Hmsc 3.0: Getting started with Hmsc: univariate models

Gleb Tikhonov Øystein H. Opedal Nerea Abrego Aleksi Lehikoinen
Melinda M. J. de Jonge Jari Oksanen Otso Ovaskainen

19 February 2021

Introduction

The Hierarchical Modelling of Species Communities (HMSC) framework is a statistical framework for analysis of multivariate data, typically from species communities. Here, we demonstrate how to get started with Hmsc-R. While HMSC is primarily meant for multivariate data, this vignette illustrates the basics with univariate models.

```
library(Hmsc)
```

We set the random number seed to make the results presented here reproducible.

```
set.seed(1)
```

Linear model

As the first case study, we use HMSC to fit a univariate linear model. To relate the results to something that we expect the reader to be familiar with, we also apply the basic `lm` function to the same data and compare the results.

Generating simulated data

For illustrative purposes, we use simulated data for which we know the parameter values.

```
n = 50
x = rnorm(n)
alpha = 0
beta = 1
sigma = 1
L = alpha + beta*x
y = L + rnorm(n, sd = sigma)
plot(x, y, las=1)
```

Here `n` is the number of data points, `x` is a continuous covariate, `alpha` and `beta` are the true parameters for intercept and slope, `L` is the linear predictor, and `y` is the response variable. We note that the data are simulated by the standard linear model with normally distributed residuals, $y_i = \alpha + \beta x_i + \epsilon_i$, where $\epsilon_i \sim N(0, \sigma^2)$.

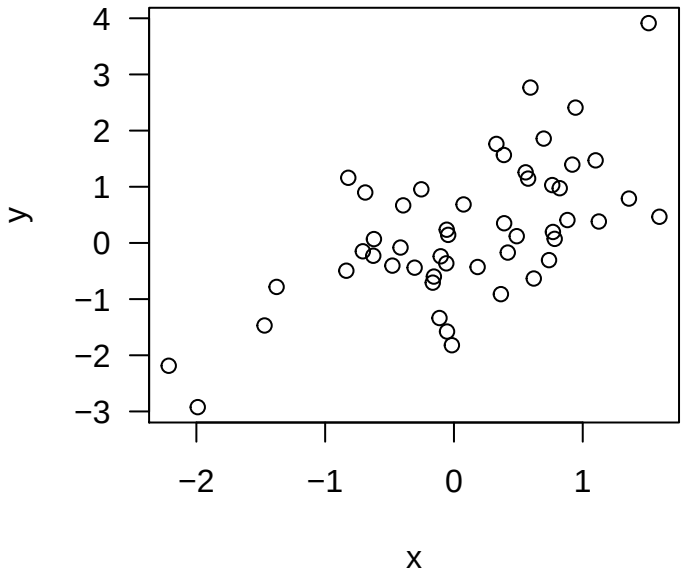

Figure 1: Scatterplot of simulated data.

Fitting models and looking at parameter estimates

The standard way to analyse these kinds of data with maximum likelihood inference is to use the `lm` function:

```
df = data.frame(x,y)
m.lm = lm(y ~ x, data=df)
summary(m.lm)
```

```
##
## Call:
## lm(formula = y ~ x, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.92760 -0.66898 -0.00225  0.48768  2.34858
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.1219     0.1394   0.875    0.386
## x             0.9545     0.1681   5.679 7.73e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9781 on 48 degrees of freedom
## Multiple R-squared:  0.4019, Adjusted R-squared:  0.3894
## F-statistic: 32.25 on 1 and 48 DF,  p-value: 7.726e-07
```

We note that, as expected, the parameter estimates roughly correspond to the values we assumed for the intercept ($\alpha = 0$) and slope ($\beta = 1$) when generating the data.

To conduct the analogous analyses with HMSC, we first construct the model as

```
Y = as.matrix(y)
XData = data.frame(x = x)
m = Hmsc(Y = Y, XData = XData, XFormula = ~x)
```

While the `lm` function constructs the model and fits it at the same time, the `Hmsc` function only constructs the model object. To fit the HMSC model with Bayesian inference, we use the `sampleMcmc` function. When calling `sampleMcmc`, we need to decide how many chains to sample (`nChains`), how many samples to obtain per chain (`samples`), how long transient (also called burn-in) to include (`transient`), and how frequently we wish to see the progress of the MCMC sampling (`verbose`). MCMC sampling can take a lot of time, so we have included two options below. By setting `test.run = TRUE`, the entire .Rmd version of the vignette can be run quickly, but the parameter estimates will not be reliable. However, that does not matter if the aim of running the vignette is e.g. to get familiar with the syntax of HMSC-R, to examine the structure of the constructed objects and the inputs and outputs of the functions. When setting `test.run = FALSE`, a much larger amount of MCMC sampling is conducted, and running the .Rmd version of the vignette will reproduce the results shown in the .pdf version of the vignette.

```
nChains = 2
test.run = FALSE
if (test.run){
   #with this option, the vignette runs fast but results are not reliable
   thin = 1
   samples = 10
   transient = 5
   verbose = 5
} else {
   #with this option, the vignette evaluates slow but it reproduces the results of the
   #.pdf version
   thin = 5
   samples = 1000
   transient = 500*thin
   verbose = 500*thin
}
```

We are now ready to call `sampleMcmc` and thus estimate the model parameters.

```
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                    nChains = nChains, verbose = verbose)
```

```
## setting updater$GammaEta=FALSE due to absence of random effects included to the model

## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent lo

## Computing chain 1
## Chain 1, iteration 2500 of 7500 (transient)
## Chain 1, iteration 5000 of 7500 (sampling)
## Chain 1, iteration 7500 of 7500 (sampling)
## Computing chain 2
## Chain 2, iteration 2500 of 7500 (transient)
## Chain 2, iteration 5000 of 7500 (sampling)
## Chain 2, iteration 7500 of 7500 (sampling)
```

As briefly explained above, the parameters (`thin`, `samples`, `transient`, `nChains`) control how the posterior distribution is sampled with the MCMC method. We will return to this issue soon. But the main thing is that now the model object `m` also includes estimated parameters, in the same way as the object `m.lm` includes the parameters estimated by the `lm` function.

To look at the parameter estimates, we extract the posterior distribution from the model object and convert it into a `coda` object.

```
mpost = convertToCodaObject(m)
summary(mpost$Beta)
```

```
##
## Iterations = 2505:7500
## Thinning interval = 5
## Number of chains = 2
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                                  Mean     SD Naive SE Time-series SE
## B[(Intercept) (C1), sp1 (S1)] 0.1187 0.1434 0.003207       0.003300
## B[x (C2), sp1 (S1)]           0.9485 0.1598 0.003573       0.003574
##
## 2. Quantiles for each variable:
##
##                                  2.5%     25%    50%    75%  97.5%
## B[(Intercept) (C1), sp1 (S1)] -0.161 0.02432 0.1189 0.2169 0.4049
## B[x (C2), sp1 (S1)]            0.632 0.84385 0.9478 1.0563 1.2674
```

To assess model fit in terms of R^2, we apply the `evaluateModelFit` function to the posterior predictive distribution computed by the function `computePredictedValues`.

```
preds = computePredictedValues(m)
evaluateModelFit(hM=m, predY=preds)
```

```
## $RMSE
## [1] 0.9583775
##
## $R2
## [1] 0.4018712
```

We note that the parameter estimates and R^2 given by HMSC are highly consistent with those given by the `lm` function. We note, however, that the two approaches are not identical as `lm` applies the maximum likelihood framework (and thus e.g. yields confidence intervals) whereas HMSC applies the Bayesian framework (and thus e.g. assumes prior distributions and yields credible intervals).

