# Joint species distribution modelling with the R-package Hmsc

## Appendix S3. Comparing the performance of the Hmsc block Gibbs sampler with Hamiltonian Monte Carlo

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

*18 December 2019*

## Introduction

The Hierarchical Modelling of Species Communities (Hmsc) framework is a statistical framework for analysis of multivariate data, typically from species communities. It uses Bayesian inference to fit latent-variable joint species distribution models. The conceptual basis of the method is outlined in (Ovaskainen et al. 2017).

Here, we compare the performance of the MCMC estimation scheme implemented in the `Hmsc` package with a few alternative sampling strategies. We follow the same protocol as we used to test the performance of the `Hmsc` sampling scheme in Vignette 5, so that we assess model-fitting performance in terms of (1) computational time, (2) mixing properties, and (3) ability to recover true parameter values. We consider a subset of the simulated experiments, presented in Vignette 5, namely Cases 1,2 and 4.

## Generating simulated data

### Set directories and load libraries

```
localDir = "."
source("load_libraries.r")
```

### The `makedata` function

The `makedata` function produces datasets based on the Hmsc model. Options include

- ns = Number of sites
- ny = Number of species
- hierarchical = TRUE/FALSE, where TRUE yields data with two nested random levels, and FALSE yields a single random level.
- spatial = TRUE/FALSE, where TRUE yields spatially explicit data with sample unit locations sampled from the unit square.

```
makedata = function(ns, ny, hierarchical=FALSE, spatial=FALSE){
  rho = 0.5
  sigma = 0.3
  study.design = matrix(NA,ny,2)
  study.design[,1] = sprintf('su_%.3d',1:ny)
  study.design[,2] = sprintf('plot_%.3d',rep(1:10,ny/10))
  colnames(study.design) = c("sampling.unit","plot")
  study.design = as.data.frame(study.design)
  xy = cbind(runif(ny), runif(ny))
```

```r
X.categorical = sample(c("A","B","C"), ny, replace=TRUE)
X.covariate = rnorm(n=ny)
X.data = data.frame(X.categorical, X.covariate)
X.formula = ~ X.categorical + X.covariate
X = model.matrix(X.formula, data=X.data)
nc = dim(X)[2]

Tr.categorical = sample(c("A","B","C"), ns, replace=TRUE)
Tr.covariate = rnorm(n=ns)
Tr.data = data.frame(Tr.categorical, Tr.covariate)
Tr.formula = ~ Tr.categorical + Tr.covariate
Tr = model.matrix(Tr.formula, data=Tr.data)
nt = dim(Tr)[2]

C = matrix(0,nrow=ns, ncol=ns)
for (i in 1:ns){
  for (j in 1:ns){
    if(floor((i-1)/5)==floor((j-1)/5)){
      C[i,j] = 0.9
    }
    if(i==j){C[i,j] = 1}
  }
}

V = (sigma/2)^2*diag(nc)
gamma = matrix(rnorm(n = nc*nt, mean=0, sd=sigma), ncol=nt, nrow=nc)
mu = tcrossprod(gamma,Tr)
Si = kronecker(V,rho*C + (1-rho)*diag(ns))
beta = matrix(mvrnorm(n=1, mu=as.vector(mu), Sigma=Si), ncol=ns, nrow=nc)
LF = X%*%beta

eta = matrix(0, ncol=2, nrow=ny)
if (spatial){
  di = as.matrix(dist(xy))
  Si.alpha = exp(-di/0.5)
  eta[,1] = mvrnorm(mu=rep(0,ny), Sigma=Si.alpha)
  Si.alpha = exp(-di/0.1)
  eta[,2] = mvrnorm(mu=rep(0,ny), Sigma=Si.alpha)
} else {
  eta[,1] = rnorm(n=ny)
  if (hierarchical){
    tmp = rnorm(n = 10)
    eta[,2] = rep(tmp,ny/10)
  } else {
    eta[,2] = rnorm(n=ny)
  }
}
lambda = matrix(rnorm(n = 2*ns, mean=0, sd=sigma), ncol=ns, nrow=2)
LR = eta%*%lambda

L = LF + LR
eps = matrix(rnorm(n = ny*ns, mean=0, sd=2*sigma), ncol=ns, nrow=ny)
Y = L + eps
```

```
  all.data = list(study.design = study.design, X.data = X.data,
                  X.formula = X.formula, Tr.data = Tr.data,
                  Tr.formula = Tr.formula, C = C, Y = Y, xy = xy)
  all.parameters = list(gamma = gamma, beta = beta, rho = rho,
                        V = V, eta = eta, lambda = lambda, sigma = sigma,
                        L = L, mu = mu, LF = LF, LR = LR, eps = eps)
  return(list(all.data, all.parameters))
}
```

### Case 1: Baseline model

As the baseline case, we consider a community of $n_s = 50$ species observed on $n_y = 200$ sampling units. We include for each sampling unit two covariates, one categorical (each unit randomized to represent one of three classes with equal probability) and one continuous (sampled from standard normal distribution). We include for each species two traits, one categorical (each species randomized to represent one of three classes with equal probability) and one continuous (sampled from standard normal distribution). We assume that the species belong to groups of 5 species, and that the phylogenetic correlation is 0.9 within a group and 0 between the groups. We sampled each element of the matrix $\mathbf{\Gamma}$ (influence of traits on expected species parameters) from $N(0, \sigma^2)$, set the matrix $\mathbf{V}$ (variation among species not explained by traits) to $\frac{\sigma^2}{4}\mathbf{I}$, where we set $\sigma = 0.3$ (for motivation behind this value, see below). We assumed the value $\rho = 0.5$ for the phylogenetic signal parameter. We generated two sampling-level latent variables, both following standard normal distributions, and sampled the latent loadings on/for these variables from $N(0, \sigma^2)$. We assumed that the data were normally distributed, obtained by adding to the linear predictor noise distributed as $N(0, 4\sigma^2)$. We set $\sigma = 0.3$, resulting in the following empirical variances for vectorized quantities: variance of data 1.30, variance of fixed effects 0.74, variance of random effects 0.22, and residual variance 0.36. Out of the fixed effects, the expectation based on traits had variance 0.25. As the data are roughly equally influenced by all model components, we expected that true values of all model parameters can be recovered if a sufficient amount of data are available.

```
set.seed(1)
source("makedata.R")

ns = 50
ny = 200
tmp = makedata(ns=ns, ny=ny)
all.data=tmp[[1]]
all.parameters = tmp[[2]]
L1 = all.parameters$L
Y1 = all.data$Y

#First five species at first five sites
all.data$Y[1:5,1:5]
```

```
##             [,1]      [,2]       [,3]       [,4]        [,5]
## 1 -0.19800205 1.0126434 -1.0216938  1.1751838 -0.36021756
## 2 -0.05294868 1.3931726 -0.3298163  1.4037167 -0.07184399
## 3  1.56562943 0.6619827  1.6723218 -0.8680332 -1.75510750
## 4  3.14674580 2.6849328  0.3252162  2.2381762  0.64636434
## 5  1.43213030 0.3684729  0.1694582  1.1808551 -0.73522366
```

```
#Random levels for first five sites
all.data$study.design[1:5,]
```

```
##   sampling.unit     plot
## 1        su_001 plot_001
## 2        su_002 plot_002
## 3        su_003 plot_003
## 4        su_004 plot_004
## 5        su_005 plot_005
```

```
#Covariates
all.data$X.data[1:5,]
```

```
##   X.categorical X.covariate
## 1             B   0.8936737
## 2             A  -1.0472981
## 3             C   1.9713374
## 4             C  -0.3836321
## 5             C   1.6541453
```

```
#Trait data
all.data$Tr.data[1:5,]
```

```
##   Tr.categorical Tr.covariate
## 1              B   -1.6845206
## 2              C   -0.1442266
## 3              B    1.1802137
## 4              C    0.6813999
## 5              A    0.1432476
```

```
c(var(as.vector(all.parameters$mu)), var(as.vector(all.parameters$LF)),
  var(as.vector(all.parameters$LR)), var(as.vector(all.parameters$eps)),
  var(as.vector(Y1)))
```

```
## [1] 0.2473474 0.7387942 0.2219114 0.3628864 1.3037621
```

```
dir.create(file.path(localDir, "data"), showWarnings=FALSE, recursive=TRUE)
save(file=file.path(localDir, "data","Case1.RData"), all.data, all.parameters)
```

### Case 2: Presence-absence model

We assumed the same linear predictor as for the baseline case but truncated the normally distributed data to 0/1 according to the probit model.

```
Y2 = 1*(L1 + matrix(rnorm(n = ny*ns), ncol=ns, nrow=ny)>0)
all.data$Y = Y2

#First five species at first five sites
all.data$Y[1:5,1:5]
```

```
##   [,1] [,2] [,3] [,4] [,5]
## 1    1    1    1    1    0
## 2    0    1    1    1    0
## 3    1    0    1    1    0
## 4    0    1    0    1    1
## 5    1    1    1    1    1
```

```
save(file=file.path(localDir, "data","Case2.RData"), all.data, all.parameters)
```

### Case 4: Large number of sites

We assumed otherwise the same model as for Case 1 but set the number of sampling units to $n_y = 2000$.

```r
tmp=makedata(ns=50, ny=2000)
all.data=tmp[[1]]
all.parameters=tmp[[2]]
save(file=file.path(localDir, "data","Case4.RData"), all.data, all.parameters)
```

## Model setup and fitting

We fitted `Hmsc` models with structure matching the data generation model, i.e. we did not examine here the influence of model misspecification or e.g. issues related to variable selection. However, we did not fix the number of latent variables to that used for generating the data, but assumed the default prior for them, as well as for all other model parameters.

