Particle Filter

```r
library(iotools)
library(ecoforecastR)
```

```
## Loading required package: rjags

## Loading required package: coda

## Warning: package 'coda' was built under R version 4.3.2

## Linked to JAGS 4.3.2

## Loaded modules: basemod,bugs
```

This exercise explores the use of the particle filter to constrain a simple ecosystem model for the Metolius Ameriflux sites in Oregon. Specifically, we'll be looking at the Metolius Intermediate Pine site (US-ME2) http://ameriflux.lbl.gov/sites/siteinfo/US-Me2 and assimilating a derived MODIS data product, LAI, into a super simple ecosystem model. In the code below we will perform three analyses:

1) Run an ensemble forecast for one year
2) Use a particle filter to analyse the existing ensemble based on MODIS LAI
3) Rerun the LAI assimilation with a resampling PF

The code below is longer than most activities, but this is not due to the complexity of the PF itself. Rather, we will spend a decent amount of code on defining the model, defining the initial conditions, and defining the prior distributions for the model parameters.

The initial code is set up to run with a small ensemble size (ne=10) and short time (17-days, nt = 816 30-minute time-steps) to allow you to be able to "Knit" this document.

The final run should be conducted with a decently large ensemble (500-5000 depending on what your computer can handle) and be for the full year `nt = length(time)`. Since this requires large simulation, if you are doing this activity for a class turn in the final HTML, not the Rmd, and feel free to answer questions separately (e.g. in another doc) so you don't have to run a second time to include your answers.

Alternatively, you could change the output type to `html_notebook` and run the analysis block-by-block.

Super Simple Ecosystem Model

Let's begin by defining our model itself, as well as a number of ancillary functions that will be useful in simulation and analysis. The model below is very simple but is complex enough to have some chance at capturing observed variability. In addition, unlike most ecosystem models, it explicitly contains process error. The model has three state variables (X) that are all expressed in terms of carbon (Mg/ha): Leaf Biomass, Structural Biomass (wood, roots, etc), and soil organic matter (SOM). The model also has only two drivers: photosynthetically active radiation (PAR), and air temperature. Within the model we first estimate LAI from Leaf Biomass and SLA. Using LAI and light we estimate GPP using a simple light use efficiency approach. GPP is then allocated to autotrophic respiration (Ra), leaf NPP, and woody NPP. These leaf and wood biomass pools can then turns over into SOM as litterfall and Coarse Woody Debris / mortality. Heterotrophic respiration is assumed to follow a standard Q10 temperature sensitivity. Finally, Normal process error is added to X.

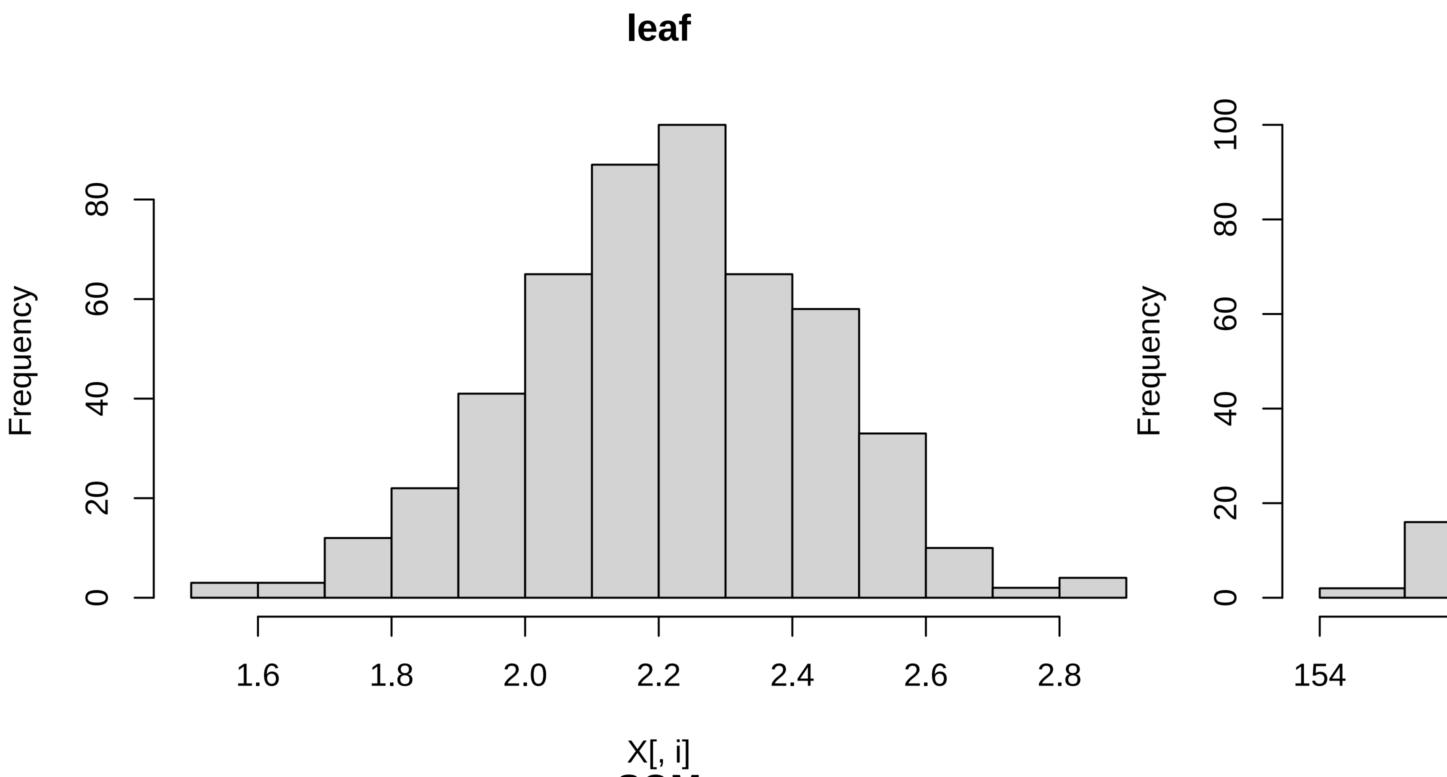
```r
library(compiler)

##' Super Simple Ecosystem Model
##' @param X        [leaf carbon, wood carbon, soil organic carbon] (units=Mg/ha)
##' @param params   model parameters
##' @param inputs   model drivers (air temperature, PAR)
##' @param timestep seconds, defaults to 30 min
SSEM.orig <- function(X, params, inputs, timestep = 1800){

  ne = nrow(X)  ## ne = number of ensemble members

  ##Unit Converstion: umol/m2/sec to Mg/ha/timestep
  k = 1e-6 * 12 * 1e-6 * 10000 * timestep #mol/umol*gC/mol*Mg/g*m2/ha*sec/timestep

  ## photosynthesis
  LAI = X[, 1] * params$SLA * 0.1  #0.1 is conversion from Mg/ha to kg/m2
  if(inputs$PAR > 1e-20){
      GPP = pmax(0, params$alpha * (1 - exp(-0.5 * LAI)) * inputs$PAR)
  } else {
      GPP = rep(0, ne)
  }

  ## respiration & allocation
  alloc = GPP * params$falloc ## Ra, NPPwood, NPPleaf
  Rh = pmax(params$Rbasal * X[, 3] * params$Q10 ^ (inputs$temp / 10), 0) ## pmax ensures SOM never goes

  ## turnover
  litterfall = X[, 1] * params$litterfall
  mortality = X[, 2] * params$mortality

  ## update states
  X1 = pmax(rnorm(ne, X[, 1] + alloc[, 3] * k - litterfall, params$sigma.leaf), 0)
  X2 = pmax(rnorm(ne, X[, 2] + alloc[, 2] * k - mortality, params$sigma.stem), 0)
  X3 = pmax(rnorm(ne, X[, 3] + litterfall + mortality - Rh * k, params$sigma.soil), 0)

  return(cbind(X1 = X1, X2 = X2, X3 = X3,
               LAI = X1 * params$SLA * 0.1,
               GPP = GPP,
               NEP = GPP - alloc[, 1] - Rh,
               Ra = alloc[, 1], NPPw = alloc[, 2], NPPl = alloc[, 3],
               Rh = Rh, litterfall = litterfall, mortality = mortality))

}
SSEM <- cmpfun(SSEM.orig)  ## byte compile the function to make it faster
```

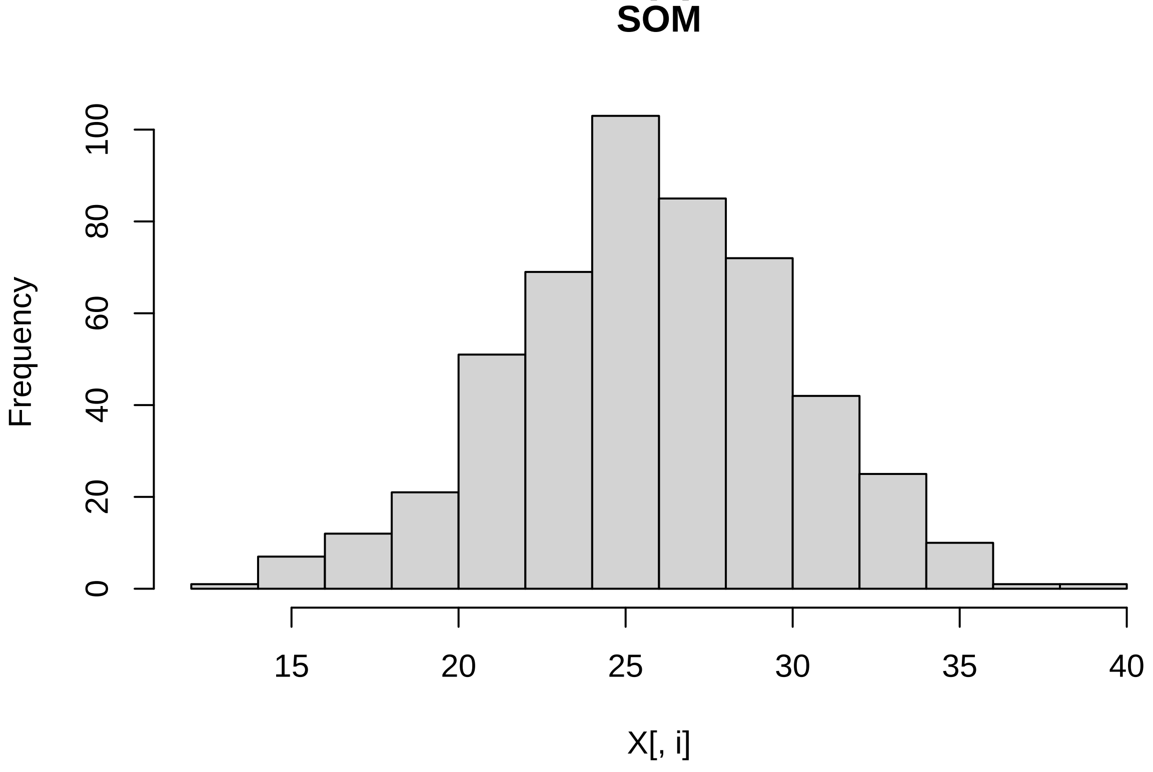
Initial Conditions

Having defined our model, the next step is to define the ensemble size and generate an ensemble estimate of the initial state variables. To do so we'll use the estimates that are reported in the Ameriflux BADM Meta-data files for the site. Since we're only relying on two different estimates of pool size to calculate our mean and standard deviation, and neither estimate has a reported error, these should be taken as "demonstration only" rather than as "Best practices". In a real application one would want to account for the sampling error associated with the number of vegetation plots or soil cores measured, the measurement error in the soil C and tree DBH, and the allometric uncertainty in converting from DBH to leaf and stem biomass. In other words, our pool sizes are likely a lot less certain than what we take them to be in this exercise.

