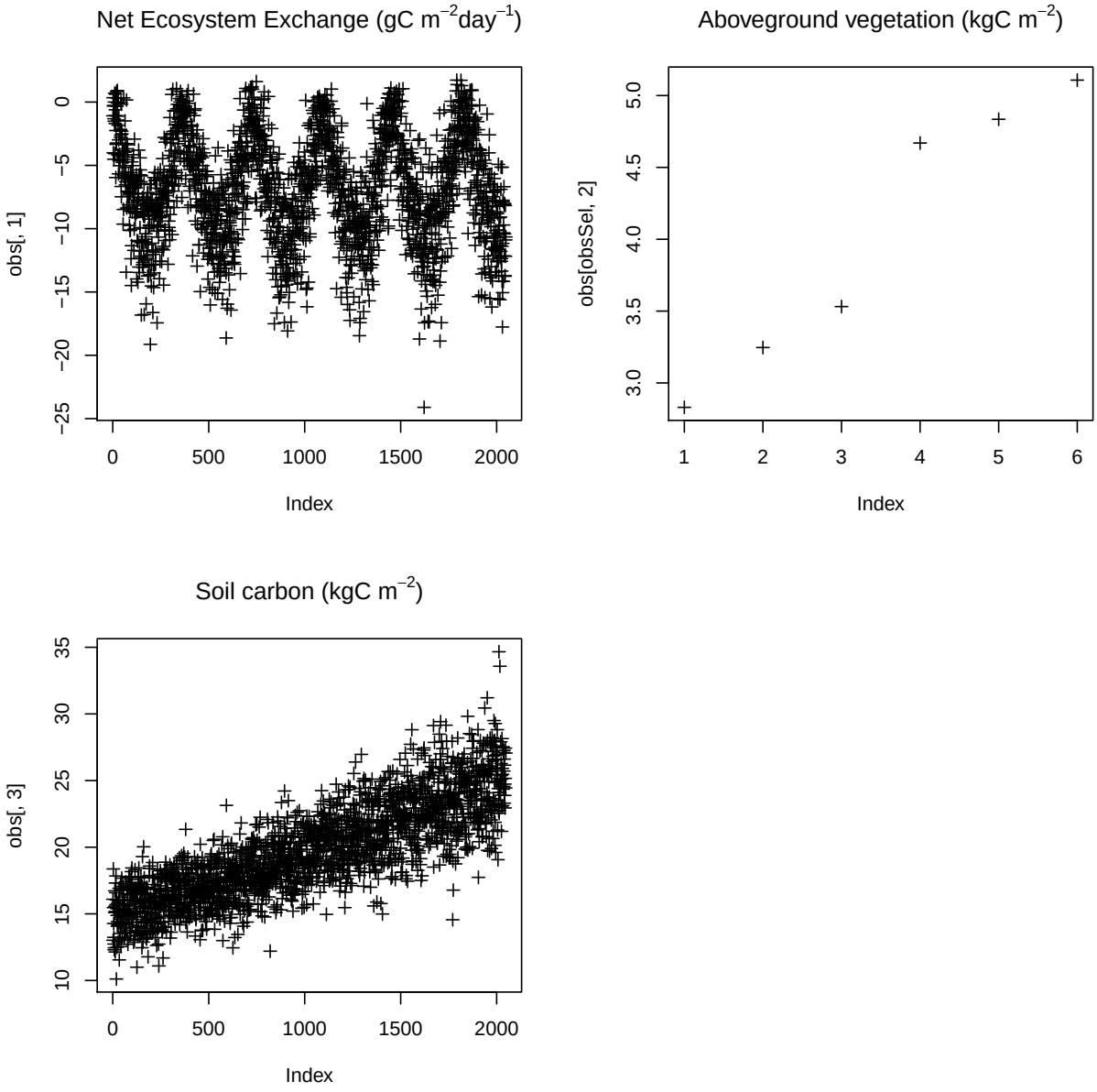
Before going on to make a calibration of VSEM with BayesianTools we need some calibration data and to specify our prior and likelihood for the problem.

```
## create some useful plot functions needed later

# Create a prediction function
createPredictions <- function(par){
  # set the parameters that are not calibrated on default values
  x = defaultPars
  x[parSel] = par
  predicted <- VSEM(x[1:11], PAR) # replace here VSEM with your model
  return(predicted[,plotfld]*plotfac )
}

# Create an error function
createError <- function(mean, par){
  return(rnorm(length(mean), mean = mean, sd = mean*par[7]))
}

createErrorNEE <- function(mean, par){
  return(rnorm(length(mean), mean = mean, sd = pmax(mean*par[7],0.5)))
}

createPlot <- function(TRUTH=TRUE){
    pred <- getPredictiveIntervals(parMatrix = parMatrix, model = createPredictions,
                                   numSamples = numSamples, quantiles = c(0.025, 0.5, 0.975),
                                   error = errMod)
    plotTimeSeries(observed = myObs[,plotfld],
                   predicted = pred$posteriorPredictivePredictionInterval[2,],
                   confidenceBand = pred$posteriorPredictiveCredibleInterval[c(1,3), ],
                   predictionBand = pred$posteriorPredictivePredictionInterval[c(1,3), ],
                   main=plotTitle)
    if (TRUTH) lines(referenceData[,plotfld],col=rgb(red=0,green=1,blue=0,alpha=0.5),lwd=1)

}

tracePlot<-function (sampler, thin = "auto", ...)
{
    codaChain = getSample(sampler, coda = T, thin = thin, ...)
    plot(codaChain,smooth=FALSE)
}
```

Calibration data

Load the calibration data into R.

```
load(file="calibrationData.RData")

oldpar <- par(mfrow = c(2,2))
plot(obs[ ,1], main = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")")),pch=3)
plot(obs[obsSel,2], main = expression(paste("Aboveground vegetation (kgC ",m^{-2},")")),pch=3)
plot(obs[ ,3], main = expression(paste("Soil carbon (kgC ",m^{-2},")")),pch=3)
```



An imbalance in the calibration data available for different variables in the system is very typical in ecology where some parts of the system such as carbon fluxes can be measured automatically whereas others such as stocks of carbon in the vegetation and soil are much more labour intensive.

Prior

Now we need to choose which parameters to calibrate and to specify a prior distribution for those parameters. BayesianTools makes this easy with helper functions for uniform, beta and truncated normal distributions. It is also possible to create a bespoke prior distribution using the function createPrior.

Here we will use a uniform prior distribution and assume no prior covariances between the model parameters.

Qu: Is this a good choice what do you think?

```r
# load reference parameter definition (upper, lower prior)
refPars <- VSEMgetDefaults()

# this adds one additional parameter for the likelihood coefficient of variance (see below)
refPars[12,] <- c(0.1, 0.001, 0.5)
rownames(refPars)[12] <- "error-cv"

## select which parameters to calibrate
parSel = c(1,3,5,6,9,10,12)
parRange <- rbind(refPars$lower[parSel],refPars$upper[parSel])

kable(refPars[parSel,])
```

	best	lower	upper
KEXT	0.500	2e-01	1e+00
LUE	0.002	5e-04	4e-03
tauV	1440.000	5e+02	3e+03
tauS	27370.000	4e+03	5e+04
Cv	3.000	0e+00	4e+02
Cs	15.000	0e+00	1e+03
error-cv	0.100	1e-03	5e-01

```r
defaultPars        <- refPars$best
names(defaultPars) <- row.names(refPars)

## optional, you can also directly provide lower, upper in the createBayesianSetup, see help
prior <- createUniformPrior(lower = refPars$lower[parSel],
                            upper = refPars$upper[parSel])
```

Likelihood function

A particular strength of BayesianTools, especially for calibrating models written in R or other languages that can be called from R, is that the likelihood function (containing a call to the model) is just a normal R function. This avoids having to recode the model as is required by many other Bayesian Calibration packages (eg WinBUGS, JAGS and Stan).

Here we have chosen a Gaussian distribution for the likelihood.

```r
likelihood <- function(par){
  # set parameters that are not calibrated on default values
  x = defaultPars
  x[parSel] = par

  predicted <- VSEM(x[-nvar], PAR) # replace here VSEM with your model
  predicted[,1] = 1000 * predicted[,1] # unit change kg -> g

  ## NEE
  diff <- c(predicted[,1] - obs[,1]) # difference betweeno observed and predicted
  llValues1 <- dnorm(diff, sd = pmax((abs(c(predicted[,1])) + 0.0000001) * x[nvar],0.5), log = TRUE)

  ## Aboveground carbon
  diff <- c(predicted[obsSel,2] - obs[obsSel,2]) # difference betweeno observed and predicted
```

```
  llValues2 <- dnorm(diff, sd = (abs(c(predicted[obsSel,2])) + 0.0000001) * x[nvar], log = TRUE)

  ## Soil carbon
  diff <- c(predicted[,3] - obs[,3]) # difference betweeno observed and predicted
  llValues3 <- dnorm(diff, sd = (abs(c(predicted[,3])) + 0.0000001) * x[nvar], log = TRUE)
  return(sum(llValues1,llValues2,llValues3))
}
```

Setup BayesianTools

Once the prior and likelihood have been specified these can be passed as a setup function to BayesianTools. We also need to create a settings list to specify how long the MCMC chain should run for and how many chains to run with.

```
bayesianSetup <- createBayesianSetup(likelihood, prior, names = rownames(refPars)[parSel])

settings <- list(iterations = 1000, nrChains = 2)
```

A first calibration with BayesianTools

```
defaultPars["Av"] <- 1.0
defaultPars["Cr"] <- 0.0
```

Run the MCMC

The last choice is to choose which sampler to run with. Whilst it is possible to choose the simple Metropolis sampler that we have already come across, for most cases more efficient MCMC samplers will provide a significant speedup without losing accuracy. Of the samplers available in BayesianTools the one that is recommended for most calibrations is DEzs.

```
out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)
```

Check for convergence

We can check whether the MCMC has converged on the posterior by visual inspection of the trace plots for each calibrated parameter and also by calculating the Gelman-Rubin statistic.

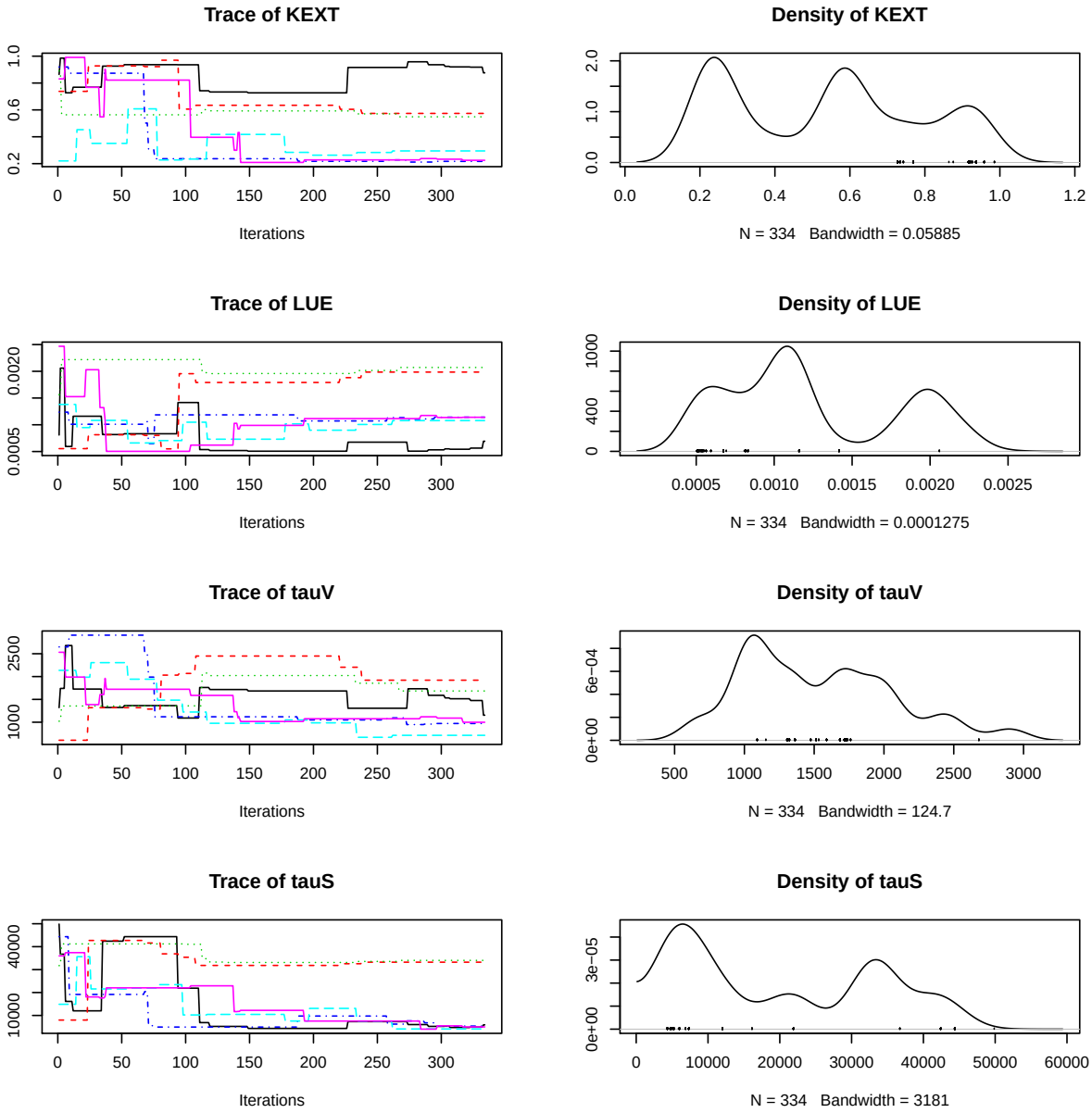
Gelman-Rubin

For this statistic we need multiple chains each with a different starting point.