We obtained 8 independent MCMC chains for each model with sampling parameters set to thin=10 and samples=1000, thus running 15,000 MCMC steps per chain. We adapted the number of latent factors during the first 4,000 iterations, and ignored from each chain the first 5,000 iterations as a transient. The resulting thinned posterior sample consisted of 8000 samples.

The code below takes a long time to run. The .Rmd file can be run much more quickly by setting `test.run=TRUE`, but the results will not be reliable.

For case 1 and 4, we set up the model with Gaussian errors, using `distr = "normal"`

```r
nChains = 8
test.run = FALSE
recomputePostFlag = FALSE
if (test.run){
   #with this option, the vignette evaluates in ca. ? minutes in a laptop
   thin = 1
   samples = 10
} else {
   #with this option, the vignette evaluates in ca. ? hrs in a laptop
   thin = 10
   samples = 1000
}
verbose = 0

if(recomputePostFlag){
  for (case in c(1,4)){
    set.seed(1)
    load(file = file.path(localDir, "data", paste("Case",toString(case),".RData", sep="")))
    m = Hmsc(Y = all.data$Y,
             XData = all.data$X.data, XFormula = all.data$X.formula,
             TrData = all.data$Tr.data, TrFormula = all.data$Tr.formula,
             C = all.data$C,
             distr = "normal", studyDesign = all.data$study.design,
             ranLevels = list(
               "sampling.unit" = HmscRandomLevel(units = all.data$study.design[,1])))
    ptm = proc.time()
    m = sampleMcmc(m, samples=samples, thin=thin, transient = ceiling(0.5*samples*thin),
                   nChains = nChains, nParallel = nChains, verbose=verbose)
    computational.time =  proc.time() - ptm
    dir.create(file.path(localDir, "models"), showWarnings=FALSE, recursive=TRUE)
    save(file=file.path(localDir, "models", paste("Case",toString(case),".RData", sep="")),
         m, computational.time)
  }
}
```

For case 2, we set up the model with probit errors, using `distr = "probit"`

```
if(recomputePostFlag){
  set.seed(1)
  load(file=file.path(localDir, "data","Case2.RData"))
  m = Hmsc(Y=all.data$Y,
           XData=all.data$X.data, XFormula=all.data$X.formula,
           TrData=all.data$Tr.data, TrFormula=all.data$Tr.formula,
           C=all.data$C,
           distr="probit", studyDesign=all.data$study.design,
           ranLevels=list(
             "sampling.unit"=HmscRandomLevel(units = all.data$study.design[,1])))
  ptm = proc.time()
  m = sampleMcmc(m, samples=samples, thin=thin, transient = ceiling(0.5*samples*thin),
                 nChains = nChains, nParallel = nChains, verbose=verbose)
  computational.time =  proc.time() - ptm
  save(file=file.path(localDir, "models","Case2.RData"), m, computational.time)
}
```

## Alternative MCMC techniques

In this section we consider some alternative posterior sampling strategies based on available software for generic Bayesian model fitting. Typically, the performance of a particular posterior sampling strategy depends on the interplay of three aspects: the selected parameterisation of the given studied statistical problem, the shape in parameter space of the posterior implied by the observed data, and the choice of sampling method. In our comparison, we focus on Hamiltonian Monte Carlo-type MCMC methods, which exploit the gradient information of the target posterior density (Neal 2012). Such methods have been shown to be more efficient for sampling from high-dimensional complex-shaped posteriors and recently became readily available through statistical software. Namely, we consider two alternative implementation options: the first one exploits a Hmsc parameterisation that closely follows the specification from the main text and performs sampling using the Stan software (Carpenter et al. 2017) and its R interface (Stan Development Team 2019), and the second one considers a marginalised Hmsc representation with sampling being conducted using the TensorFlow Probability library in Python (we originally attempted a similar Stan-based implementation, but faced major challenges caused by its autodifferentiation specifics). We use the cases 1,2 and 4 above to highlight the distinctintive properties of different implementations in different scenarios.

We stress that the current comparison does not cover all available opportunities for posterior sampling, which is currently a hot research topic with novel techniques rapidly evolving. Neither does this comparison cover all potential configurations of the Hmsc model structure, possible parameterisations and observed data, and therefore extrapolation of the presented results should be done with reasonable caution.

We consider the following alternative implementations:

A. Basic Stan-based implementation. The Stan code implementing the model closely follows the Hmsc mathematical formulation, with the exception that the $\mathbf{\Psi}$ parameters were analytically marginalized out, leading to an alternative formulation of the multiplicative Gammma Shrinkage Process Prior: $\Lambda_{hj} \sim$ Student-t$(\nu, 0, \tau_h^{-1})$. Unlike the conditional block Gibbs sampling scheme, the gradient-based samplers do not require conditional conjugacy of the distributions, and may benefit from the reduced number of model parameters. Stan model files for Gaussian-noise observation model and binary observation model are named **stan__normal__basic.stan** and **stan__probit__basic.stan**, correspondingly. The corresponding experiments are marked as cases 1-Stan, 2-Stan and 4-Stan.

B. The second type of models proceeds with the idea of marginalization and addresses the Hmsc model as a Gaussian Process with a particularly structured covariance function. The hyperparameters of such a Gaussian process are $\mathbf{V}$, $\rho$, $\delta$, $\mathbf{\Lambda}$ and $\mathbf{\Sigma}$ in Hmsc notation, the rest of the Hmsc parameters can be probabilistically expressed conditional on the hyperparameters and observed data. The corresponding experiments are marked as cased 1-TFP and 4-TFP.

The priors of the alternative models matched the `Hmsc` priors with a single exception - instead of a spike-and-slab prior for the $\rho$ parameter, the alternative models used a flat prior over $[0, 1]$

## Fitting alternative models

The Stan models were fitted using 4 chains for 1000 samples with 1000 steps discarded as burn-in using the No-U-Turn-Sampler with default sampler parameters, which is the default Stan MCMC tool. Because implementation of adaptive assessment of number of latent factors with a gradient-based sampler is a rather nontrivial task, we fixed the number of factors $n_f$ in all experiments to 5, which is the upper number of factors estimated by the `Hmsc` model fitting algorithm.