Model fit can be evaluated in many ways. The second measure of model fit returned here by the `evaluateModelFit` function is `RMSE`, i.e. the root-mean-square error.

Checking MCMC diagnostics

Let us now return to the parameters that guide posterior sampling in the MCMC algorithm. We first plot the trace plots of the β-parameters.

```
plot(mpost$Beta)
```

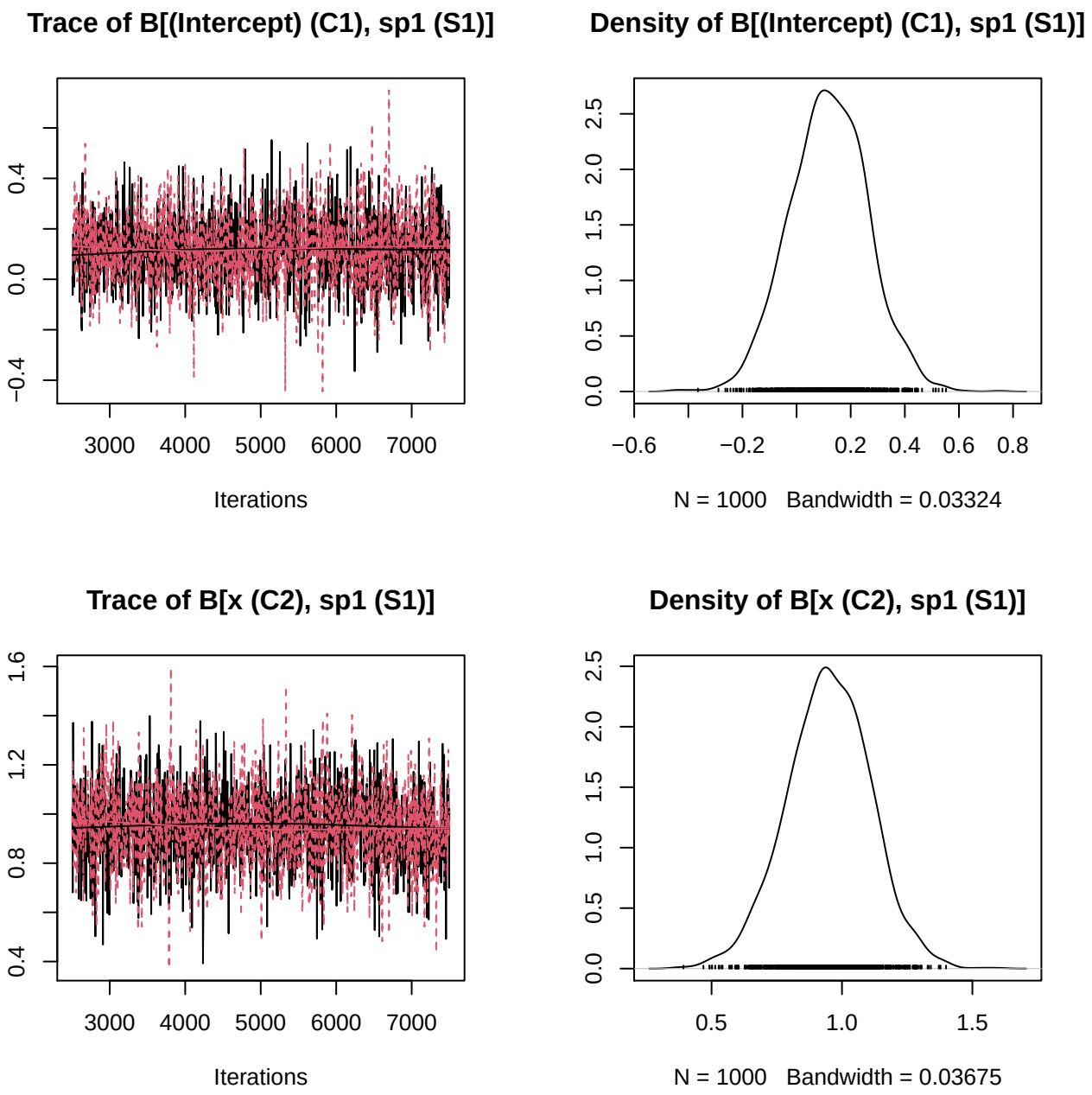

Figure 2: Posterior trace plots of Beta parameters.

The black and red lines show the (`nChains`=2) independent MCMC chains. Both chains start from iteration 2501, as by setting `transient`=2500 we have chosen to ignore iterations before that as a possible transient. The chains have run in total 7500 iterations each, as we selected to obtain 1000 samples, and we have recorded every 5:th step (`thin`=5) of the iterations.

If running the 'test.run = FALSE' version of the vignette, the trace-plots look essentially as good as they can ever look like. First of all, the two chains yield essentially identical results. Second, the chains mix very well, i.e. they go fast up and down without any apparent autocorrelation. Third, they seem to have reached a stationary distribution, as e.g. the first half of the recorded iterations looks statistically identical to the

second half of the recorded iterations.

We may also evaluate MCMC convergence more quantitatively in terms of effective sample sizes and potential scale reduction factors.

```
effectiveSize(mpost$Beta)
```

```
## B[(Intercept) (C1), sp1 (S1)]           B[x (C2), sp1 (S1)]
##                       1888.798                      2000.000
```

```
gelman.diag(mpost$Beta,multivariate=FALSE)$psrf
```

```
##                               Point est. Upper C.I.
## B[(Intercept) (C1), sp1 (S1)]   1.001248   1.001255
## B[x (C2), sp1 (S1)]             1.001690   1.008866
```

We observe that the effective sample sizes are very close to the theoretical value of the actual number of samples, which is 2000 (1000 per chain). This indicates that there is very little autocorrelation among consecutive samples. The potential scale reduction factors are very close to one, which indicates that two chains gave consistent results, as suggested by visual inspection of the trace plots.

