Demonstration of Diagnostic Tools

Contents

This tutorial demonstrates use of the R functions that we have developed to evaluate model congruency for models fitted with `ppmlasso`. Hereafter, we refer to these as diagnostic tools.

1 Prerequisites

1.1 Grid of covariate data

Fitting a species distribution model requires input of some covariates that we use to explain the observed species pattern. These may be environmental variables, or variables related to the sampling process, which we refer to as *observer bias* variables. For `ppmlasso`, we must supply this information as a data frame, with rows corresponding to locations set along a regular grid of points at a desired pixel resolution. For each row of the data frame, the values may be raw or summarised values of the variables. For more information, see Renner et al. (2015).

We will make use of the same grid used in the main manuscript, which contains variables at a 4km resolution throughout mainland France. These variables include:

- `X`: The x-coordinate in LAEA in metres
- `Y`: The y-coordinate in LAEA in metres
- `temp`: mean annual temperature, from Worldclim
- `prec`: mean annual precipitation rate, from Worldclim
- `hp`: human population density, from the Gridded Population of the World v4 database

- `percagri`: percentage of the landscape that is used for agriculture from Corine Land Cover 2012
- `perc311`: percentage of the landscape that is broad-leaved forest from Corine Land Cover 2012
- `perc312`: percentage of the landscape that is coniferous forest from Corine Land Cover 2012
- `perch10`: percentage of the landscape was covered by forest in 1910, from the 2013 HILDA database

```
load("Env_Fr_4km.RData") # loads env.4km.Fr, our grid
env = env.4km.Fr # rename grid
```

We choose to model on the km scale instead of the metre scale, so we divide the coordinates by 1000, as follows:

```
env$X = env$X/1000
env$Y = env$Y/1000
head(env)
```

```
##         X    Y     temp     prec        hp percagri      perc311  perc312
## 1573 3425 2274 63.57307 1178.204  3.064148        0  0.882492741 0.000000
## 1574 3429 2274 79.79453 1109.074  1.557800        0 53.273854305 0.000000
## 1575 3433 2274 78.57307 1107.901  1.754362        0  0.004917942 0.000000
## 1576 3437 2274 95.67902 1032.209  2.433250        0  1.910664267 0.000000
## 1577 3441 2274 81.75902 1081.475  4.513101        0 15.859292274 0.000000
## 1578 3445 2274 74.62705 1107.033 10.643269        0  0.117075533 2.476746
##       perch10
## 1573 23.92531
## 1574 45.96375
## 1575 27.61984
## 1576 42.20135
## 1577 66.19803
## 1578 75.00000
```

The models we fit require covariates to be measured at each location in the grid. If the grid has missing information for some locations, we recommend these locations be removed from the grid before proceeding any further.

1.2 Fitting a regularisation path using the full data set

Our diagnostic tools use the output from the optimal model fitted to the full set of data as a reference point for comparison. As the real point data for the stag beetle `Lucanus servus` may be sensitive, we demonstrate this on a set of 2449 points simulated from the intensity map using the full set of data in the main manuscript.

```
load("Simulated Lucanus points.RData")
sp.dat = data.frame(X = simX, Y = simY)
head(sp.dat)
```

```
##          X        Y
## 1 3805.347 2684.591
## 2 4040.644 2844.360
## 3 4059.829 2916.176
## 4 3713.923 2178.748
## 5 4018.739 2941.974
## 6 3946.906 2720.301
```

```
dim(sp.dat)
```

```
## [1] 2449    2
```

We now specify the formula we use in the model. In the manuscript, we include temperature and precipation as quadratic terms, the percentage landscape variables as linear terms, and the natural logarithm of the

human population density.

```
form = ~ poly(temp, prec, degree = 2, raw = TRUE) + perch10 + perc311 + perc312 +
    percagri + log(hp) # formula for ppmlasso
```

We are now ready to fit the regularisation path of models using `ppmlasso`. We now load the `ppmlasso` and `data.table` packages, as well as some new functions which will appear in an update of the `ppmlasso` package.

```
library(ppmlasso)
library(data.table)
source("ppmlasso_functions.R")
source("New functions.R")
```

We prepare the data for fitting with the `ppmdat()` function. Here, we specify the following arguments:

- `sp.xy`: a data frame containing the x- and y-coordinates of the species
- `sp.scale`: the spatial resolution at which to fit the model
- `back.xy`: the environmental grid
- `coord`: the names of the vectors containing the x- and y-coordinates in the `sp.xy` and `back.xy` data frames
- `sp.file`: an optional previously created file of species data to import
- `quad.file`: an optional previously created environmental grid to import
- `file.name`: by default, this function saves the output as an `.RData` file, and this argument specifies the name of the file.

```
data.po  = ppmdat(sp.xy = sp.dat, sp.scale = 4, back.xy = env, coord = c("X","Y"),
                  sp.file = NA, quad.file = NA, file.name = "TestPPM")
```

We now may fit a regularisation path of models fitted with increasing lasso penalties with the `ppmlasso()` function. See the documentation of the `ppmlasso()` function in R for full details of the arguments and the output. In essense, this function will fit `n.fits` Poisson point process models with increasing lasso penalties, and choose among them the model which optimises some `criterion` (BIC by default).

```
fitAll = ppmlasso(form, sp.xy = sp.dat, env.grid = env, sp.scale = 4, data = data.po,
                n.fits = 200, criterion = "bic", max.it = 10, lamb = NA)
```

Hence, the obejct `fitAll` contains all of the information from the optimal model fitted to the full set of 2449 points. The coefficients are contained in a vector called `beta`:

```
fitAll$beta
```

```
##   Intercept        temp      temp^2        prec   temp*prec      prec^2
## -5.90566920  8.69043000 -6.06275267  0.00000000 -1.32583534  1.55411290
##     perch10     perc311     perc312    percagri     log(hp)
##  0.27599577  0.04140629 -0.05556798 -0.10763424  0.75141565
```

We may also want to visualise a map of the predicted intensities. Because we include an observer bias variable in our model (the natural logarithm of human population density), we will want to correct for this bias using the method of Warton, Renner, and Ramp (2013). With this method, we essentially neutralise the effect of any observer bias variables by setting them equal to some constant for prediciton. To do this, we will initialise a new data frame `pred.data` to be equal to the grid used to fit the model:

```
pred.data = env
```

We then replace the observer bias variable (in this case `hp`) with a constant. Here, we will replace it with the mean human population density throughout the region.

```
bias.var = c("hp")
bias.col = match(bias.var, names(pred.data))
```

```
for (v in bias.col)
{
  pred.data[,v] = mean(pred.data[,v])
}
```

We can now calculate the corrected intensities with the `predict.ppmlasso()` function.

```
correctedmu = predict.ppmlasso(fitAll, newdata = pred.data)
```

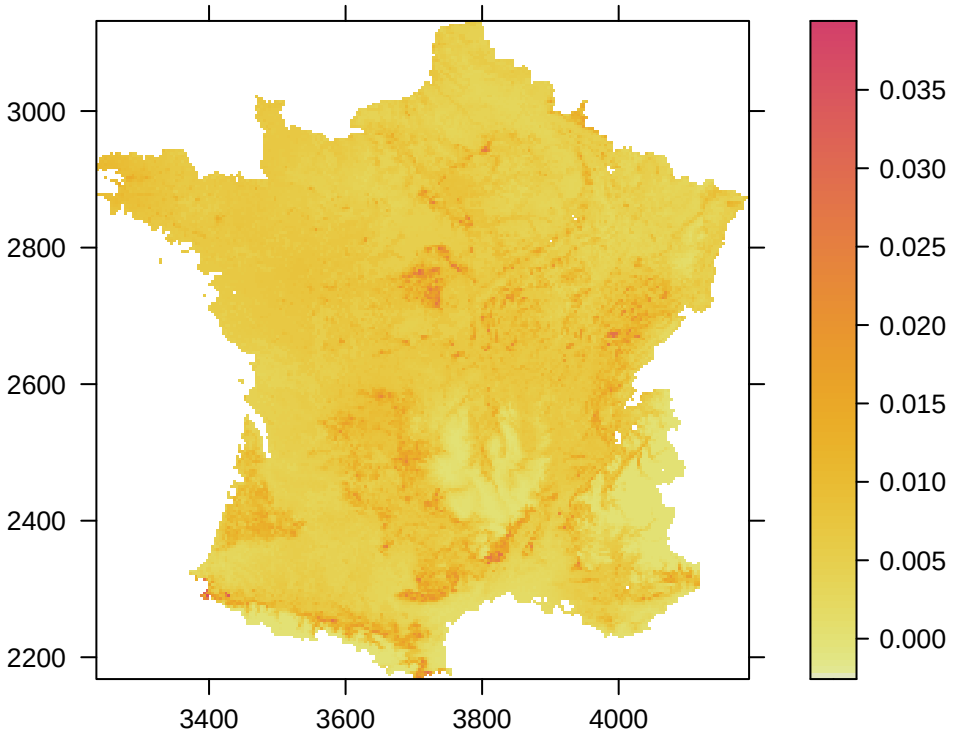
We can map these intensities using the `levelplot()` function of the `lattice` package. We will load a pre-defined color scheme for the map from the file `ColorScheme.RData`.

```
library(lattice)
```

```
##
## Attaching package: 'lattice'
```

```
## The following object is masked from 'package:spatstat':
##
##     panel.histogram
```

```
load("ColorScheme.RData")
levelplot(correctedmu ~ env$X + env$Y, cuts = 100, asp = "iso",
          col.regions = col2, xlab = "", ylab = "")
```



2 Fitting models to random subsets of points

Our diagnostic tools rely on taking random subsets of the observed species data. While the manuscript uses 1000 simulations, we only demonstrate one here for a single subset size. Full code to conduct 1000 simulations across all considered subsample sizes are included in the R script `SimulationDemo.R`.

Let us consider a subset of size 1000, stored in a data frame called `sp.sim`.

```
n.pts = 1000
sim = 1
set.seed(sim)
pts_keep = sample(1:dim(sp.dat)[1], n.pts)
sp.sim = sp.dat[pts_keep,]
```

We now fit a regularisation path to the subset of 1000 points.

```
data.po  = ppmdat(sp.xy = sp.sim, sp.scale = 4, back.xy = env, coord = c("X","Y"),
                  sp.file = NA, quad.file = NA, file.name = "TestPPM")
fit.sim = ppmlasso(form, sp.xy = sp.sim, env.grid = env, sp.scale = 4,
                   data = data.po, n.fits = 200, criterion = "bic", max.it = 10,
                   lamb = NA)
```

For each simulated subset of a nominal size, we store the fitted coefficients in an array called `beta.sims`, and the means and standard deviations of the covariates necessary for rescaling prior to making predictions in arrays called `s.means.sims` and `s.sds.sims`, respectively.

First, we initialise these arrays, as follows:

```
n_var = 11 # for creating matrix dimensions for simulations (10 variables plus 1 intercept)
n.sim = 1000 # number of simulations
subset_sizes = c(1000, 500, 200, 100, 50) # subset sizes

beta.sims = array(NA, dim = c(length(subset_sizes), n.sim, n_var))
s.means.sims = s.sds.sims = array(NA, dim = c(length(subset_sizes), n.sim, (n_var - 1)))
# n_var minus the "intercept" (-1)
dimnames(beta.sims)[[1]] = subset_sizes
dimnames(beta.sims)[[3]] = names(fitAll$beta)
```

We now fill the corresponding elements of these arrays for our subset of size 1000:

```
beta.sims[1,sim,] = fit.sim$beta
s.means.sims[1,sim,] = fit.sim$s.means
s.sds.sims[1,sim,] = fit.sim$s.sds
```

We will load the results of 1000 simulations from the file `SimulatedData.RData` across the subset sizes ($n = 50, 100, 200, 500, 1000$) to demonstrate the use of the developed tools to diagnose model congruency.

```
load("SimulatedData.RData")
```

This file contains the following objects:

- `beta.sims`: a $5 \times 1000 \times 11$ array of the fitted coefficients $\hat{\boldsymbol{\beta}}$ and $\hat{\boldsymbol{\gamma}}$. These dimensions correspond to 5 subset sizes ($n = 50, 100, 200, 500, 1000$), 1000 random subsamples, and 11 covariates. Crucially, the array is given names for the first dimension corresponding to the subset sizes as well as for the third dimension corresponding to the names of the covariates, as these names are used in plots to follow:

```
dimnames(beta.sims)
```

```
## [[1]]
## [1] "1000" "500"  "200"  "100"  "50"
##
```

```
## [[2]]
## NULL
##
## [[3]]
##  [1] "Intercept" "temp"      "temp^2"    "prec"      "temp*prec" "prec^2"
##  [7] "perch10"   "perc311"   "perc312"   "percagri"  "log(hp)"
```

- `s.means.sims`: a $5 \times 1000 \times 10$ array of the means of the 10 covariates (excluding the intercept) for the 1000 simulations across the 5 subsets.
- `s.sds.sims`: a $5 \times 1000 \times 10$ array of the standard deviations of the 10 covariates (excluding the intercept) for the 1000 simulations across the 5 subsets.

These arrays of the means and standard deviations are necessary as the `ppmlasso()` function rescales all covariates to have mean 0 and variance 1 before fitting models to ensure that the lasso penalty is applied appropriately. Hence, we need to record the means and standard deviations to appropriately rescale new input data for prediction.

3 Diagnostic tools for model intensity

Now that we have loaded the output from the models fitted to each of the random subsets, we can demonstrate the tools we have developed to assess model stability. These tools are available in the script `DiagnosticFunctions.R`.

Our first set of tools compares the fitted intensity of the model fitted to the full set of points to the rescaled intensities of the models fitted to the random subsets. The `compute_intensity()` function calculates the raw and rescaled fitted intensities for all the subset models. The function takes the following input:

- `fitN`: the model fitted to the full set of points
- `simbetas`: the array containing the coefficients of the models fitted to the random subsets
- `simmeans`: the array containing the covariate means of the models fitted to the random subsets
- `simsds`: the array containing the covariate standard deviations of the models fitted to the random subsets
- `PredData`: the covariates for prediction

```
source("DiagnosticFunctions.R")
all_mus = compute_intensity(fitN = fitAll, simbetas = beta.sims, simmeans = s.means.sims,
                            simsds = s.sds.sims, PredData = pred.data)
```

3.1 Proportion of models with a meaningful environmental effect

When we first examine the models, we may want to see the proportion of the subset models that have a meaningful environmental effect. Our models are fitted with a lasso penalty, and consequently some of the fitted coefficients may be shrunk to 0, effectively removing the effect of the corresponding covariates. If all of the coefficients of the environmental covariates are equal to 0 (i.e if $\hat{\beta}_j = 0$ for all j), there is no predicted environmental effect and the map of predicted intensities will be uniform. We would expect that as the subset size increases, the proportion of fitted models with no environmental effect will decrease.

We can check this with the `ZeroEnvEffect()` function, which takes as input an array `mus` containing the intensities computed from the `compute_intensity()` function, as follows:

```
enveffect = ZeroEnvEffect(mus = all_mus)
enveffect
```

```
##      Env Effect No Env Effect
## 1000       1000             0
## 500        1000             0
## 200         821           179
```

```
## 100          463           537
## 50           351           649
```

We see that once the subset size reaches $N = 500$, all of the fitted models have some meaningful environmental effect.

3.2 Maps of average intensity

We may visually assess model congruency by considering the average rescaled intensity throughout the study region across the various subset sizes. The rescaling is key here: as subset size increases, we would expect the intercept to increase as well, such that the maximum predicted intensity will naturally grow with subset size. As such, mapping the average raw intensities $\hat{\mu}_{\text{avg},N}(s)$ across different subset sizes N is likely to produce maps with different scales. Consequently, rescaling the intensities to have the same mean as the model fitted with all points should produce maps with comparable scales such that comparisons can be more readily made, and we therefore map the rescaled intensities $\hat{\mu}_{\text{avg},N,m}(s)$.