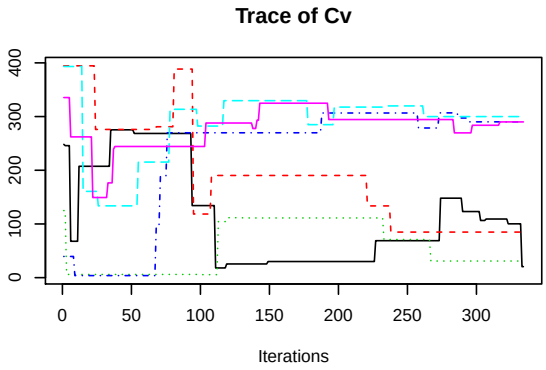
Gelman-Rubin compares variance within chains and between chains. If it is close to one, this suggests that the variance is about equal, suggesting that the MCMC has converged on the posterior. A Gelman-Rubin value much greater than one suggests that variance between chains is greater than within chains and that the MCMC has not converged.

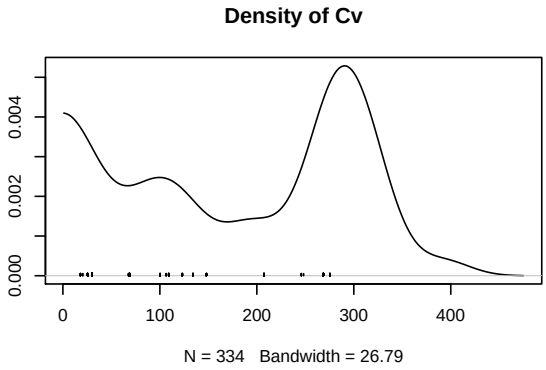
Qu: Has the calibration converged...?

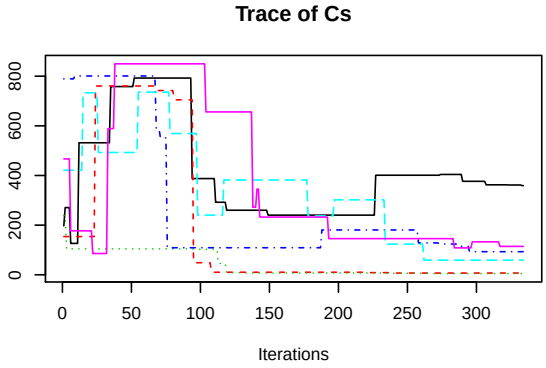
```
tracePlot(out,whichParameters=c(1:4))
```

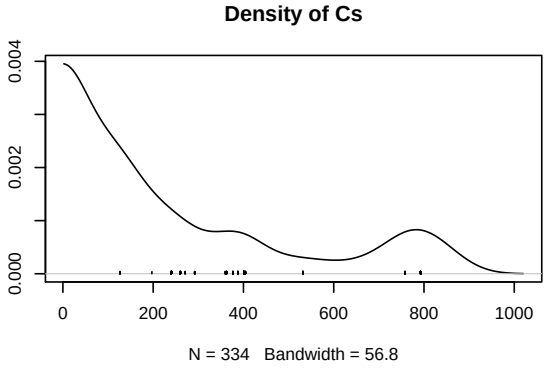


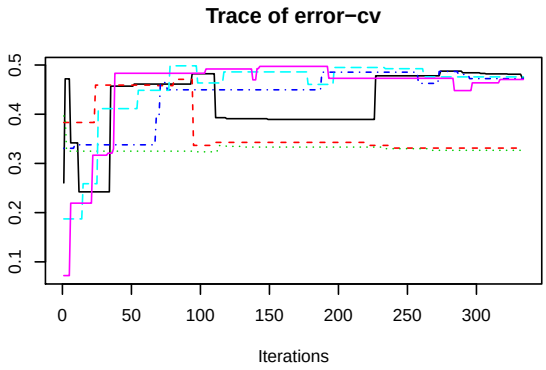

```
tracePlot(out,whichParameters=c(5:7))
```

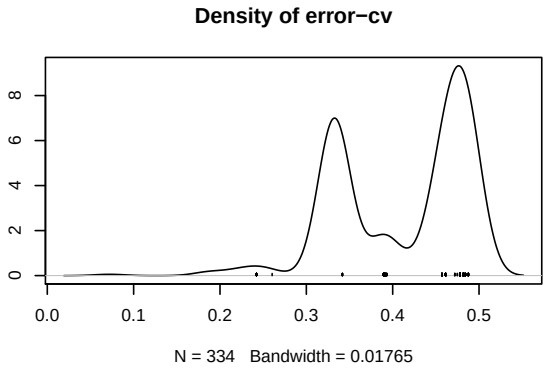













```
gelmanDiagnostics(out) # should be below 1.05 for all parameters to demonstrate convergence
```

```
## Potential scale reduction factors:
##
##          Point est. Upper C.I.
## KEXT           7.33      24.44
## LUE           10.07      17.51
## tauV           4.35       8.07
## tauS           8.43      17.70
## Cv             4.78       9.06
## Cs             2.73       6.28
## error-cv       4.54      15.41
##
## Multivariate psrf
```

```
##
## 79.4
```

A calibration with a longer chain

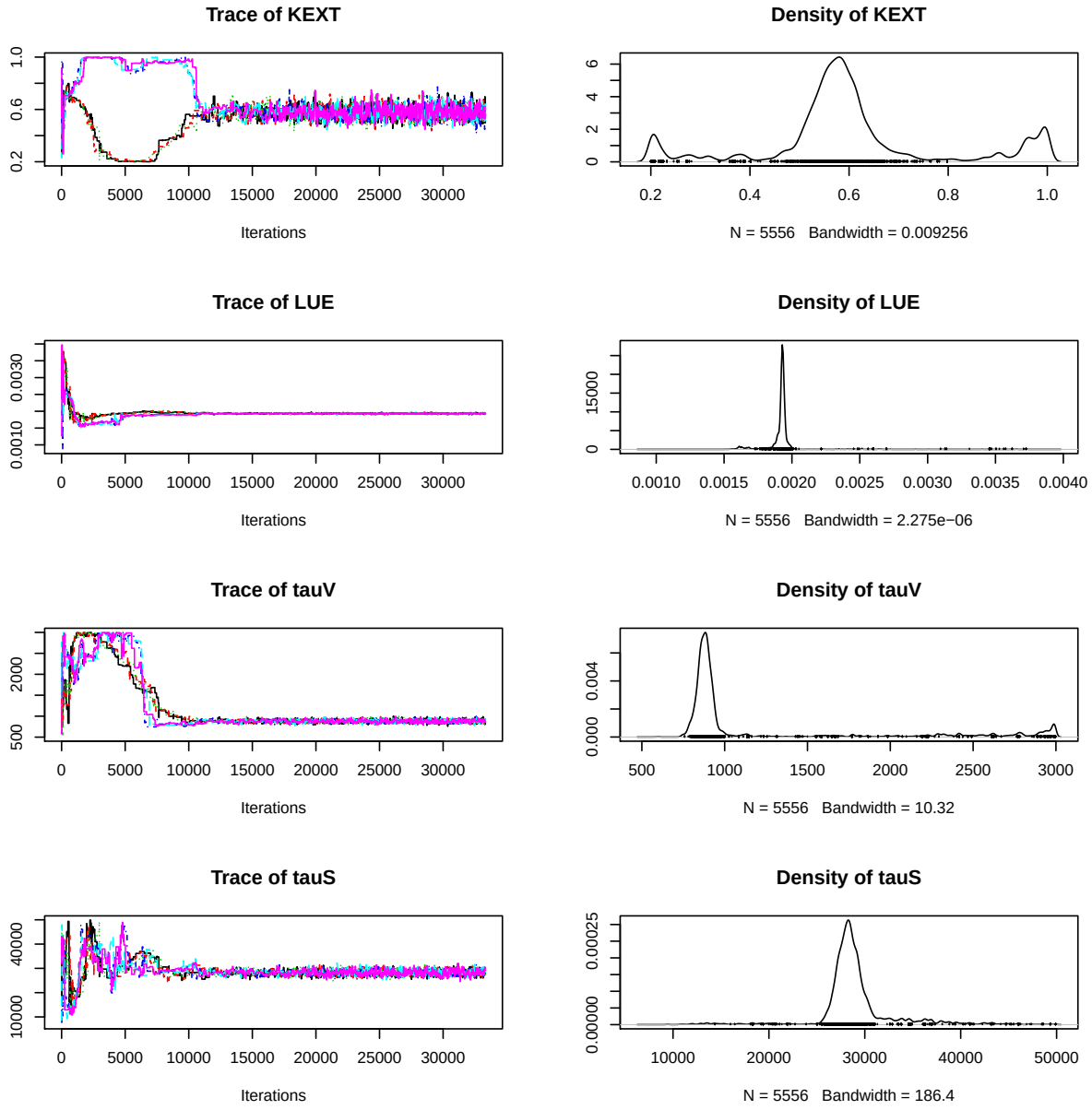
Since the calibration with just 1000 iterations was too short we'll try another calibration now with a much longer chain. This takes longer than we'll want to wait just now so we'll load in a calibration that was made earlier...

```
settings <- list(iterations = 100000, nrChains = 2)
#out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)
#save(out,file="eMod-pObs.rda")

load(file="eMod-pObs.rda")
```

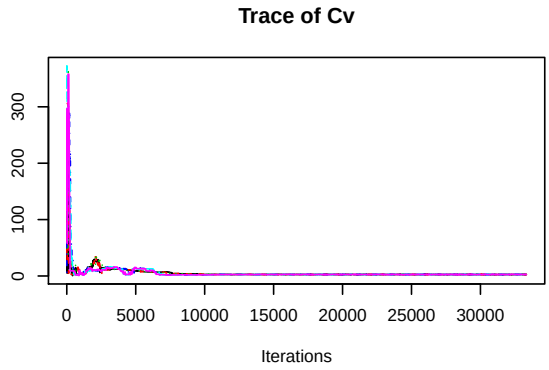
Again check for convergence

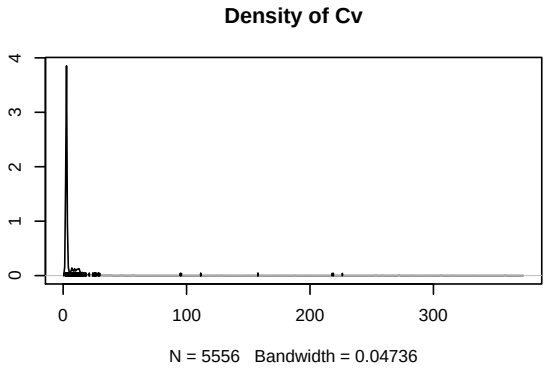
Qu: Has the calibration converged now?