Please set `recomputeStanFlag = TRUE` in order to refit the Stan models (it takes a couple of hours to run everything). If you have already fitted the models, you may use `recomputeStanFlag = FALSE`.

```r
library(rstan)
library(plyr)
recomputeStanFlag = FALSE

if(recomputeStanFlag){
  # we need one internal function from HMSC package to substitute HMSC posterior.
  # source('../HMSC/R/combineParameters.R')
  source_url('https://github.com/hmsc-r/HMSC/blob/master/R/combineParameters.R?raw=TRUE')
  options(mc.cores = parallel::detectCores())
  chainN = nChains
  iterN = 1000
  nf = 5
  thinN = 1
  modelNameVec = c("stan_normal_basic.stan","stan_probit_basic.stan","stan_normal_basic.stan")
  modelCaseVec = c(1,2,4)
  modelTypeVec = c("normal","probit","normal")
  for(mInd in 1:length(modelCaseVec)){
    modelCase = modelCaseVec[mInd]
    modelType = mapvalues(modelCase, from=modelCaseVec, to=modelTypeVec)
    print(modelCase)
    set.seed(1)
    load(file=file.path(localDir, "models", paste("Case",toString(modelCase),".RData", sep="")))

    V0 = m$V0
    f0 = m$f0
    U0 = m$UGamma
    aS = m$aSigma
    bS = m$bSigma
    nu = m$ranLevels[[1]]$nu
    a1 = m$ranLevels[[1]]$a1
    a2 = m$ranLevels[[1]]$a2
    b1 = m$ranLevels[[1]]$b1
    b2 = m$ranLevels[[1]]$b2
    dataList = list(ny=m$ny, ns=m$ns, nc=m$nc, nt=m$nt, nf=nf, Y=m$Y, X=m$XScaled, T=m$TrScaled,
                    C=m$C, V0=V0, f0=f0, U0=U0, aS=aS, bS=bS, nu=nu, a1=a1, a2=a2, b1=b1, b2=b2)
    modelName = mapvalues(modelCase, from=modelCaseVec, to=modelNameVec)
    stanmod <- stan_model(modelName)
    ptm = proc.time()
    fit <- stan(modelName, data=dataList, iter=2*thinN*iterN, chains=chainN, thin=thinN,
                refresh=100, seed=1, init_r=0.5)
    computational.time =  proc.time() - ptm

    parList = extract(fit)
    for(chain in 1:chainN){
      for(iter in 1:iterN){
        ind = iter + (chain-1)*iterN
        Beta = parList$Beta[ind,,]
        Gamma = parList$Gamma[ind,,]
        iV = solve(parList$V[ind,,])
        rho = 1
        if(modelCase=="normal"){
          iSigma = parList$sigma[ind,]^-1
        } else
          iSigma = rep(1,m$ns)
        Gamma = parList$Gamma[ind,,]
        if(!is.null(parList$Eta)){
          Eta = list(parList$Eta[ind,,])
        } else
          Eta = list(matrix(0,m$ny,nf))
        Lambda = list(parList$Lambda[ind,,])
        if(!is.null(parList$Psi)){
          Psi = list(parList$Psi[ind,,])
        } else
          Psi = list(0*parList$Lambda[ind,,])
        Delta = list(matrix(parList$delta[ind,]))
        Alpha = list(rep(1,nf))
        m$postList[[chain]][[iter]] = combineParameters(Beta=Beta,BetaSel=NULL,wRRR=NULL,
                      Gamma=Gamma,iV=iV,rho=rho,iSigma=iSigma,
                      Eta=Eta,Lambda=Lambda,Alpha=Alpha,Psi=Psi,Delta=Delta,
                      PsiRRR=NULL,DeltaRRR=NULL,
                      ncNRRR=m$ncNRRR, ncRRR=m$ncRRR, ncsel=m$ncsel,
                      XSelect=m$XSelect,
                      XScalePar=m$XScalePar, XInterceptInd=m$XInterceptInd,
                      XRRRScalePar=m$XRRRScalePar,
                      nt=m$nt, TrScalePar=m$TrScalePar,
                      TrInterceptInd=m$TrInterceptInd, rhopw=NULL)
        m$postList[[chain]][[iter]]$rho = parList$rho[ind]
      }
      m$postList[[chain]] = m$postList[[chain]][1:iterN]
    }
    m$samples = iterN
    m$transient = iterN*thinN
    m$thin = thinN

    save(file=file.path(localDir, "models", paste("Case",toString(modelCase+100),
                        ".RData", sep="")), m, computational.time)
  }
}
```

As described above, the cases 1-TFP and 4-TFP were fitted using Python and the TensorFlow-Probability package. Similarly to Stan models, we exploited the implementation of a No-U-Turn-Sampler provided by the TensorFlow-Probability package, which operated on the manually-derived marginal log-posterior-density. By neatly exploiting the structure of the Hmsc-induced covariance matrix, it is possible to bring the asymptotic scaling of the marginal likelihood computational cost from typical cubic scaling of Gaussian processes $n_y^3 n_s^3$ to become linear with respect to $n_y n_s$.

Stan and Python code implementing these cases are given at the end of this document.

## Evaluating model performance

### MCMC mixing

We used the `coda` package to compute the effective number of MCMC samples (sample size adjusted for autocorrelation) and Potential Scale Reduction Factor (Gelman and Rubin's convergence diagnostic, PSRF) for the following parameters: $\mathbf{B}$, $\mathbf{\Gamma}$, $\rho$, $\mathbf{V}$, $\mathbf{\Omega}$. Note the following useful functions

- `computePredictedValues` returns model-predicted values for the original sampling units used in model fitting.
- `evaluateModelFit` returns several measures of model fit, including RMSE (root mean square error), $R^2$ (coefficient of determination), Tjur $R^2$ (coefficient of discrimination), and AUC (area under the receiver operating characteristic curve).
- `convertToCodaObject` converts the posterior distribution produced by `Hmsc` into `coda`-format.

```r
recomputePerformanceFlag = FALSE
cases = c(1,2,4,101,102,104,201,204)

if(recomputePerformanceFlag){
  ptm = proc.time()
  for (case in cases){
    print(case)
    set.seed(1)
    load(file=file.path(localDir, "data", paste("Case",toString(case%%10),".RData", sep="")))
    load(file=file.path(localDir, "models", paste("Case",toString(case),".RData", sep="")))
    computational.time = computational.time[3]

    # predY = computePredictedValues(m, expected=FALSE)
    # MF = evaluateModelFit(m, predY)
    MF = NA

    # ASSESS MIXING
    mpost = convertToCodaObject(m)

    es.beta = effectiveSize(mpost$Beta)
    ge.beta = gelman.diag(mpost$Beta,multivariate=FALSE)$psrf

    es.gamma = effectiveSize(mpost$Gamma)
    ge.gamma = gelman.diag(mpost$Gamma,multivariate=FALSE)$psrf

    es.rho = effectiveSize(mpost$Rho)
    ge.rho = gelman.diag(mpost$Rho,multivariate=FALSE)$psrf

    es.V = effectiveSize(mpost$V)
    ge.V = gelman.diag(mpost$V,multivariate=FALSE)$psrf
```

```r
mpost$temp = mpost$Omega[[1]]
for(i in 1:length(mpost$temp)){
  mpost$temp[[i]] = mpost$temp[[i]][,1:50^2]
}

es.omega = effectiveSize(mpost$temp)
ge.omega = gelman.diag(mpost$temp,multivariate=FALSE)$psrf

es.alpha = NA
ge.alpha = NA

if (case==6){
  es.alpha = effectiveSize(mpost$Alpha[[1]])
  ge.alpha = gelman.diag(mpost$Alpha[[1]])
}

mixing = list(es.beta=es.beta, ge.beta=ge.beta,
              es.gamma=es.gamma, ge.gamma=ge.gamma,
              es.rho=es.rho, ge.rho=ge.rho,
              es.V=es.V, ge.V=ge.V,
              es.omega=es.omega, ge.omega=ge.omega)

# COMPUTE NRMSE AND COR
estimates = list()
postList=poolMcmcChains(m$postList, start = 1)
npost = length(postList)
npri = npost

for (para in 1:5){
  parameter = c("beta","gamma","rho","V","omega")[para]

  if (parameter=="beta"){
    getpar = function(a)
      return(a$Beta)
    true = all.parameters$beta
    prior = list()
    for(i in 1:npost){
      gamma = matrix(mvrnorm(n=1, mu=m$mGamma, Sigma = m$UGamma), ncol=m$nt, nrow=m$nc)
      mu = tcrossprod(gamma,m$Tr)
      rho = sample(size=1,x=m$rhopw[,1],prob=m$rhopw[,2])
      V = solve(rwish(m$f0, solve(m$V0)))
      Si = kronecker(V,rho*all.data$C + (1-rho)*diag(m$ns))
      prior[[i]] = matrix(mvrnorm(n=1, mu=as.vector(mu), Sigma=Si), ncol=m$ns, nrow=m$nc)
    }
  }

  if (parameter=="rho"){
    getpar = function(a)
      return(a$rho)
    true = all.parameters$rho
    prior = list()
    for(i in 1:npri){
      prior[[i]] = sample(size=1,x=m$rhopw[,1],prob=m$rhopw[,2])
    }
  }

  if (parameter=="gamma"){
    getpar = function(a)
      return(a$Gamma)
    true = all.parameters$gamma
    prior = list()
    for(i in 1:npri){
      prior[[i]] = matrix(mvrnorm(n=1, mu=m$mGamma, Sigma = m$UGamma), ncol=m$nt, nrow=m$nc)
    }
  }

  if (parameter=="V"){
    getpar = function(a)
      return(a$V)
    true = all.parameters$V
    prior = list()
    for(i in 1:npost){
      prior[[i]] = solve(rwish(m$f0, solve(m$V0)))
    }
  }

  if (parameter=="omega"){
    getpar = function(a)
      return(t(a$Lambda[[1]])%*%a$Lambda[[1]])
    true = t(all.parameters$lambda)%*%all.parameters$lambda
    nu = m$rL[[1]]$nu
    a1 = m$rL[[1]]$a1
    a2 = m$rL[[1]]$a2
    b1 = m$rL[[1]]$b1
    b2 = m$rL[[1]]$b2
    prior = list()
    for(i in 1:npost){
      nf = 10
      delta = rep(NA,nf)
      delta[1] = rgamma(1,a1,b1)
      for (j in 2:nf){
        delta[j] = rgamma(1,a2,b2)
      }
      tau = cumprod(delta)
      psi = matrix(rgamma(nf*m$ns, nu/2, nu/2), nf, m$ns)
      sd = sqrt(1/(psi*matrix(rep(tau,m$ns),nr=nf, nc=m$ns)))
      la = rnorm(length(sd), mean=0, sd=sd)
      dim(la) = dim(sd)
      prior[[i]] = t(la)%*%la
    }
  }

  post = lapply(postList, getpar)
  post.mean = Reduce("+", post) / length(post)
  cors = rep(0,npost)
  for (i in 1:npost){
```

```r
        cors[i] = cor(as.vector(true),as.vector(post[[i]]))
      }
      se = function(a)
        return((a-true)^2)
      post.se = lapply(post, se)
      prior.se = lapply(prior, se)
      post.rmse = sqrt(Reduce("+", post.se) / length(post.se))
      prior.rmse = sqrt(Reduce("+", prior.se) / length(prior.se))
      nrmse = post.rmse / prior.rmse

      if (parameter=="beta"){
        estimates$cor.beta = cors
        estimates$nrmse.beta = nrmse
      }
      if (parameter=="gamma"){
        estimates$cor.gamma = cors
        estimates$nrmse.gamma = nrmse
      }
      if (parameter=="rho"){
        estimates$post.rho = post
        estimates$nrmse.rho = nrmse
      }
      if (parameter=="V"){
        estimates$cor.V = cors
        estimates$nrmse.V = nrmse
      }
      if (parameter=="omega"){
        estimates$cor.omega = cors
        estimates$nrmse.omega = nrmse
      }
    }

    dir.create(file.path(localDir, "performance"), showWarnings=FALSE, recursive=TRUE)
    save(file=file.path(localDir, "performance",paste("Case",toString(case),".RData",sep="")),
         computational.time, mixing, estimates, MF)
  }
}
```