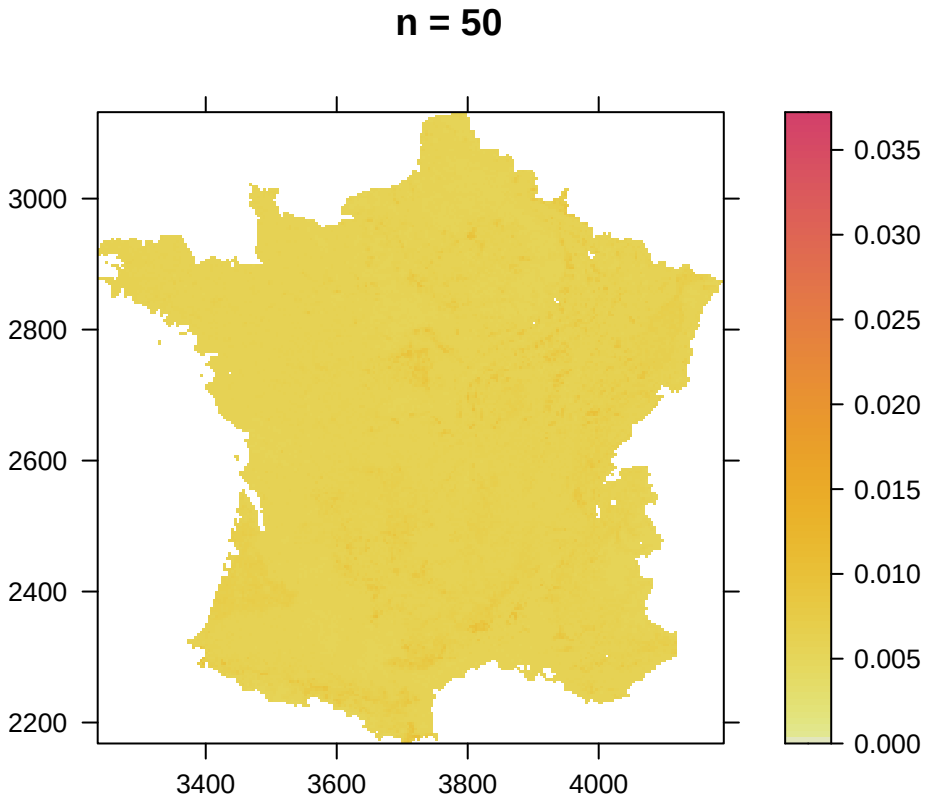
First, we determine a common upper limit for the scale:

```
zmax = rep(NA, 5) # maximum average rescaled intensity for each subset size
for (i in 1:5)
{
  zmax[i] = max(apply(all_mus$rescale_mus[i,,], 2, mean))
}
zmax = c(zmax, max(correctedmu)) # add maximum fitted intensity for model with all points
zplot = max(zmax) # Largest maximum will serve as the upper limit in the plot
```

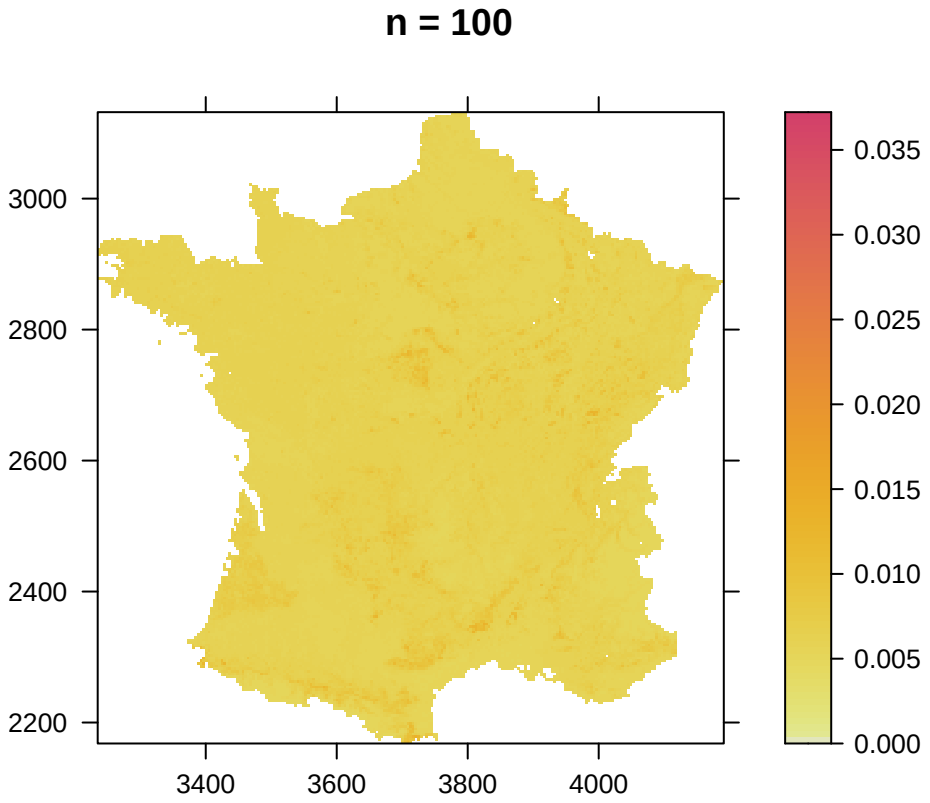
Now, we can use the `avg_mu_plot()` function to map the average intensity for each subset size, and compare it to the map of the model that uses all $n = 2449$ points. The `avg_mu_plot()` function takes the following input:

- `mus`: an array containing the intensities computed from the `compute_intensity()` function
- `subsetsize`: the subset size to use for the plot
- `XY`: a data frame containing the x- and y-coordinates for plotting
- `f`: the function to be applied to the intensity before mapping. For example, the user may enter `f = log` to produce a map of the natural logarithm of the intensities. Set to the identity function by default.
- `mu.min`: the minimum non-zero intensity to be used. All intensities less than `mu.min` are truncated to be equal to `mu.min`. If we allow the fitted intensity to be very low and plot the logarithm of the intensities, we risk the scale of the coloring being overwhelmed by the low value. By default, this is set to 10^{-5}
- `intensity`: whether to plot the raw intensities or (by default) the rescaled intensities. Entering anything other than "rescaled" will return a map of the average raw intensities.
- `...`: arguments to be passed to the `levelplot()` function such that the plot may be customised by the user.

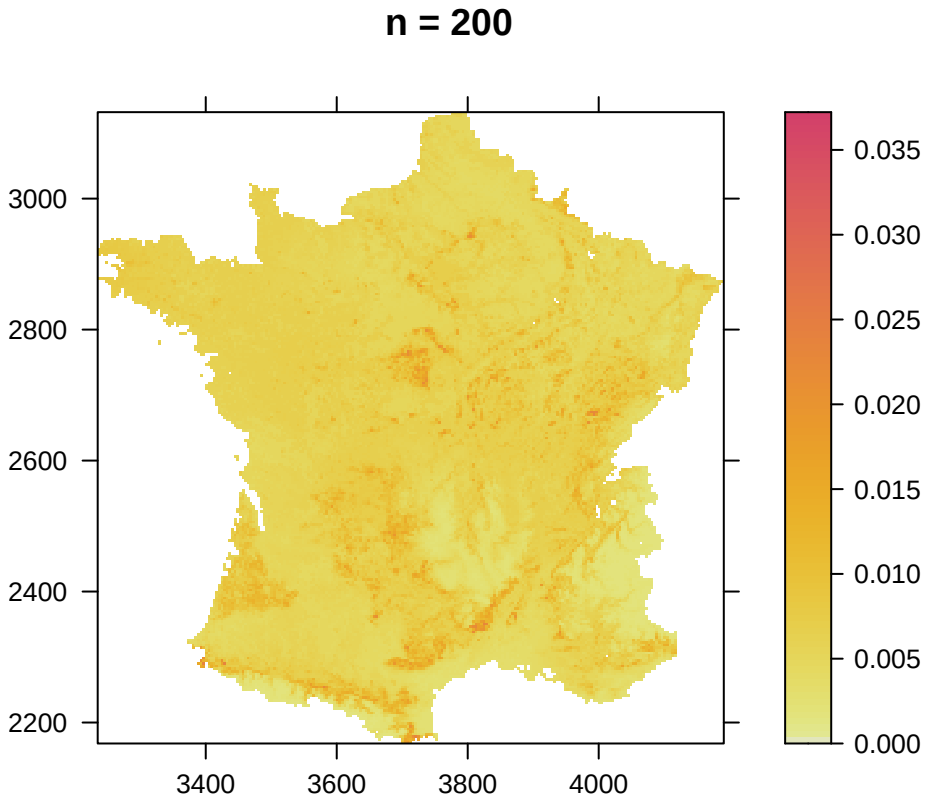
```
avg_mu_plot(mus = all_mus, subsetsize = 50, XY = env[,1:2], mu.min = 1.e-5,
            intensity = "rescaled", col.regions = col2, main = "n = 50",
            xlab = "", ylab = "", cuts = 100, at = seq(0, zplot, length.out = 99))
```



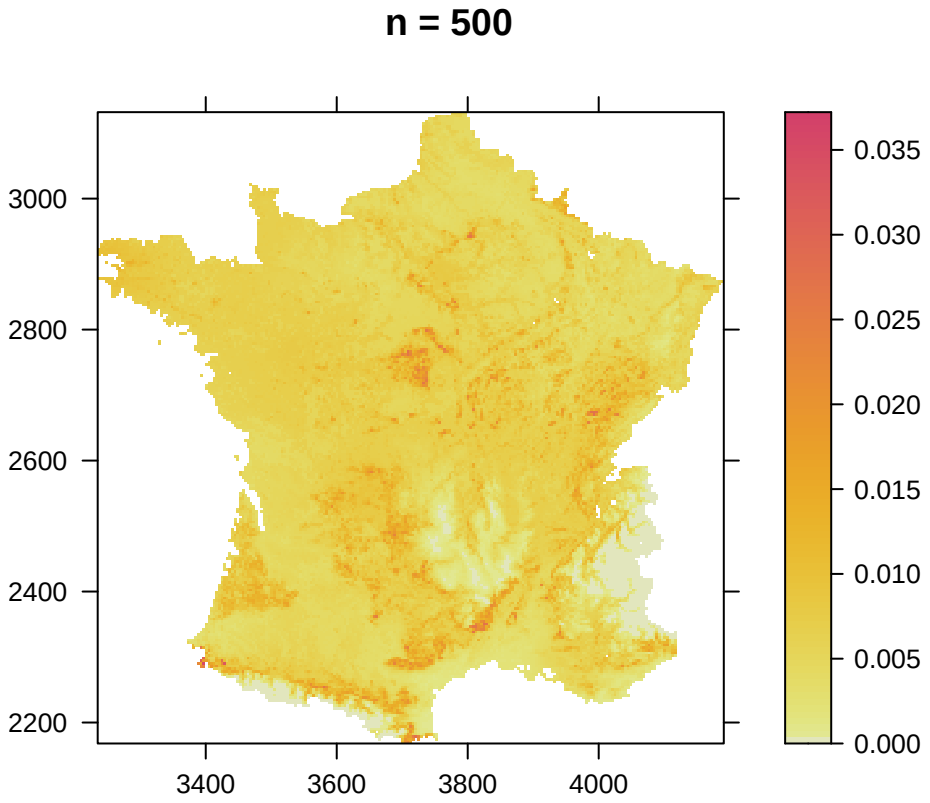

```
avg_mu_plot(mus = all_mus, subsetsize = 100, XY = env[,1:2], mu.min = 1.e-5,
            intensity = "rescaled", col.regions = col2, main = "n = 100",
            xlab = "", ylab = "", cuts = 100, at = seq(0, zplot, length.out = 99))
```



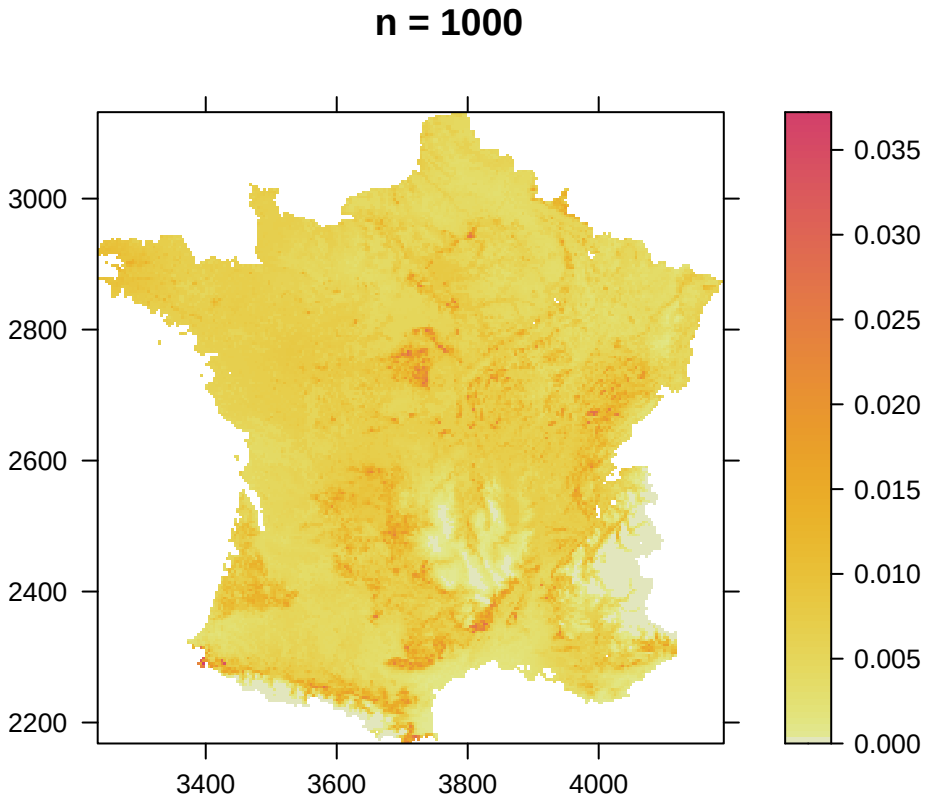

```
avg_mu_plot(mus = all_mus, subsetsize = 200, XY = env[,1:2], mu.min = 1.e-5,
            intensity = "rescaled", col.regions = col2, main = "n = 200",
            xlab = "", ylab = "", cuts = 100, at = seq(0, zplot, length.out = 99))
```



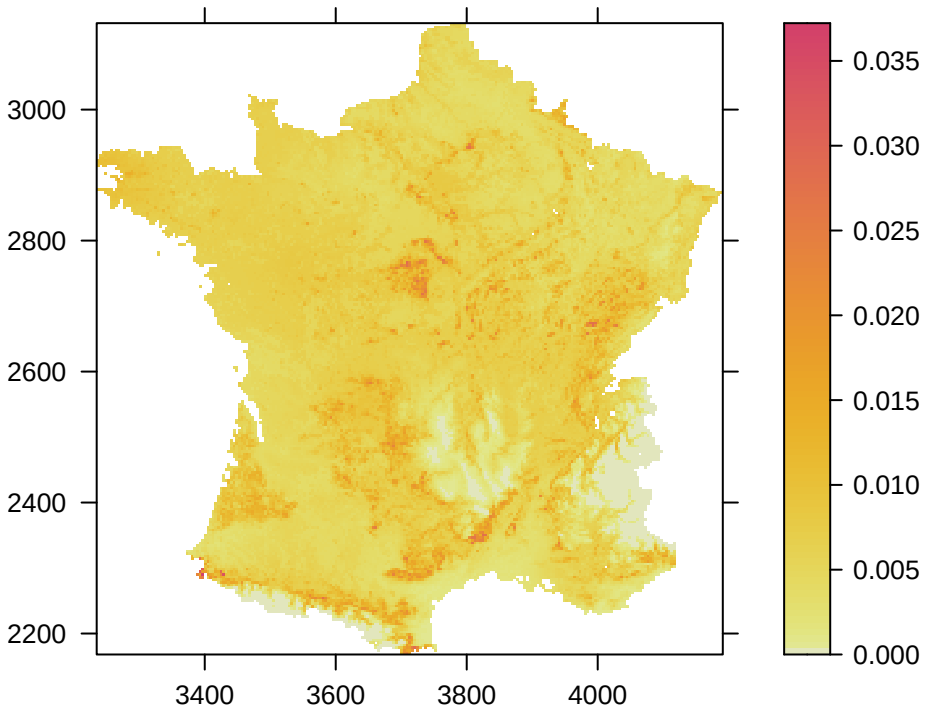

```
avg_mu_plot(mus = all_mus, subsetsize = 500, XY = env[,1:2], mu.min = 1.e-5,
            intensity = "rescaled", col.regions = col2, main = "n = 500",
            xlab = "", ylab = "", cuts = 100, at = seq(0, zplot, length.out = 99))
```




```
avg_mu_plot(mus = all_mus, subsetsize = 1000, XY = env[,1:2], mu.min = 1.e-5,
            intensity = "rescaled", col.regions = col2, main = "n = 1000",
            xlab = "", ylab = "", cuts = 100, at = seq(0, zplot, length.out = 99))
```




```
levelplot(correctedmu ~ env$X + env$Y, cuts = 100, asp = "iso",
          col.regions = col2, main = "n = 2449", xlab = "", ylab = "",
          at = seq(0, zplot, length.out = 99)) # all 2449 points
```



Visually, we see that the maps begin to resemble each other once the subset size reaches $N = 500$ points.