In summary, the MCMC diagnostics did not indicate any problems with MCMC convergence. This means that the posterior sample is likely to be representative of the true posterior distribution, and thus the inference from the model can be trusted. If the MCMC convergence would have indicated problems, we should have refitted the model using different parameters. If the trace-plots would have suggested the presence of a transient (the early iterations would have looked different from the later iterations), we should have increased the amount of transient iterations to be discarded. If the chains would have shown autocorrelations and/or differed from each other, we should have increased the number of iterations. This can be done by increasing either the number of samples, or by keeping the number of samples fixed but increasing thinning. We recommend the latter, as increasing the number of samples can make the model objects very big and lead to computationally expensive post-processing. If one needs to run a million iterations, we recommend doing so by `samples=1000` and `thin=1000` rather than `samples=1000000` and `thin=1`. In our view, 1000 samples are typically sufficient to evaluate e.g. posterior means and credible intervals with sufficient accuracy.

Checking the assumptions of the linear model

When applying the linear model, it is a good practice to examine if the assumptions of the linear model are upheld. In the context of the `lm` function, this can be done e.g. by constructing the following diagnostic plots.

```
nres.lm = rstandard(m.lm)
preds.lm = fitted.values(m.lm)
par(mfrow=c(1,2))
hist(nres.lm, las = 1)
plot(preds.lm,nres.lm, las = 1)
abline(a=0,b=0)
```

The first plot shows that the residuals conform well to the assumption of normality, and the second plot shows that the residuals are homoscedastic. This is not surprising, as the data were simulated from the linear model.

Structural model assumptions can be checked with HMSC basically in the same way as with the standard linear model. However, while for models fitted with the `lm` function simply typing `plot(m.lm)` would provide many diagnostic plots "automatically", with HMSC there is no such built-in functionality. This is because

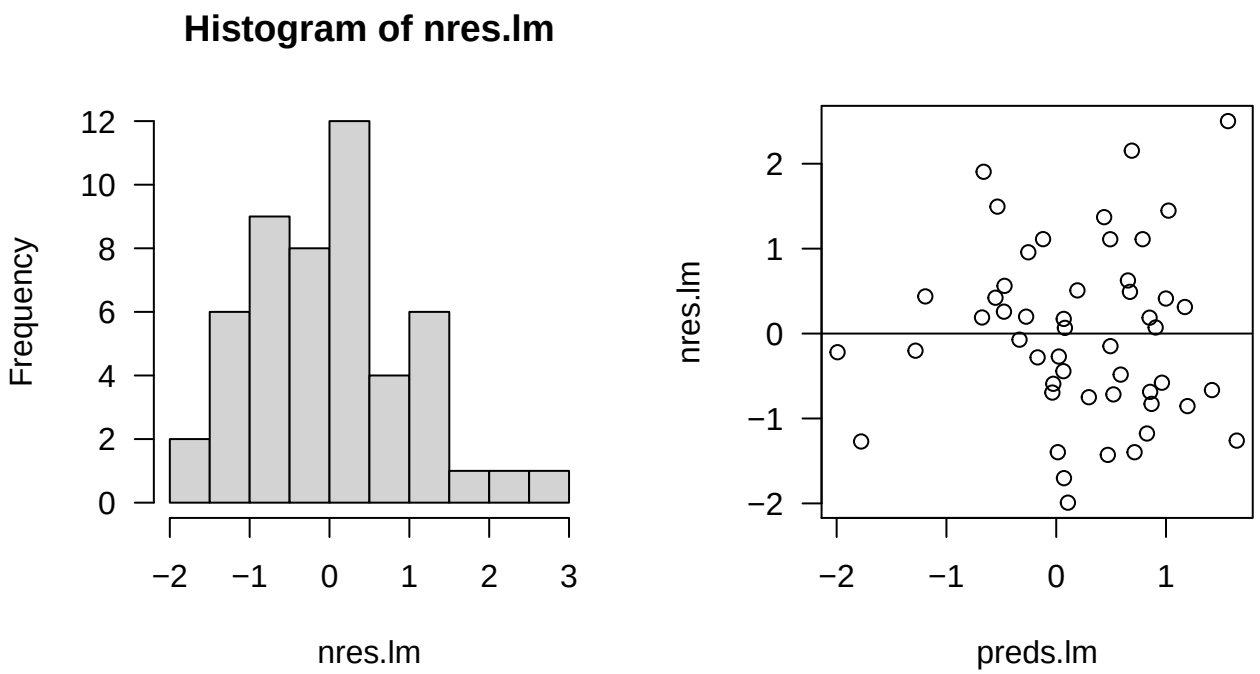

Figure 3: Residuals of the linear model fitted with the lm() function, shown as a histogram and as a function of the fitted values.

typical applications of HMSC relate to much more complex models where it is not straightforward to decide what a "standard" diagnostic plot should be. To generate diagnostic plots with HMSC, we first summarize the posterior predictive distribution into posterior mean and then extract and standardize the residuals.

```
preds.mean = apply(preds, FUN=mean, MARGIN=1)
nres = scale(y-preds.mean)
par(mfrow=c(1,2))
hist(nres, las = 1)
plot(preds.mean,nres, las = 1)
abline(a=0,b=0)
```

These diagnostic plots are essentially identical to those obtained for the linear model fitted with the `lm` function. This is to be expected, as we observed earlier that the parameter estimates (and hence the fitted values and residuals) were consistent between the two approaches.

Generalized linear models

Above, we have fitted a linear model with normally distributed residuals. As this has been set as the default option for HMSC, we defined our model above simply as

```
m = Hmsc(Y=Y, XData=XData, XFormula=~x)
```

whereas the full version that makes the assumption of normality transparent would be

```
m = Hmsc(Y=Y, XData=XData, XFormula=~x, distr = "normal")
```