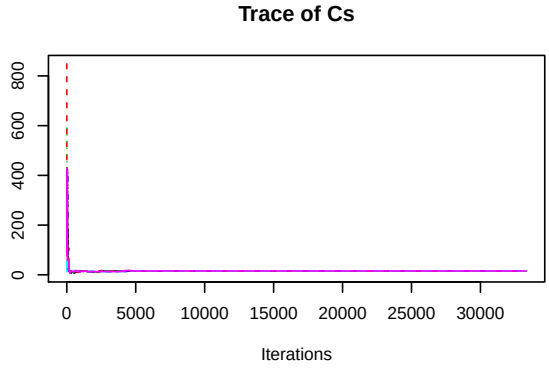
```
tracePlot(out,whichParameters=c(1:4))
```

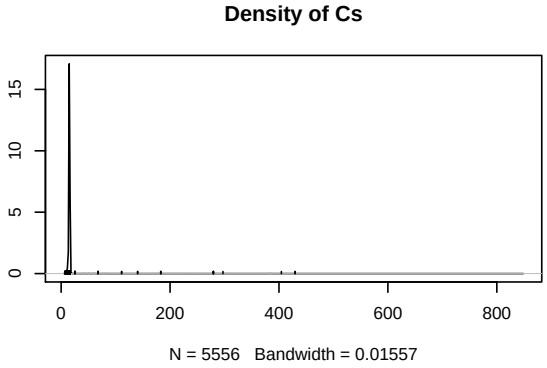


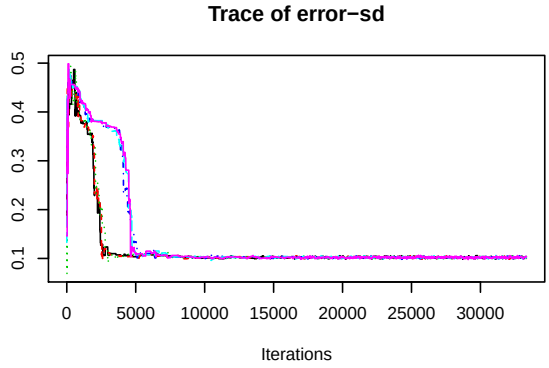

```
tracePlot(out,whichParameters=c(5:7))
```

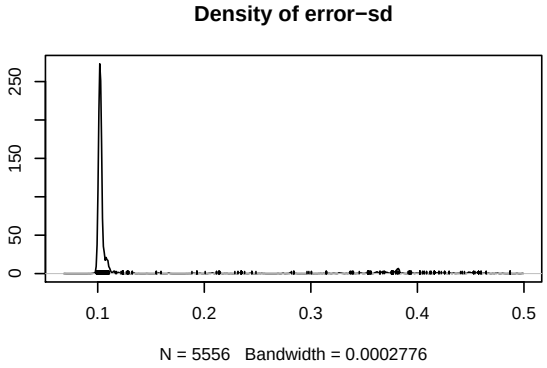













```
gelmanDiagnostics(out, start = 20000) # should be below 1.05 for all parameters to demonstrate converge
```

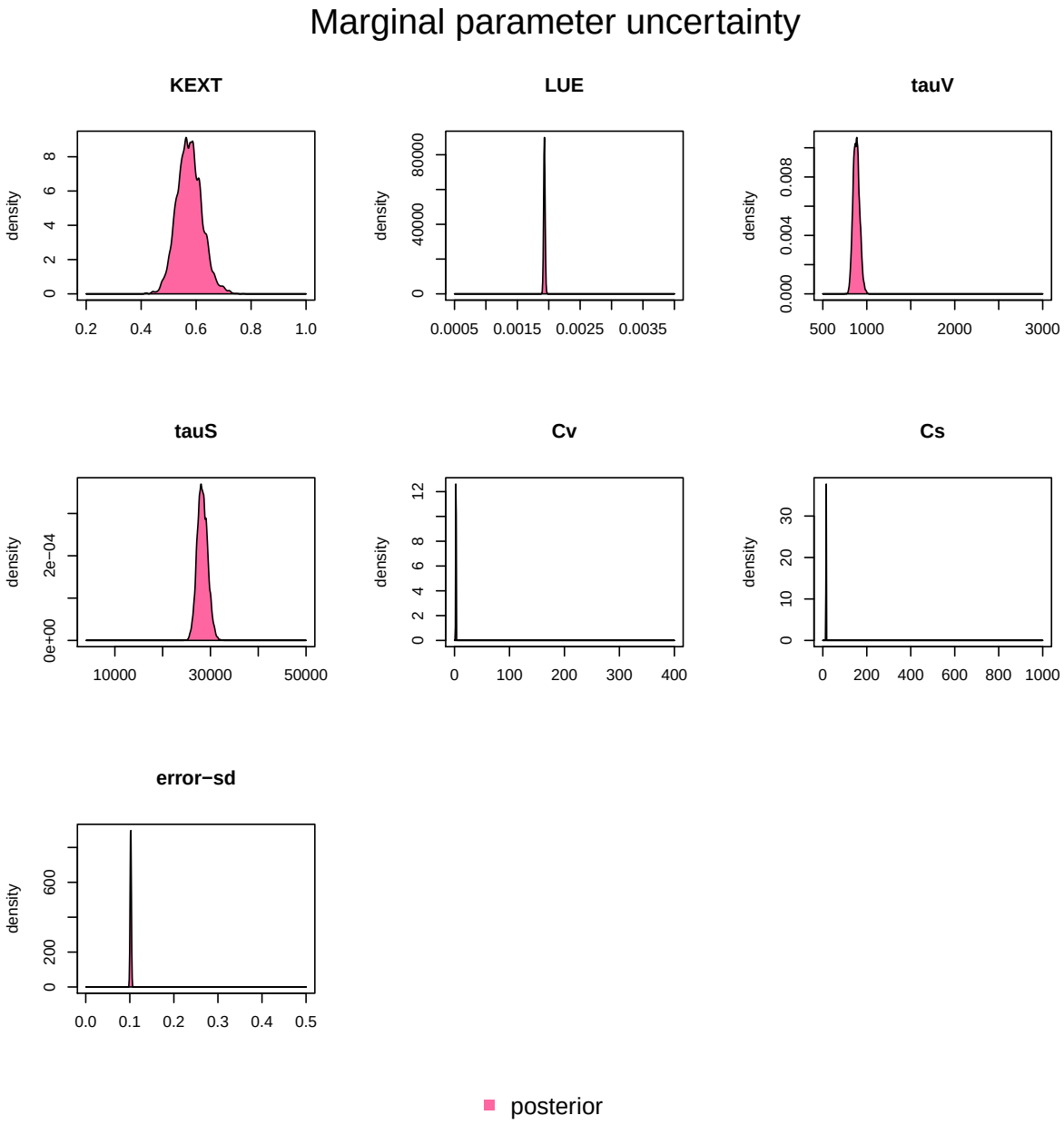
```
## Potential scale reduction factors:
##
##          Point est. Upper C.I.
## KEXT           1.00       1.01
## LUE            1.00       1.01
## tauV           1.01       1.01
## tauS           1.01       1.02
## Cv             1.01       1.01
## Cs             1.00       1.00
## error-sd       1.00       1.01
##
## Multivariate psrf
```

```
##
## 1.01
```

Parameter plots

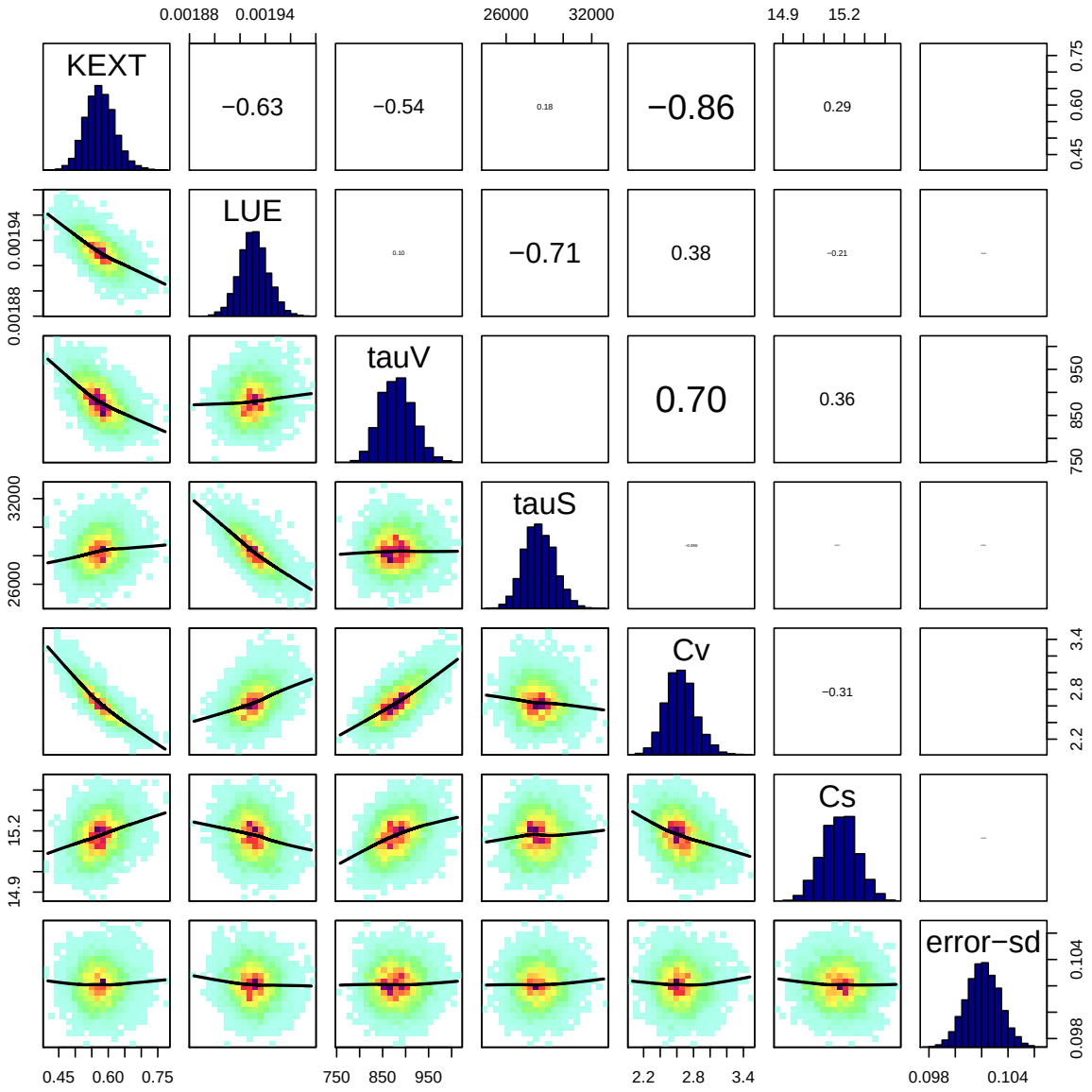
We can view plots of the posterior marginal distributions of the parameters after the calibration. The range on the x-axis on the plot shows the range of the original uniform prior. The plots show that parameter uncertainty has reduced significantly, from prior to posterior, as a result of the calibration.

```
marginalPlot(out, start=20000, singlePanel=FALSE,xrange=parRange)
```



It is also often useful to see what posterior correlations we have found between parameters. These can be viewed easily in BayesianTools.

```
correlationPlot(out, start=20000)
```