There are a few assumptions in developing the initial conditions - The mean and standard deviation are only calculated using two observations. As stated above, this is not best practice. - The wood state is the combination of stems, coarse roots, and fine roots - The SOM (soil) state is the combination of litter, downed coarse woody debris, and soil. - All values are converted from g/m2 to Mg/ha

```r
#### SET THE ENSEMBLE SIZE
ne = 500 ## production run should be 200 - 5000, depending on what your computer can handle

### Initial State (Mg/ha)
### These specific data points were extracted from the Ameriflux BADM files
Bwood = (c(11983, 12097)+c(3668, 3799)+c(161, 192)) * 1e-6 * 10000 ## sum up stems, coarse roots, and f
Bleaf = c(206, 236) * 0.01
SOM = c(1.57, 1.58) + c(0.49, 1.39) + c(2.06, 2.59) * 1e-3 * 10000 ## sum up litter, CWD, and soil; cha
X = as.matrix(c(mean(Bleaf), mean(Bwood), mean(SOM)))
### sample initial condition ensemble members
if(ne > 1){
  X = as.matrix(cbind(
      rnorm(ne, X[1], sd(Bleaf)),
      rnorm(ne, X[2], sd(Bwood)),
      rnorm(ne, X[3], sd(SOM))))
}
X.orig = X ## make a copy so that we can run different experiments

## visualize initial condition priors
pool.lab = c("leaf", "wood", "SOM")
for(i in 1:3){
  hist(X[, i], main = pool.lab[i])
}
```





Parameter Priors

Having defined the initial condition state vector, we'll next define the priors on the model parameters. Unlike in JAGS, where we define the priors in terms of named distributions, for a particle filter we want to actually **draw random samples from those prior distributions**.

For two parameters, SLA and litterfall, there are estimates reported in the Ameriflux BADM as well. Therefore, the prior on SLA was set from data, but the priors on light use efficiency (alpha), Q10, soil basal respiration, and the process uncertainties in GPP and Rh were set just based on my expert opinion – all these could be informed better by literature data (e.g. PEcAn's meta-analysis).

```
## ancillary data from Ameriflux BADM metadata
SLA = 1e3/c(114,120)      ## m2/kg
litterfall = c(71,94)*0.01*3 ## gC/m2/yr->Mg/ha/yr

### initial params
timestep = 1800 #seconds
params = list()
params$SLA = rnorm(ne, mean(SLA), sd(SLA))      ## Specific leaf area

## univariate priors: expert opinion
params$alpha = rlnorm(ne, log(0.02), 0.05)      ## light use efficiency
params$Q10 = rnorm(ne, 2.1, 0.1)                ## soil respiration Q10
params$Rbasal = rlnorm(ne, log(0.2), 1) / (params$Q10^2.5) ## Soil basal respiration (umol/m2/sec per M
```

For the allocation parameters, we can't assume they are dependent as we assume for the other parameters because the three allocation parameter have to sum to 1. Therefore if leaf NPP is high, wood NPP, and Ra must be lower. This requires a bit more calculation to solve. First, we'll assume that on average that NPP is ~50% of GPP (e.g. Litton et al 2007), and that leaf NPP is 31.5% of total NPP (which is the default allocation fraction used by Quaife et al 2008 for DALEC for this site).

To account for uncertainty we'll scale these fractions by an "effective sample size" (`Neff`) in order to specify how many observations these represent – for example, if `Neff` was 10 then the variability in the allocation to NPP vs Ra would be the equivalent to the uncertainty associated with observing 5 coin flips come up "NPP" and 5 come up "Ra". To assign different ensemble members different levels of process error we draw `Neff` from a Poisson distribution. Again, this is an underestimate and a better practice would be to derive the priors for these fractions from data and to account for the fact that the mean proportions should vary from ensemble member to ensemble members as well as the certainty.

```
## reimplimentation of the rdirichlet function from MCMCpack
## to fix bug in how it handles alpha as a matrix
rdirichlet.orig = function (n, alpha)
{
    l <- length(alpha)
    if(is.matrix(alpha)) l <- ncol(alpha)
    x <- matrix(rgamma(l * n, alpha), ncol = l)
    sm <- x %*% rep(1, l)
    return(x/as.vector(sm))
}
rdirichlet <- cmpfun(rdirichlet.orig)         ## byte compile to speed up

## multivariate prior on allocation parameters
## assume that NPP is ~50% of GPP on average (Litton et al 2007)
Ra = 0.5
## prior mean on allocation, assume leaf NPP is 31.5% of total (Quaife et al 2008)
alloc = matrix(c(Ra, (1 - 0.315) * (1 - Ra), 0.315 * (1 - Ra)), 1)
## draw effective sample size to add stochasticity to prior
Neff = matrix(rpois(ne, 100), ne)
## prior on [Ra, wood, leaf]
params$falloc = rdirichlet(ne,Neff%*%alloc)
```

For the process error we're setting Gamma priors on the *precisions* and then converting those to standard deviations.

```
## Process error: expert opinion

## prior process error in leaf biomass
params$sigma.leaf = 1 / sqrt(rgamma(ne, 10, 10 * 0.01 ^ 2))
## prior process error in stem biomass
params$sigma.stem = 1 / sqrt(rgamma(ne, 10, 10 * 0.1 ^ 2))
## prior process error in soil carbon
params$sigma.soil = 1 / sqrt(rgamma(ne, 10, 10 * 0.1 ^ 2))
```

Finally, the prior for both litterfall and mortality are set based on **moment matching** – deriving the parameters for the Beta that match a specified mean and variance.

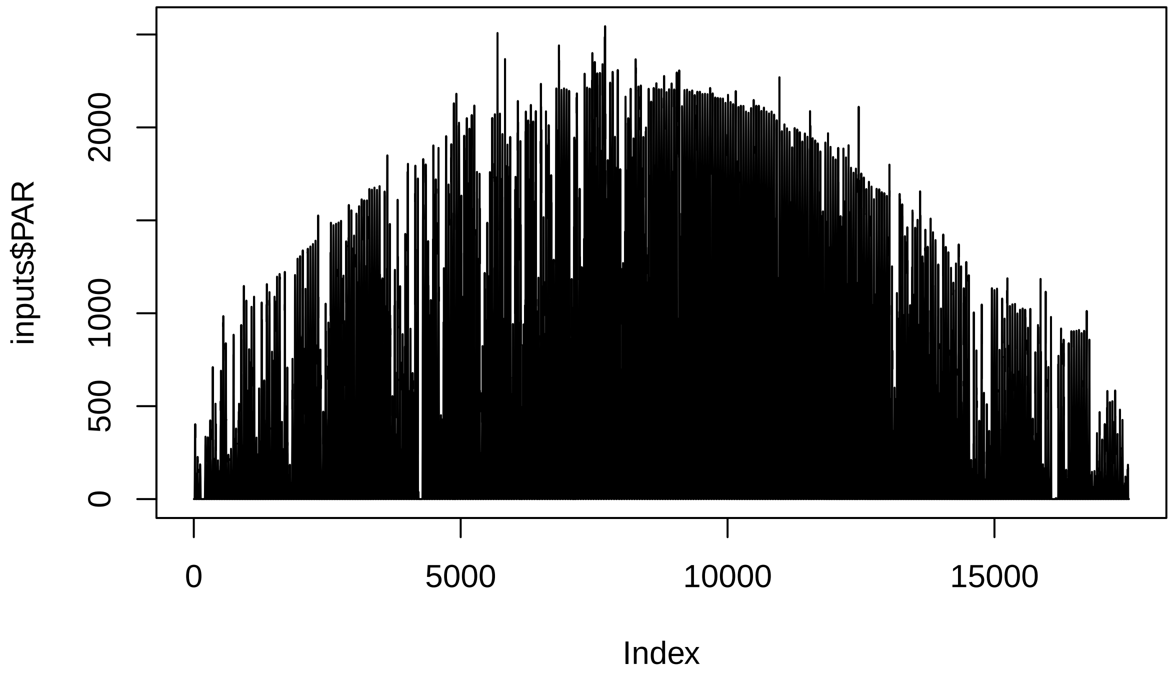
For litterfall, since this needs to be expressed as a proportion of leaves lost, this is based on comparing the variability in the observed annual litterfall rate to the observed leaf biomass. For mortality this is done based on the mean background tree mortality rate for temperate forests reported in Dietze et al (2011) (1/142) and assuming a CV of 50%. The latter could be much improved with species and system specific data. The approach for the litterfall rate could also be improvedwith additional data and accounting for the sampling uncertainty in both the numerator and denominator.

```
## moment matching beta prior on turnover times
beta.match <- function(mu, var){   ## Beta distribution moment matching
  a = mu * ((mu * (1 - mu) / var) - 1)
  b = a * (1 - mu) / mu
  return(data.frame(a = a, b = b))
}

## simulate litterfall turnover based on observed
## litterfall rate and Bleaf prior (initial condition)
lit = rnorm(10000,mean(litterfall),sd(litterfall)/sqrt(2))/
  rnorm(10000,mean(Bleaf),sd(Bleaf)/sqrt(2))
## draw prior mean and sd; convert turnover per year -> turnover per timestep
lit.mu = rnorm(ne,mean(lit),sd(lit))*timestep/86400/365
lit.sd = 1/sqrt(rgamma(ne,10,10*var(lit)))*timestep/86400/365
litterfall.param = beta.match(lit.mu,lit.sd^2)
## match moments and draw litterfall prior
params$litterfall = rbeta(ne,litterfall.param$a,litterfall.param$b)

## draw prior mean based on background tree mortality rate of 1/142 per year (Dietze et al 2011)
mortality.mu = 1/rpois(ne,142)*timestep/86400/365
## draw prior sd assuming a 50% CV
mortality.sd = rbeta(ne,4,4)*mortality.mu*timestep/86400/365
## match moments and draw mortality prior
mortality.param = beta.match(mortality.mu,mortality.sd^2)
params$mortality = rbeta(ne,mortality.param$a,mortality.param$b)
```

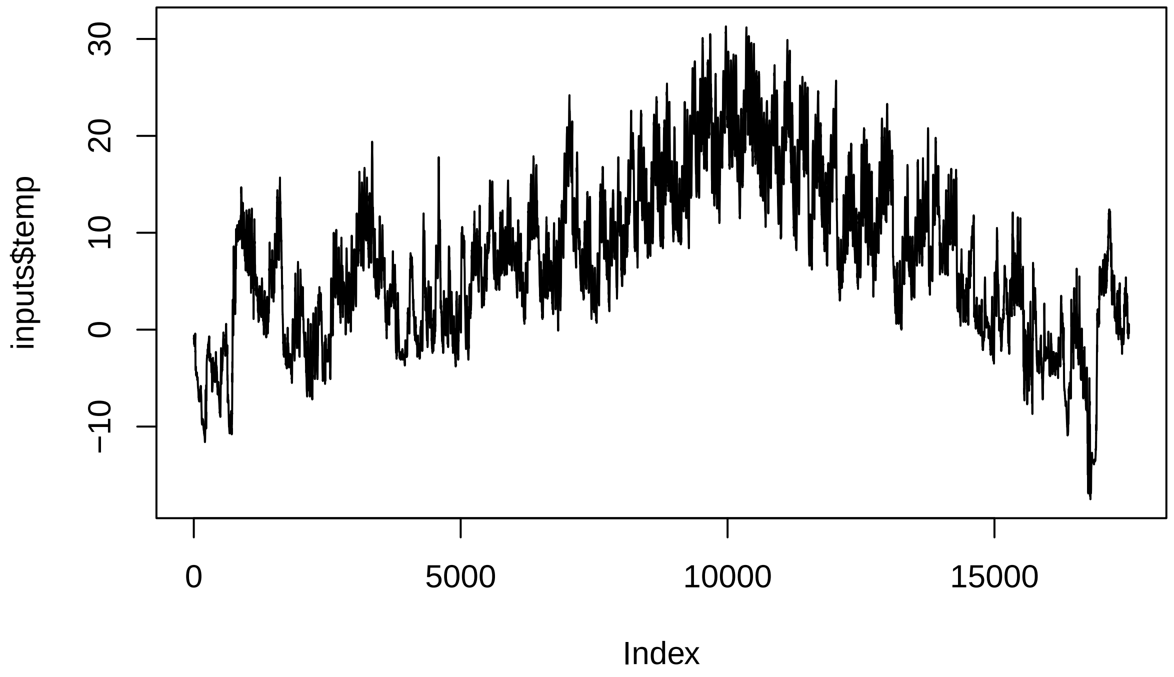
Drivers

Next, we need to load the observed meteorology from the flux tower to provide our input drivers. The data are for a full year (2 per hour * 24 hours * 365 days). For simplicity, we won't worry about the uncertainty in the met data.

```
## load met data
load("data/Lab10_inputs.RData")
plot(inputs$PAR,type='l')
```



```
plot(inputs$temp,type='l')
```