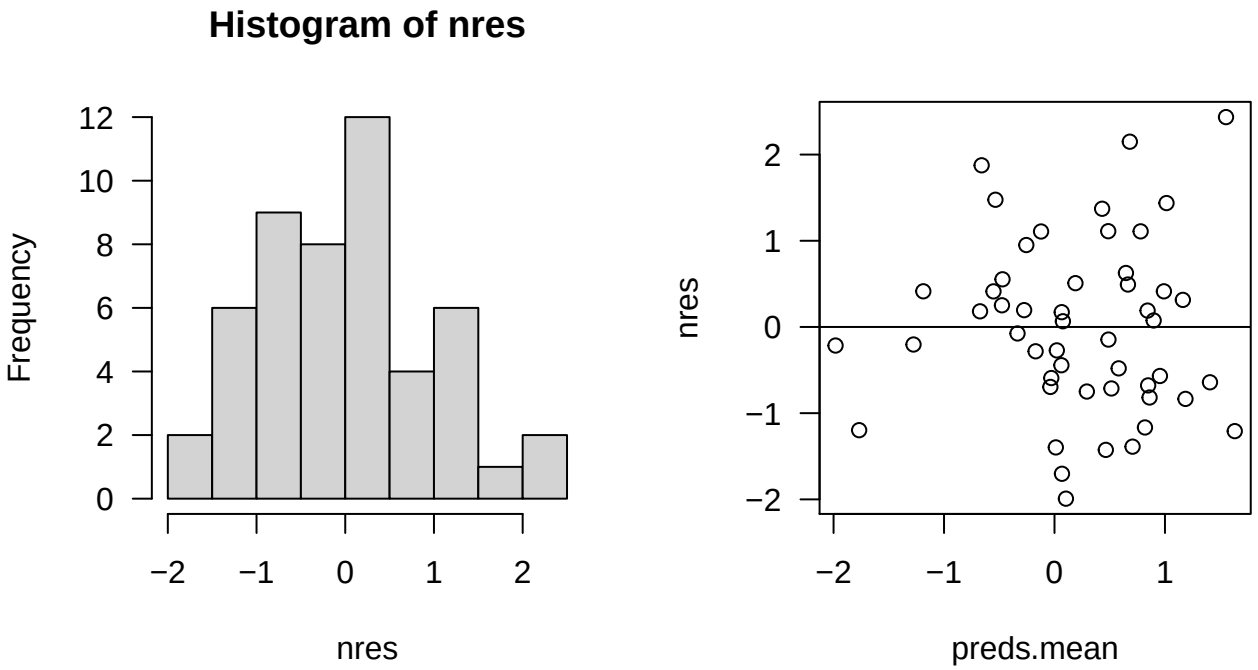

Figure 4: Residuals of the linear model fitted with HMSC, shown as a histogram and as a function of the fitted values.

In addition to the normal distribution, HMSC-R 3.0 allows three other types of link functions and error distributions: probit model for presence-absence data, and Poisson and lognormal Poisson models for count data. We note that while also many other kinds of link functions and error distributions can be quite straightforwardly implemented in the univariate framework, this is not the case for HMSC due to its multivariate and hierarchical nature. Thus, implementing other kinds of link functions and error distributions is a challenge for the future.

Probit model for presence-absence data

A common case with community ecology data is that of presence-absence data, meaning that the response variable is either 0 (the species is absent) or 1 (the species is present). In HMSC-R, such data can be modelled with probit regression. For those readers not familiar with probit regression, we note that it is similar to logistic regression, it just applies a slightly different link-function (probit instead of logit). We next repeat the above exercise but do so for data that conform to the assumptions of probit regression.

```
y = 1*(L+ rnorm(n, sd = 1)>0)
plot(x,y, las = 1)
```

```
Y=as.matrix(y)
m = Hmsc(Y=Y, XData=XData, XFormula=~x, distr="probit")
verbose = 0
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                    nChains = nChains, verbose = verbose)
```

```
## setting updater$GammaEta=FALSE due to absence of random effects included to the model

## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l
```

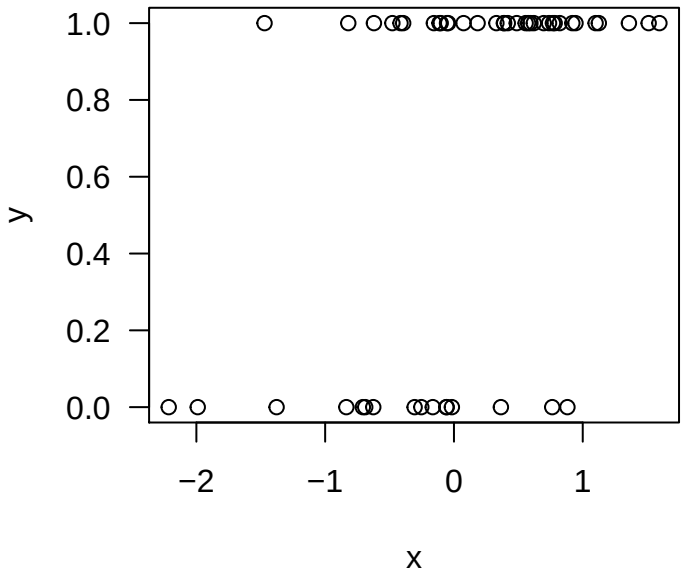

Figure 5: Scatterplot of simulated binary data.

```
## Computing chain 1
## Computing chain 2
```

```
mpost = convertToCodaObject(m)
summary(mpost$Beta)
```

```
##
## Iterations = 2505:7500
## Thinning interval = 5
## Number of chains = 2
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                                  Mean     SD Naive SE Time-series SE
## B[(Intercept) (C1), sp1 (S1)] 0.4594 0.1977 0.004422       0.004987
## B[x (C2), sp1 (S1)]           0.8689 0.2745 0.006137       0.006621
##
## 2. Quantiles for each variable:
##
##                                   2.5%    25%    50%    75%  97.5%
## B[(Intercept) (C1), sp1 (S1)] 0.07323 0.3270 0.4566 0.5854 0.8547
## B[x (C2), sp1 (S1)]           0.36155 0.6767 0.8691 1.0473 1.4365
```

```
effectiveSize(mpost$Beta)
```

```
## B[(Intercept) (C1), sp1 (S1)]           B[x (C2), sp1 (S1)]
##                      1597.230                       1719.138
```

```
gelman.diag(mpost$Beta, multivariate=FALSE)$psrf
```

```
##                               Point est. Upper C.I.
## B[(Intercept) (C1), sp1 (S1)]  1.0000493  1.0028940
## B[x (C2), sp1 (S1)]            0.9994034  0.9995767
```

```
preds = computePredictedValues(m)
evaluateModelFit(hM=m, predY=preds)
```

```
## $RMSE
## [1] 0.4133796
##
## $AUC
## [1] 0.7794118
##
## $TjurR2
## [1] 0.2108186
```

We note that we have now set `verbose = 0` to suppress the output on progress of MCMC sampling. Generally we recommend the user not to set `verbose = 0` to see that MCMC sampling is moving as it should and to be able to estimate how long it will take before the sampling finishes. We have suppressed the output just to make the vignette a bit more compact.

Compared to the case of the linear model, we observe that the effective sample size is somewhat smaller (achieving MCMC convergence is generally more challenging in non-normal models), and the parameter estimates include more uncertainty (0/1 data are less informative than normally distributed data). We further note that instead of the R^2 of the linear model, model fit is now evaluated in terms of `AUC` and Tjur R^2.