3.3 Maps of the standard deviation

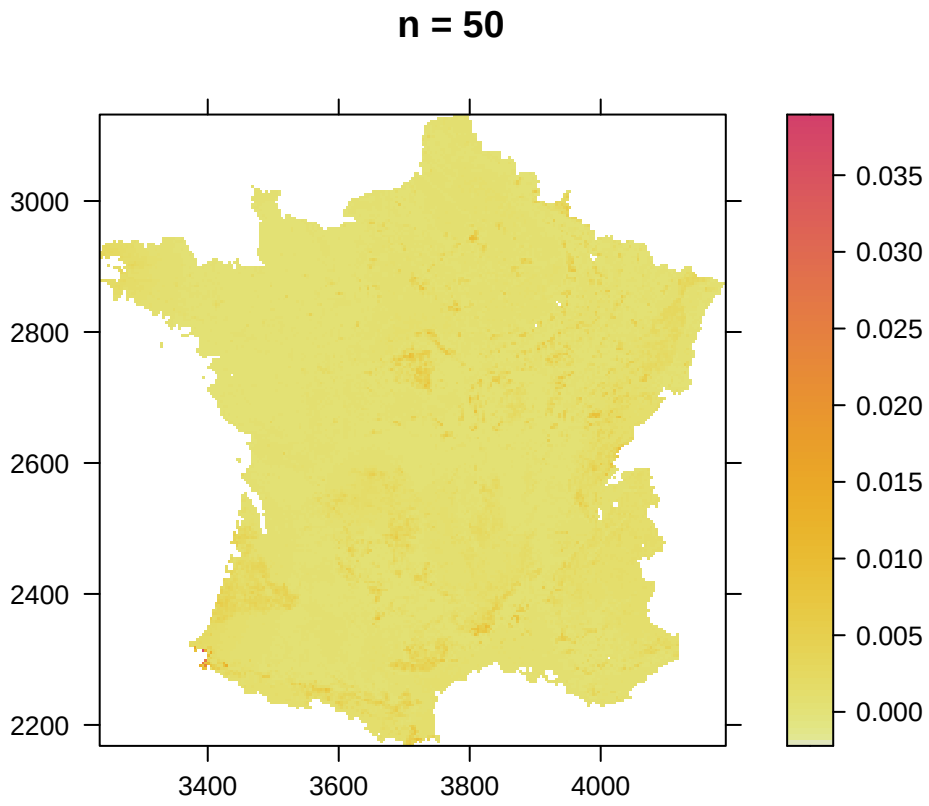
While mapping the average rescaled intensities as above gives us a snapshot of the mean, it does not tell us how precise these point estimates are. We may wish to quantify the uncertainty of these point estimates.

Many software packages, including `spatstat`, produce standard errors for both coefficients and the fitted intensities. However, there is no theoretical standard error derivation for coefficients or intensities when models are fitted with a lasso penalty. Nonetheless, we can map the standard deviation of the rescaled intensities at each subset size as a way to quantify their uncertainty.

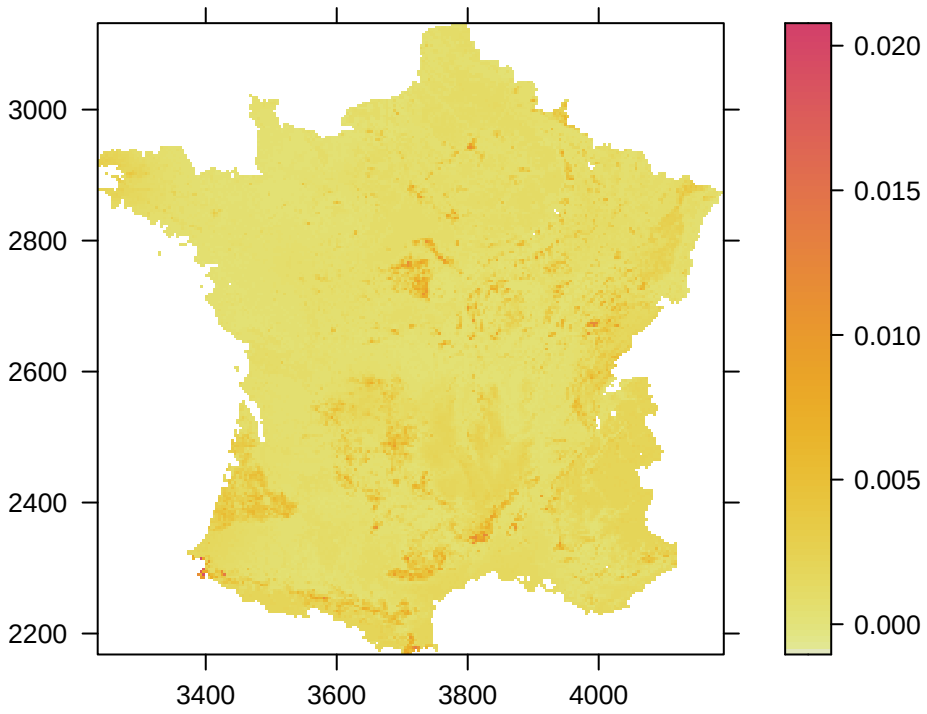
The `sd_plot()` function allows us to map standard deviations computed from the simulated subsets, as follows. The function takes as input:

- `mus`: an array containing the intensities computed from the `compute_intensity()` function
- `subsetsize`: the subset size to use for the plot
- `XY`: a data frame containing the x- and y-coordinates for plotting
- `f`: the function to be applied to the standard deviation before mapping. For example, the user may enter `f = log` to produce a map of the natural logarithm of the standard deviations. Set to the identity function by default.
- `intensity`: whether to plot the standard deviation of the raw intensities or (by default) the rescaled intensities. Entering anything other than "rescaled" will return a map of the standard deviation of the raw intensities.
- `...`: arguments to be passed to the `levelplot()` function such that the plot may be customised by the user.

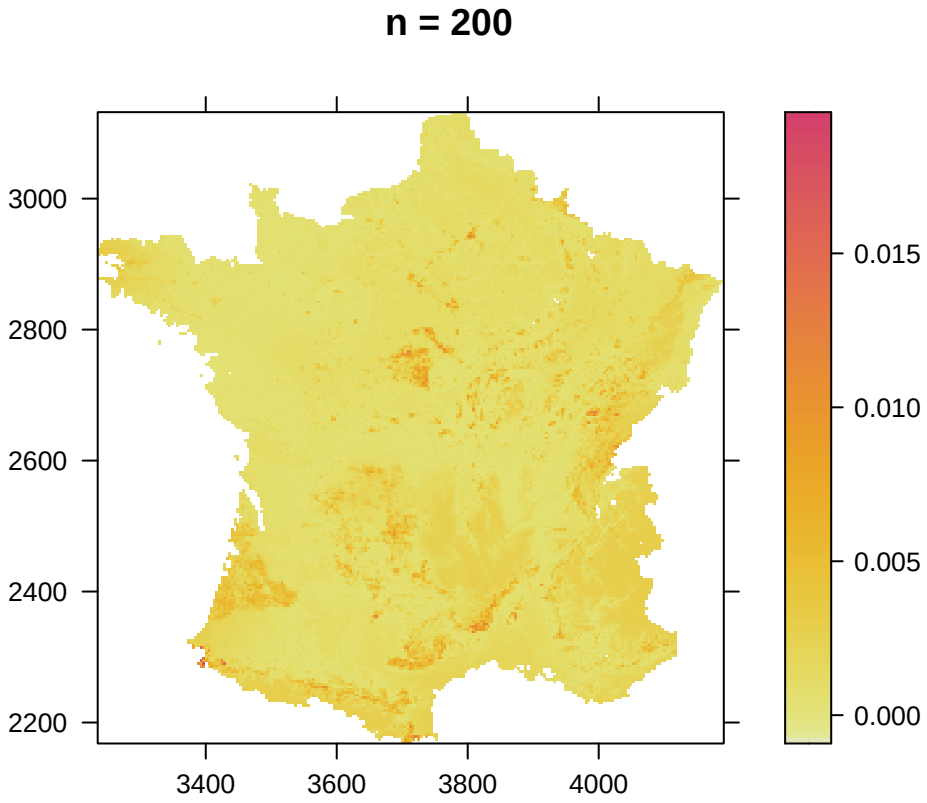
```
sd_plot(mus = all_mus, subsetsize = 50, XY = env[,1:2], col.regions = col2,
        main = "n = 50", xlab = "", ylab = "", cuts = 100)
```



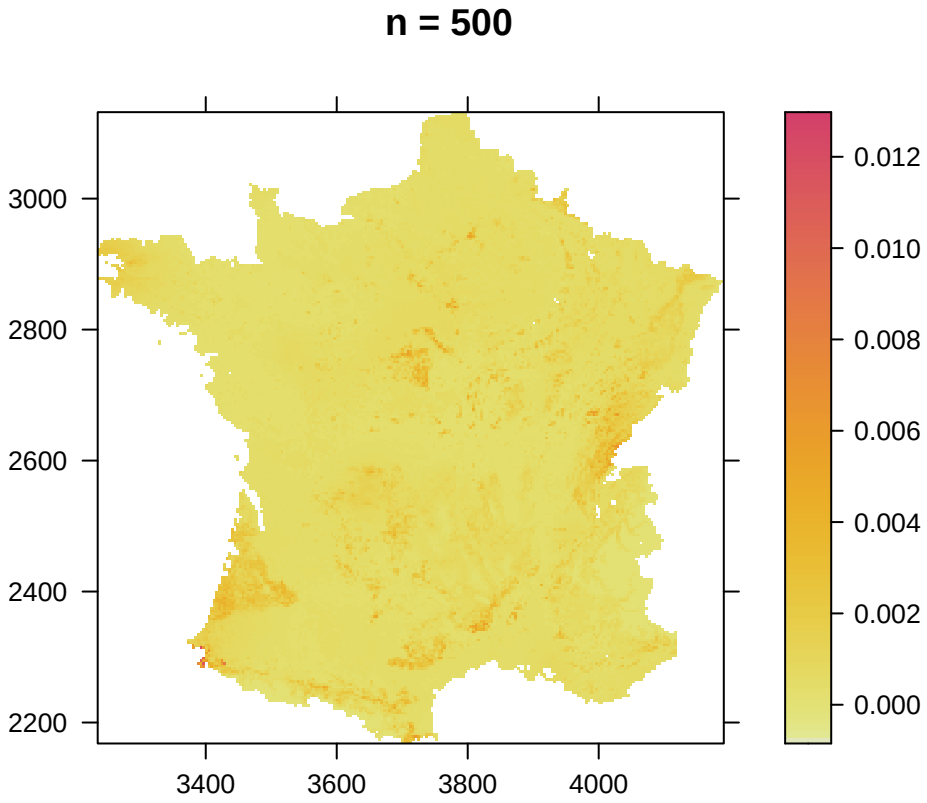

```
sd_plot(mus = all_mus, subsetsize = 100, XY = env[,1:2], col.regions = col2,
        main = "n = 100", xlab = "", ylab = "", cuts = 100)
```



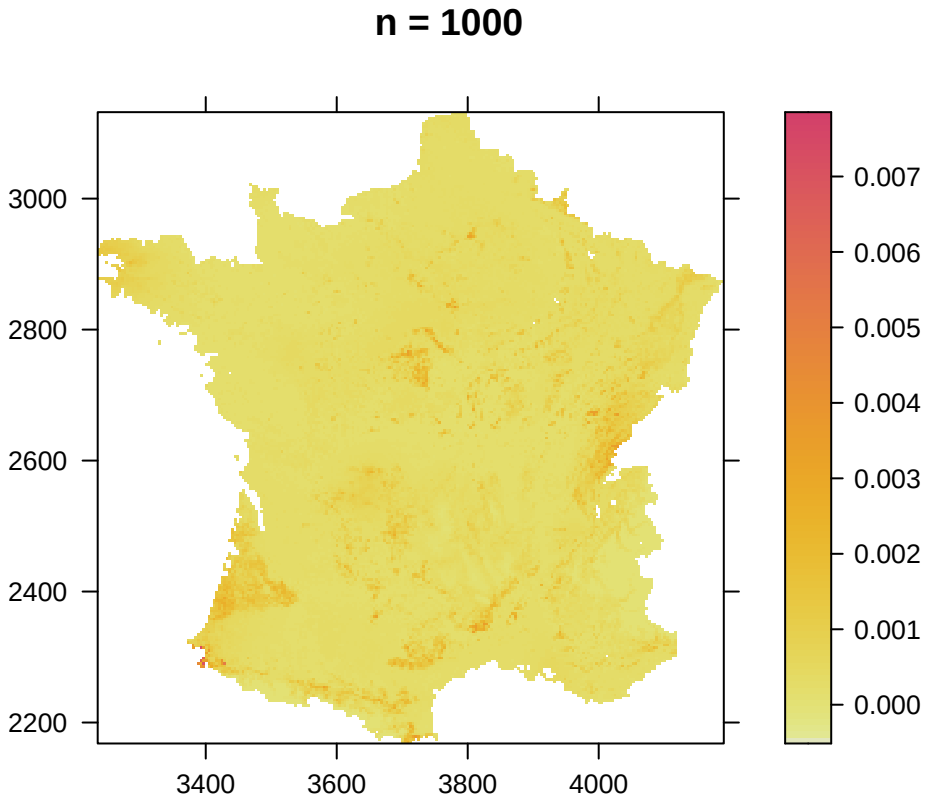

```
sd_plot(mus = all_mus, subsetsize = 200, XY = env[,1:2], col.regions = col2,
        main = "n = 200", xlab = "", ylab = "", cuts = 100)
```




```
sd_plot(mus = all_mus, subsetsize = 500, XY = env[,1:2], col.regions = col2,
        main = "n = 500", xlab = "", ylab = "", cuts = 100)
```




```
sd_plot(mus = all_mus, subsetsize = 1000, XY = env[,1:2], col.regions = col2,
        main = "n = 1000", xlab = "", ylab = "", cuts = 100)
```



We see that for each subset size, the standard deviation tends to be highest where the intensity is highest. This is not surprising, as we are fitting Poisson point process models, which implies that the variance grows proportionally with the mean.

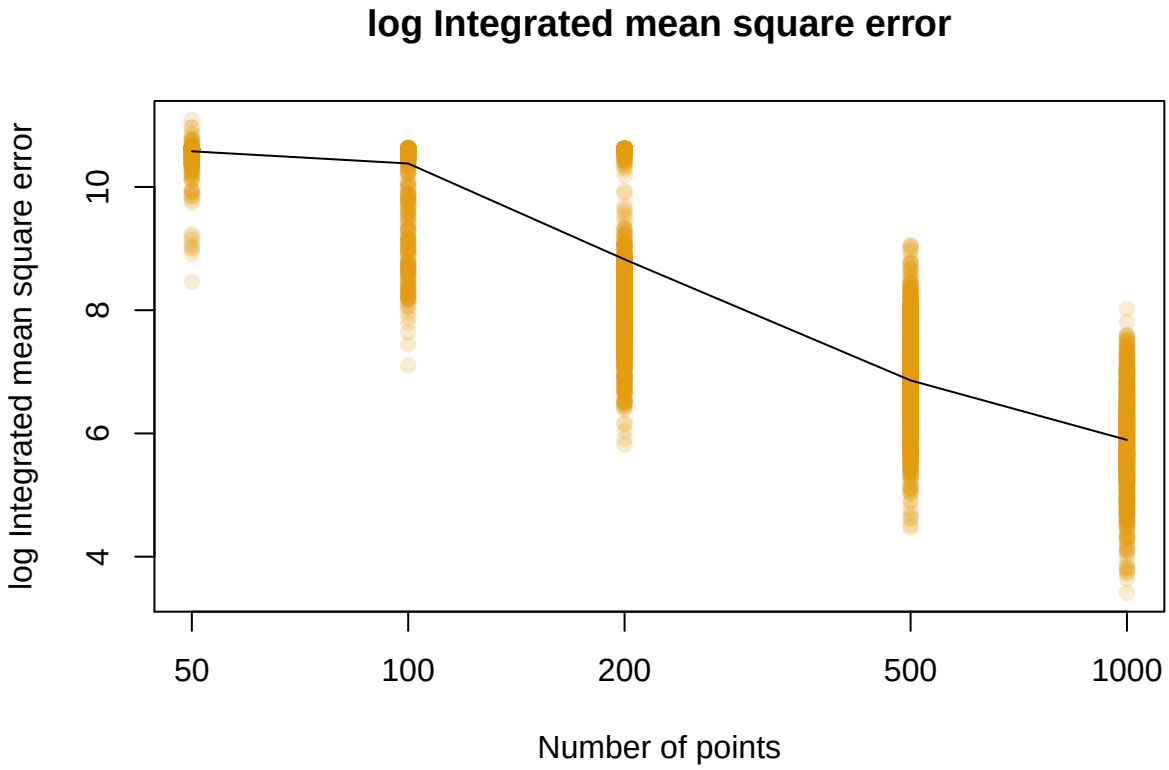
3.4 Integrated mean square error

The integrated mean square error (ISME) provides a single number to measure alignment between the fitted intensities of the model using all the points ($\hat{\mu}(s)$) with the rescaled intensities of the models fitted to the subsets ($\hat{\mu}_{i,N,m}(s)$). The lower the IMSE, the greater the alignment.