Initial forecast

Now we're ready to produce our initial ensemble forecast for the system. To do this we'll set up some storage and loop over time where we calling the model each time step. After this we'll generate some basic diagnostic plots for the model.

```
X = X.orig                                ## initial condition
nt = nrow(inputs)#17 * 48                                ## 17 days of 30min; production run should be nrow(
print(nt)
```

```
## [1] 17520
```

```
output = array(0.0, c(nt, ne, 12))      ## output storage [time step,ensembles,variables]

## foreward ensemble simulation
for(t in 1:nt){
  output[t, , ] <- SSEM(X, params, inputs[t, ])  ## run model, save output
  X <- output[t, , 1:3]                          ## set most recent prediction to be the next IC
  if((t %% 336) == 0) print(t / 336)             ## counter: weeks elapsed (7*48 = 1 week)
}
```

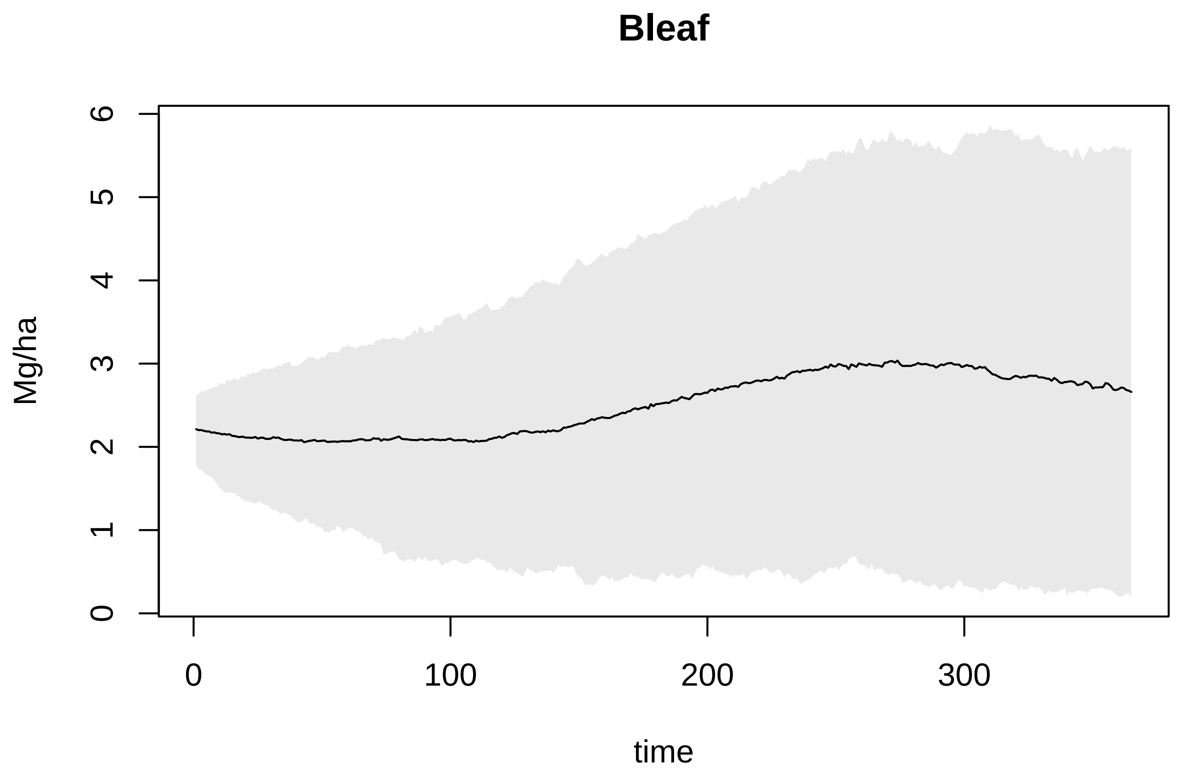
```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
## [1] 35
## [1] 36
## [1] 37
## [1] 38
## [1] 39
## [1] 40
## [1] 41
## [1] 42
## [1] 43
## [1] 44
## [1] 45
## [1] 46
## [1] 47
## [1] 48
## [1] 49
## [1] 50
## [1] 51
## [1] 52
```

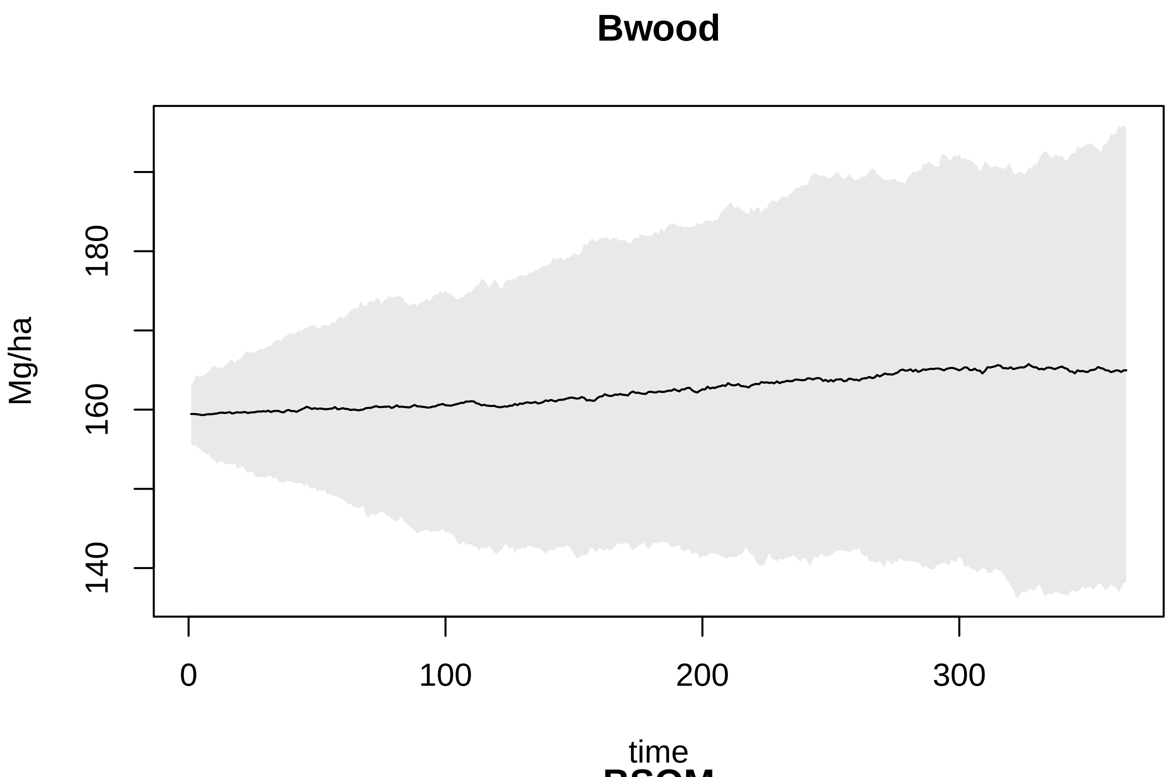
```
output[is.nan(output)] = 0
output[is.infinite(output)] = 0
output.ensemble = output                          ## save original ensemble projection

## average the output to daily
bin = 86400 / timestep          ## seconds per day / timestep = number of timesteps per day
out.daily = array(0.0, c(ceiling(nt/bin), ne, 12))  ## [day, ensemble, variables]
for(i in 1:12){                 ## loop over each variable
  print(i)
  out.daily[,,i] <- apply(output[,,i],2, tapply, rep(1:365,each=bin)[1:nt], mean)
}
```

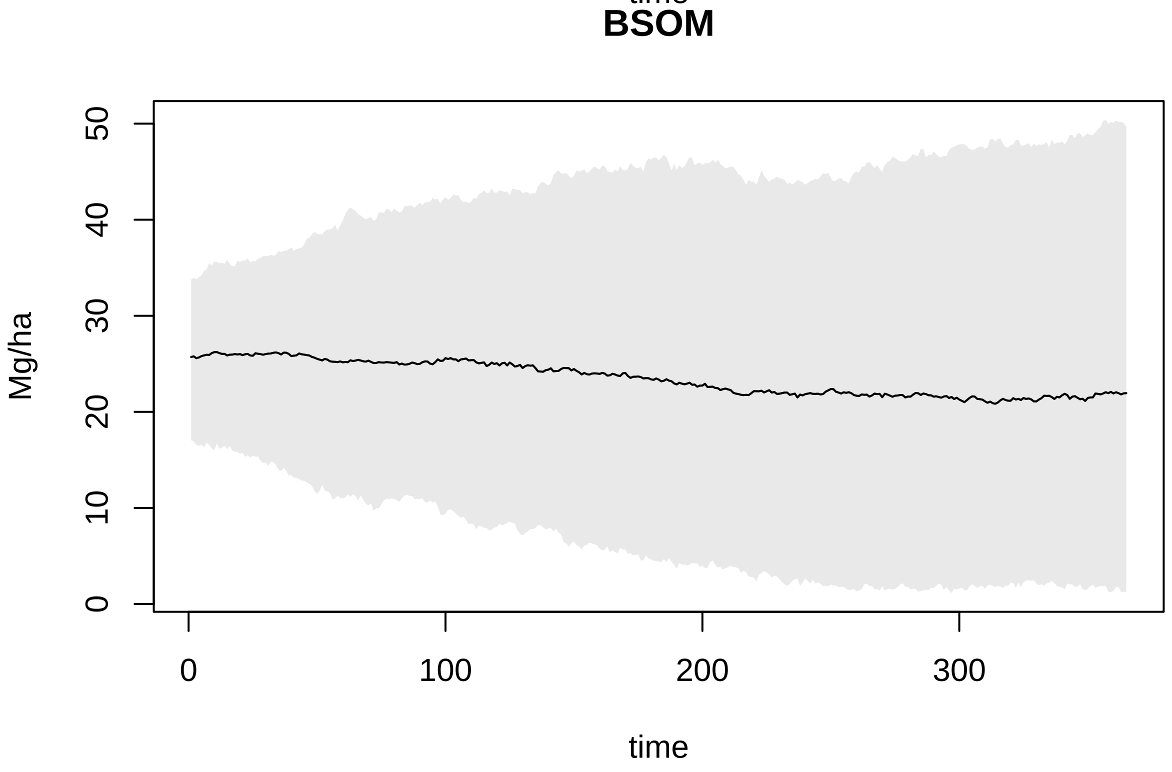
```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
```

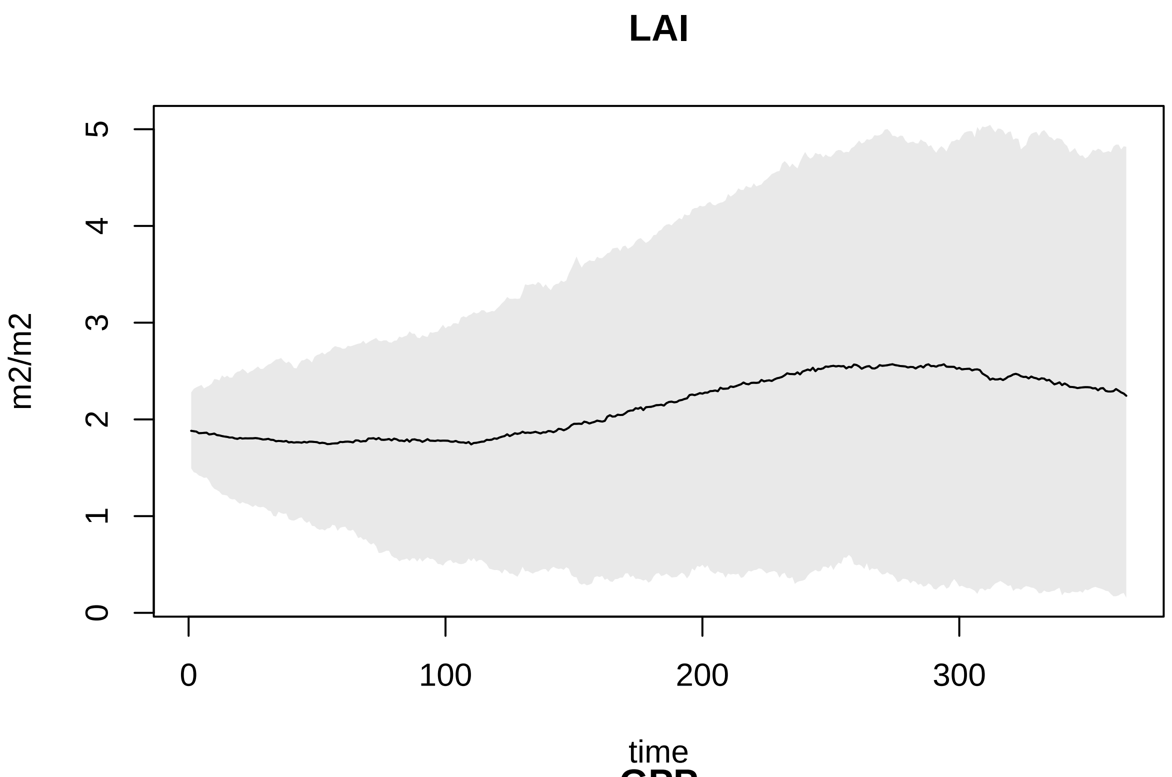
```
## Basic time-series visualizations
varnames <- c("Bleaf","Bwood","BSOM","LAI","GPP","NEP","Ra",
              "NPPw","NPPl","Rh","litterfall","mortality")
units <- c("Mg/ha","Mg/ha","Mg/ha","m2/m2","umol/m2/sec","umol/m2/sec",
           "umol/m2/sec","umol/m2/sec","umol/m2/sec","umol/m2/sec",
           "Mg/ha/timestep","Mg/ha/timestep")
```