Poisson model for count data

Another common case with community ecology data is that the response variable is a non-negative integer 0,1,2,3,..., representing the count of individuals. In HMSC-R, such data can be modelled with Poisson regression. We next repeat the above exercise but do so for data that conform the assumptions of Poisson regression.

```
y = rpois(n, lambda = exp(L))
plot(x,y,las=1)
```

```
Y=as.matrix(y)
m = Hmsc(Y=Y, XData=XData, XFormula=~x, distr="poisson")
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                    nChains = nChains, verbose = verbose)
```

```
## setting updater$GammaEta=FALSE due to absence of random effects included to the model
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l
```

```
## Computing chain 1
## Computing chain 2
```

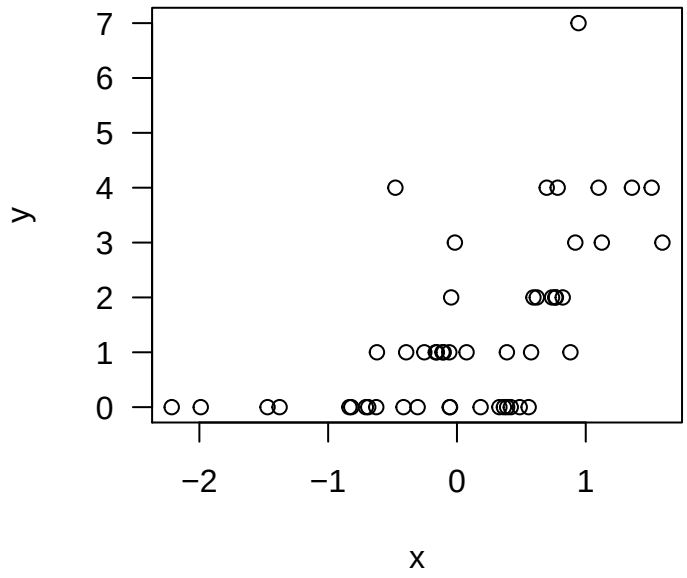

Figure 6: Scatterplot of simulated count data.

```
mpost = convertToCodaObject(m)
summary(mpost$Beta)
```

```
##
## Iterations = 2505:7500
## Thinning interval = 5
## Number of chains = 2
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                                  Mean     SD Naive SE Time-series SE
## B[(Intercept) (C1), sp1 (S1)] -0.2075 0.1837 0.004108        0.04833
## B[x (C2), sp1 (S1)]            1.1549 0.1799 0.004022        0.03345
##
## 2. Quantiles for each variable:
##
##                                  2.5%     25%     50%      75% 97.5%
## B[(Intercept) (C1), sp1 (S1)] -0.5932 -0.3192 -0.1968 -0.08444 0.137
## B[x (C2), sp1 (S1)]            0.8275  1.0353  1.1491  1.25811 1.565
```

```
effectiveSize(mpost$Beta)
```

```
## B[(Intercept) (C1), sp1 (S1)]           B[x (C2), sp1 (S1)]
##                      30.77823                      37.97361
```

```
gelman.diag(mpost$Beta, multivariate=FALSE)$psrf
```

```
##                               Point est. Upper C.I.
```

```
## B[(Intercept) (C1), sp1 (S1)]   1.213076   1.705965
## B[x (C2), sp1 (S1)]             1.066215   1.206109
```

```
preds = computePredictedValues(m, expected = FALSE)
evaluateModelFit(hM=m, predY=preds)
```

```
## $RMSE
## [1] 1.232883
##
## $SR2
## [1] 0.4585446
##
## $O.AUC
## [1] 0.8133333
##
## $O.TjurR2
## [1] 0.2723417
##
## $O.RMSE
## [1] 0.4173407
##
## $C.SR2
## [1] 0.3726606
##
## $C.RMSE
## [1] 1.441995
```

In this case, the MCMC diagnostics are much worse than for the probit model. Unfortunately, achieving MCMC convergence for a Poisson model in the HMSC context is highly challenging. To get an unbiased sample from the posterior and results that can be fully trusted, one should increase the thinning and thus run the model longer.

For the Poisson model, many kinds of measures of model fit are given. The measure `SR2` can be viewed as a pseudo-R^2. Whereas the R^2 of a linear model can be computed as the squared Pearson correlation between predicted and true values, we have computed `SR2` as the squared Spearman correlation between observed and predicted values. As the Poisson model predicts counts, it can be used to separate whether a species is present (count>0) or absent (count=0). For evaluating how well species occurrences (indicated by `O.`) are predicted, the same measures (`AUC` and Tjur R^2) can be computed as for the probit model. It is also of interest to examine how well the model is able to predict abundance variation in cases where the species is present, and thus ignoring absences. This is done with the measures indicated by `C.`, where C refers to "conditional on presence". Note that here we have generated the predictions with the option `expected = FALSE`. In this case, the predictions are not expected values (e.g. on average we expect to see 2.3 individuals), but a posterior predictive distribution of data, i.e., actual counts 0,1,2,3,... that include the Poisson sampling error. We have selected `expected = FALSE` so that presence-absences can be inferred from the predictions, enabling us to compute also the `O.`and `C.` variants of measures of model fit.

Lognormal Poisson model for count data

The Poisson model is often not a good model for ecological count data as it does not allow sufficient amount of variation around the expectation. To allow for more variation, HMSC-R includes the possibility to fit a lognormal Poisson model. We note that another standard option would be to fit the Negative Binomial model, but this model is not included currently in HMSC-R. We next repeat the above exercise but do so for data that conform to the assumptions of lognormal Poisson regression.

```
y = rpois(n, lambda = exp(L+rnorm(n, sd=2)))
plot(x,y, las = 1)
```