The function `IMSE_plot()` produces a plot of the integrated mean square error for the various subset models, along with a trace plot of the mean. It takes as input the following arguments:

- `muN`: the intensity surface $\hat{\mu}(s)$ of the model fitted to all points
- `mus`: the array of rescaled intensities produced by the `compute_intensity()` function
- `mu.min`: the minimum non-zero intensity to be used in computing IMSE. If we allow the fitted intensity to be very low and define the IMSE on the logarithm of the intensities as in the main manuscript, we risk overweighting those fitted intensities which are very small. By default, this is set to 10^{-5}.
- `f`: a function to apply to the computed IMSE for plotting. Options include "log" and "sqrt", and the default is set to `NULL` such that no transformation is carried out.
- `col`: a color vector set as a proportion from 0 to 1 of red, green, and blue such that the `rgb()` function may produce appropriately transparent colours for plotting
- `...`: other arguments to be passed to an internal function `IMSE()`. For example, the `IMSE()` function computes the integrated mean square error based on the natural logarithm of the intensities by default. This can be changed by setting `f_intensity = sqrt`, for instance, to compute the IMSE based on the square root of the intensities instead, or `f_intensity = identity` to compute the IMSE based on the raw intensities.

```
plotIMSE = IMSE_plot(muN = correctedmu, mus = all_mus, f = "log")
```




```
##       1000        500        200        100         50
##   5.894893   6.860903   8.824520 10.381110 10.578243
```

We see that the IMSE tends to decrease as subset size increases, suggesting greater alignment between the models produced using larger subsets and the model fitted using the complete data set.

The `IMSE_plot()` function returns both the calculated integrated mean square error for each subset as well as the mean for each subset size.

3.5 Correlation measures

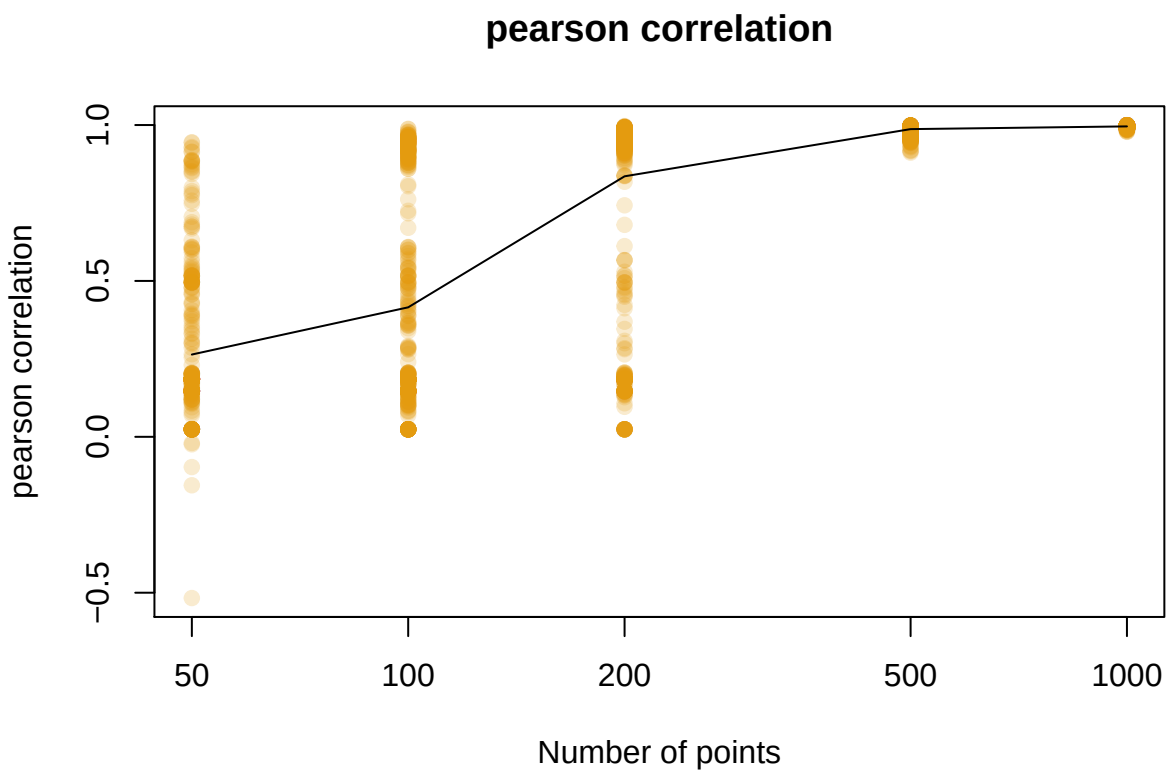
Correlation provides another measure of the alignment between the fitted intensities of the model using all the points ($\hat{\mu}(s)$) with the rescaled intensities of the models fitted to the subsets ($\hat{\mu}_{i,N,m}(s)$).

The function `Corr_plot()` produces a plot of the chosen correlation measure for the various subset models, along with a trace plot of the mean, analogous to the `IMSE_plot()` function. It takes as input the following arguments:

- `muN`: the intensity surface $\hat{\mu}(s)$ of the model fitted to all points
- `mus`: the array of rescaled intensities produced by the `compute_intensity()` function
- `mu.min`: the minimum non-zero intensity to be used in computing IMSE. If we allow the fitted intensity to be very low and define the IMSE on the logarithm of the intensities as in the main manuscript, we risk overweighting those fitted intensities which are very small. By default, this is set to 10^{-5}.
- `method`: which correlation method to compute. Pearson correlations are produced by default, but entering "spearman" or "kendall" will produce plots of Spearman's ρ or Kendall's τ.

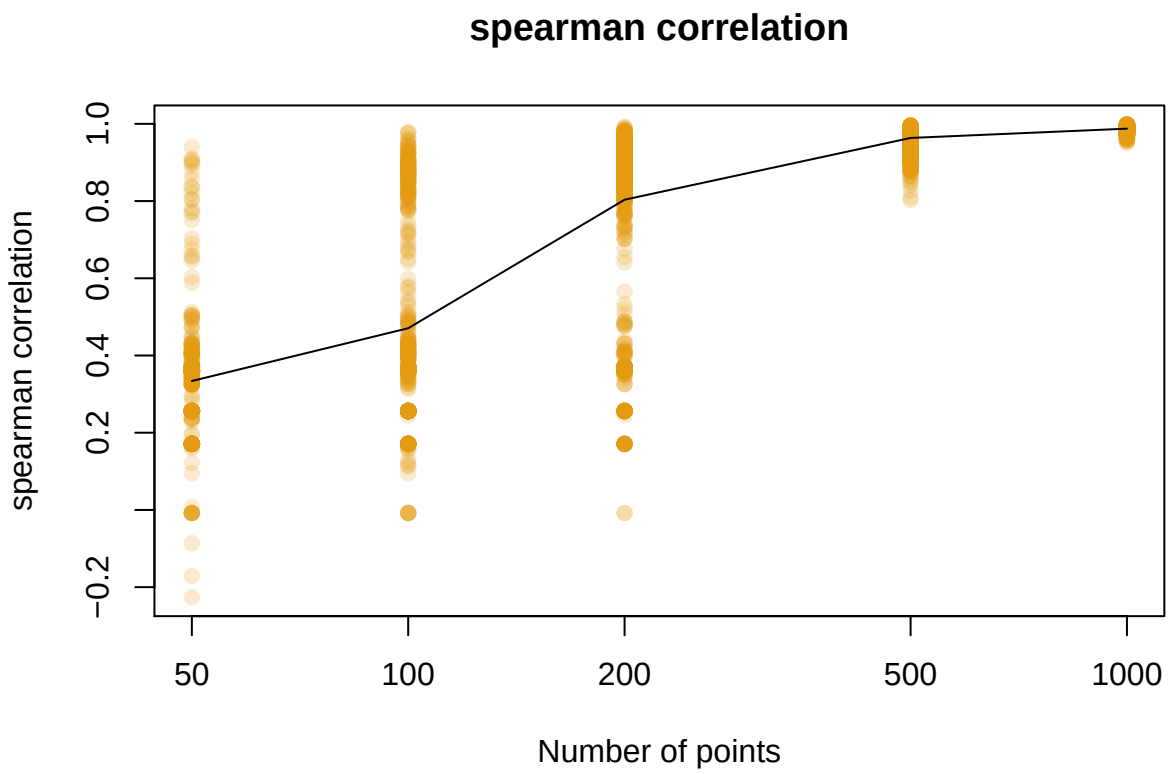
- `f`: a function to apply to the computed correlation for plotting. Options include "log" and "sqrt", and the default is set to `NULL` such that no transformation is carried out.
- `col`: a color vector set as a proportion from 0 to 1 of red, green, and blue such that the `rgb()` function may produce appropriately transparent colours for plotting
- `...`: other arguments to be passed to an internal function `CorrSims()`. For example, we could compute correlations between the natural logarithm of the intensities by setting `f_intensity = log`.

```
pearson = Corr_plot(muN = correctedmu, mus = all_mus, mu.min = 1.e-5, f_intensity = log)
```




```
##      1000       500       200       100        50
## 0.9956352 0.9869218 0.8355402 0.4153048 0.2639445
```

```
spearman = Corr_plot(muN = correctedmu, mus = all_mus, mu.min = 1.e-5, method = "spearman")
```




```
##      1000       500       200       100        50
## 0.9874881 0.9634526 0.8034872 0.4706448 0.3339161
```

We see that as subset size increases, both the Pearson correlation of the log intensities and Spearman's ρ increase. Once the subset size reaches $N = 500$, the alignment is quite strong, with the Pearson correlation greater than 0.98 and Spearman's ρ greater than 0.96.

The `Corr_plot()` function returns both the calculated correlation measure for each subset as well as the mean for each subset size.

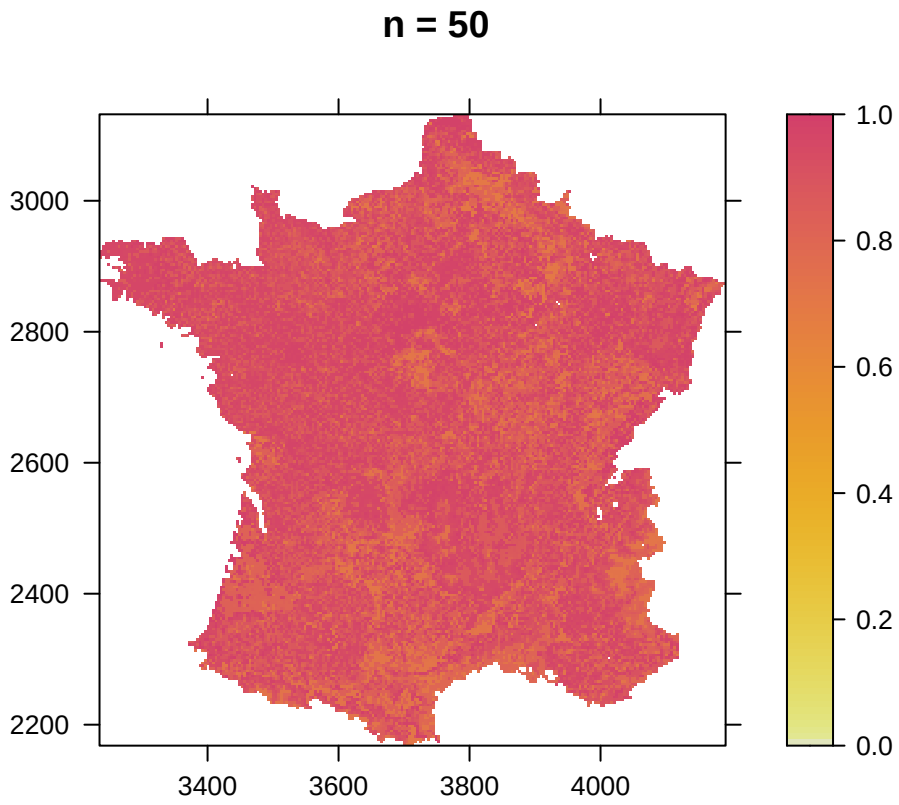
3.6 Quantile Matching

While the previous tools are useful summaries of the overall alignment between the fitted intensities of the model using all the points ($\hat{\mu}(s)$) with the rescaled intensities of the models fitted to the subsets ($\hat{\mu}_{i,N,m}(s)$), they do not indicate where the intensity surfaces differ.

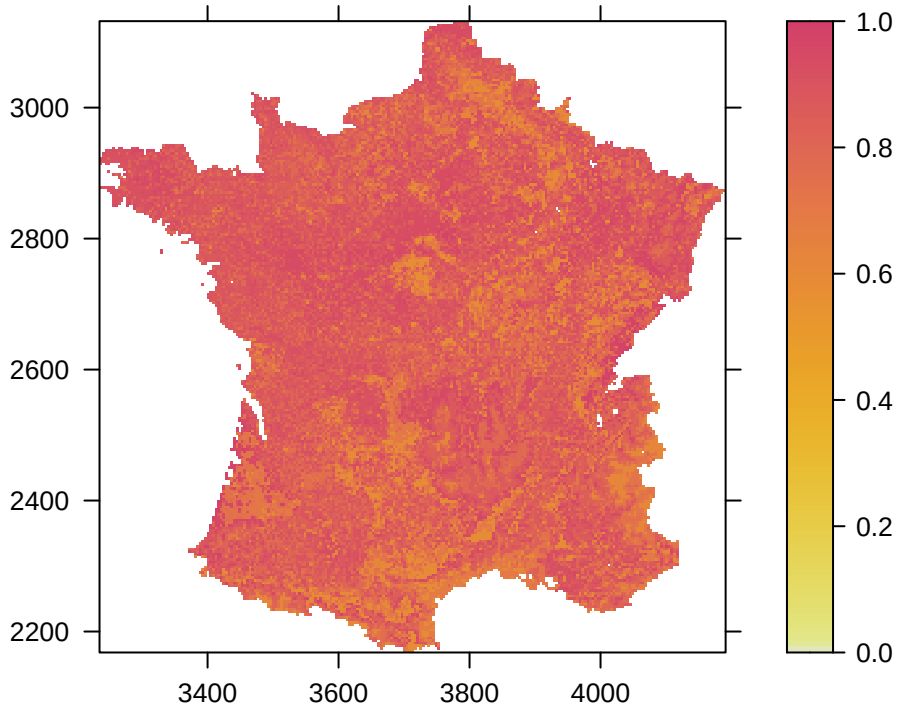
The function `quantilematch()` produces a map of the proportion of subsets which place the locations into different categories than the categories from the model which uses the full set of available data as defined by user-supplied quantiles. In this way, it indicates regions where categories of intensity are most likely to differ between the models fitted to random subsets and the model fitted with all of the data points. It takes as input the following arguments:

- `muN`: the intensity surface $\hat{\mu}(s)$ of the model fitted to all points
- `mus`: the array of rescaled intensities produced by the `compute_intensity()` function
- `subsetsize`: the subset size to use for the plot
- `XY`: a data frame containing the x- and y-coordinates for plotting
- `quantiles`: a vector of quantiles which defines the categories, by default set to $(0.2, 0.4, 0.6, 0.8)$

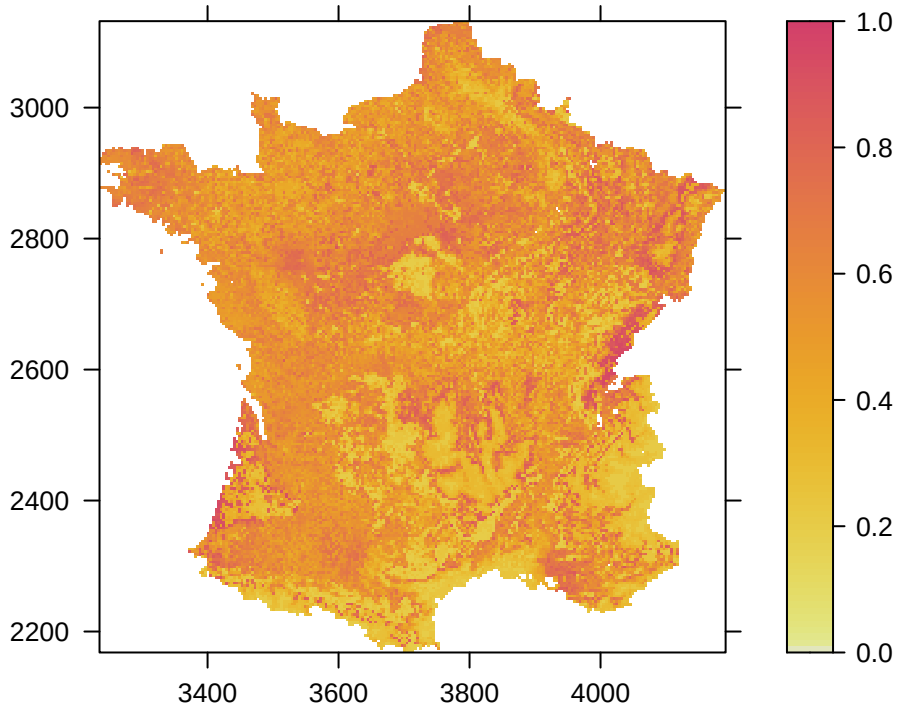
```
q50 = quantilematch(correctedmu, all_mus, 50, env[,1:2],
              col.regions = col2, main = "n = 50",
              xlab = "", ylab = "", cuts = 100,
              at = seq(0, 1, length.out = 99))
```