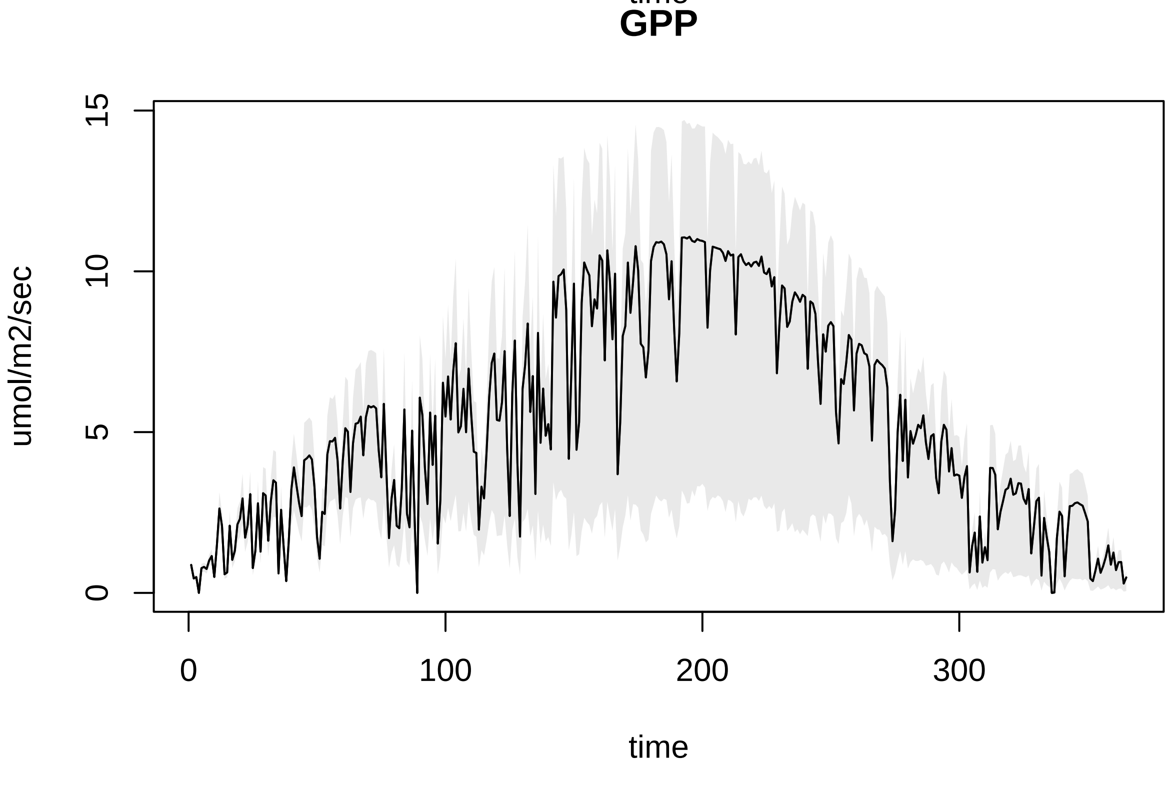
```
for(i in 1:12){  ## loop over variables
  ci = apply(out.daily[, , i], 1, quantile, c(0.025, 0.5, 0.975))    ## calculate CI over ensemble membe
  plot(ci[2, ], main = varnames[i],
       xlab = "time", ylab = units[i], type='l',ylim  =range(ci))
  ciEnvelope(1:ncol(ci), ci[1, ], ci[3, ], col = col.alpha("lightGrey", 0.5)) ## plot interval
  lines(ci[2, ])                                                              ## plot median
}
```