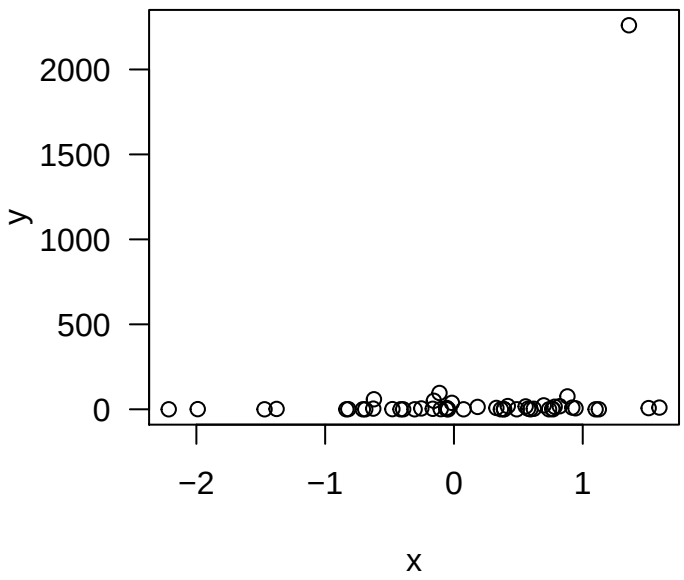

Figure 7: Scatterplot of simulated lognormal count data.

```
Y=as.matrix(y)
m = Hmsc(Y=Y, XData=XData, XFormula=~x, distr="lognormal poisson")
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                    nChains = nChains, verbose = verbose)
```

```
## setting updater$GammaEta=FALSE due to absence of random effects included to the model

## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l

## Computing chain 1
## Computing chain 2
```

```
mpost = convertToCodaObject(m)
summary(mpost$Beta)
```

```
##
## Iterations = 2505:7500
## Thinning interval = 5
## Number of chains = 2
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                                  Mean     SD Naive SE Time-series SE
## B[(Intercept) (C1), sp1 (S1)] 0.3957 0.3903 0.008727        0.03189
## B[x (C2), sp1 (S1)]           1.0843 0.4654 0.010406        0.02666
```

```
##
## 2. Quantiles for each variable:
##
##                                    2.5%    25%    50%    75% 97.5%
## B[(Intercept) (C1), sp1 (S1)] -0.3796 0.1443 0.4099 0.6561 1.126
## B[x (C2), sp1 (S1)]            0.1888 0.7642 1.0869 1.3884 2.022
```

```
effectiveSize(mpost$Beta)
```

```
## B[(Intercept) (C1), sp1 (S1)]           B[x (C2), sp1 (S1)]
##                      156.6817                      305.4497
```

```
gelman.diag(mpost$Beta, multivariate=FALSE)$psrf
```

```
##                                Point est. Upper C.I.
## B[(Intercept) (C1), sp1 (S1)]   1.007102   1.036059
## B[x (C2), sp1 (S1)]             1.012604   1.059545
```

```
preds = computePredictedValues(m, expected = FALSE)
evaluateModelFit(hM=m, predY=preds)
```

```
## $RMSE
## [1] 319.4837
##
## $SR2
## [1] 0.06079167
##
## $O.AUC
## [1] 0.5911458
##
## $O.TjurR2
## [1] 0.04102951
##
## $O.RMSE
## [1] 0.4753874
##
## $C.SR2
## [1] 0.09984074
##
## $C.RMSE
## [1] 399.084
```

We note that the MCCM mixing for the lognormal Poisson model is again much worse that for the normal model. As illustrated above, the same types of output can be given for the lognormal Poisson model as for the standard Poisson model.

Hurdle models

Ecological count data are often dominated by zeros, i.e. they are zero-inflated. One standard model that can be fitted to such data is the Zero-Inflated Poisson model. While HMSC-R 3.0 does not include this or

other zero-inflated models, we note that it is always possible to apply the closely-related Hurdle approach. In the Hurdle model, one analyses separately occurrence data `1*(Y>0)` and abundance conditional on presence `Y[Y==0] = NA`. We note that these two aspects of the data (occurrence and abundance conditional on presence) are statistically independent of each other, and thus it is interesting to compare results obtained for them, e.g. to see if variation in occurrence and variation in abundance are explained by the same environmental covariates. In HMSC-R 3.0, occurrence data are analysed by the probit model, whereas the best model for abundance (conditional on presence) depends on the type of the data.

How to select the best model?

When fitting generalized linear models in the maximum likelihood framework, a common way of comparing models is to use AIC or some other information criteria. For example, in the case of count data, AIC could be used to compare the Poisson model, the Negative Binomial model, and the Zero-Inflated Poisson model. In the case of HMSC-R, model selection is recommended to be conducted through evaluating predictive performance through cross-validation. We illustrate cross-validation strategies below in the context of mixed models.

Mixed models: fixed and random effects

We next turn to mixed models, i.e. models with both fixed and random effects. For simplicity, we consider here linear models only, even if similar analyses could be done with probit, Poisson or lognormal Poisson models.

Hierarchical structure: sampling units within plots

We first consider a hierarchical study design in which 100 sampling units belonging to a discrete set of `np=10` plots. We assume that the plots have an additive effect to the response variable, i.e. we consider a random intercept model.

```
n = 100
x = rnorm(n)
alpha = 0
beta = 1
sigma = 1
L = alpha + beta*x
np = 10
sigma.plot = 1
sample.id = 1:n
plot.id = sample(1:np, n, replace = TRUE)
ap = rnorm(np, sd = sigma.plot)
a = ap[plot.id]
y = L + a + rnorm(n, sd = sigma)
plot.id = as.factor(plot.id)
plot(x,y,col = plot.id, las = 1)
```

```
XData = data.frame(x = x)
Y = as.matrix(y)
```

In the maximum likelihood framework, one option for fitting a mixed model would be to apply the `lmer` function as `lmer(y~x+(1|plot.id),data=da)`. Fitting a similar model with HMSC can be done as follows.

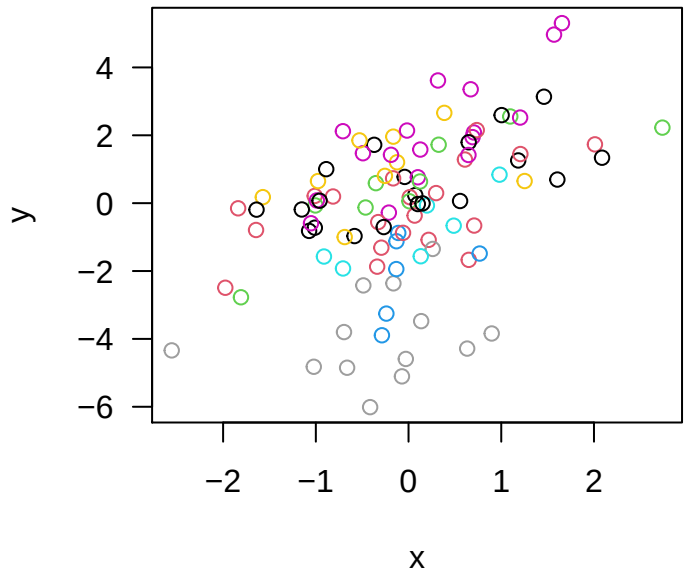

Figure 8: Scatterplot of simulated data originating from ten plots, as indicated by colors.

```
studyDesign = data.frame(sample = as.factor(sample.id), plot = as.factor(plot.id))
rL = HmscRandomLevel(units = studyDesign$plot)
m = Hmsc(Y=Y, XData=XData, XFormula=~x,
         studyDesign=studyDesign, ranLevels=list("plot"=rL))
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                    nChains = nChains, verbose = verbose)
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l
```

```
## Computing chain 1
## Computing chain 2
```

Here we have included two new input parameters. First, `studyDesign` defines the nature of the study design, here the individual sampling units and the plots to which the sampling units belong. Second, `ranLevels` includes a list of those aspects of the study design to which a random effect will be included, here the plot. The random effect object `rL` was created with the function `HmscRandomLevel` assuming the default options for latent variables. We note that in typical applications of HMSC, the data are multivariate, and the latent factor approach is used to reduce dimensionality of residual association matrices by having a smaller number of latent factors than the number of species. As we assume a univariate model, it is not necessary to have more than one latent variable, and in general it is not necessary to have more latent variables than there are speciess. This is because having the same number of latent variables as species is sufficient to generate a full rank species association matrix. For this reason, when calling `Hmsc`, the default option is to reduce the maximal number of factors to be at most the number of species, and thus in the univariate case the above script fixes the number of the underlying latent factors to one.

Let us now evaluate model fit as we have done above.

```
preds = computePredictedValues(m)
MF = evaluateModelFit(hM=m, predY=preds)
MF$R2
```

```
## [1] 0.7873702
```

Here, we have predicted the same data that were used to fit the model, and thus R^2 measures explanatory power rather than predictive power. If we would add more predictors, we could make R^2 approach one even if the predictors would represent just random noise, a phenomenon known as overfitting. For this reason, the predictive power of the model should be evaluated more critically by cross-validation.