The effective sample size (Figure 1) was close to 8000 for the $\mathbf{B}$, $\mathbf{\Gamma}$, and $\mathbf{V}$ parameters, except for `Hmsc` model fit for Case 2 (probit model), for which a longer MCMC chain may be needed for obtaining robust results. Mixing was in general more challenging to obtain for the $\mathbf{\Omega}$ and $\rho$ parameters, for which the effective sample size varied much among the elements of the matrix also for the Gaussian models.

We can also assess sampling efficiency in units of effective samples per minute (Figure 2). The most striking pattern is that the Stan implementations were most efficient for cases 1 and 2, but less efficient for case 4. The `Hmsc` and TensorFlow implementations performed roughly at the same level.

The potential scale reduction factors (Gelman-Rubin diagnostics; Figure 3) indicated certain complications with inter-chain convergence for the $\mathbf{\Omega}$ parameters. This pattern was apparently independent of the posterior sampling method, and varied considerably among the different cases.

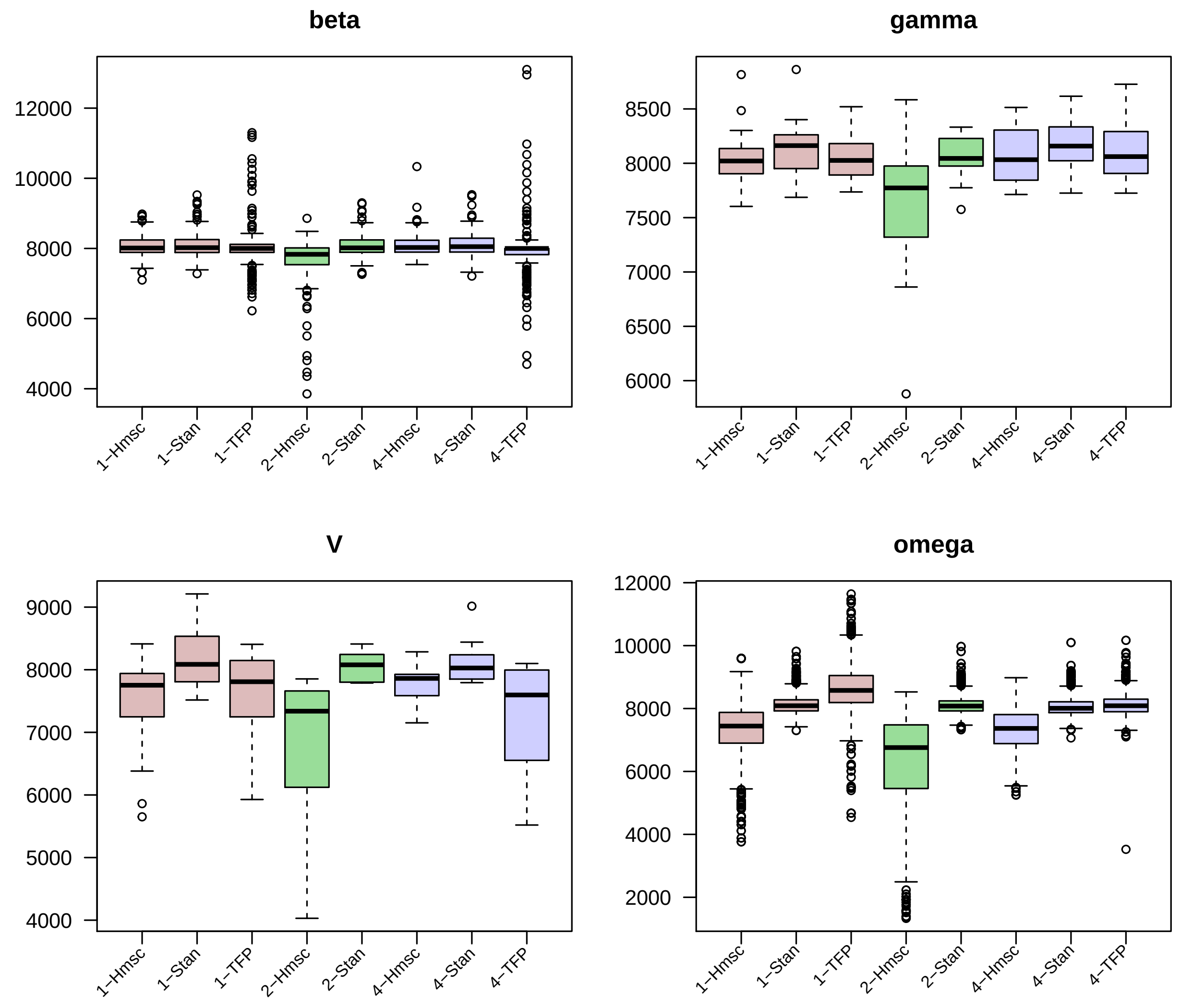

Figure 1: Distributions of effective sample sizes of multivariate parameters.