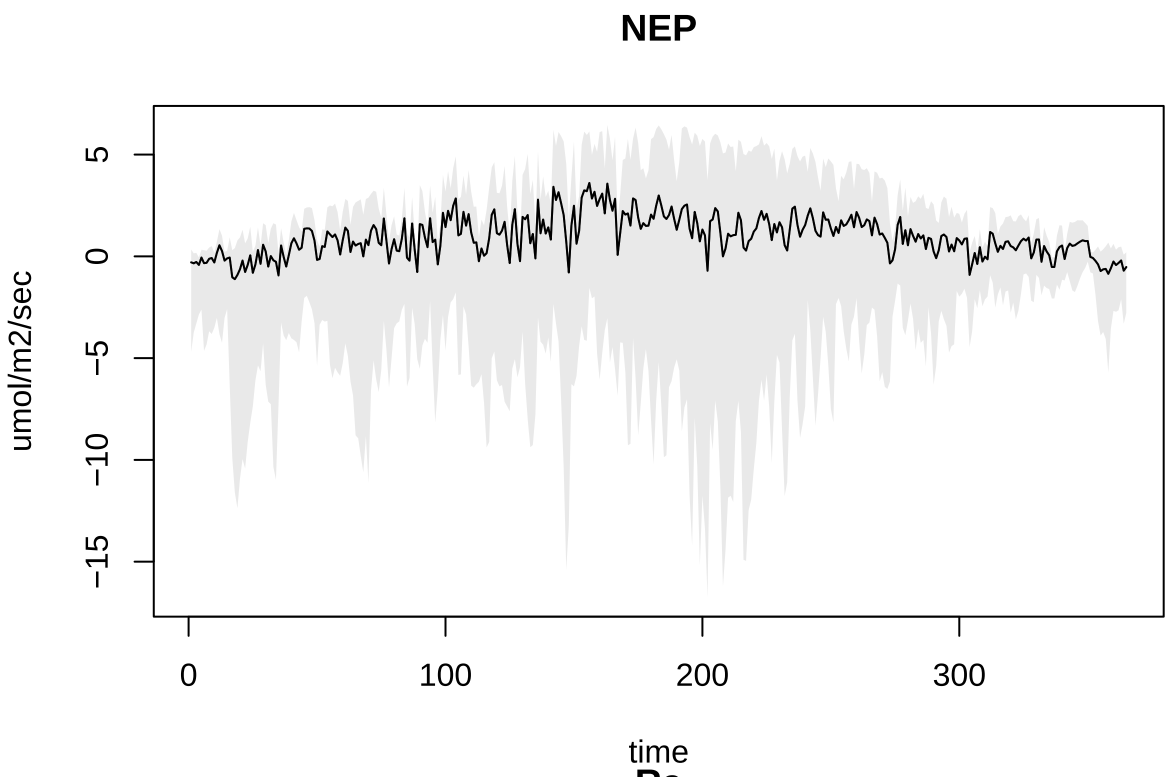


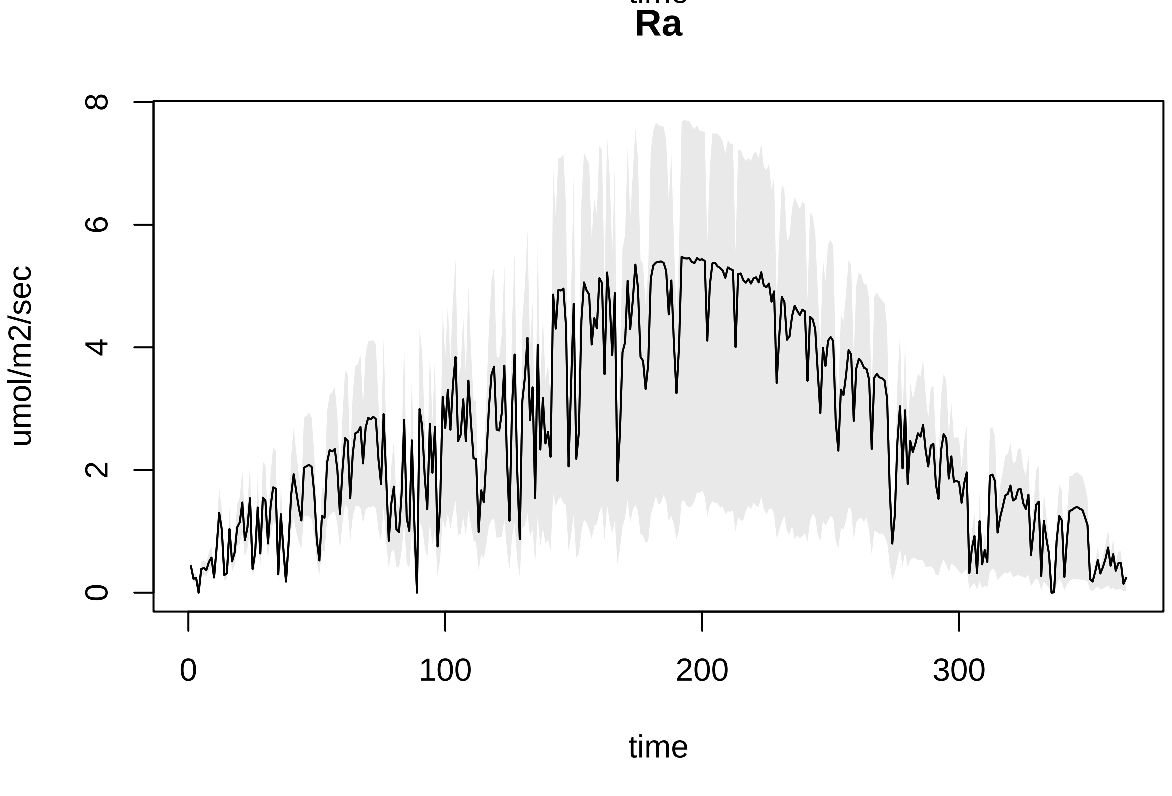

Bwood
Mg/ha
140
160
180
0
100
200
300
time

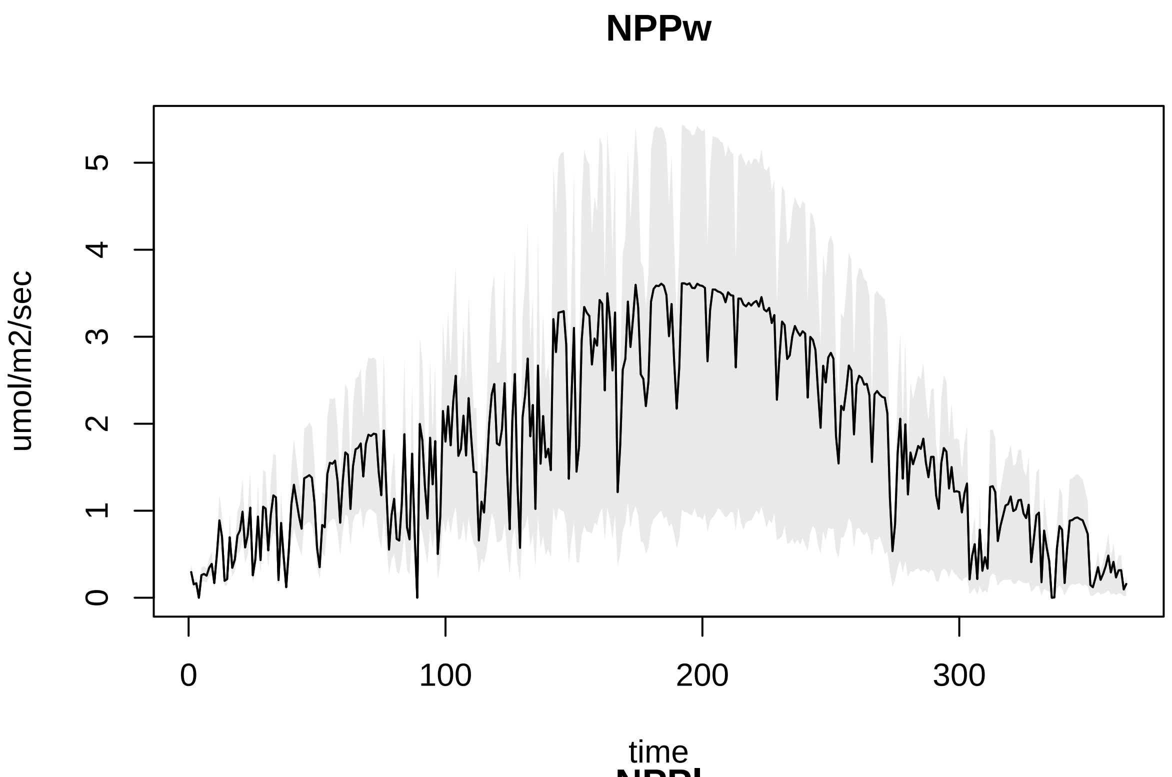

BSOM
Mg/ha
0
10
20
30
40
50
0
100
200
300
time

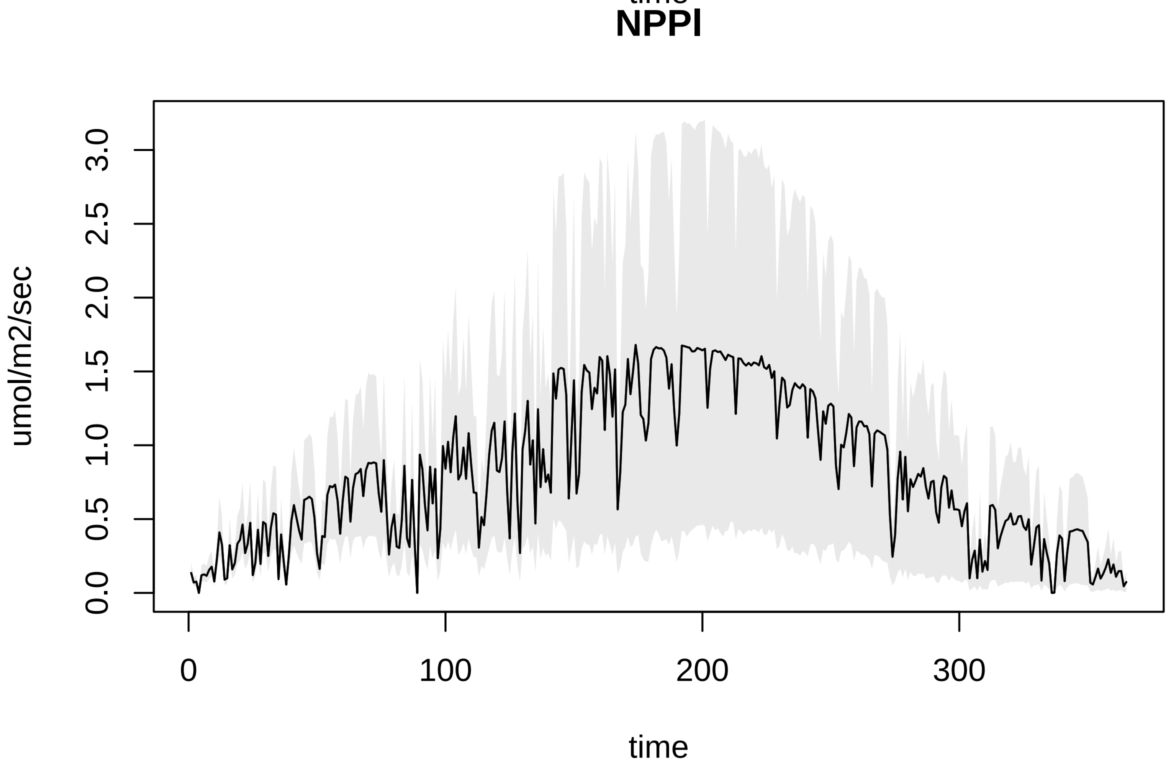

LAI
m2/m2
0
1
2
3
4
5
0
100
200
300
time

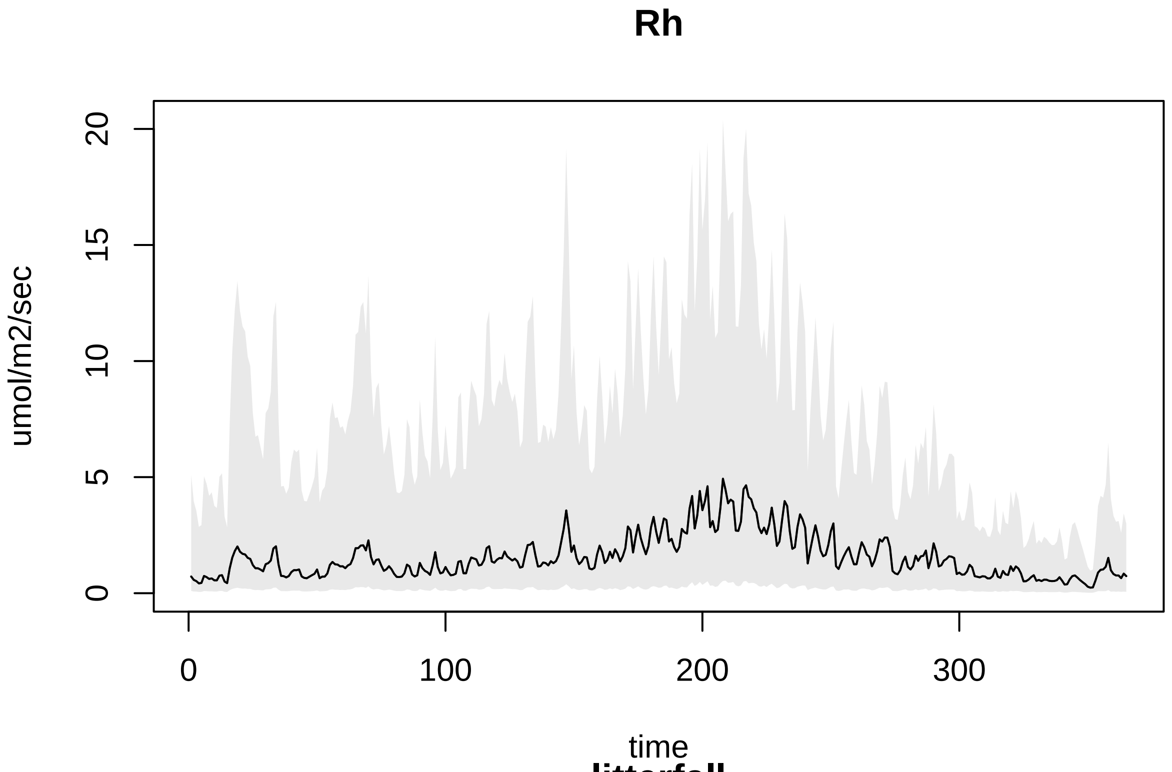

GPP
umol/m2/sec
0
5
10
15
0
100
200
300
time

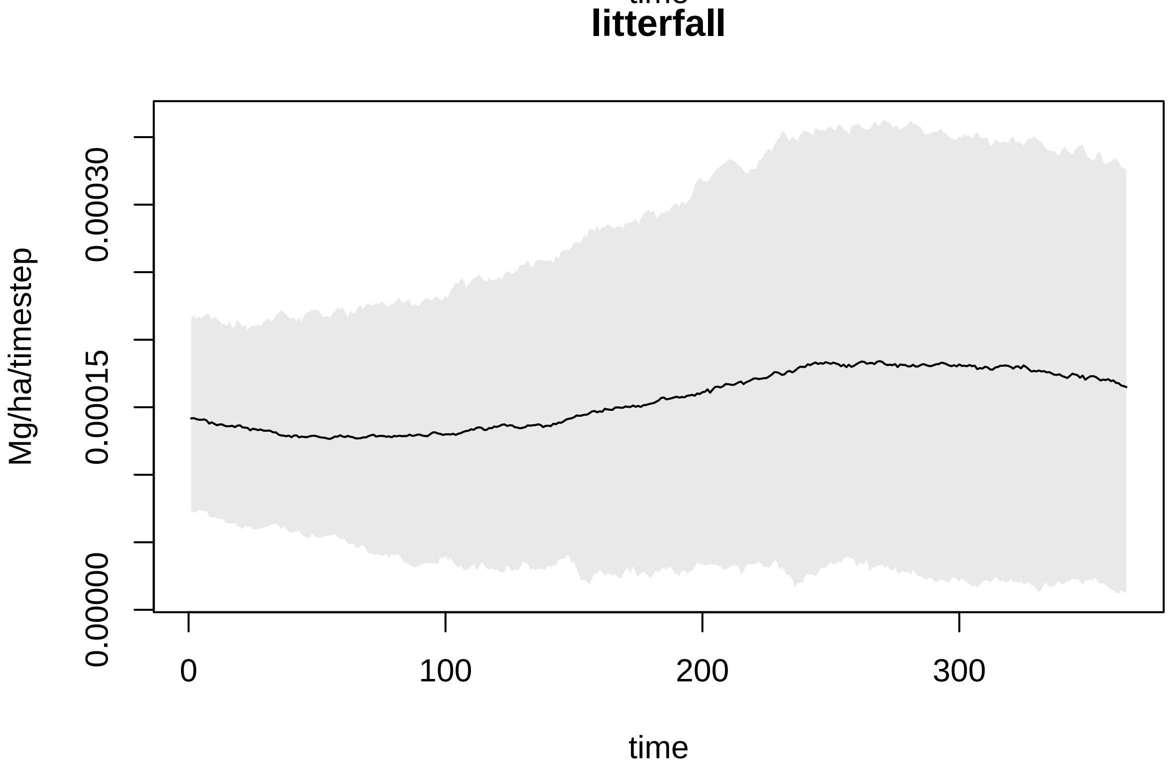

NEP
umol/m2/sec
5
0
−5
−10
−15
0
100
200
300
time

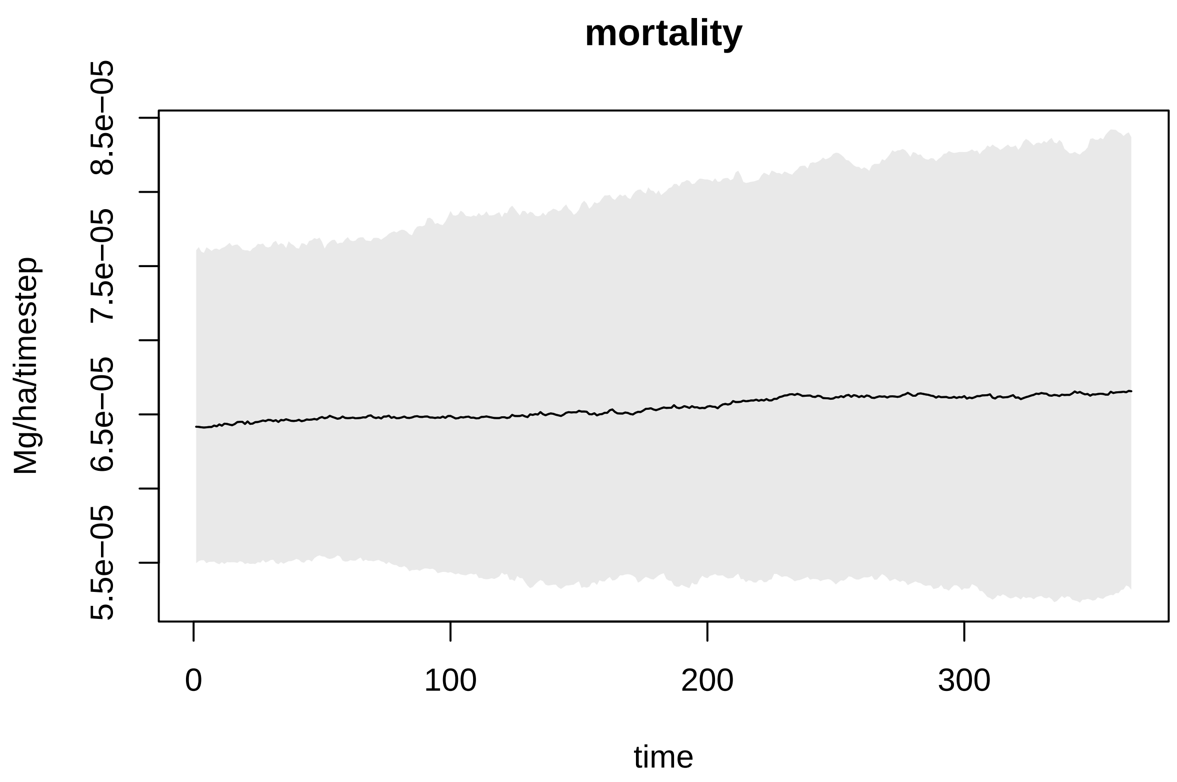

Ra
umol/m2/sec
8
6
4
2
0
0
100
200
300
time


NPPw
umol/m2/sec
0
1
2
3
4
5
0
100
200
300
time


NPPl
umol/m2/sec
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
100
200
300
time


Rh
umol/m2/sec
0
5
10
15
20
0
100
200
300
time


litterfall
Mg/ha/timestep
0.00000
0.00015
0.00030
0
100
200
300
time



MODEL VALIDATION: MODIS LAI

Next, let's load and clean the MODIS LAI data. The following is necessary to understand the data.