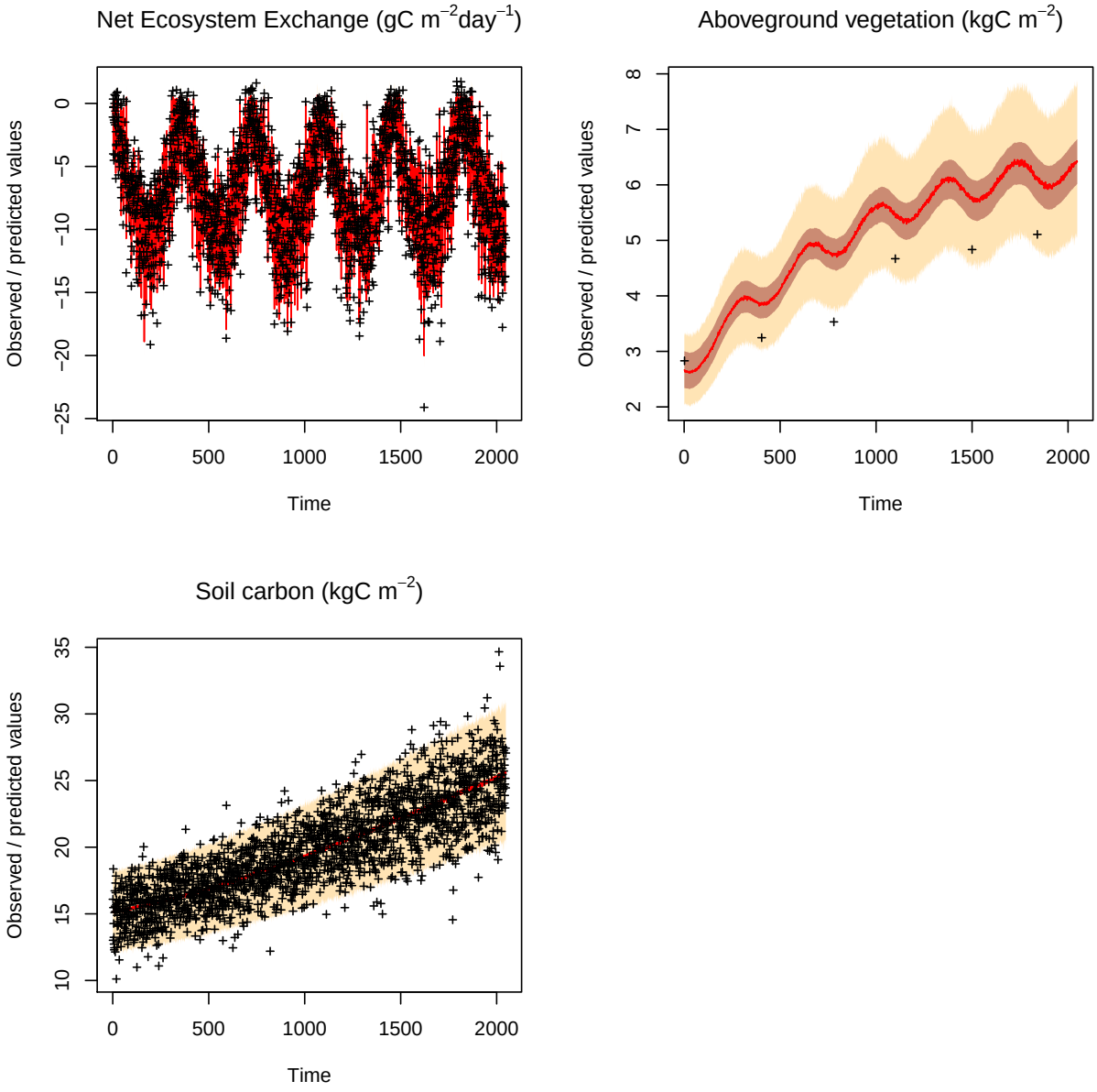
Sample from the posterior and plot posterior model predictions

To see what the model is predicting after calibration we sample the posterior. In practise we do this by taking a random sample from the MCMC chain after the burnin. A sample size of 1000 is normally sufficient. The model is then rerun with the sample and plots created to show the model predictions with uncertainty.

Shown in the plots are the three model outputs for which we have calibration data. The red line marks the mode of the posterior. The dark shading marks the 2.5 and 97.5 quantiles of the posterior. This is sometimes known as the credible interval. For the light shading the observational error model has been added on. This is sometimes known as the predictive interval as this is what the combined model and observational error model would predict for any new observations.

Qu: What do you think about this calibration. Is it a good one?

```
parMatrix  = getSample(out, start=20000,numSamples = 6000)
numSamples = min(1000, nrow(parMatrix))

par(mfrow=c(2,2))

plotfld    = 1
plotfac    = 1000.0
plotTitle  = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")"))
errMod     = createErrorNEE
myObs      = obs
createPlot(TRUTH=FALSE)

plotfld    = 2
plotfac    = 1
plotTitle  = expression(paste("Aboveground vegetation (kgC ",m^{-2},")"))
errMod     = createError
myObs[-obsSel,] <- NA
createPlot(TRUTH=FALSE)

plotfld    = 3
plotTitle  = expression(paste("Soil carbon (kgC ",m^{-2},")"))
errMod     = createError
myObs      = obs
createPlot(TRUTH=FALSE)
```



Diagnosing why the calibration is poor with imbalanced data.

Using virtual data

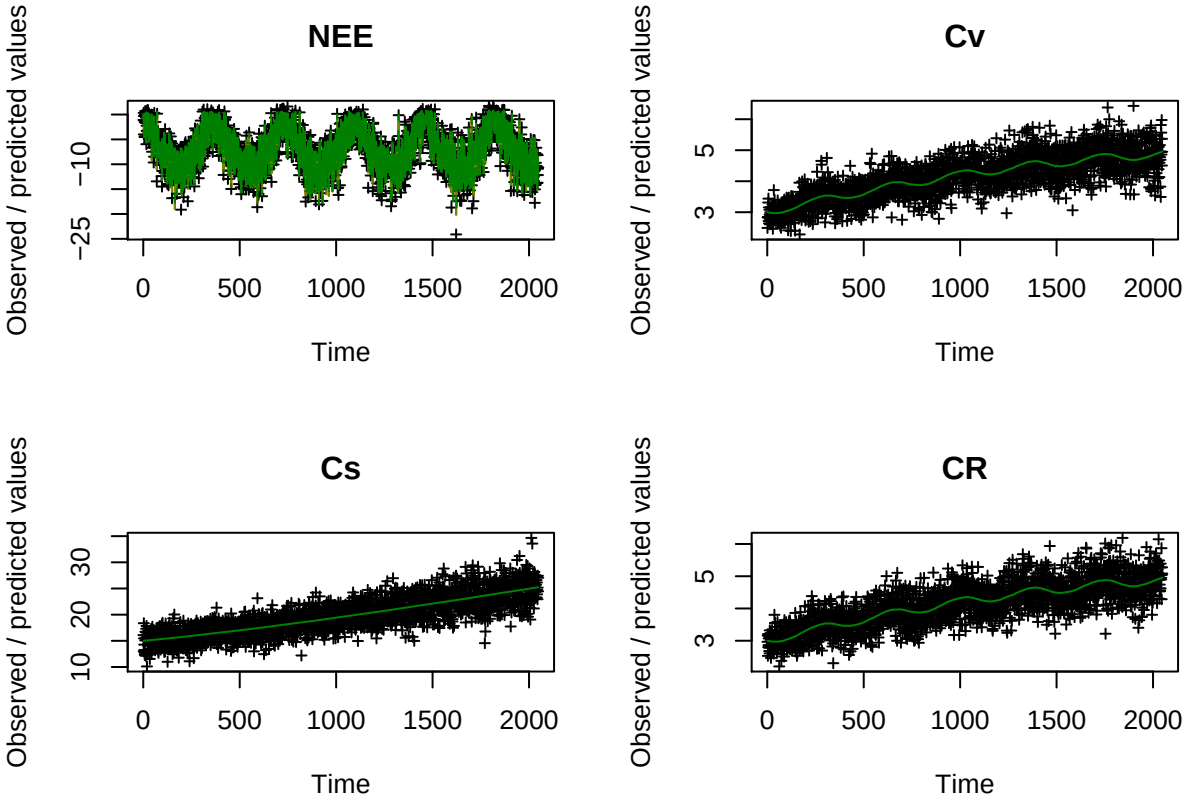
For the purposes of this tutorial we want to be able to control for model error. Therefore, we created 'virtual' data derived from the output of the model so that we can perform the calibration with a perfect model or one with a known error.

We generated the data by running the model with a known parameter vector ("refPars$best"). To the resulting output from VSEM we added noise from a normal distribution to simulate 'stochastic' error that would be present in observed data.

The plot shows the virtual data that we have generated. The model output (or "truth" line in this context)

is shown in green.

```
oldpar <- par(mfrow = c(2,2))
for (i in 1:4) {
               plotTimeSeries(observed = obs[,i],
                              predicted = referenceData[,i], main = colnames(referenceData)[i])
               lines(referenceData[,i],col=rgb(red=0,green=1,blue=0,alpha=0.5),lwd=1)
               }
```



Model error

The calibration that we made before was with a model with a known error

```
defaultPars["Av"] <- 1.0
defaultPars["Cr"] <- 0.0
```

In essence we took away the root pool so that the model was missing an important process.

If we reset the parameters to their original values, then we will have what we never have in the real world, a perfect model. That is to say the model that generated the data.

```
defaultPars["Av"] <- 0.5
defaultPars["Cr"] <- 3.0
```

Perfect model and six aboveground vegetation observations

What if we had a perfect model and retain the same imbalance in data?

Run the calibration

Rerun the calibration now with a perfect model.

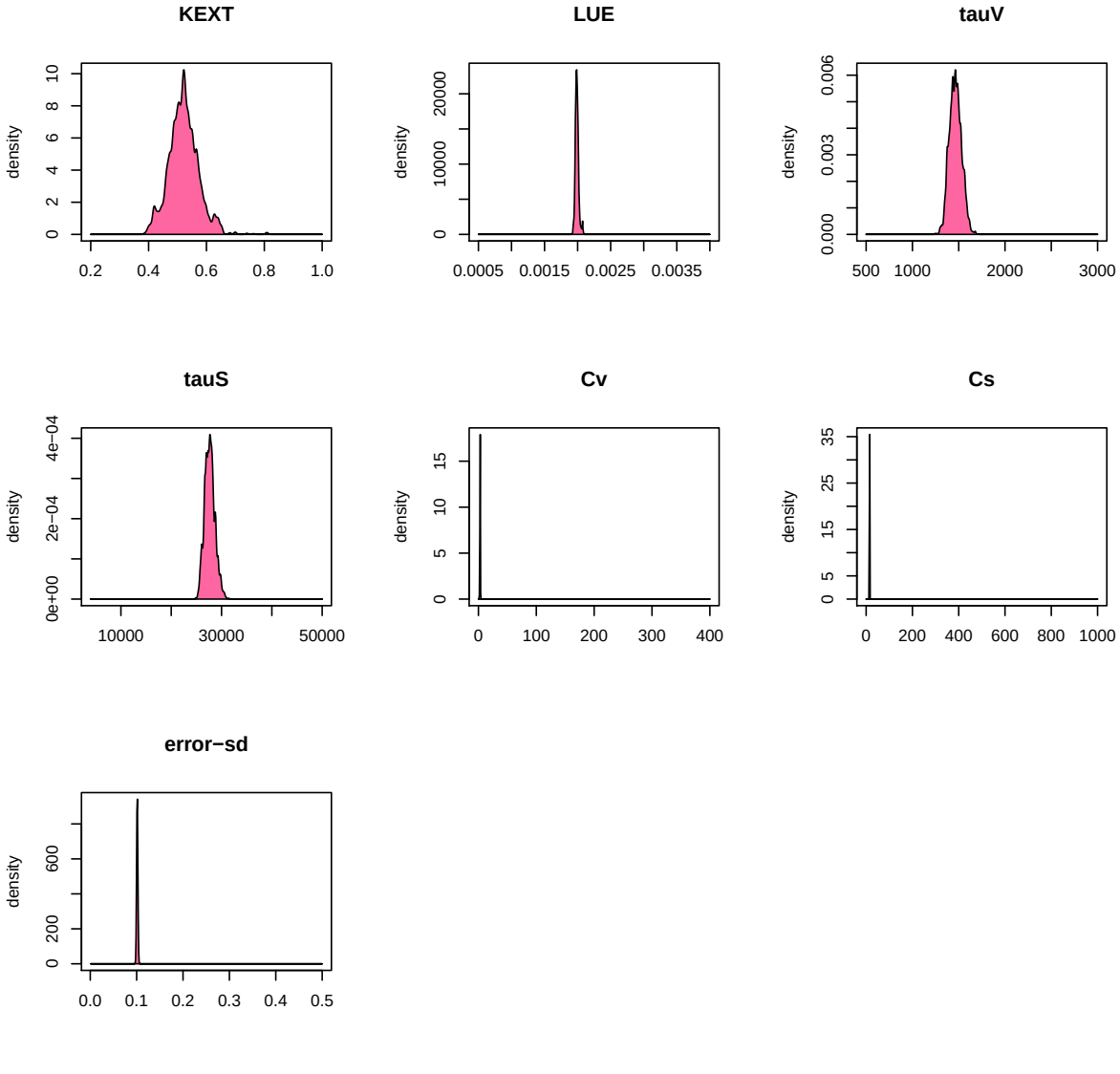
```
obsSel <- c(1,202,390,550,750,920)*2.0
#out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)
#save(out,file="pMod-pObs.rda")

load(file="pMod-pObs.rda")
```

Parameter plots

Since we know the 'true' values of the parameters we can compare these with what we see in the posterior.