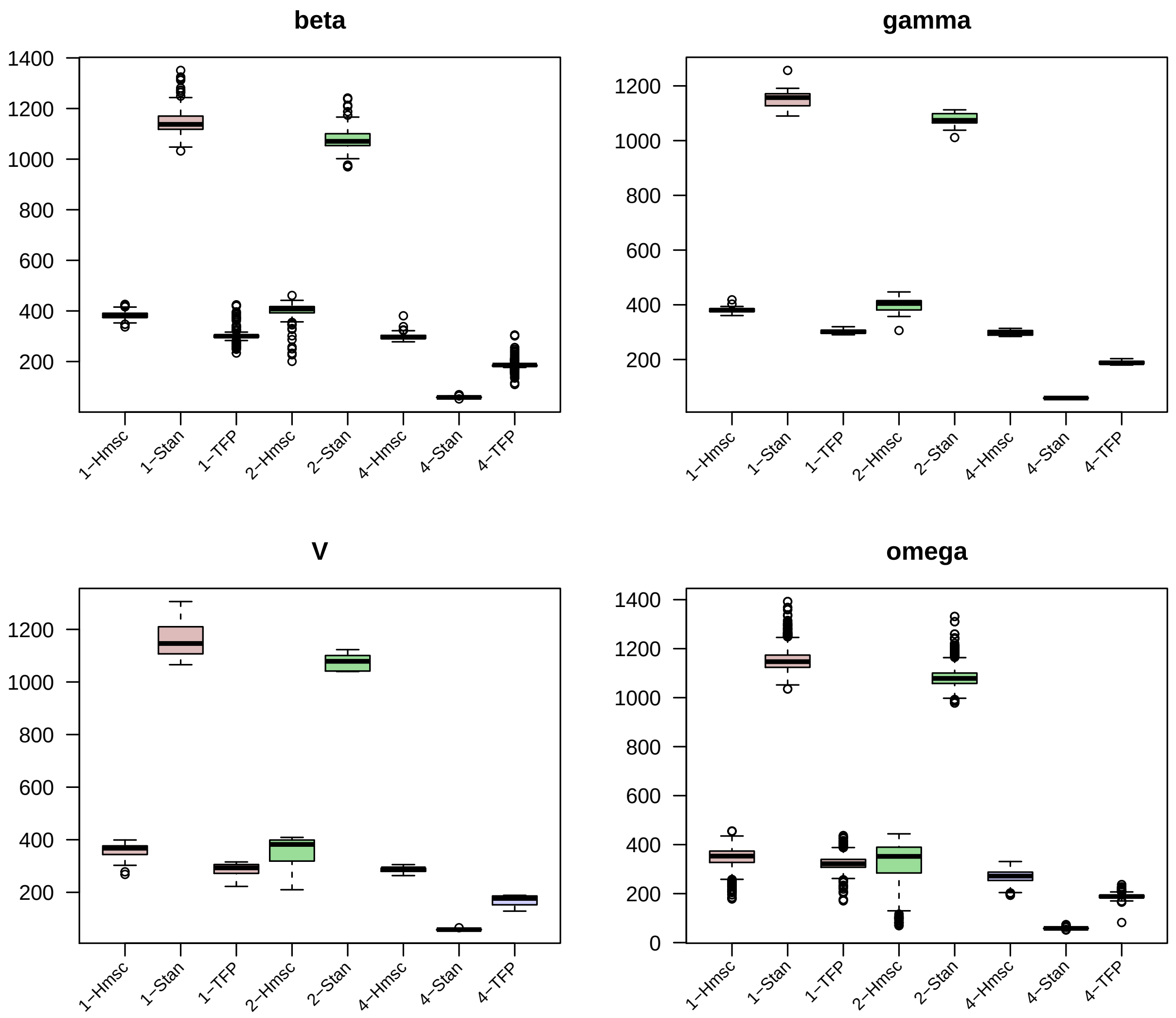

Figure 2: Distributions of effective sample sizes per minute of multivariate parameters.

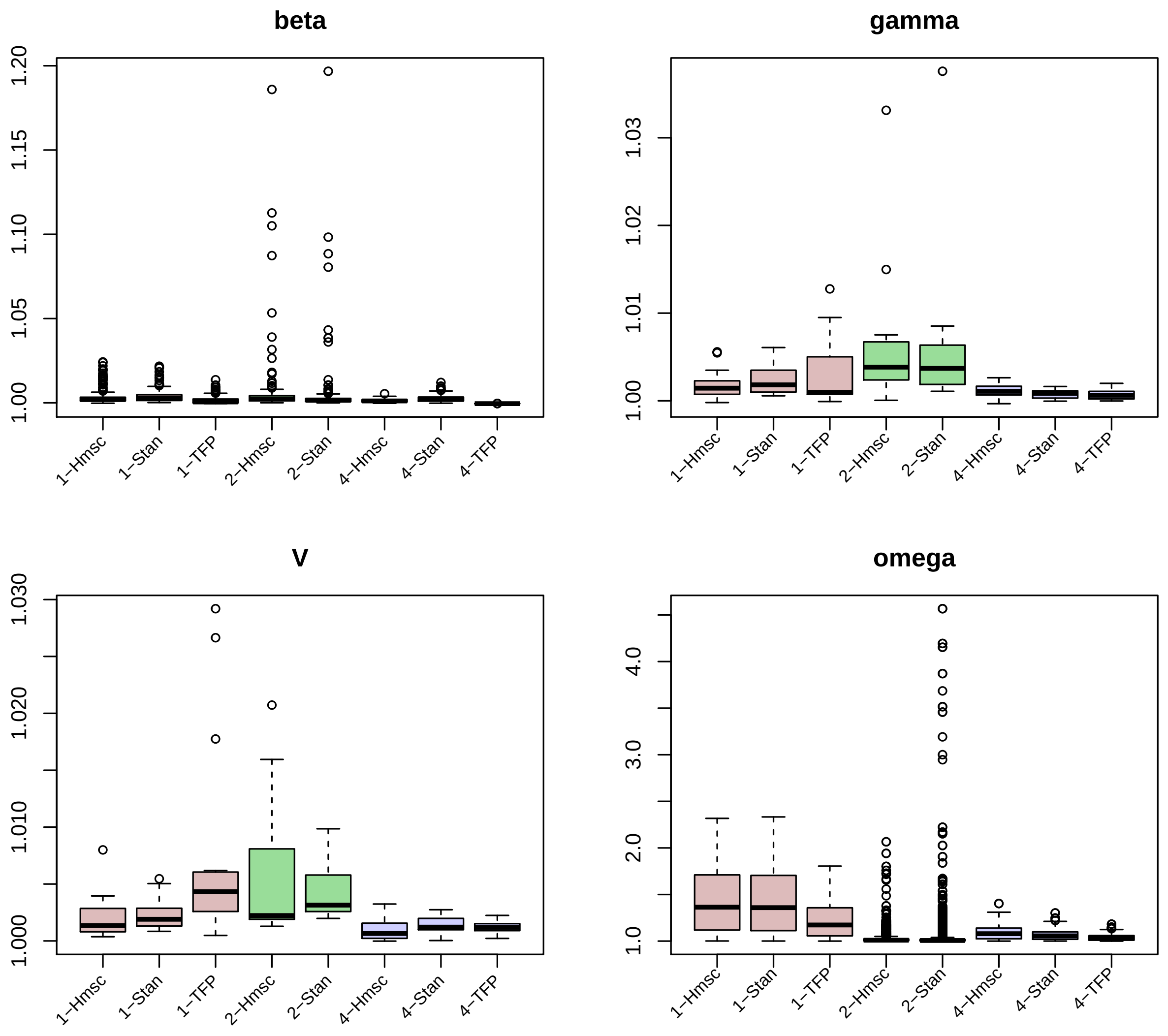

Figure 3: Distributions of upper C.I. for potential scale reduction factors for multivariate parameters.