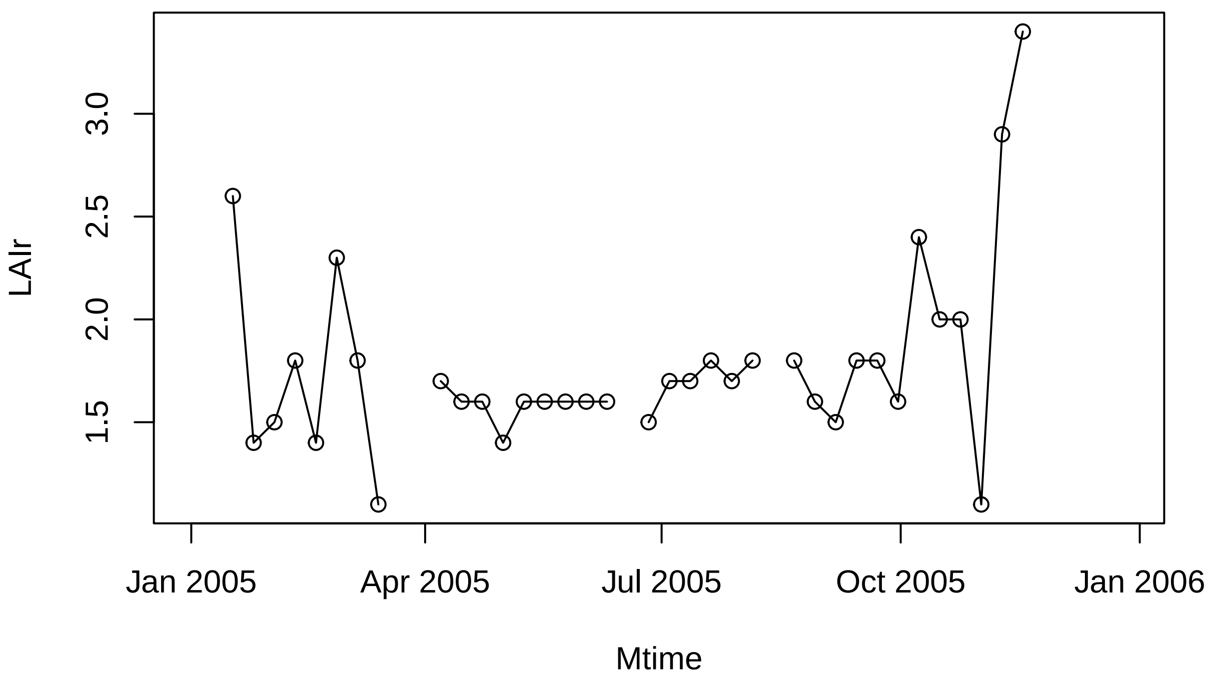
- Each row is a different variable at a different time. The key variables are LAI estimate, QC code, and standard devivation on the LAI estimate.
- The date for each row variable is embedded in the file name in column 1
- The value for each variable is in column 10
- Each LAI estimate is an 8-day average.

```
## open MODIS data and extract remotely-sensed LAI (LAIr),
## the standard deviation, and the QAQC flags
MODIS = read.csv("data/Lat44.45230Lon-121.55740Start2000-01-01End2012-12-31_MOD15A2.asc",
                 header = FALSE, as.is = TRUE, na.string = "-3000")
MODvar = substr(MODIS[, 1], 43, 52)                                 ## extract variable names
Mtime.raw = substr(MODIS[which(MODvar == "Lai_1km"), 3], 2, 8)  ## extract and process dates
Mtime = as.Date(Mtime.raw, format="%Y%j")
QC = MODIS[which(MODvar == "FparLai_QC"), 10]                       ## extract Quality Control flags
LAIr = MODIS[which(MODvar == "Lai_1km"), 10] * 0.1                  ## extract LAI data, Convert units
LAIr.sd = MODIS[which(MODvar == "LaiStdDev_"), 10] * 0.1            ## extract LAI SD, Convert units

## apply QC
LAIr[QC > 1] = NA
LAIr.sd[QC > 1] = NA
LAIr.sd[LAIr.sd < 0.66] = 0.66  ## use a published validation estimate to set floor on LAI uncertainty

## select year
yr = grep("2005", Mtime.raw)
```

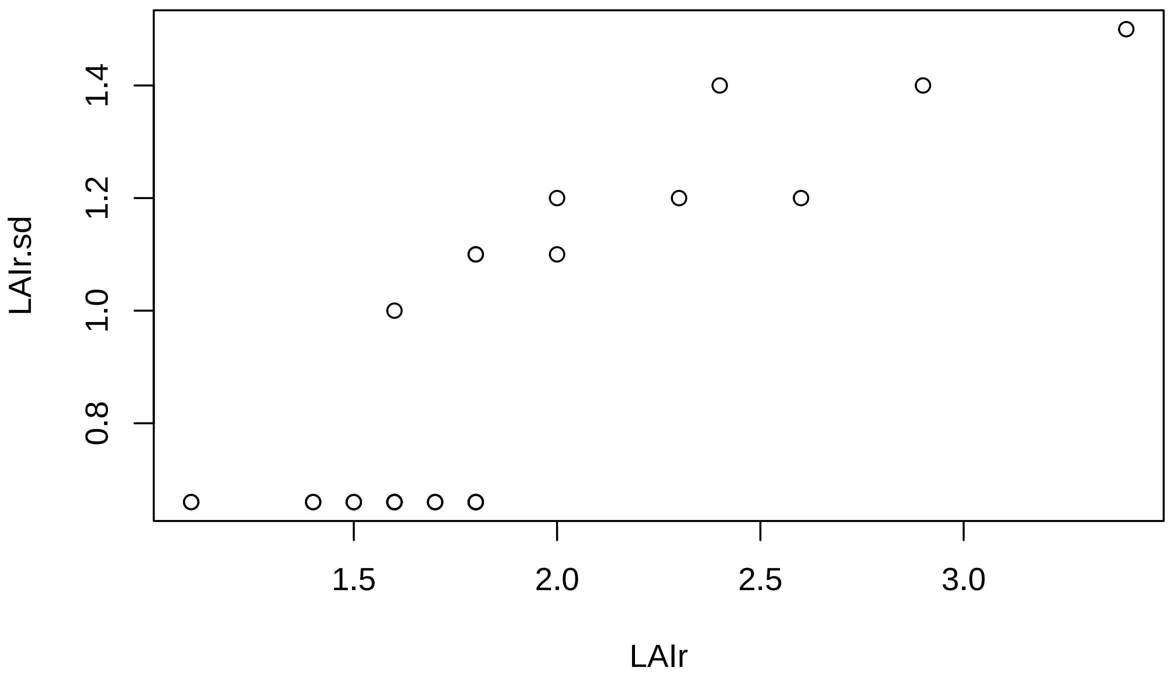
```
LAIr = LAIr[yr]
LAIr.sd = LAIr.sd[yr]
QC = QC[yr]
Mtime = Mtime[yr]
```

Plot a time-series of LAI and the relationship between the MODIS-LAI and its standard deviation. Remember that this is an evergreen forest, so we're not expecting much true variability or seasonality in LAI.

```
plot(Mtime,LAIr,type = 'o')
```



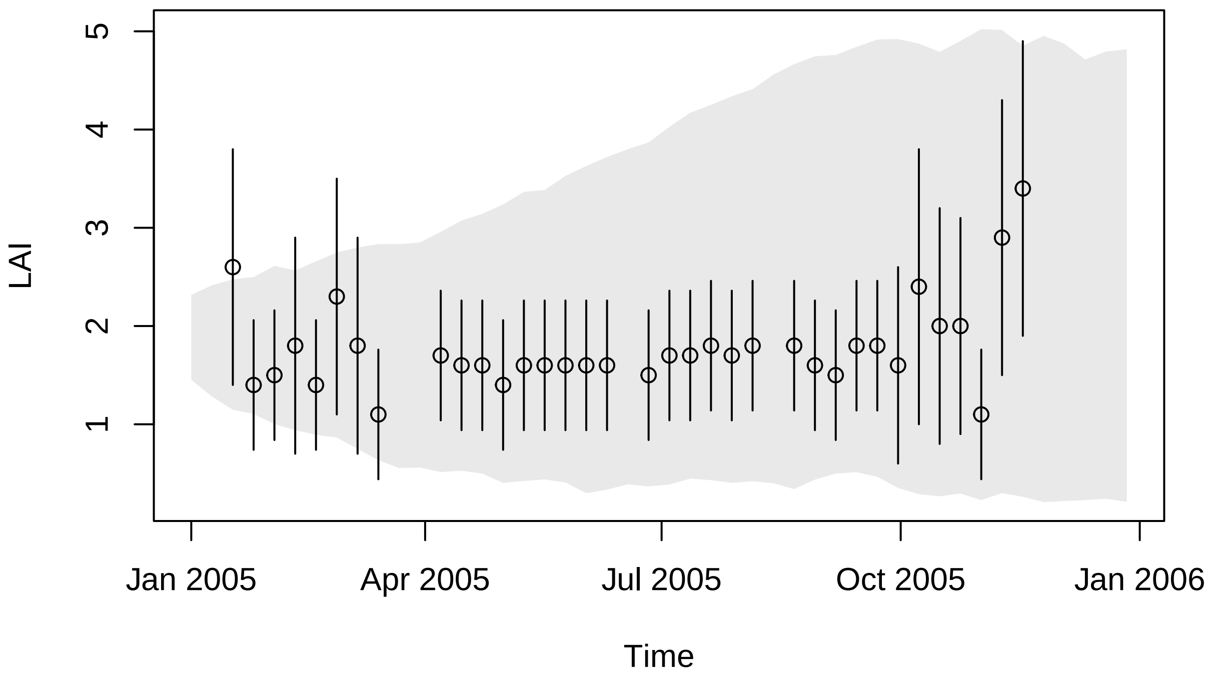
```
plot(LAIr, LAIr.sd)
```



Remember that the model output is at the 30-minute time-step but the MODIS LAI is an 8-day average. To be able to compare model to data, let's calculate the time-averaged LAI from the model for the same periods as MODIS.