Qu: How well do the posterior modes match the original parameter values used to generate the data in the table refPars$best

```
marginalPlot(out, start=20000, singlePanel=FALSE, xrange=parRange)
```



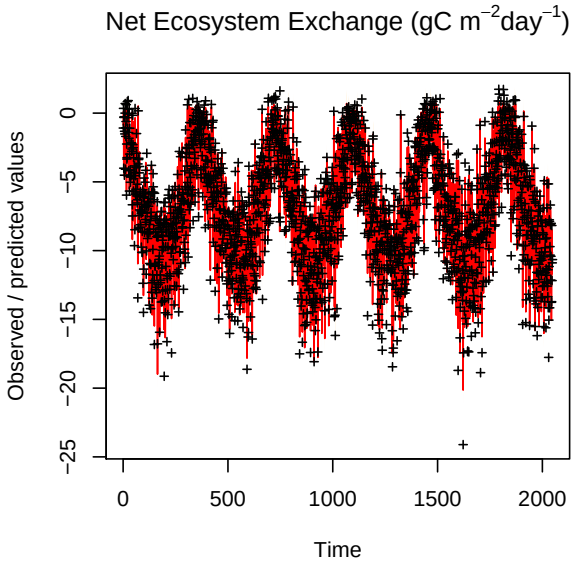

```
kable(refPars[parSel,])
```

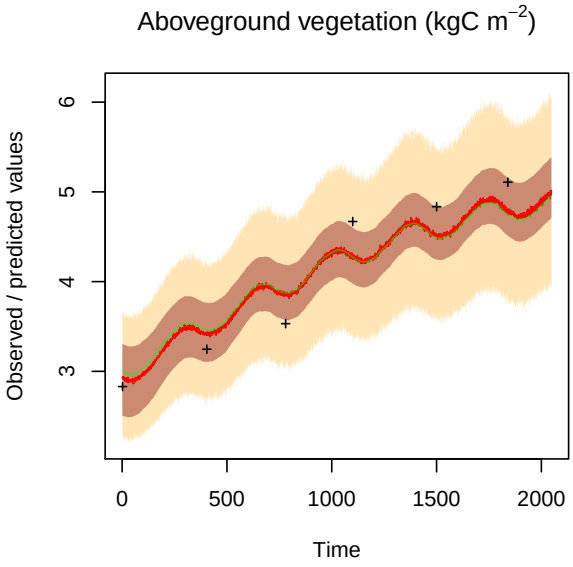
	best	lower	upper
KEXT	0.500	2e-01	1e+00
LUE	0.002	5e-04	4e-03
tauV	1440.000	5e+02	3e+03
tauS	27370.000	4e+03	5e+04
Cv	3.000	0e+00	4e+02
Cs	15.000	0e+00	1e+03
error-cv	0.100	1e-03	5e-01

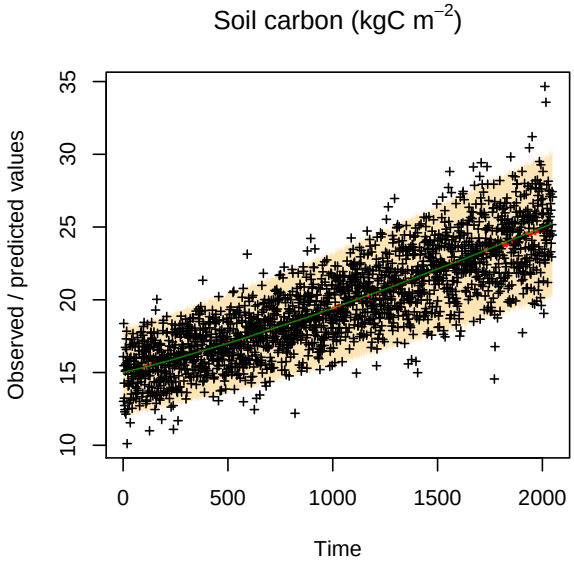
Model predictions

Qus: Is this a good calibration? Does it seem to be important that there is an imbalance in the data?

```
parMatrix  = getSample(out, start=20000,numSamples = 6000)
numSamples = min(1000, nrow(parMatrix))

par(mfrow=c(2,2))

plotfld    = 1
plotfac    = 1000.0
plotTitle  = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")"))
errMod     = createErrorNEE
myObs      = obs
createPlot(TRUTH=FALSE)

plotfld    = 2
plotfac    = 1
plotTitle  = expression(paste("Aboveground vegetation (kgC ",m^{-2},")"))
errMod     = createError
myObs[-obsSel,] <- NA
createPlot()

plotfld    = 3
plotTitle  = expression(paste("Soil carbon (kgC ",m^{-2},")"))
errMod     = createError
myObs      = obs
createPlot()
```







Model with error and all observations

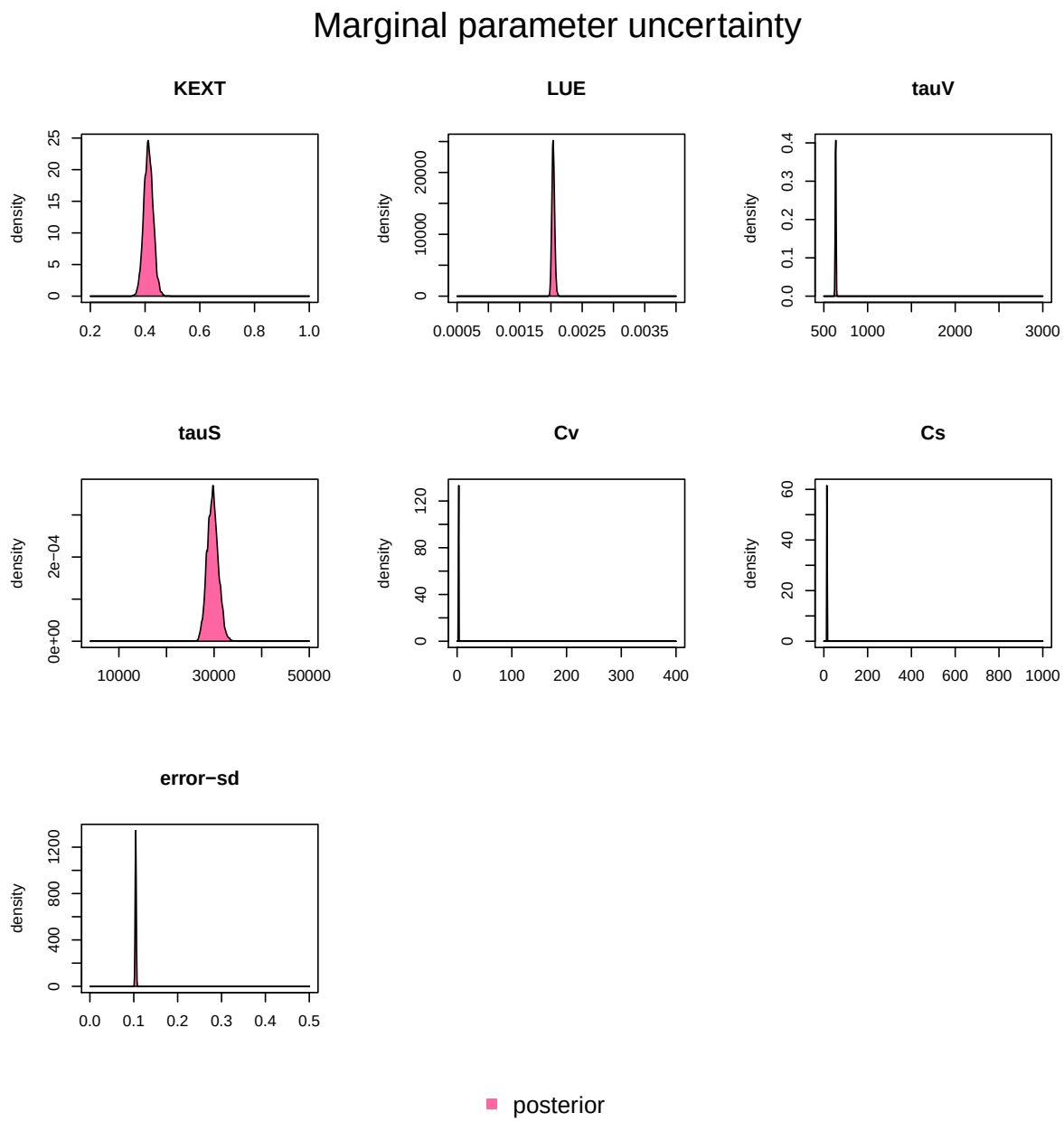
What happens if bring back the model error that we had before but now have balanced numbers of observations?

```
obsSel <- 1:ndays
defaultPars["Av"] <- 1.0
defaultPars["Cr"] <- 0.0

settings <- list(iterations = 100000, nrChains = 2)
#out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)
#save(out,file="eMod-aObs.rda")

load(file="eMod-aObs.rda")
```

Parameter plots

Qu: How well do the posterior modes match the original parameter values used to generate the data in the table refPars$best

```
marginalPlot(out, start=20000, singlePanel=FALSE, xrange = parRange)
```




```
kable(refPars[parSel,])
```

	best	lower	upper
KEXT	0.500	2e-01	1e+00
LUE	0.002	5e-04	4e-03
tauV	1440.000	5e+02	3e+03
tauS	27370.000	4e+03	5e+04

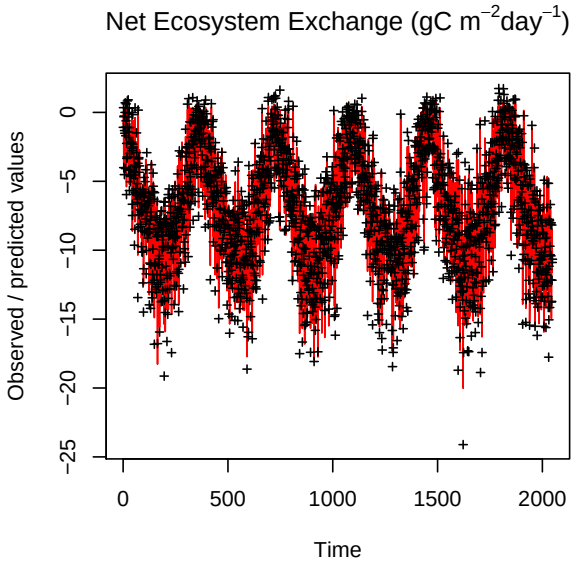
	best	lower	upper
Cv	3.000	0e+00	4e+02
Cs	15.000	0e+00	1e+03
error-cv	0.100	1e-03	5e-01

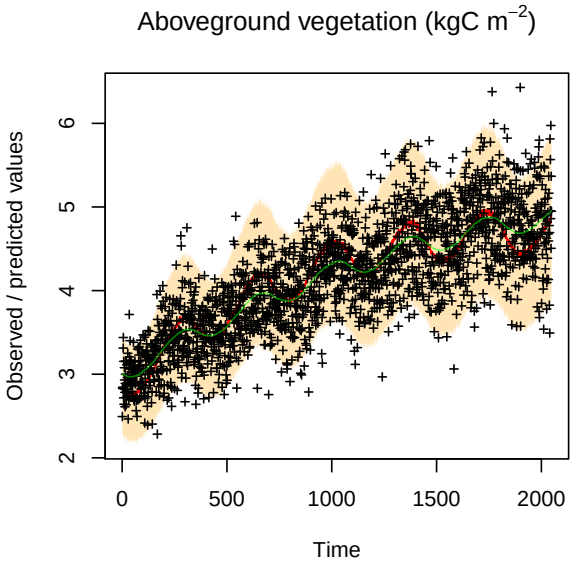
Model predictions

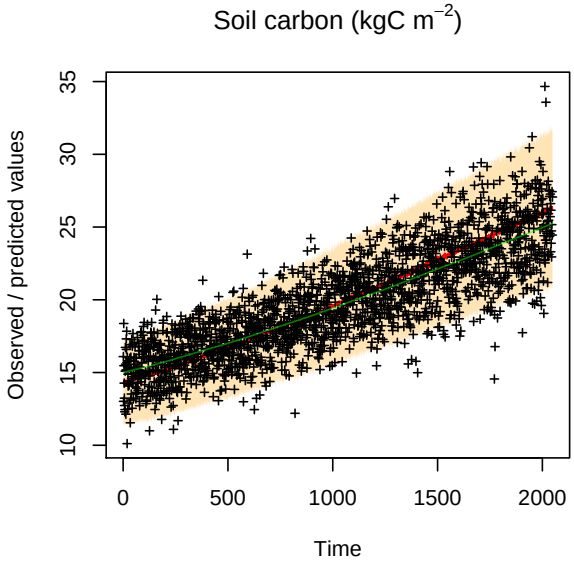
Qus:

1) **Why is the calibrated model closer to the data than for the calibration with unbalanced data?**
2) **The modes of some of the parameters departed quite strongly from the 'true' values what role might this have played in the model predictions**

```
parMatrix  = getSample(out, start=20000,numSamples = 6000)
numSamples = min(1000, nrow(parMatrix))

par(mfrow=c(2,2))

plotfld    = 1
plotfac    = 1000.0
plotTitle  = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")"))
errMod     = createErrorNEE
myObs      = obs
createPlot(TRUTH=FALSE)

plotfld    = 2
plotfac    = 1
plotTitle  = expression(paste("Aboveground vegetation (kgC ",m^{-2},")"))
errMod     = createError
myObs[-obsSel,] <- NA
createPlot()

plotfld    = 3
plotTitle  = expression(paste("Soil carbon (kgC ",m^{-2},")"))
errMod     = createError
myObs      = obs
createPlot()
```







Model with error and balanced observations

This time we'll use just six observations for all of NEE, Cv and Cs. This ad hoc approach is what is often done when there are issues in the calibration with unbalanced data.