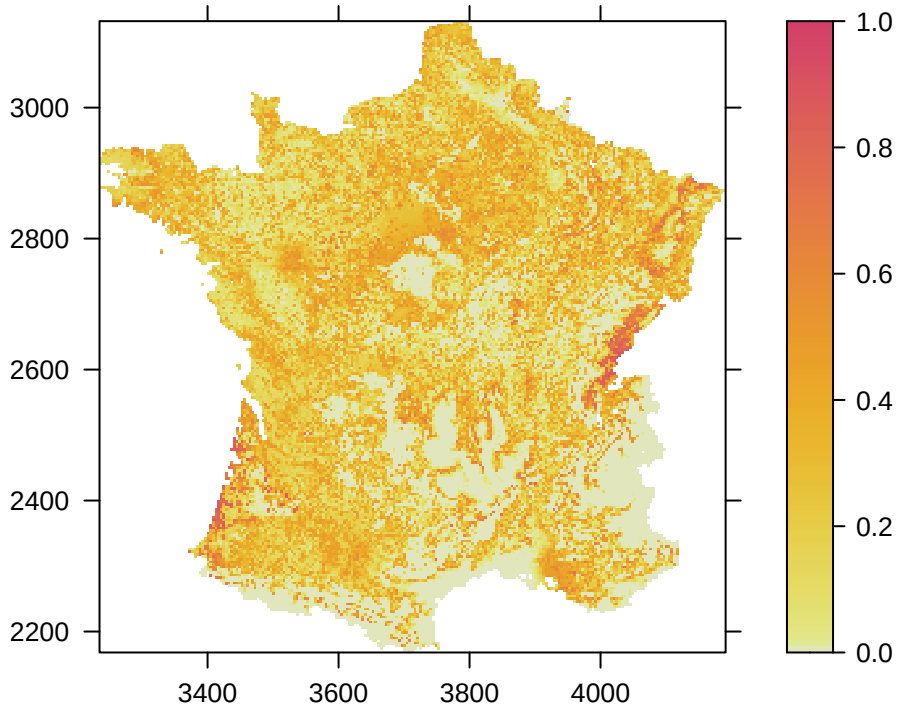

```
q100 = quantilematch(correctedmu, all_mus, 100, env[,1:2],
              col.regions = col2, main = "n = 100",
              xlab = "", ylab = "", cuts = 100,
              at = seq(0, 1, length.out = 99))
```



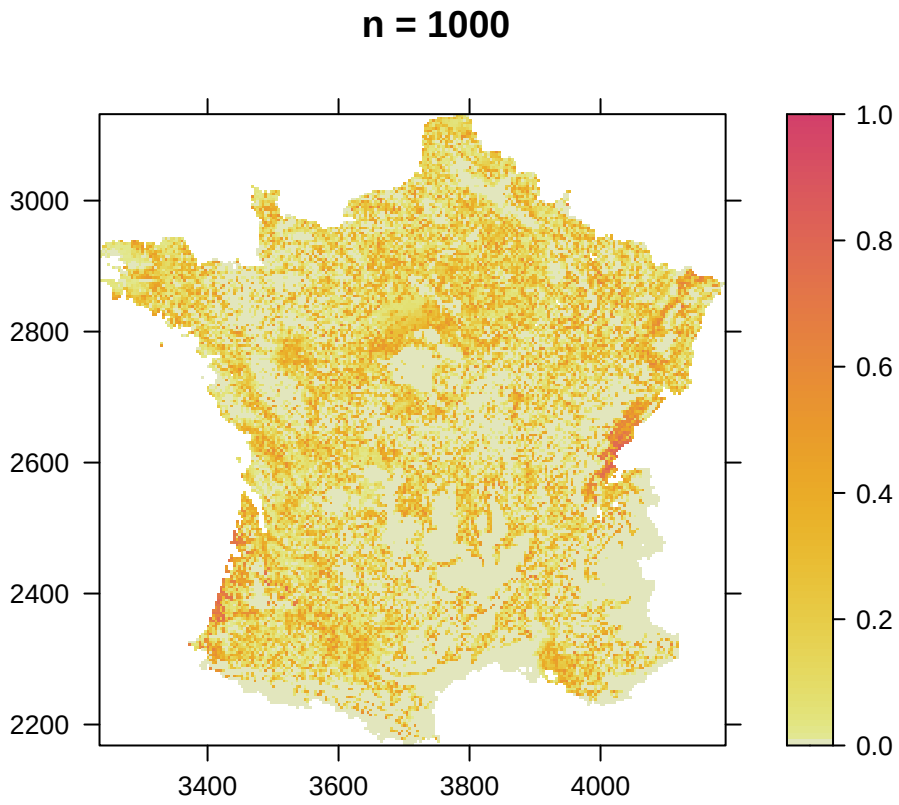

```
q200 = quantilematch(correctedmu, all_mus, 200, env[,1:2],
              col.regions = col2, main = "n = 200",
              xlab = "", ylab = "", cuts = 100,
              at = seq(0, 1, length.out = 99))
```




```
q500 = quantilematch(correctedmu, all_mus, 500, env[,1:2],
              col.regions = col2, main = "n = 500",
              xlab = "", ylab = "", cuts = 100,
              at = seq(0, 1, length.out = 99))
```




```
q1000 = quantilematch(correctedmu, all_mus, 1000, env[,1:2],
              col.regions = col2, main = "n = 1000",
              xlab = "", ylab = "", cuts = 100,
              at = seq(0, 1, length.out = 99))
```



We see that the level of misalignment is initially very high, because most of the models for subset sizes $N = 50$ and $N = 100$ set all coefficients to 0. Once we reach a subset size of $N = 500$, the level of misalignment is much lower. Even at $N = 1000$, however, there are certain regions (in particular in the east near the border with Switzerland and in the southwest along the Atlantic coast) where there is relatively high misalignment.

The `quantilematch()` function returns the assigned groups for both the model using all of the available data as well as for each of the subsets, along with the misalignment proportions plotted in the map.

4 Diagnostic tools for model coefficients

A second class of diagnostic tools explores changes in the fitted coefficients $\hat{\boldsymbol{\beta}}$ and $\hat{\boldsymbol{\gamma}}$ as subset size changes. These fitted coefficients are used to produce the intensity maps, but their values may be of ecological interest, as their sign and magnitude provides insight into the direction and strength of the effect.

4.1 Scatterplots of the fitted coefficients

The function `coef_plot()` produces a scatterplot of the different fitted coefficients $\hat{\beta}_{i,N}$ and $\hat{\gamma}_{i,N}$ for the models produced by the random subsets of different sizes, along with a trace plot of the mean.

This function takes as arguments the following:

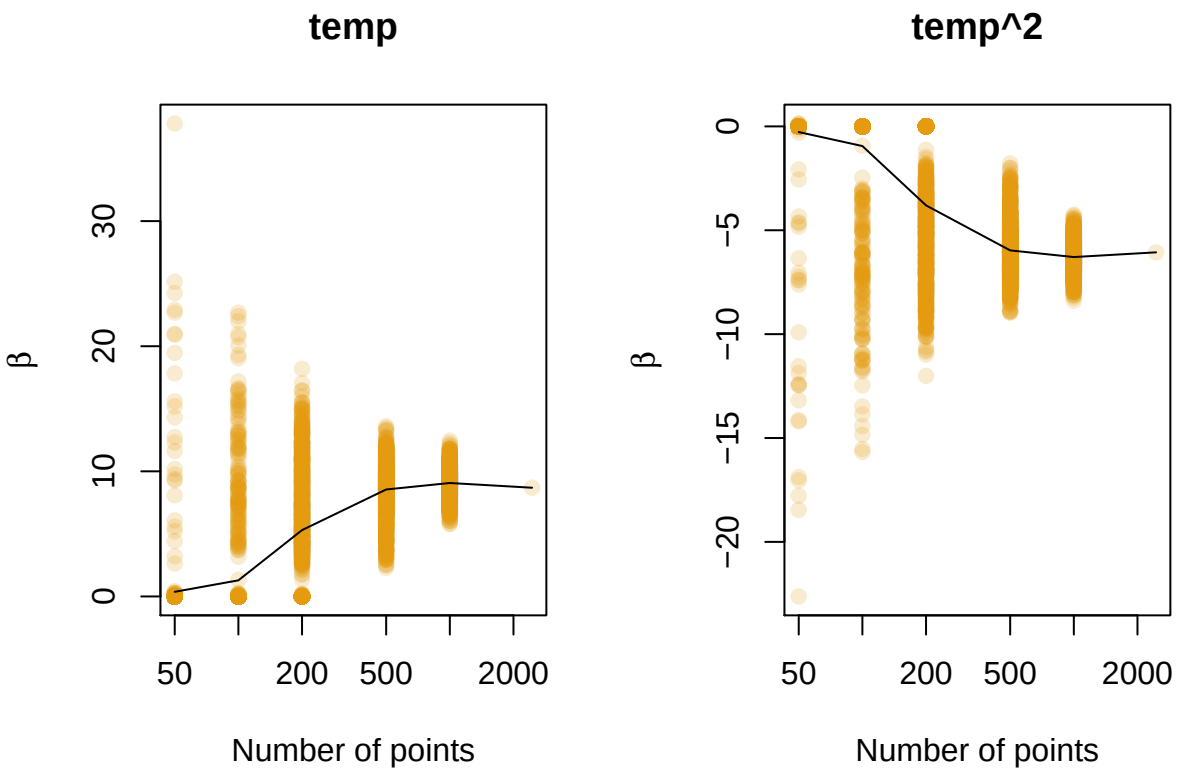
- `v`: the index of the variable to plot
- `fitN`: the model fitted with the `ppmlasso()` function to the full set of species data
- `simbetas`: the array containing the coefficients of the models fitted to the random subsets
- `col`: a color vector set as a proportion from 0 to 1 of red, green, and blue such that the `rgb()` function may produce appropriately transparent colours for plotting

We would expect that the coefficient estimates $\hat{\beta}_{i,N}$ and $\hat{\gamma}_{i,N}$ will converge toward the values $\hat{\beta}$ and $\hat{\gamma}$ computed from the model using the full set of points.

```
par(mfrow = c(1, 2))
coef_plot(v = 2, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##      1000       500       200       100        50
## 9.0671513 8.5494916 5.3014784 1.2856954 0.3697148
##
## $sds
##     1000      500      200      100       50
## 1.179833 2.096353 4.573805 3.714536 2.613488
```

```
coef_plot(v = 3, fitN = fitAll, simbetas = beta.sims)
```



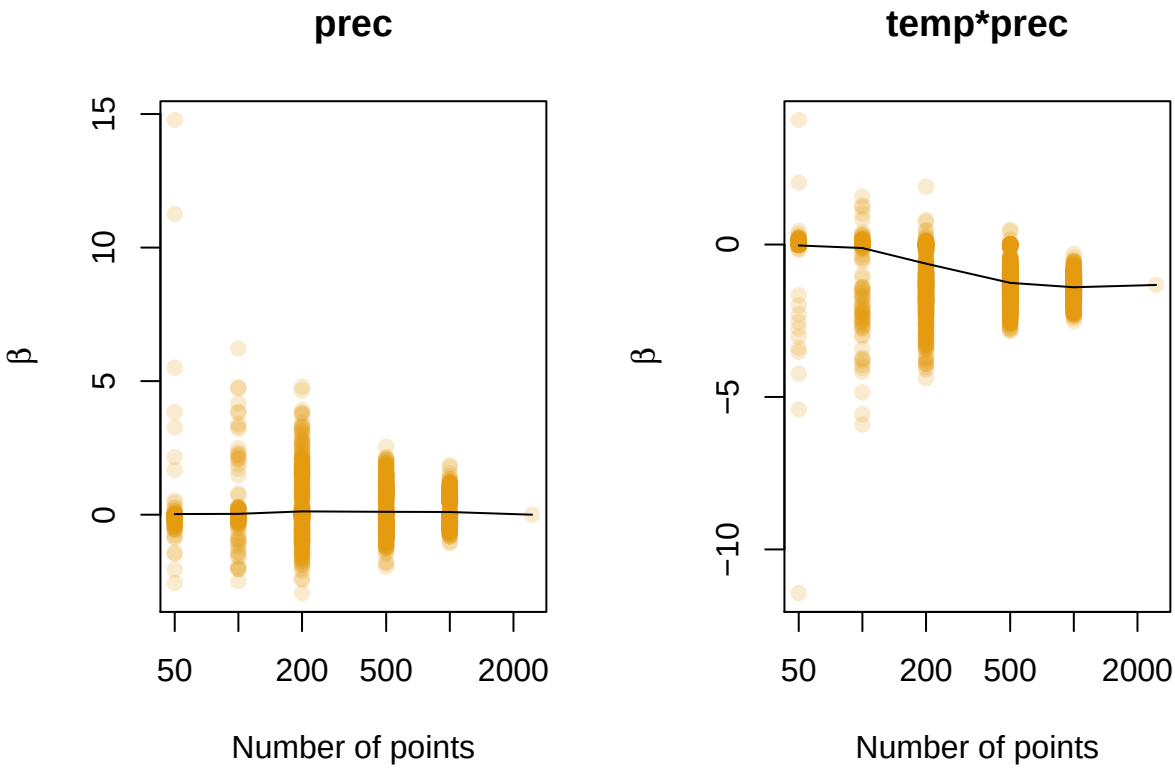

```
## $means
##       1000        500        200        100         50
## -6.2889317 -5.9686808 -3.8020318 -0.9452758 -0.2711572
##
## $sds
##      1000       500       200       100        50
## 0.7522407 1.3043520 3.1136723 2.6624306 1.8686630
```

```
coef_plot(v = 4, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##       1000        500        200        100         50
```

```
## 0.10067990 0.10716991 0.12364323 0.03086042 0.02377771
##
## $sds
##      1000       500       200       100        50
## 0.3991470 0.6214569 0.7883640 0.4988617 0.6552004
```

```
coef_plot(v = 5, fitN = fitAll, simbetas = beta.sims)
```

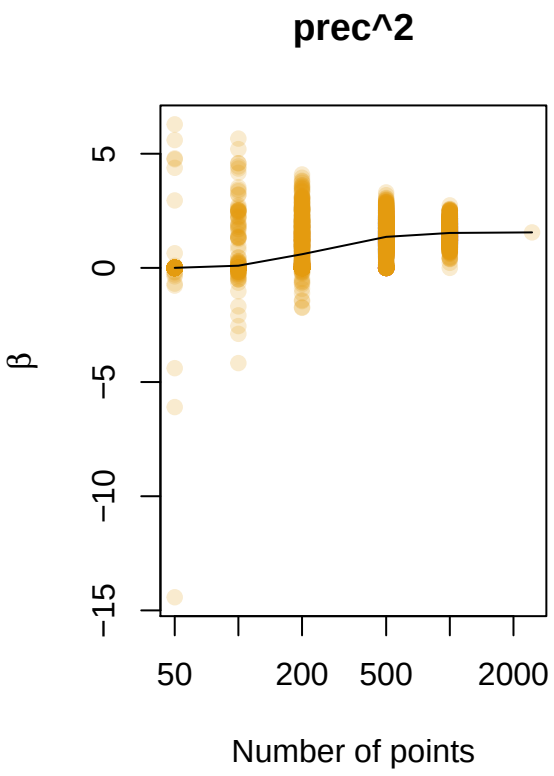


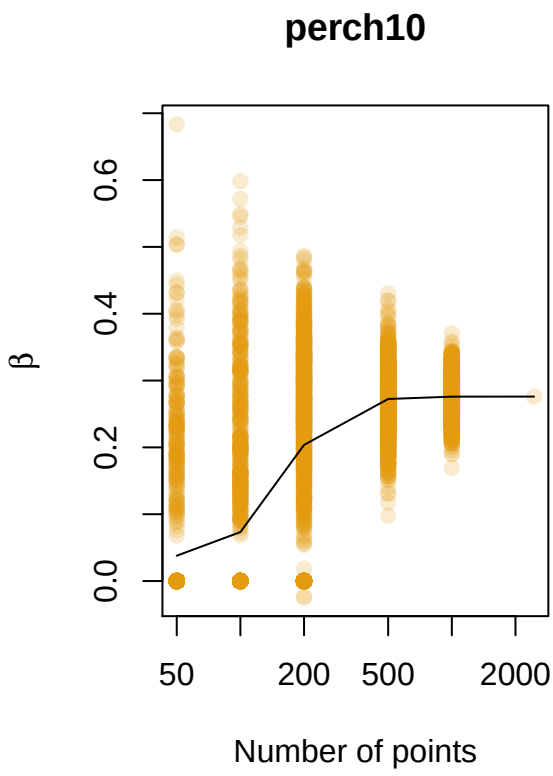

```
## $means
##       1000        500        200        100         50
## -1.4011691 -1.2567924 -0.6272815 -0.1147934 -0.0307759
##
## $sds
##      1000       500       200       100        50
## 0.3383578 0.6155705 0.9603745 0.6044825 0.5081410
```

```
coef_plot(v = 6, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##        1000         500         200         100          50
## 1.530081783 1.358955673 0.598402343 0.096332739 0.002090839
##
## $sds
##      1000       500       200       100        50
## 0.3926644 0.7134463 0.9481074 0.6179656 0.6411718
```

```
coef_plot(v = 7, fitN = fitAll, simbetas = beta.sims)
```