Examples of cross-validation strategies

To apply cross-validation, we first divide the samples randomly into a number of groups ('folds'). For example, we may apply two-fold cross-validation across the samples by making the following partition

```
partition = createPartition(m, nfolds = 2, column = "sample")
partition
```

```
##   [1] 2 2 2 1 2 1 2 1 1 1 1 2 2 1 2 1 1 2 2 2 1 2 2 2 1 1 1 1 1 2 2 2 1 1 1 1 1
##  [38] 1 1 2 1 1 1 2 1 2 1 1 2 2 2 2 1 1 1 1 2 2 1 1 1 1 1 1 2 2 2 1 2 1 2 2 2 1
##  [75] 2 1 2 1 2 2 2 2 2 2 2 1 2 1 2 2 1 1 1 2 2 1 2 2 2 1
```

The idea of cross-validation is that when making predictions for a particular fold, data from that fold is not used for parameter estimation. Model fitting and predictions are made separately for each fold, so that in the end a prediction is obtained for each sampling unit.

```
preds = computePredictedValues(m, partition = partition)
```

```
## Cross-validation, fold 1 out of 2

## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l

## Computing chain 1
## Computing chain 2
## Cross-validation, fold 2 out of 2

## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l

## Computing chain 1
## Computing chain 2
```

```
MF = evaluateModelFit(hM = m, predY = preds)
MF$R2
```

```
## [1] 0.7246587
```

As expected, the cross-validation-based predictive R^2 is lower than the explanatory R^2.

Performing the two-fold cross-validation required fitting the model twice, and more generally performing k-fold cross-validation requires fitting the model k times, which can become computationally intensive. For this reason, we have applied here two-fold cross-validation rather than e.g. leave-one-out cross-validation. Increasing the number of folds means that more data are available for fitting the model, which can be expected to lead to greater predictive performance. For this reason, the predictive power estimated by two-fold cross-validation is likely to underestimate the true predictive power of the full model fitted to all data.

Cross-validation can be performed in many ways, and how exactly it should be performed depends on which aspect of predictive power one wishes to measure. To illustrate, let us partition the plots rather than the sampling units into different folds.

```
partition = createPartition(m, nfolds = 2, column = "plot")
t(cbind(plot.id, partition)[1:15,])
```

```
##           [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## plot.id     10    2    2    1    3    6   10    7    5     3     7     8     1
## partition    2    1    1    1    2    2    2    1    1     2     1     2     1
##           [,14] [,15]
## plot.id       8     5
## partition     2     1
```

As seen by examining the correspondence between plots and the partition (shown here for the first 15 sampling units), now all sampling units belonging to a particular plot have been included in the same fold.

```
preds = computePredictedValues(m, partition=partition)
```

```
## Cross-validation, fold 1 out of 2
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l
```

```
## Computing chain 1
## Computing chain 2
## Cross-validation, fold 2 out of 2
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l
```

```
## Computing chain 1
## Computing chain 2
```

```
MF = evaluateModelFit(hM = m, predY = preds)
MF$R2
```

```
## [1] 0.07879934
```

The predictive power of the model is lower when assigning entire plots rather than individual sampling units into different cross-validation folds. This is because the prediction task is now more difficult: when making a prediction for a particular sampling unit, the model was fitted without training data from any other sampling units in the same plot. Thus the model has no possibility to estimate the actual random effect for the focal plot, and thus its predictive power is based solely on the fixed effects.

Spatial structure: including coordinates of sampling units

We next consider spatial data by assuming that each sampling unit is associated with two-dimensional spatial coordinates. We generate the data by assuming spatially structured residuals with an exponentially decreasing spatial covariance function.

```
sigma.spatial = 2
alpha.spatial = 0.5
sample.id = rep(NA,n)
for (i in 1:n){
   sample.id[i] = paste0("location_",as.character(i))
}
sample.id = as.factor(sample.id)
xycoords = matrix(runif(2*n), ncol=2)
rownames(xycoords) = sample.id
colnames(xycoords) = c("x-coordinate","y-coordinate")
a = MASS::mvrnorm(mu=rep(0,n),
            Sigma = sigma.spatial^2*exp(-as.matrix(dist(xycoords))/alpha.spatial))
y =  L + a + rnorm(n, sd = sigma)
Y=as.matrix(y)
colfunc = colorRampPalette(c("cyan", "red"))
ncols = 100
cols = colfunc(100)
par(mfrow=c(1,2))
for (i in 1:2){
   if (i==1) value = x
   if (i==2) value = y
   value = value-min(value)
   value = 1+(ncols-1)*value/max(value)
   plot(xycoords[,1],xycoords[,2],col=cols[value],pch=16,main=c("x","y")[i], asp=1)
}
```

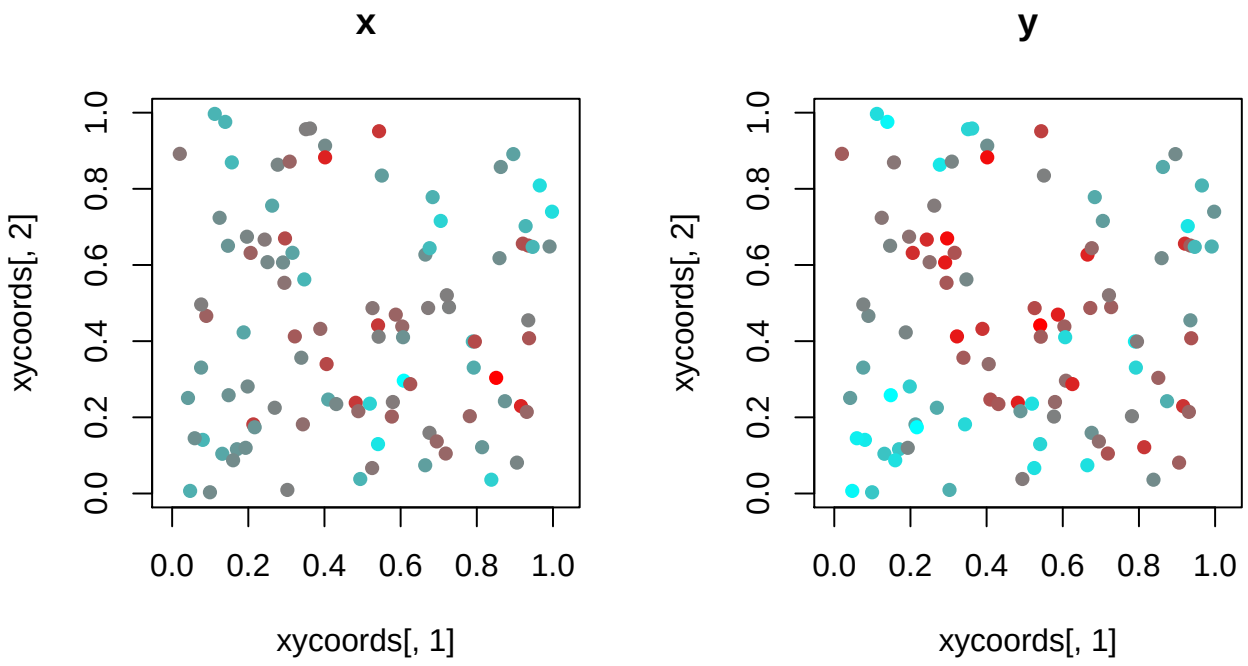

Figure 9: Plots of simulated spatially structured data.