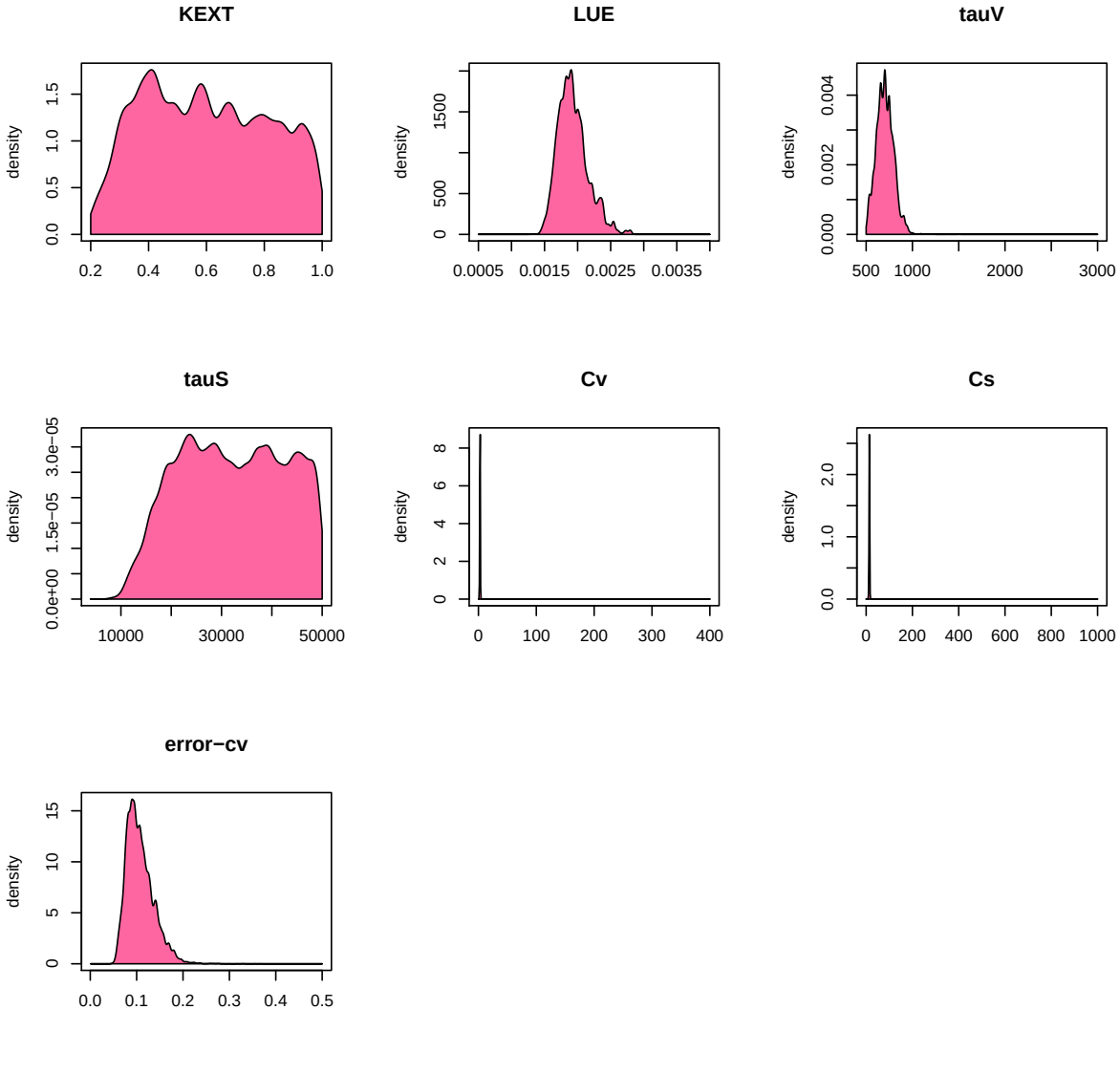
```
obsSel <- c(1,202,390,550,750,920)*2.0

likelihood <- function(par){
  # set parameters that are not calibrated on default values
  x = defaultPars
  x[parSel] = par

  predicted <- VSEM(x[-nvar], PAR) # replace here VSEM with your model
```

```
  predicted[,1] = 1000 * predicted[,1] # unit change kg -> g

  ## NEE
  diff <- c(predicted[obsSel,1] - obs[obsSel,1]) # difference betweeno observed and predicted
  llValues1 <- dnorm(diff, sd = pmax((abs(c(predicted[obsSel,1])) + 0.0000001) * x[nvar],0.5), log = TRU

  ## Aboveground carbon
  diff <- c(predicted[obsSel,2] - obs[obsSel,2]) # difference betweeno observed and predicted
  llValues2 <- dnorm(diff, sd = (abs(c(predicted[obsSel,2])) + 0.0000001) * x[nvar], log = TRUE)

  ## Soil carbon
  diff <- c(predicted[obsSel,3] - obs[obsSel,3]) # difference betweeno observed and predicted
  llValues3 <- dnorm(diff, sd = (abs(c(predicted[obsSel,3])) + 0.0000001) * x[nvar], log = TRUE)
  return(sum(llValues1,llValues2,llValues3))
}

#settings <- list(iterations = 100000, nrChains = 2)
#out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)
#save(out,file="eMod-bObs.rda")
```

Parameter plots

Qu: What has increased here? Also are we close to the 'true' parameter values?

```
load(file="eMod-bObs.rda")

marginalPlot(out, start=20000, singlePanel=FALSE, xrange=parRange)
```




```
kable(refPars[parSel,])
```

	best	lower	upper
KEXT	0.500	2e-01	1e+00
LUE	0.002	5e-04	4e-03
tauV	1440.000	5e+02	3e+03
tauS	27370.000	4e+03	5e+04
Cv	3.000	0e+00	4e+02
Cs	15.000	0e+00	1e+03
error-cv	0.100	1e-03	5e-01

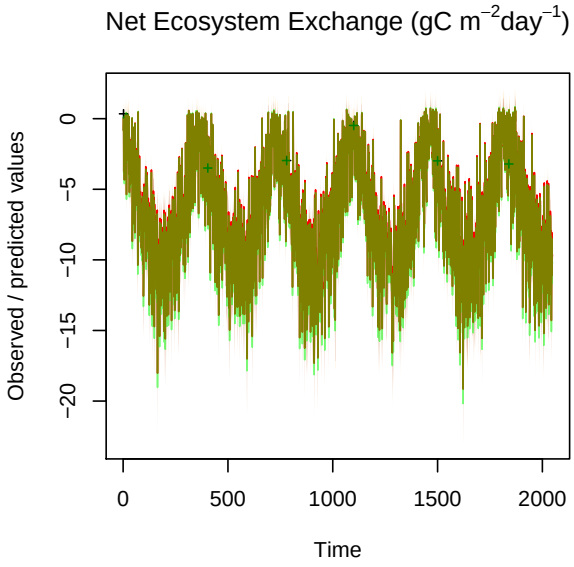
Model predictions

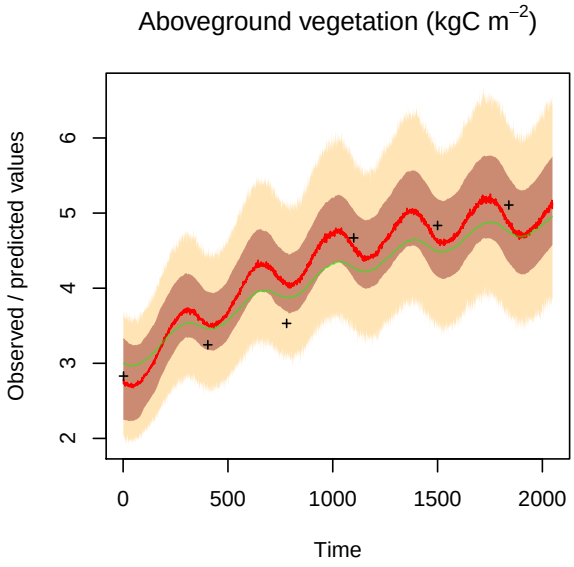
Qu: Has this solved the problem with the unbalanced data?

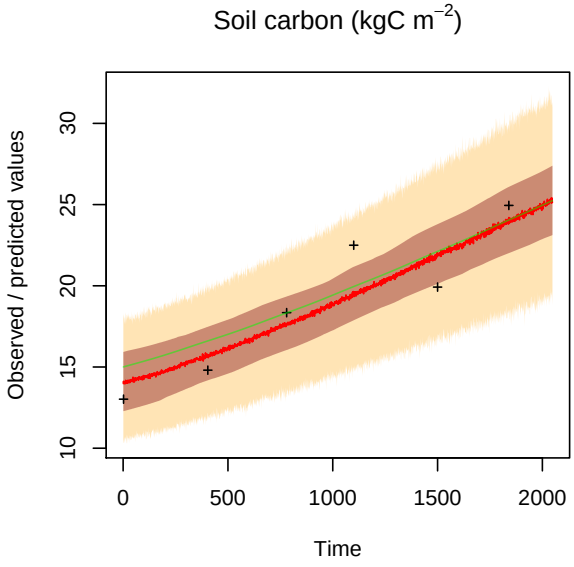
The issue is that lots of potentially useful observations are discarded. Therefore, the posterior uncertainty is larger than it needs to be.

If thinning like this improves the calibration of parts of the system that are sparsely observed, this is a sign that model error is large enough to have a detrimental influence on the calibration.

```
parMatrix  = getSample(out, start=20000,numSamples = 6000)
numSamples = min(1000, nrow(parMatrix))

par(mfrow=c(2,2))

plotfld    = 1
plotfac    = 1000.0
plotTitle  = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")"))
errMod     = createErrorNEE
myObs[-obsSel,] <- NA
createPlot()

plotfld    = 2
plotfac    = 1
plotTitle  = expression(paste("Aboveground vegetation (kgC ",m^{-2},")"))
errMod     = createError
myObs[-obsSel,] <- NA
createPlot()

plotfld    = 3
plotTitle  = expression(paste("Soil carbon (kgC ",m^{-2},")"))
errMod     = createError
myObs[-obsSel,] <- NA
createPlot()
```







Conclusion

So the issue doesn't in the end appear to be that we have an imbalance in the number of observations but that there are structural errors in the model. In this case we know what that error is but generally the model structural error is unknown. The general principle is that all uncertainty should be represented in the calibration. What if we try to model structural error in the calibration so that it can be quantified?