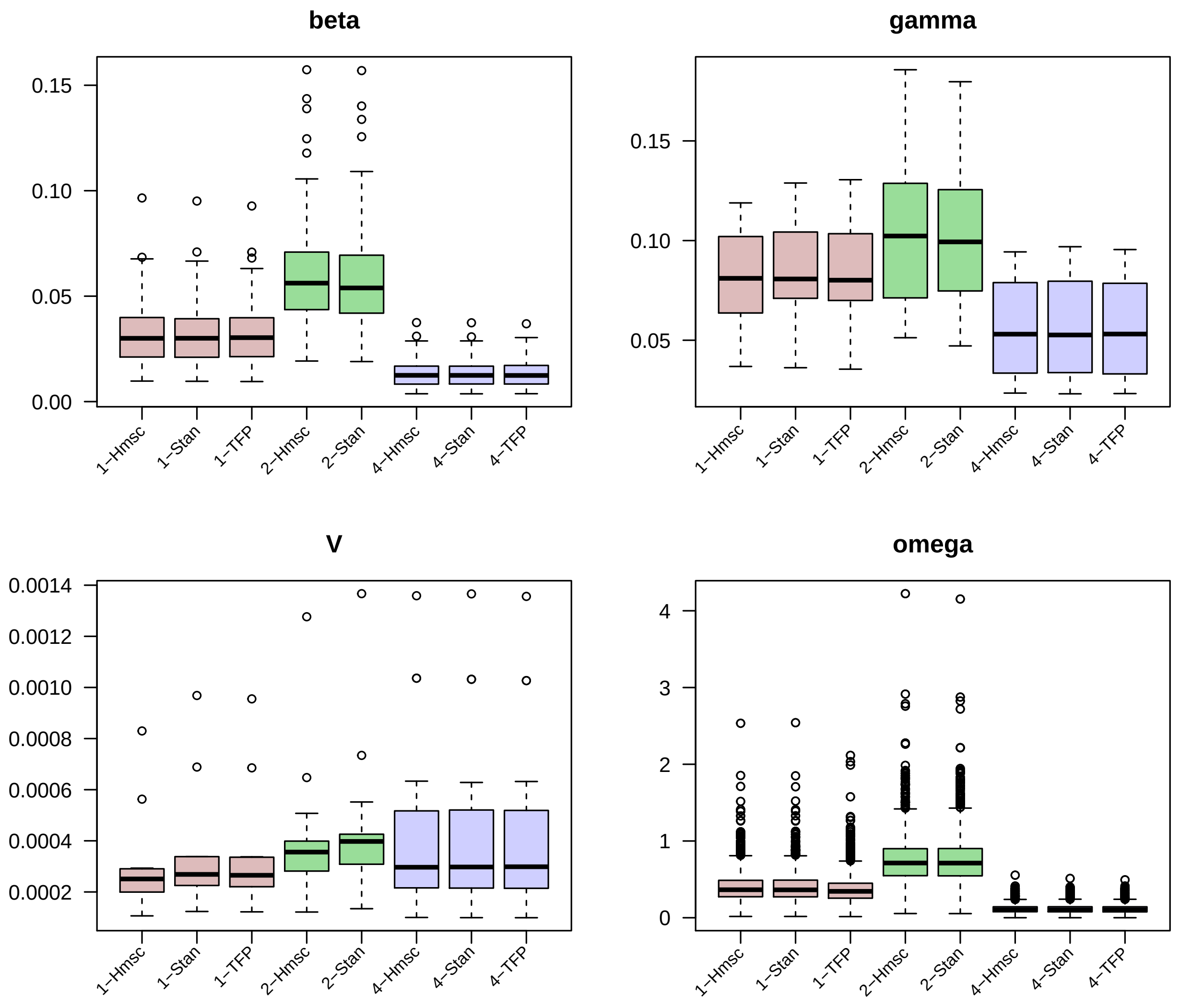

Figure 4: Distributions of normalized root mean squared errors (NRMSE) for multivariate parameters. Values smaller than one indicate that the posterior distribution is closer to the true value than the prior distribution.

### Ability to recover true parameter values

For the same parameters considered for MCMC mixing, we asked how much closer the posterior distributions were to the true values than were the prior distributions. We quantified this by the normalized root mean squared errors (NRMSE; Figure 4), where the normalization was done with respect to the root mean squared error under the prior distribution. Further, for the matrix-valued parameters $\mathbf{B}$, $\mathbf{\Gamma}$, $\rho$, $\mathbf{V}$, $\mathbf{\Omega}$ we computed the posterior distribution of the Mantel correlation between the estimated and true values (Figure 5). For the univariate parameter $\rho$ we compared the true value to the boxplot of the posterior distribution (Figure 6).

These diagnostics indicated only minor differences in the obtained model fits with different MCMC strategies, further validating the robustness of the produced results.

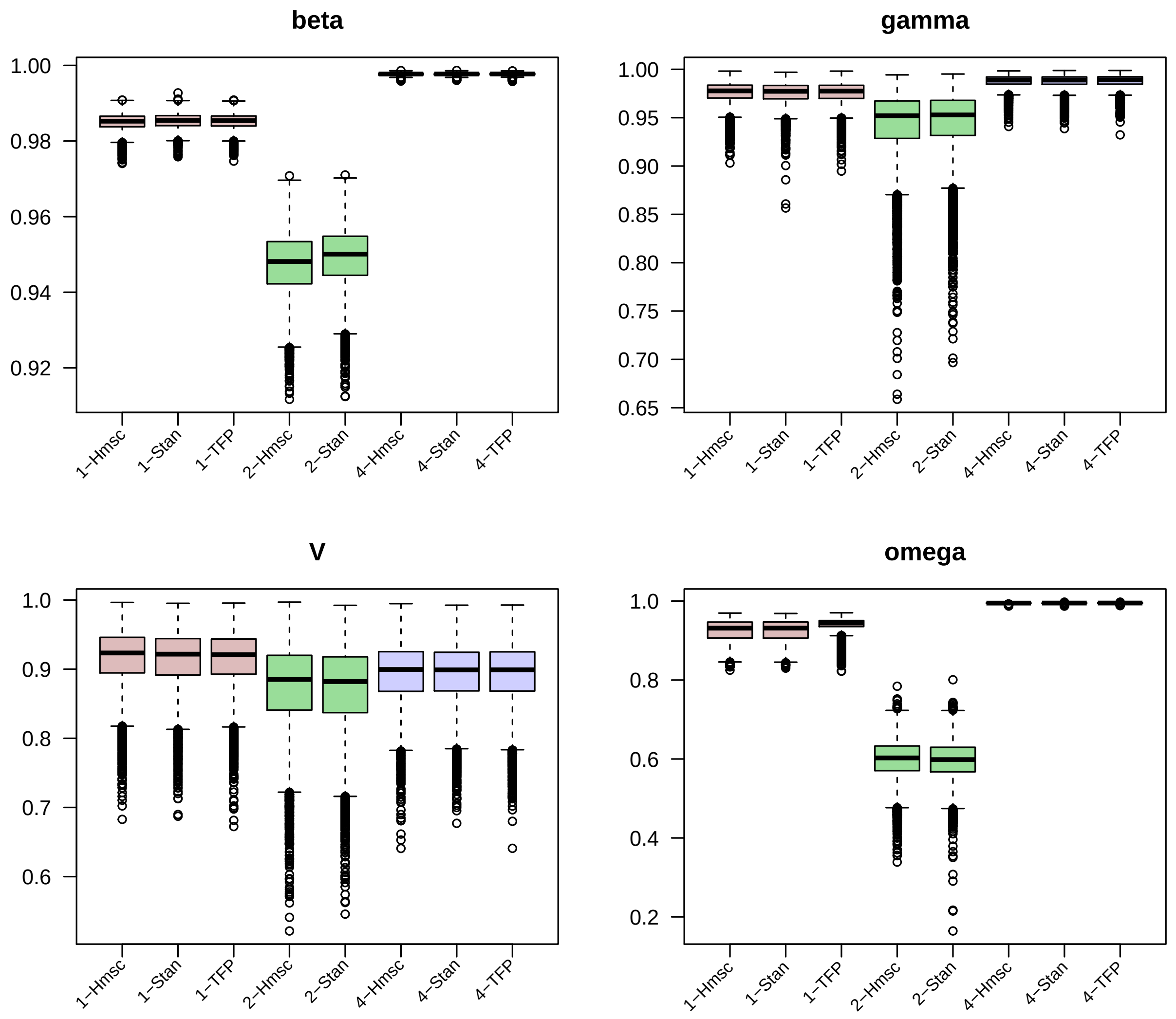

Figure 5: Posterior distributions of correlation between true and estimated values for multivariate parameters.

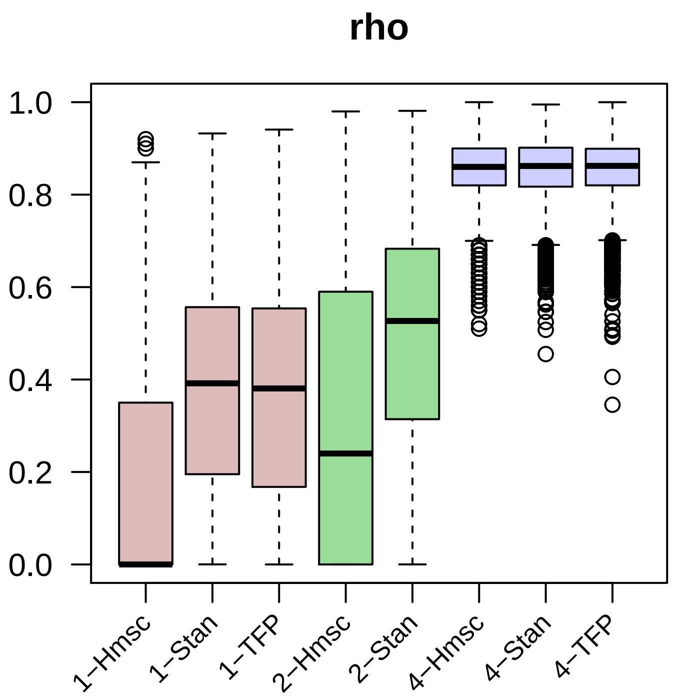

Figure 6: Posterior distributions of the phylogenetic signal parameter.