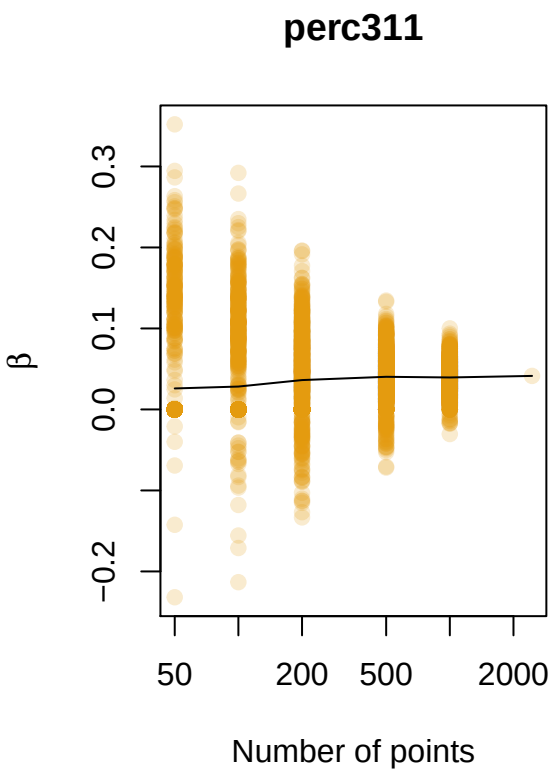


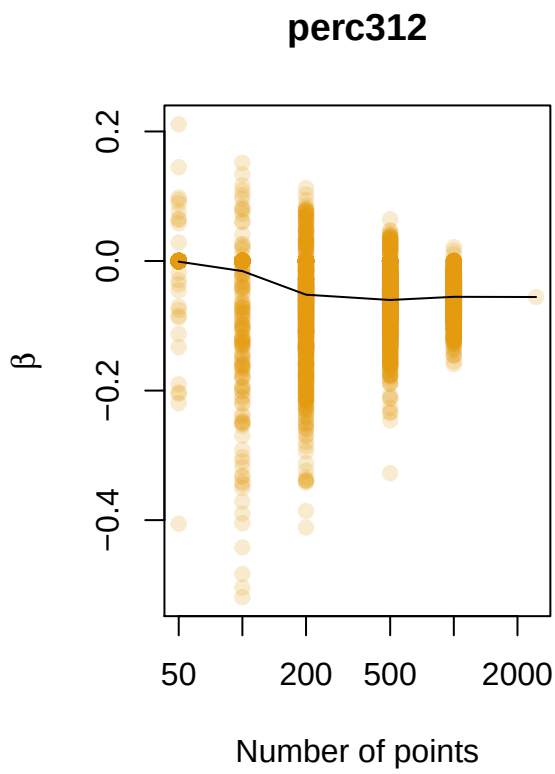

```
## $means
##       1000        500        200        100         50
## 0.27590565 0.27242454 0.20365113 0.07324764 0.03786603
##
## $sds
##       1000        500        200        100         50
## 0.02972891 0.04916710 0.13787098 0.12742810 0.09440406
```

```
coef_plot(v = 8, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##       1000        500        200        100         50
## 0.03949374 0.04029454 0.03627113 0.02824430 0.02591501
##
## $sds
##       1000        500        200        100         50
## 0.01915148 0.03071741 0.04956738 0.05751201 0.06217679
```

```
coef_plot(v = 9, fitN = fitAll, simbetas = beta.sims)
```



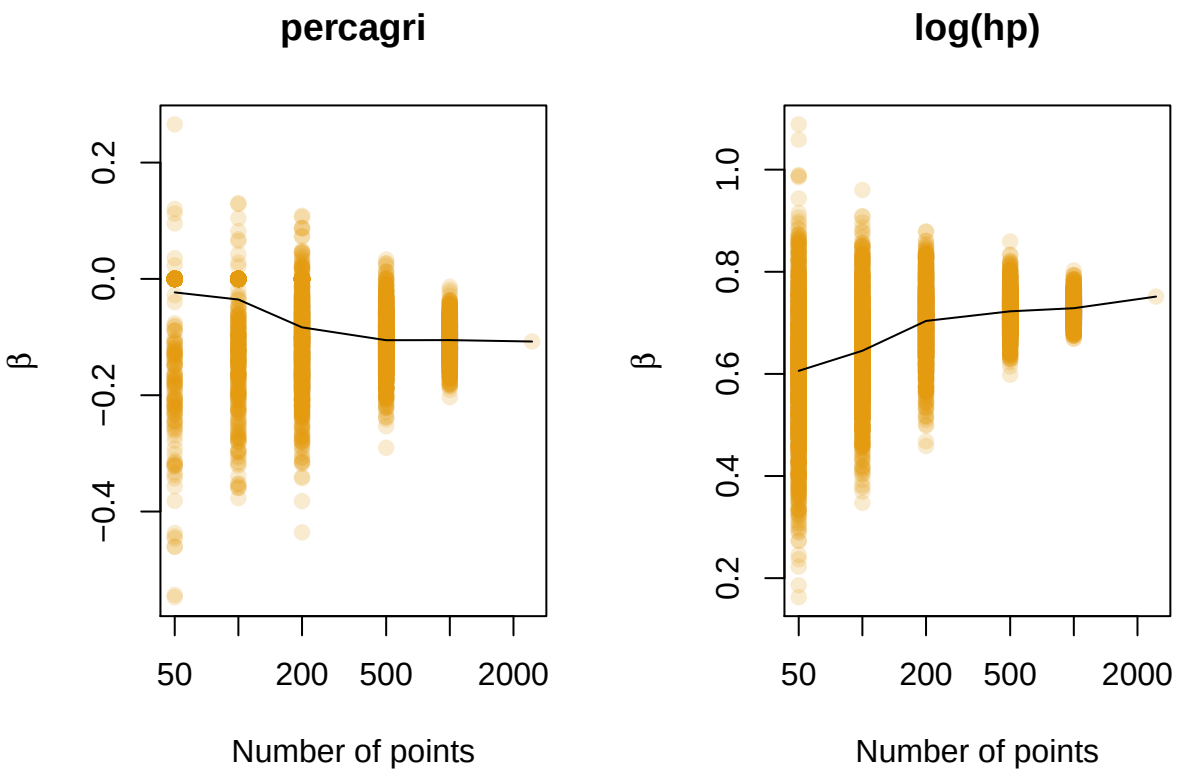



```
## $means
##          1000           500           200           100            50
## -0.0552693216 -0.0601793353 -0.0519973903 -0.0155730424 -0.0009872116
##
## $sds
##       1000        500        200        100         50
## 0.02842399 0.04816696 0.07667386 0.06306837 0.02240544
```

```
coef_plot(v = 10, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##        1000         500         200         100          50
## -0.10522984 -0.10537931 -0.08328826 -0.03543225 -0.02320848
##
## $sds
##       1000        500        200        100         50
## 0.02860646 0.04682451 0.08359032 0.07719108 0.07546346
```

```
coef_plot(v = 11, fitN = fitAll, simbetas = beta.sims)
```




```
## $means
##      1000       500       200       100        50
## 0.7286742 0.7225707 0.7037357 0.6453685 0.6059896
##
## $sds
##       1000        500        200        100         50
## 0.02198436 0.03658391 0.06832620 0.09675035 0.13309729
```

We see that the coefficient estimates tend to converge to the values obtained from the model fitted using all 2449 points, and that the variation in the estimates decreases with increasing subset size. In most cases, the coefficient estimates appear stable after the subset size $N = 500$.

We also see that there is a general tendency for the standard deviation for the coefficients to increase from $N = 50$ to $N = 200$ and then decrease thereafter. This may appear counterintuitive, but as many of the coefficient estimates are set to 0 for models of subset size $N = 50$, this has the effect of decreasing the standard deviation across all simulations. Those coefficients which are non-zero tend to have a large spread, as shown in the graph with a large range of values $\hat{\beta}_{j,i,50}$. In other words, there is a strong pull toward 0 for $N = 50$ (as shown in the analysis of coefficient signs in the next subsection), but those coefficients which are not set to 0 tend to be more variable. As subset size increases, the range of the fitted coefficients $\hat{\beta}_{j,i,N}$ tends to decrease, but as fewer coefficients are set to 0, the overall standard deviation tends to be higher across all 1000 simuluated subsets for $N = 100$ and $N = 200$. Once the subset size reaches 500, however, the range becomes small enough that the overall standard deviation starts to decrease, despite very few coefficients being set to 0.

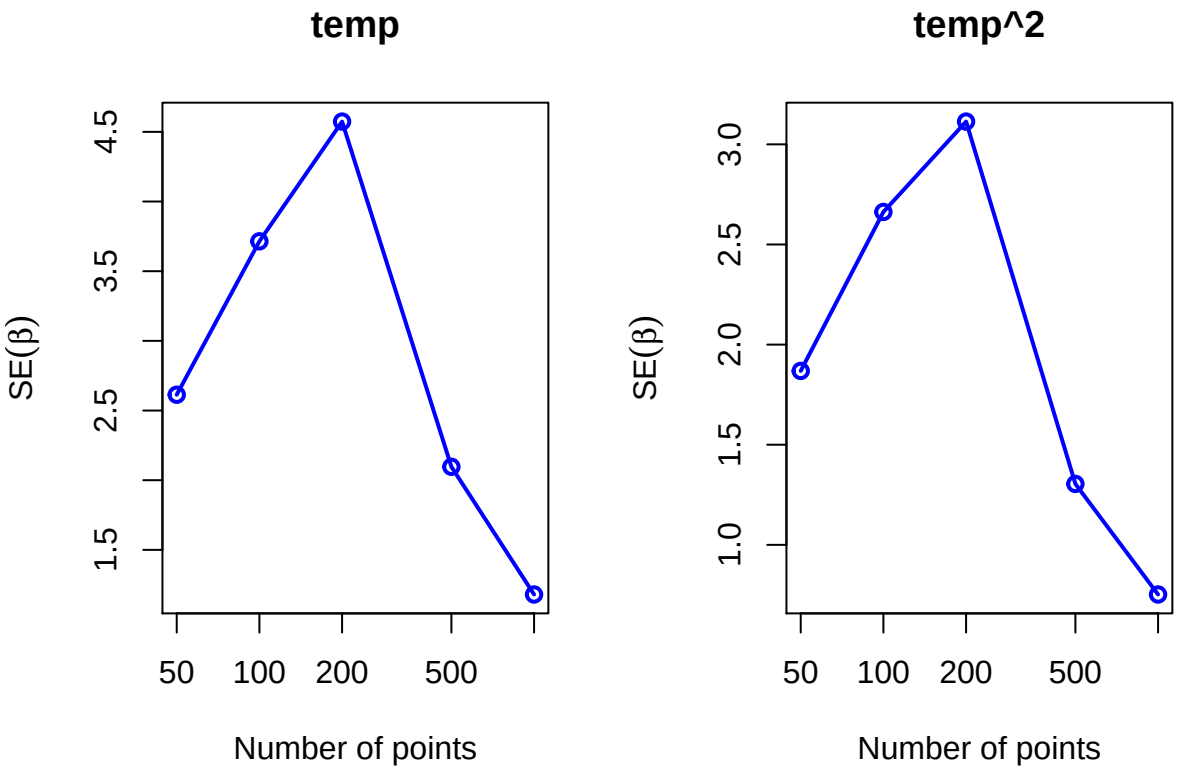
4.2 Coefficient standard error plots

We can produce a trace plot of the standard deviation of the coefficient estimates with the `coef_se_plot()` function:

```
par(mfrow = c(1, 2))
coef_se_plot(v = 2, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##      1000       500       200       100        50
## 9.0671513 8.5494916 5.3014784 1.2856954 0.3697148
##
## $sds
##     1000      500      200      100       50
## 1.179833 2.096353 4.573805 3.714536 2.613488
```

```
coef_se_plot(v = 3, fitN = fitAll, simbetas = beta.sims)
```



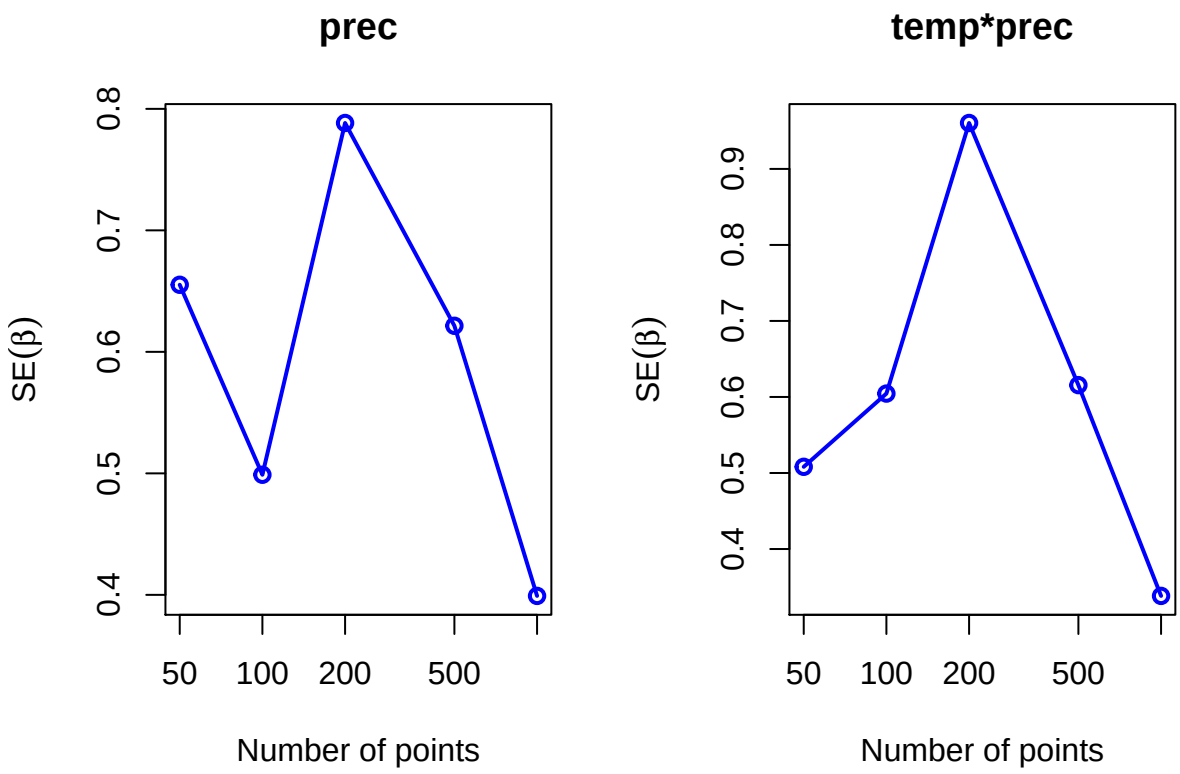

```
## $means
##       1000        500        200        100         50
## -6.2889317 -5.9686808 -3.8020318 -0.9452758 -0.2711572
##
## $sds
##      1000       500       200       100        50
## 0.7522407 1.3043520 3.1136723 2.6624306 1.8686630
```

```
coef_se_plot(v = 4, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
```

```
##       1000        500        200        100         50
## 0.10067990 0.10716991 0.12364323 0.03086042 0.02377771
##
## $sds
##      1000       500       200       100        50
## 0.3991470 0.6214569 0.7883640 0.4988617 0.6552004
```

```
coef_se_plot(v = 5, fitN = fitAll, simbetas = beta.sims)
```