Add terms to represent model structural error to the calibration

Add multiplicative and additive parameters to a new likelihood function

Perhaps the simplest way to do this would be to model discrepancy as a global time invariant bias that could be additive or multiplicative. To add this to the calibration we would simply add additive and multiplicative terms on the two model outputs for which we have lots of calibration data. This adds four new parameters to our calibration MultplicativeBiasNEE, MultplicativeBiasCs, AdditiveBiasNEE + AdditiveBiasCs)

TrueNEE = NEE_VSEM * MultplicativeBiasNEE + AdditiveBiasNEE

TrueCs = Cs_VSEM * MultplicativeBiasCs + AdditiveBiasCs

The likelihood function is modified to add the new terms representing model error.

```
addPars                    <- refPars
addPars[nvar+1,]           <- c(1.0, 0.1, 5.0)
addPars[nvar+2,]           <- c(1.0, 0.1, 5.0)
addPars[nvar+3,]           <- c(0.0, -10, 10.)
addPars[nvar+4,]           <- c(0.0, -10.0, 10.0)
rownames(addPars)[nvar+1]   <- "modmultNEE"
rownames(addPars)[nvar+2]   <- "modmultCs"
rownames(addPars)[nvar+3]   <- "modaddNEE"
rownames(addPars)[nvar+4]   <- "modaddCs"
newPars <- addPars$best
names(newPars) = row.names(addPars)

parSel = c(1,3,5,6,9,10,12,nvar+1,nvar+2,nvar+3,nvar+4)

parRange = rbind(addPars$lower[parSel], addPars$upper[parSel])

likelihood <- function(par){
  x = newPars
  x[parSel] = par

  predicted <- VSEM(x[1:(nvar-1)], PAR)
  predicted[,1] = 1000 * predicted[,1] # this is just rescaling


  diff       <- c((predicted[,1]*x[nvar+1] + x[nvar+3]) - obs[,1])
  llValues1  <- dnorm(diff, sd = pmax((abs(c(predicted[,1])) + 0.0000001) * x[nvar],0.5), log = T)
  diff       <- c(predicted[obsSel,2] - obs[obsSel,2])
  llValues2  <- dnorm(diff, sd = (abs(c(predicted[obsSel,2])) + 0.0000001) * x[nvar], log = T)
  diff       <- c(((predicted[1,3] + (predicted[,3] - predicted[1,3])*x[nvar+2]) + x[nvar+4]) - obs[,3]
  llValues3  <- dnorm(diff, sd = (abs(c(predicted[,3])) + 0.0000001) * x[nvar], log = T)

  return(sum(llValues1,llValues2,llValues3))
}
```

Run MCMC with new likelihood function

A longer MCMC chain is required to find convergence for this new more complicated model.

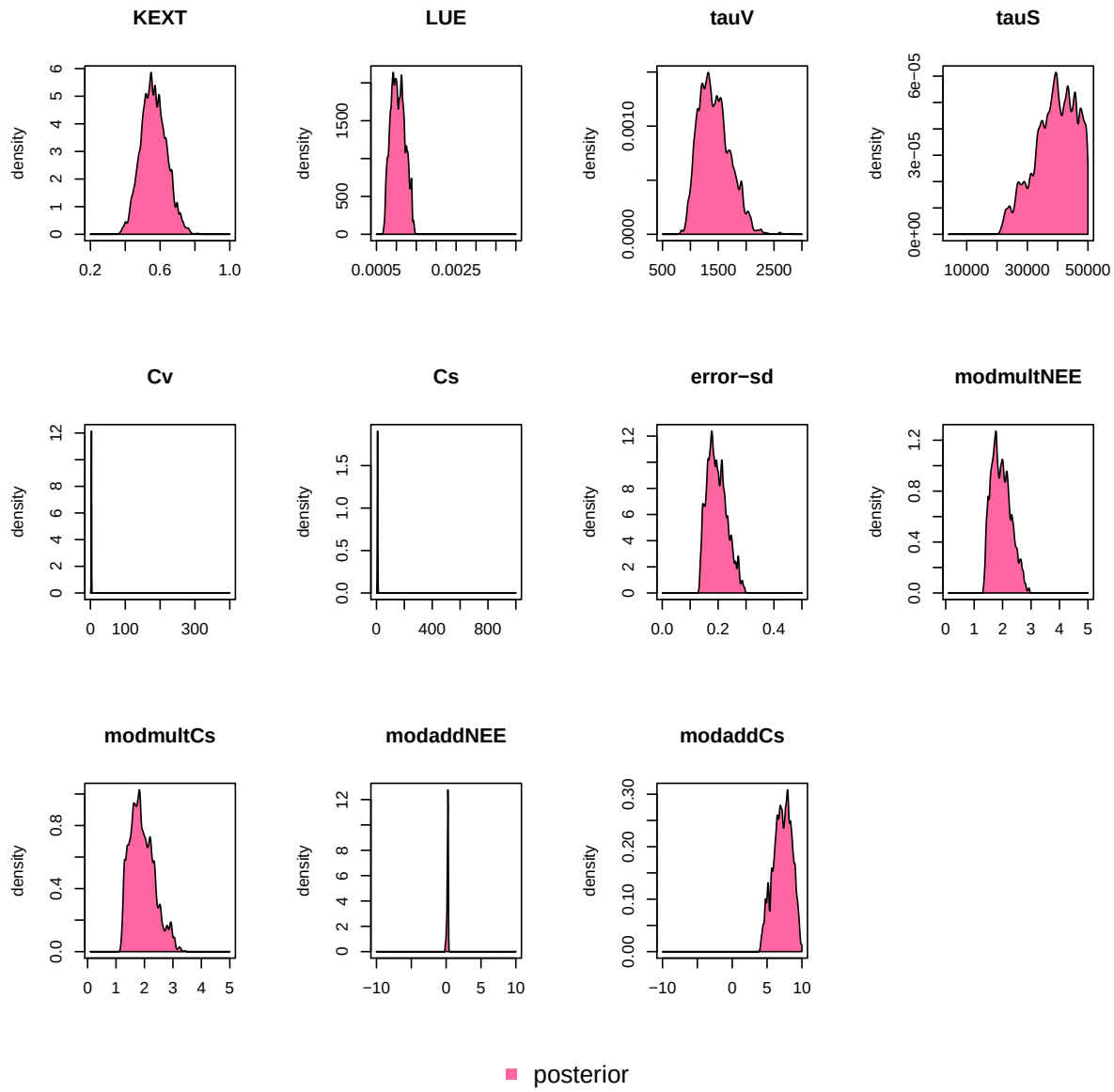
```
prior <- createUniformPrior(lower = addPars$lower[parSel],
                            upper = addPars$upper[parSel], best = addPars$best[parSel])
```

```
bayesianSetup <- createBayesianSetup(likelihood, prior, names = rownames(addPars)[parSel])

obsSel <- c(1,202,390,550,750,920)*2
newPars["Av"]  <- 1.0
newPars["Cr"]  <- 0.0

settings <- list(iterations = 350000, nrChains = 2)

#out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)
#save(out,file="eMod-pObs-nL.rda")

load("eMod-pObs-nL.rda")
```

Parameter plots

Qu: What difference has adding the new terms made?

```
marginalPlot(out, start=60000, singlePanel=FALSE, xrange=parRange)
```




```
kable(addPars[parSel,])
```

	best	lower	upper
KEXT	0.500	2e-01	1e+00
LUE	0.002	5e-04	4e-03
tauV	1440.000	5e+02	3e+03
tauS	27370.000	4e+03	5e+04
Cv	3.000	0e+00	4e+02
Cs	15.000	0e+00	1e+03
error-cv	0.100	1e-03	5e-01
modmultNEE	1.000	1e-01	5e+00
modmultCs	1.000	1e-01	5e+00
modaddNEE	0.000	-1e+01	1e+01
modaddCs	0.000	-1e+01	1e+01

	best	lower	upper

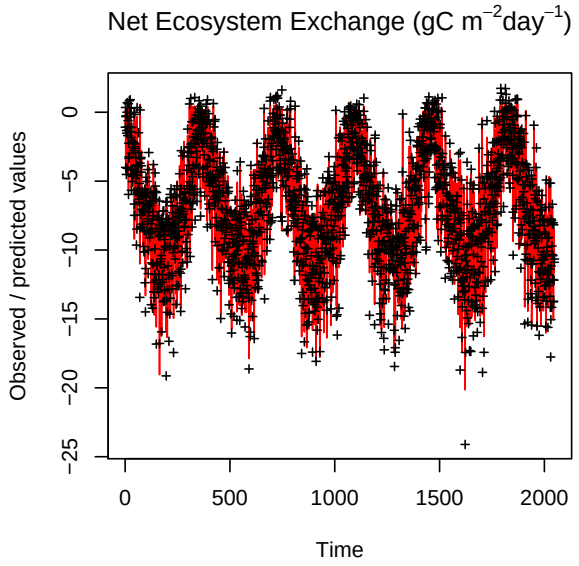
Model predictions

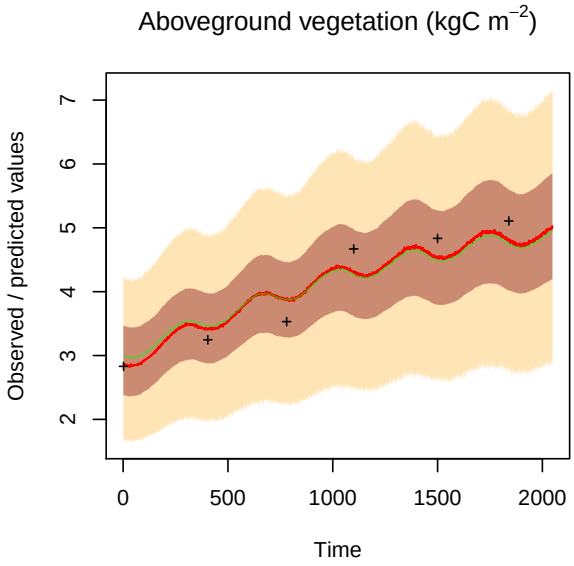
Qu: How about the model predictions. Is this a better calibration than we had before - why?

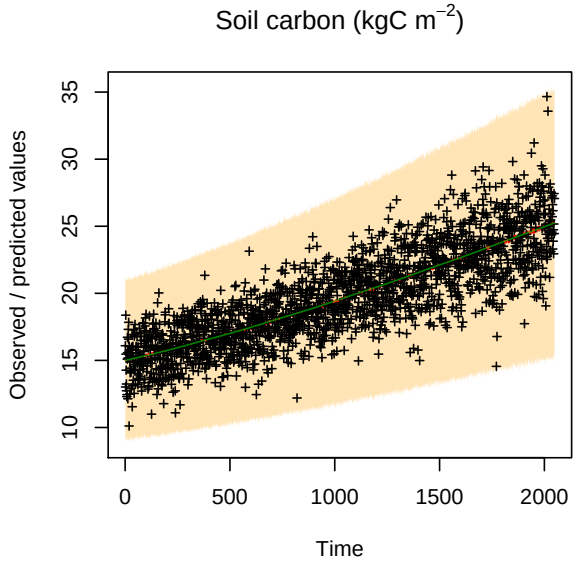
```
## Modifications requited to plotting function with the new likelihood

createPredictionsModLik <- function(par){
                                        # set the parameters that are not calibrated on default values
    x = newPars
    x[parSel] = par
    predicted <- VSEM(x[1:(nvar-1)], PAR) # replace here VSEM with your model

    modModel <- predicted
    modModel[,1] <- (predicted[,1]*1000.)*x[nvar+1] + x[nvar+3]
    modModel[,3] <- (predicted[1,3] + (predicted[,3] - predicted[1,3])*x[nvar+2]) + x[nvar+4]

    return(modModel[,plotfld])
}

createPlot <- function(TRUTH=TRUE){
    pred <- getPredictiveIntervals(parMatrix = parMatrix, model = createPredictionsModLik,
                                   numSamples = numSamples, quantiles = c(0.025, 0.5, 0.975),
                                   error = errMod)
    plotTimeSeries(observed = myObs[,plotfld],
                   predicted = pred$posteriorPredictivePredictionInterval[2,],
                   confidenceBand = pred$posteriorPredictiveCredibleInterval[c(1,3), ],
                   predictionBand = pred$posteriorPredictivePredictionInterval[c(1,3), ],
                   main=plotTitle)
    if (TRUTH) lines(referenceData[,plotfld],col=rgb(red=0,green=1,blue=0,alpha=0.5),lwd=1)

}
```

```
parMatrix  = getSample(out, start=60000,numSamples = 6000)
numSamples = min(6000, nrow(parMatrix))

par(mfrow=c(2,2))

plotfld    = 1
plotfac    = 1000.0
plotTitle  = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")"))
errMod     = createErrorNEE
myObs      = obs
createPlot(TRUTH=FALSE)

plotfld    = 2
plotfac    = 1
plotTitle  = expression(paste("Aboveground vegetation (kgC ",m^{-2},")"))
errMod     = createError
myObs[-obsSel,] <- NA
createPlot()
```

```
plotfld    = 3
plotTitle  = expression(paste("Soil carbon (kgC ",m^{-2},")"))
errMod     = createError
myObs      = obs
createPlot()
```







Model predictions without bias parameters

If we run the VSEM now without the bias parameters we get good predictions for aboveground vegetation but poor predictions for the soil pool and NEE (remember the green line is the truth).