Table 1: Performance of Hmsc and alternative sampling strategies for estimating the phylogenetic signal parameter.

| | 1-Hmsc | 1-Stan | 1-TFP | 2-Hmsc | 2-Stan | 4-Hmsc | 4-Stan | 4-TFP |
|---|---|---|---|---|---|---|---|---|
| Effective sample size | 4368.00 | 8024.00 | 4732.00 | 2161.00 | 7912.00 | 7282.00 | 8336.00 | 7127.00 |
| Upper C.I. of PSRF | 1.00 | 1.01 | 1.16 | 1.07 | 1.05 | 1.00 | 1.00 | 1.00 |
| NRMSE | 0.99 | 0.61 | 0.65 | 0.89 | 0.58 | 0.88 | 0.88 | 0.88 |

**Computational time**

All examples were run on the same 16-core machine operating under a MacOS system, which had 64 GB of RAM memory. With `Hmsc` we ran the 8 chains in parallel using the option `nParallel = nChains`, with Stan we ran 8 chains using the recommended `options(mc.cores = parallel::detectCores())` and the Python code was simultaneously run for each chain separately using an external script. We recorded the elapsed running time as the maximum time elapsed executing individual chains.

Unlike the model-fit convergence properties evaluated above, the different fitted models differed dramatically in the time elapsed for model fitting (Figure 6). The most rapid model fitting was achieved by the advanced Stan models with $n_y = 200$ data. However, the same Stan model for normal residuals ran several times slower than `Hmsc` on the $n_y = 2000$ data for case 4. Somewhat unexpectedly, the fully-marginal TensorFlow Probability model was considerably slower than the Stan version and even slower than `Hmsc` for case 1, but substantially outperformed the Stan models for case 4. This leads us to speculate that such models are better capable of scaling to larger datasets, but that this comes at the cost of certain overheating. Further, more extensive comparisons will be needed before making definitive statements.

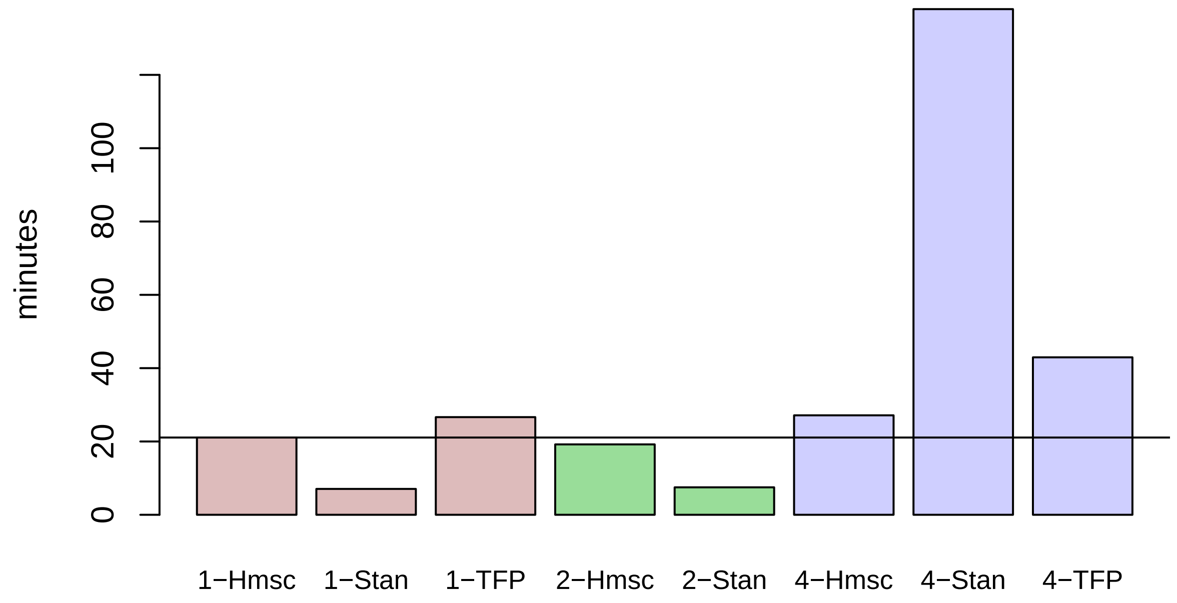

Figure 7: Computational times to perform the MCMC sampling.

In summary, the present experiment shows that none of the posterior sampling strategies clearly outperformed the others for all three cases. This suggests that different approaches would be more efficient for different model structures. However, given that many practitioners in ecology are not fully familiar with the whole spectrum of Bayesian model-fitting techniques, we consider it valuable to provide within `Hmsc` some robust tools for automation of these underlying computations, even if such a tool would be a suboptimal choice under certain scenarios.

## Stan code for cases 1-Stan, 2-Stan, 4-Stan

### Basic Gaussian model

```
data{
  int<lower=1> ny;
  int<lower=1> ns;
  int<lower=1> nc;
  int<lower=1> nt;
  int<lower=1> nf;
  matrix[ny,ns] Y;
  matrix[ny,nc] X;
  matrix[ns,nt] T;
  corr_matrix[ns] C;

  real<lower=nc> f0;
  cov_matrix[nc] V0;
  cov_matrix[nc*nt] U0;
  vector<lower=0>[ns] aS;
  vector<lower=0>[ns] bS;
  real<lower=0> nu;
  real<lower=0> a1;
  real<lower=0> a2;
  real<lower=0> b1;
  real<lower=0> b2;
}

transformed data{
  matrix[nc*nt,nc*nt] LU0 = cholesky_decompose(U0);
}

parameters{
  matrix[nc,ns] Beta;
  matrix[nc,nt] Gamma;
  cov_matrix[nc] V;
  vector<lower=0>[ns] sigma;
  matrix[ny,nf] Eta;
  matrix[nf,ns] Lambda0;
  vector<lower=0>[nf] delta;
  real<lower=0,upper=1> rho;
}

transformed parameters{
  vector[nf] tau = exp(cumulative_sum(log(delta)));
  matrix[nf,ns] Lambda = Lambda0 ./ rep_matrix(sqrt(tau),ns);
}

model{
  matrix[ny,ns] L = X*Beta + Eta*Lambda;
  matrix[ns,ns] Q = rho*C + diag_matrix(rep_vector(1-rho,ns));
  matrix[nc,ns] Mu = Gamma * T';

  # observation model
  for(j in 1:ns){
    Y[,j] ~ normal(L[,j], sqrt(sigma[j]));
  }

  # priors
  # to_vector(Beta) ~ multi_normal(to_vector(Mu), kronecker(Q,V));
  {
    real logPriorBeta;
    matrix[nc,ns] A = Beta - Mu;
    matrix[ns,ns] tmp = (Q\A') * (V\A);
    logPriorBeta = -0.5*trace(tmp) - 0.5*nc*log_determinant(Q) - 0.5*ns*log_determinant(V);
    target += logPriorBeta;
  }
  to_vector(Gamma) ~ multi_normal_cholesky(rep_vector(0,nc*nt), LU0);
  V ~ inv_wishart(f0, V0);
  for(h in 1:nf){
    Eta[,h] ~ std_normal();
    Lambda0[h,] ~ student_t(nu,0,1);
  }
  delta[1] ~ gamma(a1,b1);
  delta[2:nf] ~ gamma(a2,b2);
  sigma ~ inv_gamma(aS,bS);
}
```

### Basic Probit model

```
data{
  int<lower=1> ny;
  int<lower=1> ns;
  int<lower=1> nc;
  int<lower=1> nt;
  int<lower=1> nf;
  int<lower=0,upper=1> Y[ny,ns];
  matrix[ny,nc] X;
  matrix[ns,nt] T;
  corr_matrix[ns] C;

  real<lower=nc> f0;
  cov_matrix[nc] V0;
  cov_matrix[nc*nt] U0;
  vector<lower=0>[ns] aS;
  vector<lower=0>[ns] bS;
  real<lower=0> nu;
  real<lower=0> a1;
  real<lower=0> a2;
  real<lower=0> b1;
  real<lower=0> b2;
}

transformed data{
  matrix[nc*nt,nc*nt] LU0 = cholesky_decompose(U0);
}

parameters{
  matrix[nc,ns] Beta;
  matrix[nc,nt] Gamma;
  cov_matrix[nc] V;
  matrix[ny,nf] Eta;
  matrix[nf,ns] Lambda0;
  vector<lower=0>[nf] delta;
  real<lower=0,upper=1> rho;
}

transformed parameters{
  vector[nf] tau = exp(cumulative_sum(log(delta)));
  matrix[nf,ns] Lambda = Lambda0 ./ rep_matrix(sqrt(tau),ns);
}

model{
  matrix[ny,ns] L = X*Beta + Eta*Lambda;
  matrix[ns,ns] Q = rho*C + diag_matrix(rep_vector(1-rho,ns));
  matrix[nc,ns] Mu = Gamma * T';

  # observation model
  for(j in 1:ns){
    Y[,j] ~ bernoulli(Phi(L[,j]));
  }

  # priors
  # to_vector(Beta) ~ multi_normal(to_vector(Mu), kronecker(Q,V));
  {
    real logPriorBeta;
    matrix[nc,ns] A = Beta - Mu;
    matrix[ns,ns] tmp = (Q\A') * (V\A);
    logPriorBeta = -0.5*trace(tmp) - 0.5*nc*log_determinant(Q) - 0.5*ns*log_determinant(V);
    target += logPriorBeta;
  }
  to_vector(Gamma) ~ multi_normal_cholesky(rep_vector(0,nc*nt), LU0);
  V ~ inv_wishart(f0, V0);
  for(h in 1:nf){
    Eta[,h] ~ std_normal();
    Lambda0[h,] ~ student_t(nu,0,1);
  }
  delta[1] ~ gamma(a1,b1);
  delta[2:nf] ~ gamma(a2,b2);
}
```