The spatial structure is visible in the data as nearby sampling units have similar responses `y`, even if the predictor `x` is not (due to our assumptions) spatially autocorrelated.

To fit a spatially explicit model with HMSC, we construct the random effect using the 'sData' input argument.

```
studyDesign = data.frame(sample = sample.id)
rL = HmscRandomLevel(sData = xycoords)
m = Hmsc(Y=Y, XData=XData, XFormula=~x,
         studyDesign=studyDesign, ranLevels=list("sample"=rL))
```

Model fitting and evaluation of explanatory and predictive power can be done as before.

```
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                     nChains = nChains, verbose = verbose)
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent lo
```

```
## Computing chain 1
## Computing chain 2
```

```
preds = computePredictedValues(m)
MF = evaluateModelFit(hM=m, predY=preds)
MF$R2
```

```
## [1] 0.9842297
```

```
partition = createPartition(m, nfolds = 2, column = "sample")
preds = computePredictedValues(m, partition=partition)
```

```
## Cross-validation, fold 1 out of 2
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent lo
```

```
## Computing chain 1
## Computing chain 2
## Cross-validation, fold 2 out of 2
```

```
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent lo
```

```
## Computing chain 1
## Computing chain 2
```

```
MF = evaluateModelFit(hM=m, predY=preds)
MF$R2
```

```
## [1] 0.4498061
```

As the model includes a spatially structured random effect, its predictive power is based on both the fixed and the random effects. Concerning the latter, the model can utilize observed data from nearby sampling units included in model fitting when predicting the response for a focal sampling unit that is not included in model fitting.

The estimated spatial scale of the estimated random effect is given by the parameter `Alpha[[1]]`, for which we show the MCMC trace plot.

```
mpost = convertToCodaObject(m)
plot(mpost$Alpha[[1]])
```

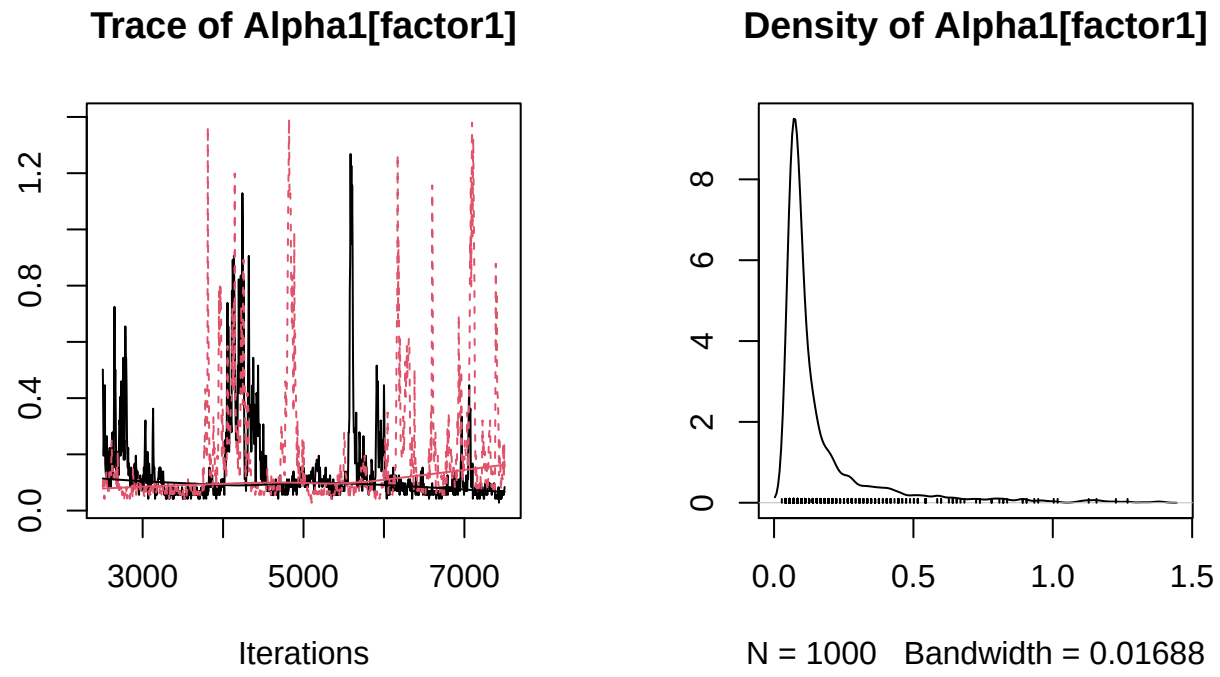

Figure 10: Posterior trace plot of the spatial scale parameter.

```
summary(mpost$Alpha[[1]])
```

```
##
## Iterations = 2505:7500
## Thinning interval = 5
## Number of chains = 2
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##        0.16581        0.19184         0.00429        0.02175
##
## 2. Quantiles for each variable:
##
##    2.5%     25%     50%     75%   97.5%
## 0.04181 0.06968 0.09756 0.16724 0.79475
```

For comparison, let us fit a non-spatial model to the same data.

```
m = Hmsc(Y=Y, XData=XData, XFormula=~x)
m = sampleMcmc(m, thin = thin, samples = samples, transient = transient,
                    nChains = nChains, verbose = verbose)
```

```
## setting updater$GammaEta=FALSE due to absence of random effects included to the model

## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l

## Computing chain 1
## Computing chain 2
```

```
preds = computePredictedValues(m)
MF = evaluateModelFit(hM=m, predY=preds)
MF$R2
```

```
## [1] 0.2674676
```

```
preds = computePredictedValues(m, partition=partition)
```

```
## Cross-validation, fold 1 out of 2

## setting updater$GammaEta=FALSE due to absence of random effects included to the model
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l

## Computing chain 1
## Computing chain 2
## Cross-validation, fold 2 out of 2

## setting updater$GammaEta=FALSE due to absence of random effects included to the model
## setting updater$latentLoadingOrderSwap=0 disabling full-conditional swapping of consecutive latent l

## Computing chain 1
## Computing chain 2
```

```
MF = evaluateModelFit(hM=m, predY=preds)
MF$R2
```

```
## [1] 0.2411381
```

We observe that both the explanatory and the predictive power is lower than for the model with a spatial random effect.