Qus:

1) **Has adding terms that represent model structural uncertainty into the calibration we**

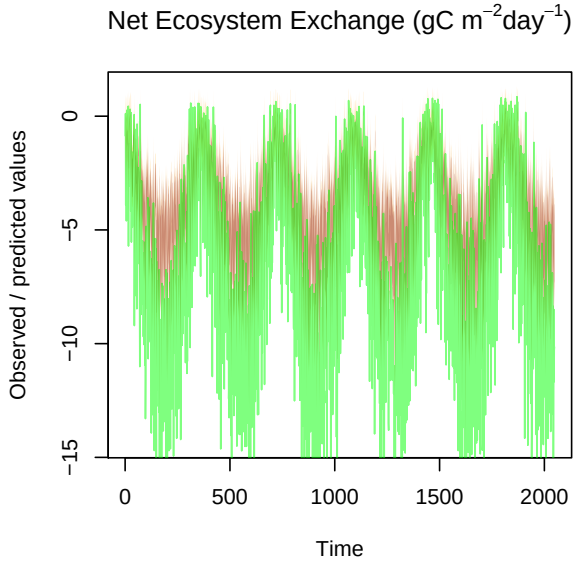
made the model worse?

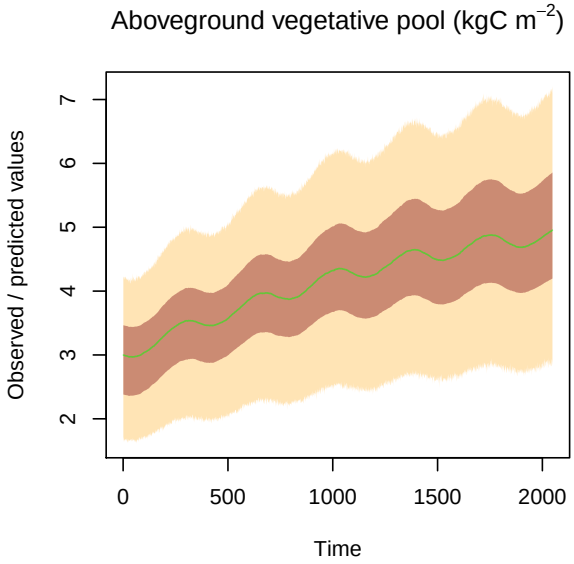
2) What else do you notice about the plot versus the one we had before. What has changed and why?

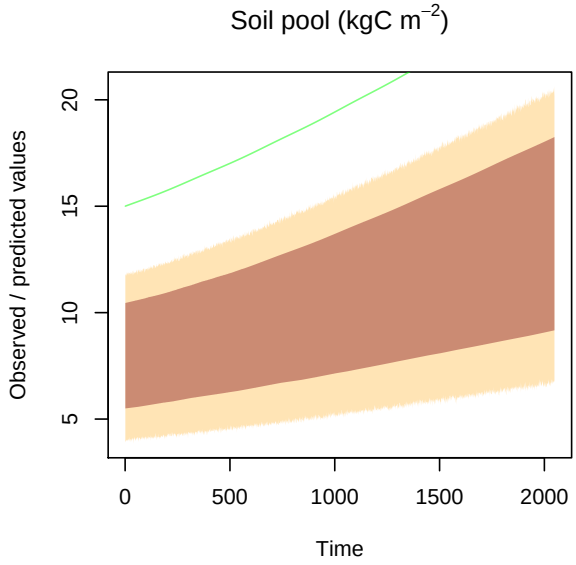
```
createPredictionsModLik <- function(par){
                                        # set the parameters that are not calibrated on default values
    x = newPars
    x[parSel] = par
    predicted <- VSEM(x[1:(nvar-1)], PAR) # replace here VSEM with your model

    modModel <- predicted
    modModel[,1] <- predicted[,1]*1000.
    #modModel[,1] <- (predicted[,1]*1000.)*x[nvar+1] + x[nvar+3]
    #modModel[,3] <- (predicted[1,3] + (predicted[,3] - predicted[1,3])*x[nvar+2]) + x[nvar+4]

    return(modModel[,plotfld])
}

createPlot <- function(TRUTH=TRUE){
    pred <- getPredictiveIntervals(parMatrix = parMatrix, model = createPredictionsModLik,
                                   numSamples = numSamples, quantiles = c(0.025, 0.5, 0.975),
                                   error = errMod)
    ## plotTimeSeries(observed = myObs[,plotfld], predicted = pred$posteriorPredictivePredictionInterval
    ##                confidenceBand = pred$posteriorPredictiveCredibleInterval[c(1,3), ],
    ##                predictionBand = pred$posteriorPredictivePredictionInterval[c(1,3), ],
    ##                main=plotTitle)
    plotTimeSeries(observed = myObs[,plotfld],
                   confidenceBand = pred$posteriorPredictiveCredibleInterval[c(1,3), ],
                   predictionBand = pred$posteriorPredictivePredictionInterval[c(1,3), ],
                   main=plotTitle)
    if (TRUTH) lines(referenceData[,plotfld],col=rgb(red=0,green=1,blue=0,alpha=0.5),lwd=1)

}
```

```
parMatrix  = getSample(out, start=60000,numSamples = 6000)
numSamples = min(6000, nrow(parMatrix))

par(mfrow=c(2,2))

plotfld    = 1
plotfac    = 1000.0
plotTitle  = expression(paste("Net Ecosystem Exchange (gC ",m^{-2} * day^{-1},")"))
errMod     = createErrorNEE
#myObs      = obs
#myObs[-obsSel,] <- NA
myObs[,] <- NA
#createPlot(TRUTH=FALSE)
createPlot()

plotfld    = 2
plotfac    = 1
plotTitle  = expression(paste("Aboveground vegetative pool (kgC ",m^{-2},")"))
errMod     = createError
#myObs[-obsSel,] <- NA
myObs[,] <- NA
```

```
createPlot()

plotfld    = 3
plotTitle  = expression(paste("Soil pool (kgC ",m^{-2},")"))
errMod     = createError
#myObs[-obsSel,] <- NA
#myObs      = obs
myObs[,] <- NA
createPlot()
```







Conclusion

1. No model represents the true system perfectly. Often we don't know what this error (discrepancy) is so we need to quantify this uncertainty.

2. If we ignore this uncertainty we will often end up with a model that is confidently wrong.

3. Another consequence of this could be that the model is over-fitted to parts of the system for which there are many observations at the expense of the few.
4. If thinning the observations improves the calibration this diagnoses that model structural error is causing problems in the calibration.
5. Adding terms to the calibration to represent uncertainty in model errors improves the parametric fit of the model. Uncertainty also increases reflecting the fact that the model has errors and that we don't know what they are.
6. The 'discrepancy model' could be useful for diagnosing model structural errors.

Appendix: Very quick recipe on preparing a model for calibration

Just for your info not to be run just now.

Two things are crucial:

1. That model parameters can be changed by the MCMC algorithm
2. Output from the model run with the different parameter vectors is available in the likelihood function.

Step1: adapting your model to accept the parameters to be calibrated.

To make parameter variation of calibrations possible from R, the first thing we need to do is to be able to execute the model with a given set of parameters. We will assume that we want to run the model with parameters coded in an object called *par*, which should be either a (named) vector or a list.

This parameter vector then needs to be communicated to the model in some way. This could be in the header of the model calling subroutine as in this FORTRAN example

```
Subroutine BASFOR(FORTYPE,RESTRT,PARAMS,MATRIX_WEATHER, &
                  CALENDAR_FERT,CALENDAR_NDEP,CALENDAR_PRUNT,CALENDAR_THINT, &
                  NDAYS,NOUT, &
                  y)

real                        :: PARAMS(NPAR)

BETA      = PARAMS(1)! (-)                          Sensitivity of LUE to [CO2]
CBTREE0   = PARAMS(2)! (kg C tree-1)                Initial C in branches
CLTREE0   = PARAMS(3)! (kg C tree-1)                Initial C in leaves
CRTREE0   = PARAMS(4)! (kg C tree-1)                Initial C in roots
.
.
.
```

Or also look at the VSEM version written in C (vsemC) included with BayesianTools for an example in C.

https://github.com/florianhartig/BayesianTools/blob/master/BayesianTools/src/vsem.cpp

or if your model is in R see how VSEM is called with pars.

```
VSEM <- function (pars = c(KEXT = 0.5, LAR = 1.5, LUE = 0.002, GAMMA = 0.4,
    tauV = 1440, tauS = 27370, tauR = 1440, Av = 0.5, Cv = 3,
    Cs = 15, Cr = 3), PAR, C = TRUE)
{
...
        KEXT = pars[1]
        LAR = pars[2]
        LUE = pars[3]
        GAMMA = pars[4]
...
}
```

Step2: making your model callable from R

How you do this depends on how your model is programmed

1. Model in R, or R interface existing
2. Model accepts parameters via the command line
3. Model can be linked as dll
4. Model reads parameters only via parameter file

1) If your model is already coded in R (eg VSEM)

Usually, you will have to do nothing. Just make sure you can call your model like this.

```
runMyModel(par)
```

2) If you have a compiled model, parameters set via command line

If your model can receive parameters via the command line, try something like this

```
runMyModel(par){

  # Create here a string with what you would write to call the model from the command line
  systemCall <- paste("model.exe", par[1], par[2])

  # this
  system(systemCall)

}
```

3) If you have a compiled dll, parameters are set via dll interface

Use the dyn.load function to link to your model

```
dyn.load(model)
runMyModel(par){

  # model call here

}
```

4) If you have a compiled model, parameters set via parameter file

Many models read parameters with a parameter file. In this case you want to do something like this

```
runMyModel(par){

  writeParameters(par)

  system("Model.exe")

}
writeParameters(par){

  # e.g.
  # read template parameter file
  # replace strings in template file
  # write parameter file
}
```

How you design the write parameter function depends on the file format you use for the parameters. In general, you probably want to create a template parameter file that you use as a base and from which you change parameters

1. If your parameter file is in an *.xml format*, check out the xml functions in R
2. If your parameter file is in a *general text format*, the best option may be to create a template parameter file, place a unique string at the locations of the parameters that you want to replace, and then use string replace functions in R, e.g. grep to replace this string.

Step 3: Returning model output to R

For simple models, you might consider returning the model output directly with the runMyModel function. This is probably so for cases 1) and 2) above, i.e. model is already in R, or accepts parameters via command line.

Back to the FORTRAN example. Model outputs are written to an array (y)

```
Subroutine BASFOR(FORTYPE,RESTRT,PARAMS,MATRIX_WEATHER, &
                  CALENDAR_FERT,CALENDAR_NDEP,CALENDAR_PRUNT,CALENDAR_THINT, &
                  NDAYS,NOUT, &
                  y)
.
.
.
real                        :: y(NDAYS,NOUT)
.
.
.
  y(day, 1)  = year + (doy-0.5)/366                  ! "Time" = Decimal year (approximation)
  y(day, 2)  = year
  y(day, 3)  = doy
  y(day, 4)  = T                                     ! degC
  y(day, 5)  = RAIN                                  ! mm d-1
  y(day, 6)  = CR          * 10                      ! t C ha-1
  y(day, 7)  = (CS+CB)     * 10                      ! t C ha-1
  y(day, 8)  = CL          * 10                      ! t C ha-1
```

.
.
.

See also the VSEM C example https://github.com/florianhartig/BayesianTools/blob/master/BayesianTools/src/vsem.cpp

```
    out(i,0) = NEE;
    out(i,1) = Cv;
    out(i,2) = Cs;
    out(i,3) = Cr;
  }

  return out;
```

and also the VSEM in R

```
            out[i, ] = c(NEE, Cv, Cs, Cr)
        }
        return(out)
    }
}
```

Model writes file output

If the model writes output to a file, write a getModelOutput function that reads in the model outputs and returns the data in the desired format, typically the same that you would use to represent your field data.

```
getModelOutput(type = X){

  read.csv(xxx)

  # do some transformation

  # return data in desidered format
}
```

Resultant code in R

From R, you should now be able to do something like this similar to what we had above for the VSEM.

```
par = c(1,2,3,4 ..)
runMyModel(par)
output <- getModelOutput(type = DesiredType)
plot(output)
```