In the 17-day period, there could be three MODIS LAI values but the first two do not pass the QC test so they are not used. Therefore, only the MODIS LAI estimate for the end of the 17-day period is shown in the plot below. Once you increase the length of the model run, more observations will show up.

```r
## Calculate model ensemble means for same periods
window = rep(1:(length(yr)), each=48*8, length=nt)
LAIm = t(apply(output[, , 4], 2, tapply, window,mean))
LAIm.ci  = apply(LAIm, 2, quantile, c(0.025, 0.5, 0.975))

## plot model and observations
Msel = 1:ncol(LAIm.ci)
plot(Mtime[Msel], LAIm.ci[2,], ylab = "LAI", xlab = "Time",
     ylim=range(c(range(LAIm.ci), range(LAIr, na.rm=TRUE))), type = 'n')
ciEnvelope(Mtime[Msel], LAIm.ci[1, ], LAIm.ci[3, ],col = col.alpha("lightGrey", 0.5))
points(Mtime, LAIr)
for(i in 1:length(LAIr)){
  if(!is.na(QC[i])){
    lines(rep(Mtime[i], 2), LAIr[i]+c(-1, 1) * LAIr.sd[i])
  }
}
```

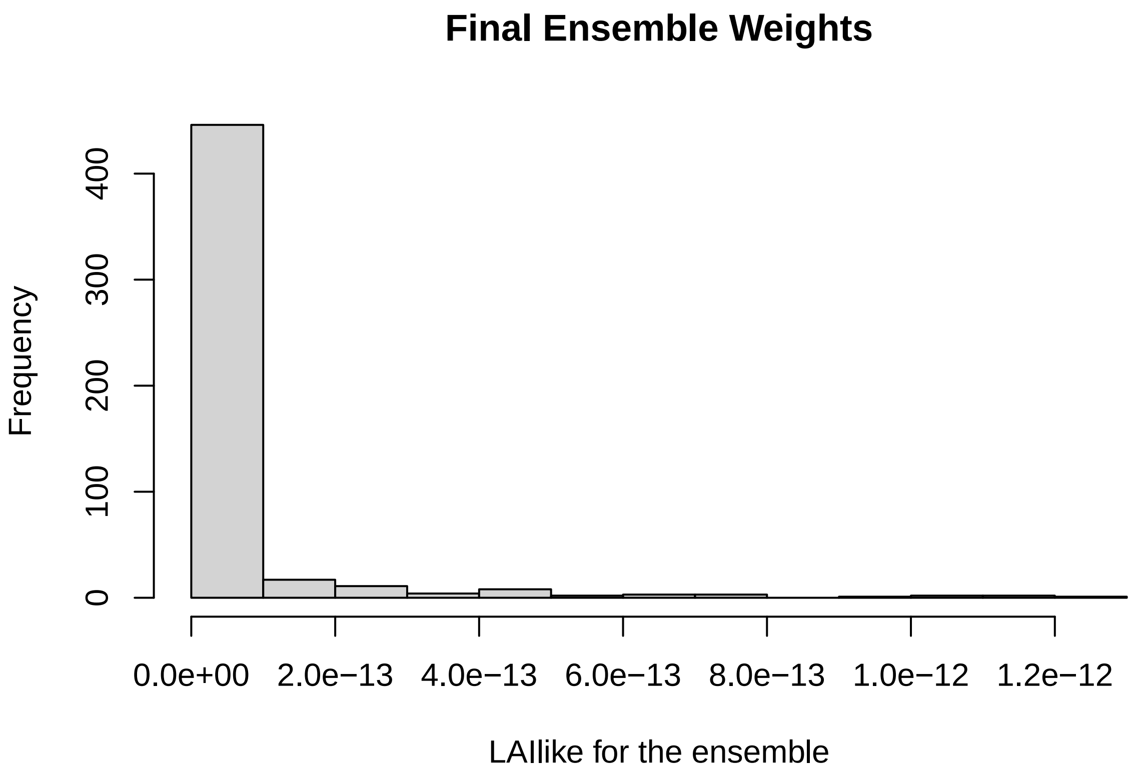


Non-resampling Particle Filter

Given this ensemble we can next apply a **non-resampling particle filter** to this existing ensemble forecast by calculating the cumulative likelihood of the observations for each ensemble member. Note that in the code below LAIm is the model (initial ensemble), LAIpf is the non-resampling particle filter, and LAIr is the remotely sensed observations.

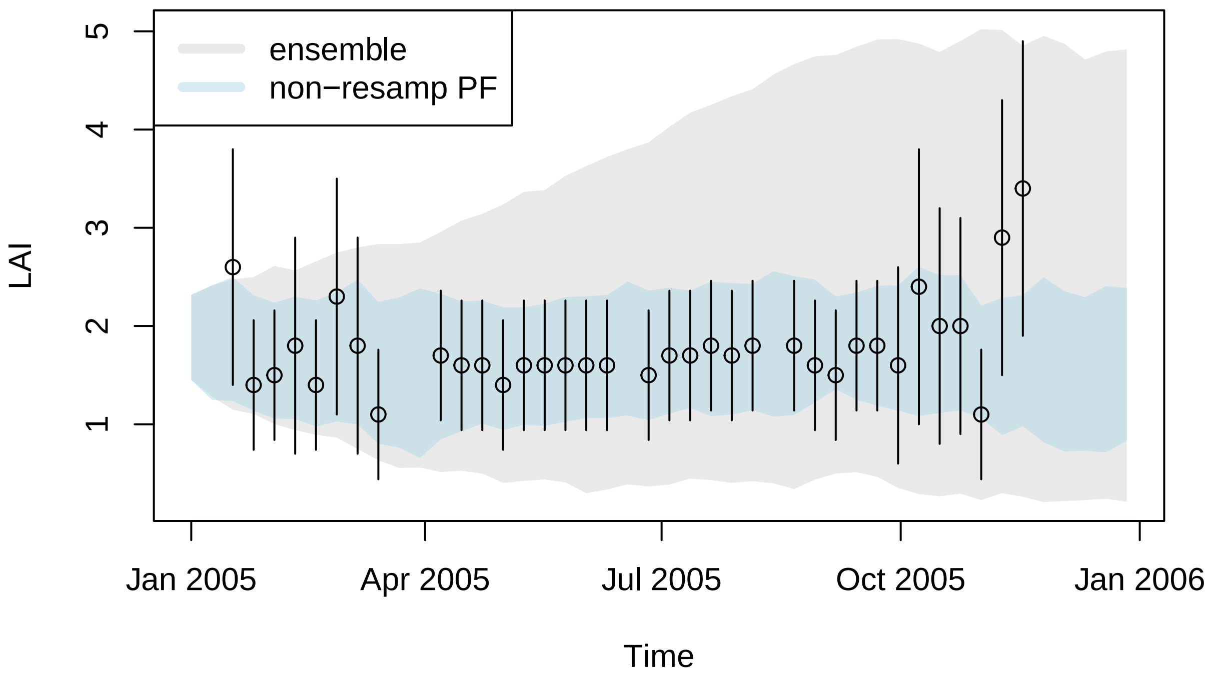
First, calculate the likelihood of each ensemble member

```
## calculate the cumulative likelihoods
## to be used as PF weights
LAIlike = array(NA, dim(LAIm))  ## storage, same size as model [ensemble, MODIS time points]
sel = 1:ncol(LAIm.ci)
for(i in 1:ne){
  LAIlike[i, ] = dnorm(LAIm[i, ], LAIr[sel], LAIr.sd[sel], log = TRUE)  ## calculate log likelihoods
  LAIlike[i, is.na(LAIlike[i, ])] = 0       ## missing data as weight 1; log(1)=0
  LAIlike[i, ] = exp(cumsum(LAIlike[i, ]))  ## convert to cumulative log likelihood and take out of log
}
hist(LAIlike[,ncol(LAIlike)],main="Final Ensemble Weights", xlab = "LAIlike for the ensemble")
```



Plot the forecast using the weights from LAIlike. This code calculates the weighted quantile using the `wtd.quantile()` function.

```
## Non-resampling Particle Filter
## calculation of CI
nobs = ncol(LAIlike)                        ## number of observations
LAIpf = matrix(NA, 3, nobs)                 ## storage [intervals,time]
wbar = apply(LAIlike, 2, mean)              ## mean weight at each time point
for(i in 1:nobs){
  ## calculate weighted median and CI
  LAIpf[, i] = wtd.quantile(LAIm[, i], LAIlike[, i] / wbar[i], c(0.025, 0.5, 0.975))
}

## plot original ensemble and PF with data
col.pf   = c(col.alpha("lightGrey", 0.5), col.alpha("lightBlue", 0.5),
             col.alpha("lightGreen", 0.5))                     ## color sequence
names.pf = c("ensemble", "non-resamp PF", "resamp PF")         ## legend names
plot(Mtime[Msel],LAIm.ci[2, ], ylim = range(c(range(LAIm.ci), range(LAIr, na.rm = TRUE))),
     type='n',ylab="LAI",xlab="Time")
ciEnvelope(Mtime[Msel], LAIm.ci[1, ], LAIm.ci[3, ], col = col.pf[1])  ## original ensemble
ciEnvelope(Mtime[Msel], LAIpf[1, ], LAIpf[3, ], col = col.pf[2])      ## non-resampling Particle Filter
points(Mtime, LAIr)                                                   ## observations
for(i in 1:length(LAIr)){                                             ## observation uncertainty
  if(!is.na(QC[i])){
    lines(rep(Mtime[i], 2), LAIr[i]+c(-1, 1) * LAIr.sd[i])            ## data is +/- 1 SD; NOT 95%
  }
}
legend("topleft", legend = names.pf[1:2], col = col.pf[1:2], lwd = 5)
```



Resampling Particle Filter

Before we move on to the resampling particle filter, we need to define an ancillary function, `update.params`, that will help us re-assign the parameter values to different ensemble members when we resample from the ensemble members.

```
update.params <- function(params, index){
  params$falloc = params$falloc[index, ]
  params$SLA = params$SLA[index]
  params$alpha = params$alpha[index]
  params$Q10 = params$Q10[index]
  params$Rbasal = params$Rbasal[index]
  params$litterfall = params$litterfall[index]
  params$mortality = params$mortality[index]
  params$sigma.leaf = params$sigma.leaf[index]
  params$sigma.stem = params$sigma.stem[index]
  params$sigma.soil = params$sigma.soil[index]
  return(params)
}
```

Finally, let’s implement the resampling particle filter. The code for this is organized in a way similar to the Kalman Filter activity – we loop over time and alternate between a forecast step and an analysis step. Since the observations only occur every 8 days, we only repeat the analysis step every 8 days (which for a 30 min timestep is every 48*8 timesteps).

```
hist.params = list()                 ## since we resample parameters, create a record (history) of what val
hist.params[[1]] = params            ## initialize with original parameters
X = X.orig                           ## reset state to the initial values, not the final values from the pr

### resampling particle filter
sample = 1                             ## counter
```

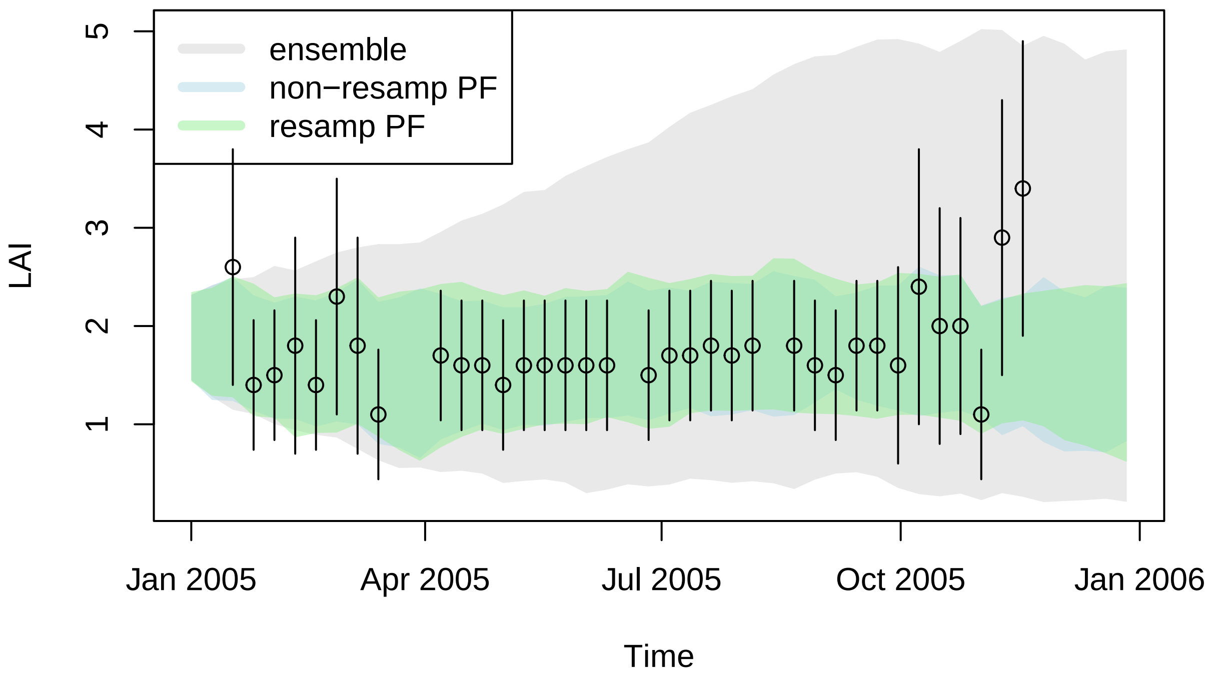
```
for(t in 1:nt){

  ## forward step
  output[t, , ]=SSEM(X, params, inputs[t, ])
  X = output[t, , 1:3]

  ## analysis step
  if(t%%(48*8) == 0){             ## if at data frequency (remainder == 0)
    sample = sample + 1           ## increment counter
    print(sample)
    if(!is.na(LAIr[sample])){     ## if observation is present

      ## calulate Likelihood (weights)
      Lm = apply(output[t + 1-(48 * 8):1, ,4], 2, mean)## average model LAI over obs period
      wt = dnorm(LAIr[sample], Lm, LAIr.sd[sample])    ## calculate likelihood (weight)

      ## resample ensemble members in proportion to their weight
      index = sample.int(ne, ne, replace = TRUE, prob = wt)

      X = X[index, ]                                  ## update state
      params = update.params(params, index)           ## update parameters
    }
    hist.params[[sample+1]] = params                 ## save parameters
  }

}
```

```
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
## [1] 35
## [1] 36
## [1] 37
## [1] 38
## [1] 39
## [1] 40
## [1] 41
## [1] 42
## [1] 43
## [1] 44
## [1] 45
## [1] 46
```

```
## save all the output
#save(output,output.ensemble,LAIlike,hist.params,inputs,file="Ex10.output.RData")
```