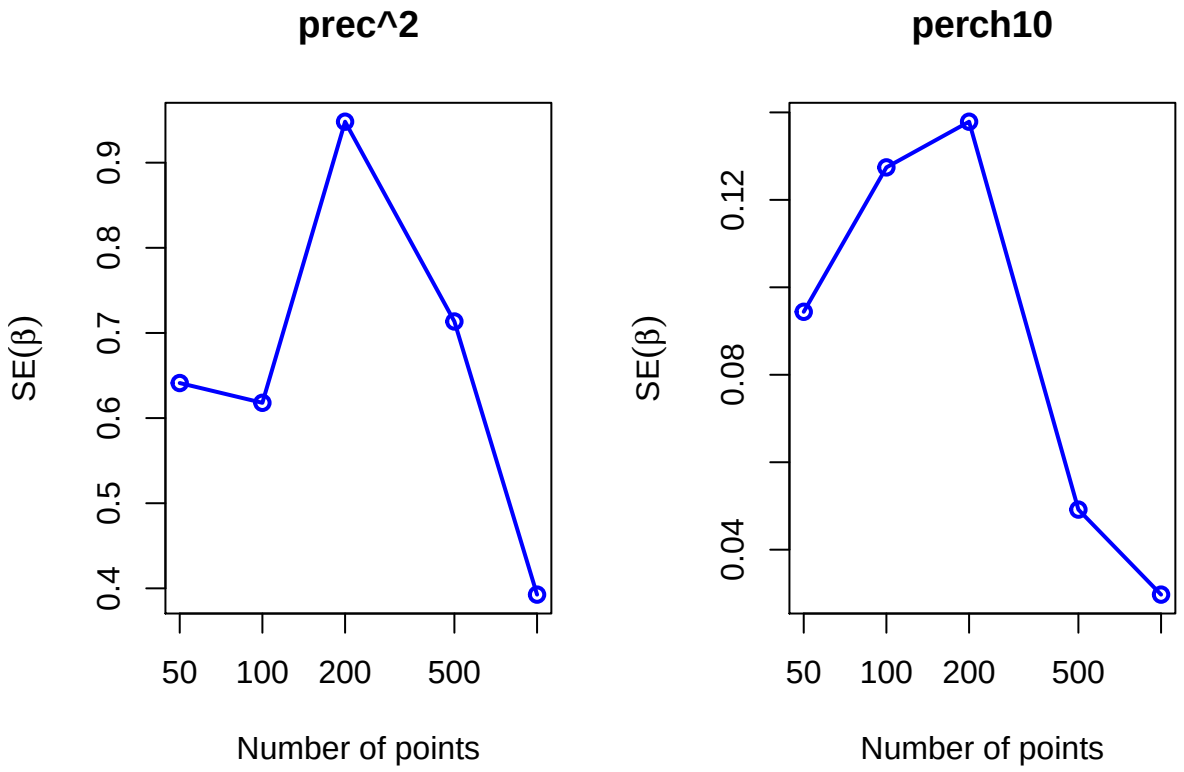



```
## $means
##       1000        500        200        100         50
## -1.4011691 -1.2567924 -0.6272815 -0.1147934 -0.0307759
##
## $sds
##      1000       500       200       100        50
## 0.3383578 0.6155705 0.9603745 0.6044825 0.5081410
```

```
coef_se_plot(v = 6, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##        1000         500         200         100          50
## 1.530081783 1.358955673 0.598402343 0.096332739 0.002090839
##
## $sds
##      1000       500       200       100        50
## 0.3926644 0.7134463 0.9481074 0.6179656 0.6411718
```

```
coef_se_plot(v = 7, fitN = fitAll, simbetas = beta.sims)
```

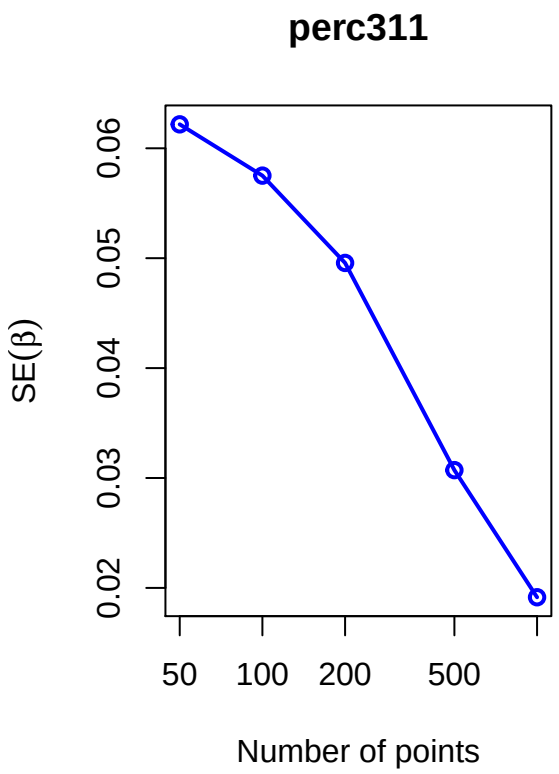


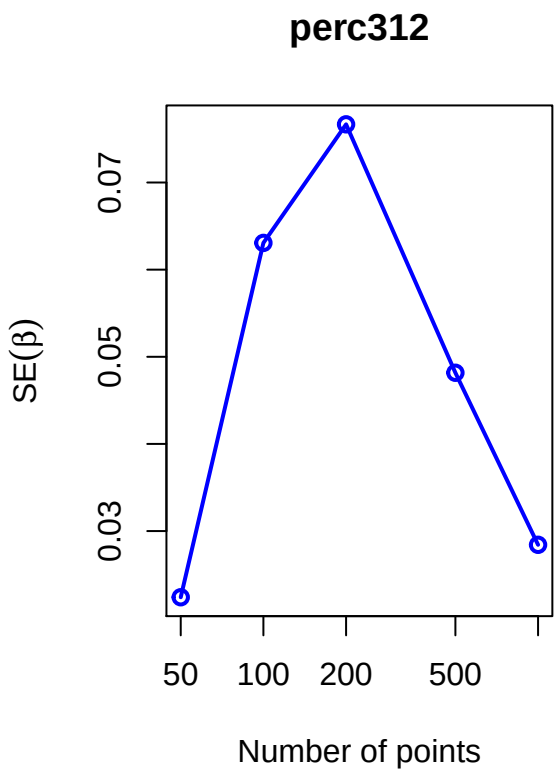

```
## $means
##       1000        500        200        100         50
## 0.27590565 0.27242454 0.20365113 0.07324764 0.03786603
##
## $sds
##       1000        500        200        100         50
## 0.02972891 0.04916710 0.13787098 0.12742810 0.09440406
```

```
coef_se_plot(v = 8, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##       1000        500        200        100         50
## 0.03949374 0.04029454 0.03627113 0.02824430 0.02591501
##
## $sds
##       1000        500        200        100         50
## 0.01915148 0.03071741 0.04956738 0.05751201 0.06217679
```

```
coef_se_plot(v = 9, fitN = fitAll, simbetas = beta.sims)
```



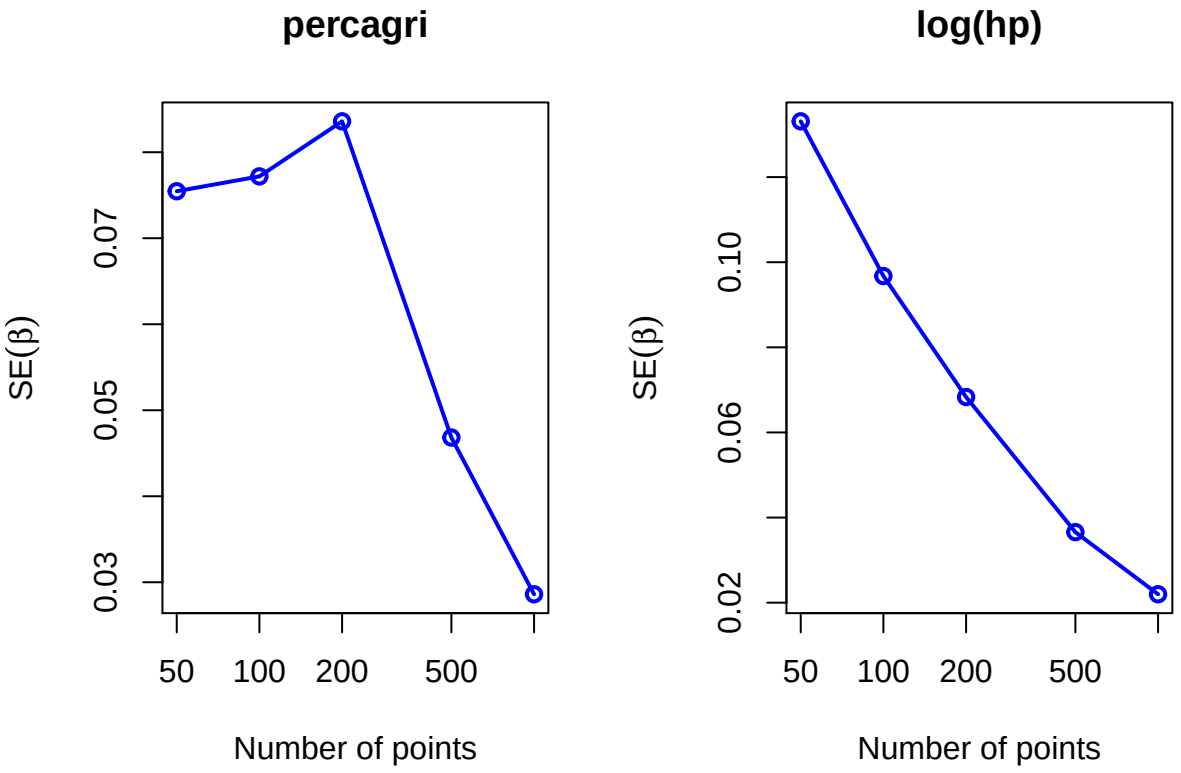



```
## $means
##          1000           500           200           100            50
## -0.0552693216 -0.0601793353 -0.0519973903 -0.0155730424 -0.0009872116
##
## $sds
##       1000        500        200        100         50
## 0.02842399 0.04816696 0.07667386 0.06306837 0.02240544
```

```
coef_se_plot(v = 10, fitN = fitAll, simbetas = beta.sims)
```

```
## $means
##        1000         500         200         100          50
## -0.10522984 -0.10537931 -0.08328826 -0.03543225 -0.02320848
##
## $sds
##       1000        500        200        100         50
## 0.02860646 0.04682451 0.08359032 0.07719108 0.07546346
```

```
coef_se_plot(v = 11, fitN = fitAll, simbetas = beta.sims)
```




```
## $means
##      1000       500       200       100        50
## 0.7286742 0.7225707 0.7037357 0.6453685 0.6059896
##
## $sds
##       1000        500        200        100         50
## 0.02198436 0.03658391 0.06832620 0.09675035 0.13309729
```

These plots confirm the general trends discussed in the previous section. As the subset size increases from $N = 50$ to $N = 200$, the standard deviations tend to increase as fewer of the estimated coefficients $\hat{\beta}_{j,i,N}$ are set to 0, and then decreases thereafter with increasing subset size.

4.3 Signs of the fitted coefficients

The sign of the coefficients may be of particular importance, as it provides insight into the nature of the effects of the corresponding covariates. A positive coefficient $\hat{\beta}_i$ implies that increasing values of covariate i are associated with higher intensities of the species, while a negative coefficient $\hat{\beta}_j$ implies that increasing values of covariate j are associated with lower intensities of the species. Meanwhile, a zero coefficient $\hat{\beta}_k = 0$ implies that covariate k has no effect on the species intensity.

The function `signcoefs()` summarises the sign information when supplied with `simbetas`, the array containing the coefficients of the models fitted to the random subsets.

```
signcoefs(simbetas = beta.sims)
```

```
## , , -
##
##      Intercept temp temp^2 prec temp*prec prec^2 perch10 perc311 perc312
```

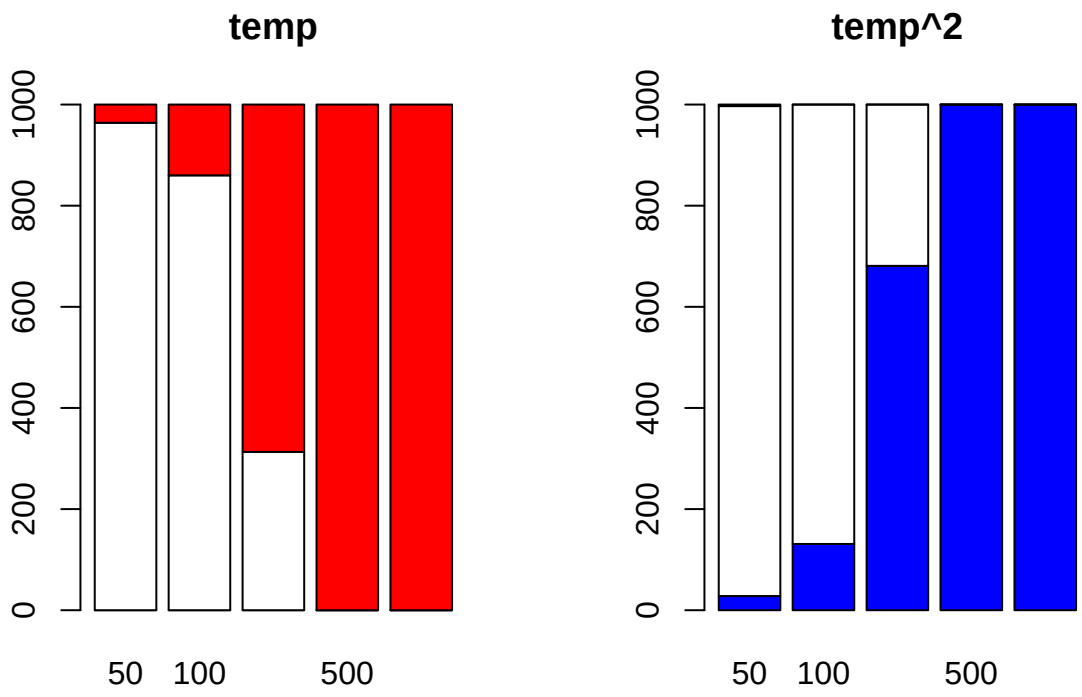
```
## 1000      1000    0   1000  196      1000      0       0      11     979
## 500       1000    0   1000  235       927      0       0      57     877
## 200       1000    0    681  146       420     21       2      85     536
## 100       1000    0    131   60        60     27       0      22     105
## 50        1000    0     28   44        14      9       0       5      15
##      percagri log(hp)
## 1000     1000       0
## 500       986       0
## 200       662       0
## 100       226       0
## 50        114       0
##
## , , 0
##
##      Intercept temp temp^2 prec temp*prec prec^2 perch10 perc311 perc312
## 1000         0    0      0  571         0      1       0      14      18
## 500          0    0      0  515        69     71       0      49      75
## 200          0  313    319  676       546    547     238     315     393
## 100          0  860    869  900       907    916     698     707     877
## 50           0  964    969  942       949    983     836     817     975
##      percagri log(hp)
## 1000        0       0
## 500         5       0
## 200       305       0
## 100       765       0
## 50        880       0
##
## , , +
##
##      Intercept temp temp^2 prec temp*prec prec^2 perch10 perc311 perc312
## 1000         0 1000      0  233         0    999    1000     975       3
## 500          0 1000      0  250         4    929    1000     894      48
## 200          0  687      0  178        34    432     760     600      71
## 100          0  140      0   40        33     57     302     271      18
## 50           0   36      3   14        37      8     164     178      10
##      percagri log(hp)
## 1000        0    1000
## 500         9    1000
## 200        33    1000
## 100         9    1000
## 50          6    1000
```

Here we see that an expected drift away from 0 with increasing subset size. This makes sense as models informed by more data points tend to require less shrinkage via the LASSO as judged by BIC. We can see unanimously positive effects of `temp`, `perch10`, and `log(hp)` when $N = 1000$, and unanimously negative effects of `temp^2`, `percagri`, and the interaction term `temp*prec`. Meanwhile, the coefficients for `prec^2` and `perc311` are nearly always positive with $N = 1000$, while the coefficient for `perc312` is nearly always negative. The only covariate without such a strong signal is `prec`, which is still estimated to be 0 in a majority of subsets of size $N = 1000$, and nearly equally balanced between positive and negative signs otherwise.

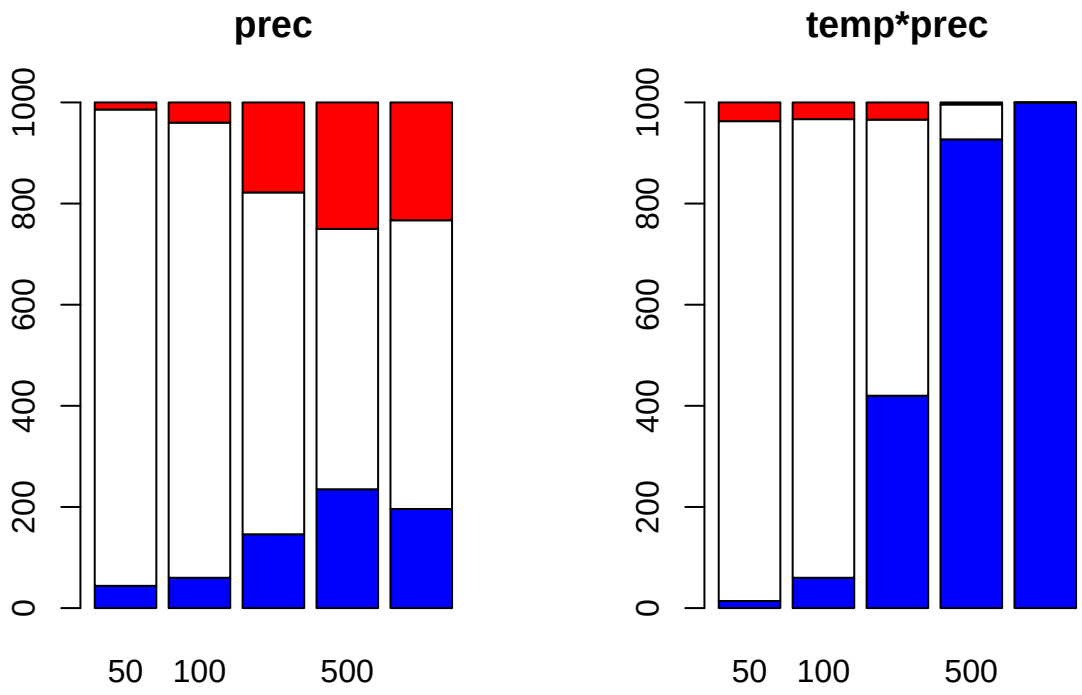
4.4 Bar plot of coefficient signs

The `signplot()` function visualises the signs of the fitted coefficients as a bar plot, with one bar for each subset size. The red, white, and blue portions of each bar correspond to positive, zero, and negative coefficient estimates.

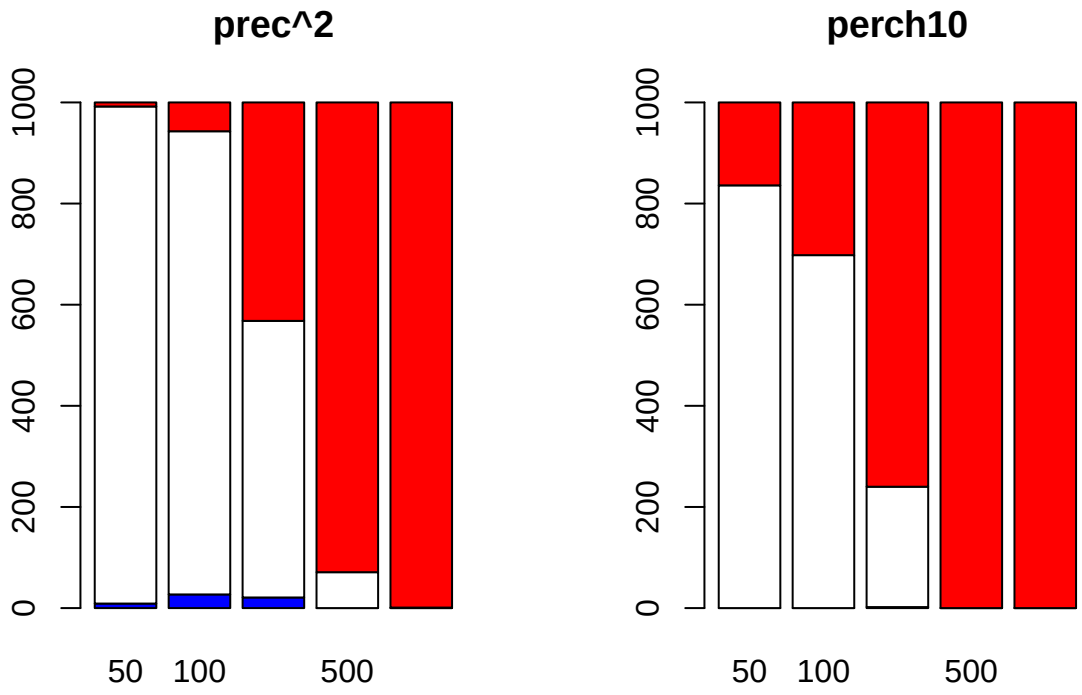
```
par(mfrow = c(1, 2))
signplot(simbetas = beta.sims, v = 2)
signplot(simbetas = beta.sims, v = 3)
```



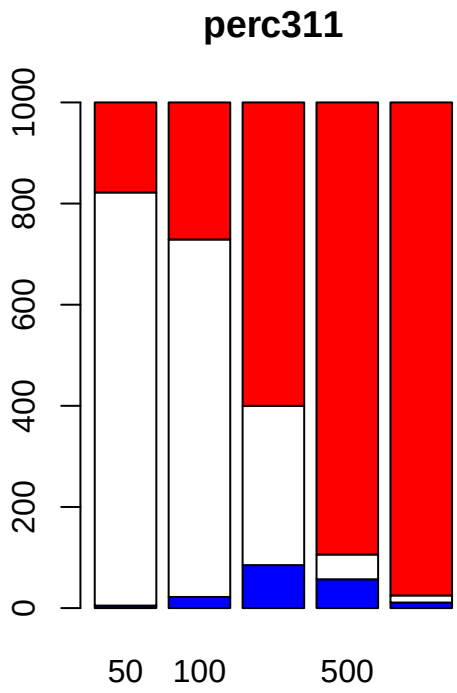

```
signplot(simbetas = beta.sims, v = 4)
signplot(simbetas = beta.sims, v = 5)
```

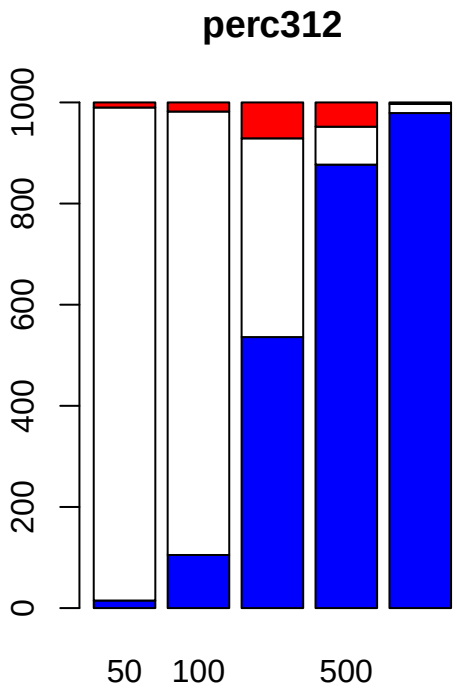



```
signplot(simbetas = beta.sims, v = 6)
signplot(simbetas = beta.sims, v = 7)
```

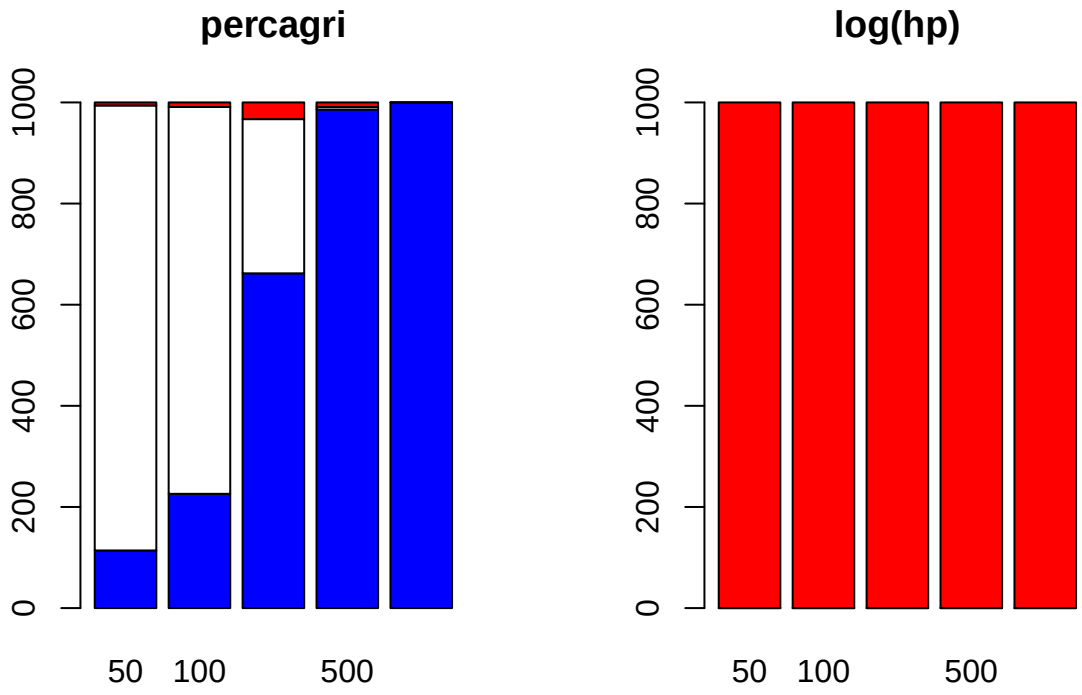



```
signplot(simbetas = beta.sims, v = 8)
signplot(simbetas = beta.sims, v = 9)
```






```
signplot(simbetas = beta.sims, v = 10)
signplot(simbetas = beta.sims, v = 11)
```



The drift away from 0 with increasing subset size is apparent with the vanishing white portions of the bars. We likewise confirm that even with a subset size of $N = 1000$, the estimated coefficients for `prec` vary in sign.

5 Creating raster objects

Users may wish to store some of the mapped values as raster objects. To this end, the function `makeraster` may be used to create a raster object in `R`, and to optionally save it as a `.tif` file. To create a raster object in `R`, we must load the `raster` package.

```
library(raster)
```

This function takes as arguments the following:

- `v_frame`: a data frame containing columns `X` and `Y` with the x and y coordinates as well as a column `v` of the values of the desired variable to be contained in the raster
- `proj`: the projection of the x and y coordinates specified in `v_frame`. By default, this is set to `NULL`, such that there is no projection associated with the raster.
- `toproj`: the desired transformed projection of the raster. By default, this is set to `NULL`, such that the output projection matches the projection set by the `proj` argument, if any.
- `saveTIFF`: if set to `TRUE`, a `.tif` file will be saved. By default, set to `FALSE`.
- `TIFFname`: the name of the `.tif` file to be created if `saveTIFF` is set to `TRUE`. By default, this is set to `"myTIFF.tif"`.

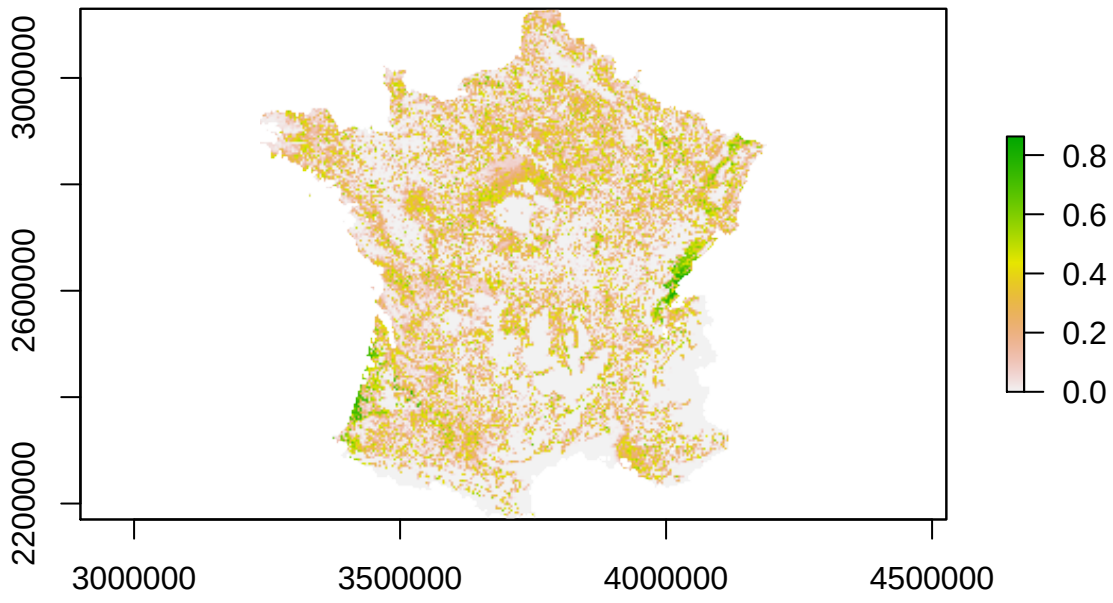
Our x and y coordinates were derived from the Lambert azimuthal equal-area (LAEA) projection, centred over Europe. We will define this projection in `R` as follows:

```
library(rgdal) # package for defining the projection with the CRS function
LAEA = CRS("+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000
```

```
           +ellps=GRS80 +units=m +no_defs ")
```

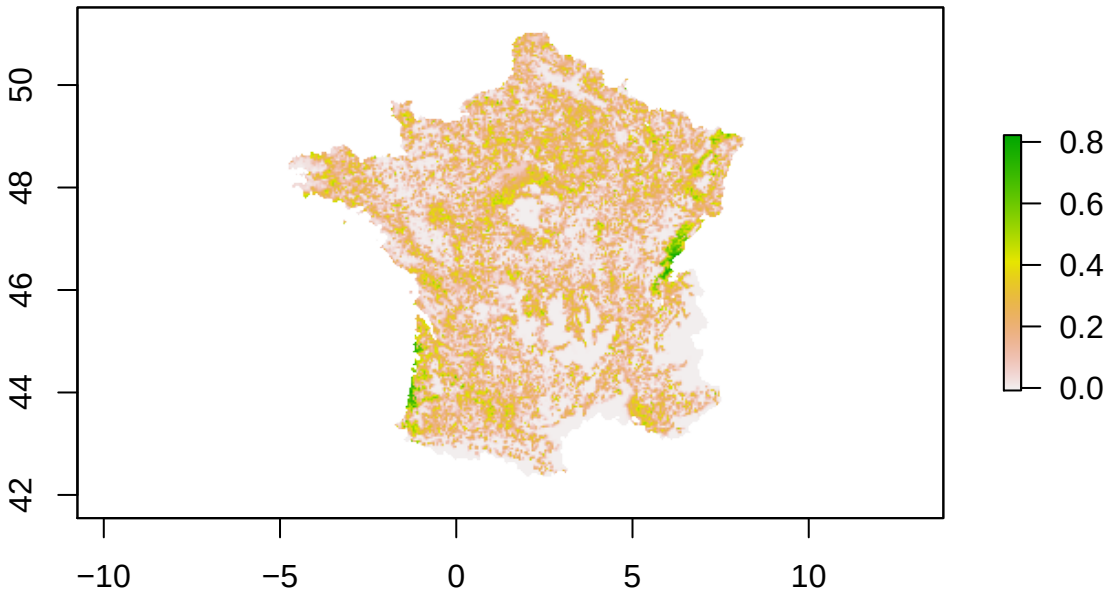
We can now make a raster of the quantile misalignment map based on subsets of size $N = 1000$:

```
q1000_frame = data.frame(X = env.4km.Fr$X, Y = env.4km.Fr$Y, v = q1000$prop_no_match)
q1000_raster = makeraster(q1000_frame, proj = LAEA)
plot(q1000_raster)
```



We can convert to another projection, such as World Geodetic System of 1984 (WGS84), as follows:

```
WGS84 = CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
q1000_raster_WGS84 = makeraster(q1000_frame, proj = LAEA, toproj = WGS84)
plot(q1000_raster_WGS84)
```



Some users may wish to save the raster to a file that can be used with other mapping software, such as QGIS. We can make a `.tif` file of the raster as follows:

```
q1000_raster_savetiff = makeraster(q1000_frame, proj = LAEA,
                          saveTIFF = TRUE, TIFFname = "Q1000_LAEA.tif")
```

References

Renner, Ian W, Jane Elith, Adrian Baddeley, William Fithian, Trevor Hastie, Steven J Phillips, Gordana Popovic, and David I Warton. 2015. "Point Process Models for Presence-Only Analysis." *Methods in Ecology and Evolution* 6 (4): 366–79.

Warton, David I, Ian W Renner, and Daniel Ramp. 2013. "Model-Based Control of Observer Bias for the Analysis of Presence-Only Data in Ecology." *PloS One* 8 (11): e79168.