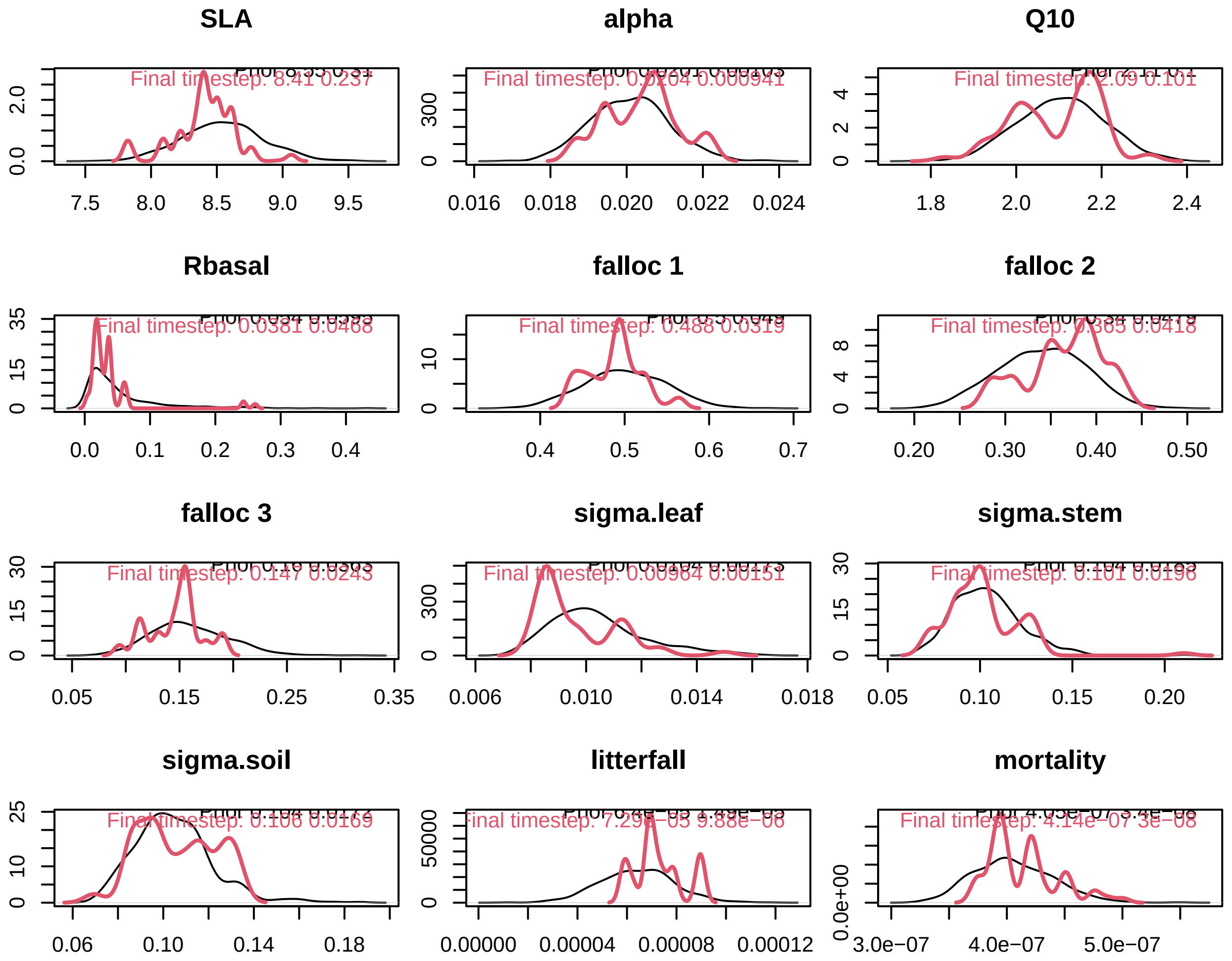
Next, let's compare the resampling particle filter to the ensemble and the non-resampling PF.

```
## Extract and summarize LAI (pr = PF, resampling)
LAIpr = t(apply(output[, , 4], 2, tapply, window, mean))     ## summarize PF LAI at measurement frequen
LAIpr.ci = apply(LAIpr, 2, quantile, c(0.025,0.5,0.975))     ## calculate median and CI

## plot time-series
plot(Mtime[Msel], LAIm.ci[2, ],
     ylim = range(c(range(LAIm.ci), range(LAIr, na.rm = TRUE))),
     type='n', ylab = "LAI", xlab = "Time")
ciEnvelope(Mtime[Msel], LAIm.ci[1, ], LAIm.ci[3, ], col=col.pf[1])     ## original ensemble
ciEnvelope(Mtime[Msel], LAIpf[1, ], LAIpf[3, ], col = col.pf[2])       ## non-resampling PF
ciEnvelope(Mtime[Msel], LAIpr.ci[1, ], LAIpr.ci[3, ], col = col.pf[3]) ## resampling PF
points(Mtime, LAIr)                                                    ## data
for(i in 1:length(LAIr)){
  if(!is.na(QC[i])){
    lines(rep(Mtime[i], 2), LAIr[i] + c(-1, 1) * LAIr.sd[i])
  }
}
legend("topleft",legend = names.pf, col=col.pf, lwd=5)
```



Finally, the resampling PF also updates the parameter distributions, so let's plot the posterior parameter distributions. In doing so it is important to remember that the technique we used does not introduce new parameter draws into the ensemble, so it is possible for the parameter distributions to become degenerate, placing too much weight on a small number of parameter combinations. More advanced methods exist for proposing new parameter values but they're beyond the scope of this primer.

Note, the **digits on the posterior and prior plots are the mean and standard deviation** for the prior (top row, black line), and the posterior (second row, red line)



Exercises

Question 1: Increase the number of particles to 500. How does this change your results?

- In this case I ran the code first with 50 particles and then with 1000 particles. Specifically, with 1000 particles, the time series visualizations of Leaf Biomass (Bleaf), Structural Biomass (Bwood), Soil Organic Matter (BSOM), Leaf Area Index (LAI), litterfall, and mortality exhibit smoother curves and narrower confidence intervals. This enhancement indicates more stable and precise estimations over time, reflecting a reduction in the variability and uncertainty inherent in the model predictions for these variables. However, for other fluxes and state variables such as Gross Primary Productivity (GPP), Net Ecosystem Exchange (NEP), Autotrophic Respiration (Ra), Net Primary Productivity of wood (NPPw), and leaves (NPPl), and Heterotrophic Respiration (Rh), no significant changes were detected. This suggests that these variables might be less sensitive to particle count. In the case of the non resampling particle filter, the results show that the 1000 particles iteration has better estimation results than the 50 particles iteration. We also observe this behavior in the resampling particle filter. The distribution of the Final Ensmable Weights present a lot of differences between the models

Question 2: After 17-days, how different are the simulations with and without particle resampling?

- In general, after 17 days, the simulations with and without particle resampling are very similar. Observing the confidence intervals we see that the non resampling particle filter have smaller regions of uncertainty, but the resampling particle filter has smoother confidence intervals, which implies that resampling helps maintain a consistent ensemble by potentially eliminating outlier particles that could cause abrupt changes. With only 17 days of simulation is difficult to find significant differences between the two methods.

Question 3: With the particles at 500, increase the length of the simulation from 17 days to 365 days. Do the results match your expectations?

- The results match my expectations. We can clearly observe that the ensamble uncertainty increases due to the variations of the data since the ecosystem experiences a wider range of environmental conditions due to seasonal changes. Anyway, the non resampling particle filter is a litle bit better describing the data with higher variations.

Question 4: After 365-days, how different are the simulations with and without particle resampling?

- After 365 days, the simulations with and without particle resampling are very similar. Anyway, if we observe the confidence intervals we can see that the non resampling particle filter has better capacity to describe the data with higher variations. This make sense, since the resampling particle filter removes poor-performing particles. This means that the resampling tends to focus the ensemble on the most likely state trajectories, potentially increasing the accuracy of the ensemble, but loosing the natural variability of the system.

Question 5: How well are the three simulations able to constrain the other fluxes and state variables that are output by the model (which should be updated in proportion to how much they covary with LAI)?

- Basic Ensemble Forecast: The basic ensamble provides a baseline by projecting the state and fluxes based on the model alone. It has the widest spread of outcomes but is making a good job capturing the range.
- Non-resampling Particle Filter: This method adjusts the weight of each particle based on the fit to observed LAI data but keeps all particles. It improves the ensemble mean accuracy but retain some error in the trajectories and more variation in the confidence intervals. This could affect how well it constrains state variables that are strongly correlated with LAI.
- Resampling Particle Filter: By resampling based on weights from the fit to LAI, this method improved the model outputs with observed data. It offers the best constraint on the related state variables. However, this is reducing ensemble diversity, which might limit the model's ability to explore possible future states like we observe in the simulations.

Question 6: Which parameters were most constrained by LAI? Does this make sense?

- Based on the observations, the Specific Leaf Area (SLA), alpha, and the basal rate of soil respiration (R_basal) appear to be the parameters most constrained by LAI. This correlation is logical given their roles in ecosystem processes. The SLA, which is the ratio of leaf area to leaf dry mass, directly influences LAI calculations. The alpha parameter, representing light use efficiency, is crucial for converting absorbed light into biomass, which is directly linked to LAI through its impact on photosynthetic capacity. However, the connection between R_basal and LAI is more indirect.

Extra Credit

Question 7: Convert the Analysis step of the resampling PF to an EnKF, rerun and compare to the previous runs. For the sake of keeping this activity simple, don't worry about updating parameters.

Hints: (a) The ensemble Forecast step stays the same; (b) Exercise 09 contains the `KalmanAnalysis` function that you need to be able to perform the Analysis step (instead of calculating Likelihoods and weights); (c) you can calculate mu.f and P.f from the ensemble forecast sample mean and cov; (d) The observation operator, H, should be a 1x12 matrix that's all zeros except for a 1 in the column that matches the observed data (LAI) in the model output; (e) Once you have an update mu.a and P.a, you either need to sample new ensemble members from that (e.g. rmvnorm), or use ensemble adjustment to nudge your current ensemble members to the correct mean and covariance.

Applying this to your own forecast

To extend the particle filter and Ensemble Kalman Filter approaches to develop iterative forecasts for you own models, you would need to apply the following steps:

1) Develop an ensemble forecast for your own model that propagates uncertainties. See the "Applying this to your own forecast" section in the Chapter 11 activity. If your model exists outside of R (e.g. compiled computer code), you'll want to make sure that this model has the capacity to restart from new initial conditions.
2) Load and organize the data you'll be using to constrain you model, and understanding the mapping between your model's state variables and your observations (in EnKF this means constructing the matrix H).

3A) If you are using an EnKF, the Analysis code you developed above should work out-of-the-box for many problems. For very large data problems you may need to address challenges such as inverting large matrices and localization.

3B) If you are using PF, you will need to construct the Likelihood (i.e. data model) appropriate for your data constraints.