## Python code for cases 1-TFP and 4-TFP

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 12 20:39:05 2019

@author: gtikhonov
"""
import numpy as np
import pickle
import pandas as pd
import os
import time
import tensorflow as tf
import tensorflow_probability as tfp
from matplotlib import pyplot as plt
from time import localtime, strftime
from scipy.sparse import csr_matrix
from scipy.stats import invwishart
from tqdm import tqdm
import argparse
tfd = tfp.distributions
tfb = tfp.bijectors
dtype = np.float32
tf.reset_default_graph()
plotFlag = True

parser=argparse.ArgumentParser()
parser.add_argument('--nf')
parser.add_argument('--methodInd')
parser.add_argument('--likeInd')
parser.add_argument('--samN')
parser.add_argument('--thinN')
parser.add_argument('--maxTreeDepth')
parser.add_argument('--cInd')
parser.add_argument('--obsUsed')
parser.add_argument('--calcFlag')
parser.add_argument('--caseNumber')
args=parser.parse_args()

nf = 5
methodInd = 1
likeInd = 4
samN = 1000
thinN = 1
maxTreeDepth = 10
cInd = 0
obsUsed = 1*2000
calcFlag = 1
caseNumber = 1
step_size_init = 0.01
```

```
if(args.nf is not None): nf = int(args.nf)
if(args.methodInd is not None): methodInd = int(args.methodInd)
if(args.likeInd is not None): likeInd = int(args.likeInd)
if(args.samN is not None): samN = int(args.samN)
if(args.thinN is not None): thinN = int(args.thinN)
if(args.maxTreeDepth is not None): maxTreeDepth = int(args.maxTreeDepth)
if(args.cInd is not None): cInd = int(args.cInd)
if(args.obsUsed is not None): obsUsed = int(args.obsUsed)
if(args.calcFlag is not None): calcFlag = bool(args.calcFlag)
if(args.caseNumber is not None): caseNumber = int(args.caseNumber)

dataDir = "data/simulated"
XData = pd.read_csv(os.path.join(dataDir,"C%d_X.csv" % caseNumber),sep=",",index_col=0)
YData = pd.read_csv(os.path.join(dataDir,"C%d_Y.csv" % caseNumber),sep=",",index_col=0)
TData = pd.read_csv(os.path.join(dataDir,"C%d_Tr.csv" % caseNumber),sep=",",index_col=0)
CData = pd.read_csv(os.path.join(dataDir,"C%d_C.csv" % caseNumber),sep=",",index_col=0)
PiData = pd.read_csv(os.path.join(dataDir,"C%d_Pi.csv" % caseNumber),sep=",",index_col=0)

ny, ns = YData.shape
ny = np.max([np.min([ny,obsUsed]),0])
XData = XData[:ny]
YData = YData[:ny]
PiData = PiData[:ny]

Y = YData.values.astype(dtype)
X = XData.values.astype(dtype)
T = TData.values.astype(dtype)
C = CData.values.astype(dtype)
Pi = PiData.values.flatten()-1
nq = np.unique(Pi).shape[0]
P = csr_matrix((np.ones(ny), (np.arange(ny), Pi)), shape=(ny,nq))
nc = X.shape[1]
nt = T.shape[1]
Yo = (~np.isnan(Y)).astype(dtype)
YoN = np.sum(Yo,0)

# priors
f0,V0 = nc+1,np.eye(nc,dtype=dtype)
U = np.eye(nt,dtype=dtype)
nu, a1, a2, b1, b2 = dtype(3), dtype(50), dtype(50), dtype(1), dtype(1)
aS, bS = dtype(1), dtype(5)
priorDict = {"f0":f0,"V0":V0, "nu":nu,"a1":a1,"b1":b1,"a2":a2,"b2":b2, "aS":aS,"bS":bS}


####################### - Probability functions - #######################

def log_like4(rho,V,U,Lambda,sigma2, Y,X,Pi,T,C, dtype):
  ny,ns = Y.shape
  nc,nf = X.shape[1], int(Lambda.shape[0])
  if not np.size(np.unique(Pi))==ny:
    raise Exception("This likelihood is compatible only with np.size(np.unique(Pi))==ny")
  Yo = (~np.isnan(Y)).astype(dtype)
  if not (Yo==True).all():
    raise Exception("This likelihood is compatible only with fully observed data")
  CM = rho*C + (1-rho)*tf.eye(ns,dtype=dtype)
  M1 = tf.reshape(tf.transpose(tf.reshape(V,[nc,nc,1,1])*tf.reshape(CM,[1,1,ns,ns]),
                              [0,2,1,3]),[nc*ns,nc*ns])
  LU = tf.cholesky(U)
  TLU = tf.matmul(T,LU)
  TUT = tf.matmul(TLU,TLU,transpose_b=True)
  M2 = tf.reshape(tf.transpose(tf.matrix_diag(tf.tile(tf.expand_dims(TUT,-1),[1,1,nc])),
                              [2,0,3,1]), [nc*ns,nc*ns])
  M = M1 + M2
  LM = tf.cholesky(M)
  iM = tf.cholesky_solve(LM, tf.eye(nc*ns,dtype=dtype))
  XTX = np.matmul(X.T,X)
  XTiDX = XTX[:,:,None]/sigma2
  A11 = tf.reshape(tf.transpose(tf.matrix_diag(XTiDX),[0,2,1,3]), [nc*ns,nc*ns],name="A11")
  LambdaiD05 = Lambda * tf.rsqrt(sigma2)
  LambdaiD = Lambda / sigma2
  LambdaiDLambdaT = tf.matmul(LambdaiD05, LambdaiD05, transpose_b=True, name="LambdaiDLambdaT")
  B = tf.eye(nf,dtype=dtype) + LambdaiDLambdaT
  LB = tf.cholesky(B, name="LB")
  iLBLambdaiD = tf.matrix_triangular_solve(LB, LambdaiD)
  H = tf.matmul(iLBLambdaiD, iLBLambdaiD, transpose_a=True)
  A12iBA21 = tf.reshape(tf.transpose(XTX[:,:,None,None]*tf.expand_dims(tf.expand_dims(H,0),0),
                                    [0,2,1,3]),[nc*ns,nc*ns])
  Q = iM + A11 - A12iBA21
  LQ = tf.cholesky(Q)
  logDetD = ny*tf.reduce_sum(tf.math.log(sigma2))
  logDetM = 2*tf.reduce_sum(tf.math.log(tf.matrix_diag_part(LM)))
  logDetLF = 0
  logDetQ = 2*tf.reduce_sum(tf.math.log(tf.matrix_diag_part(LQ)))
  logDetB = ny*2*tf.reduce_sum(tf.math.log(tf.matrix_diag_part(LB)))
  qF1 = tf.reduce_sum(np.square(Y)/sigma2)
  iDY = Y / sigma2
  Z1 = tf.matmul(X,iDY,transpose_a=True)
  Z2 = tf.matmul(iDY,Lambda,transpose_b=True)
  z1 = tf.reshape(Z1,[nc*ns,1])
  iBZ2 = tf.transpose(tf.cholesky_solve(LB, tf.transpose(Z2)))
  z = z1 - tf.reshape(tf.matmul(tf.matmul(X, iBZ2, transpose_a=True), LambdaiD), [nc*ns,1])
  qF2 = tf.reduce_sum(tf.square(tf.matrix_triangular_solve(LQ, z)))
  qF3 = tf.reduce_sum(Z2*iBZ2)
  constAdd = -0.5*np.sum(Yo)*np.log(2*np.pi)
  res = -0.5*(logDetD+logDetM+logDetLF+logDetQ+logDetB +qF1-qF2-qF3) + constAdd
  return res

def log_prob(rho,V,U, deltaLog,Lambda0, sigma2, Y,X,Pi,T,C, likeInd, priorDict, dtype):
  ny,ns = Y.shape
  nc = X.shape[1]
  f0,V0 = priorDict["f0"],priorDict["V0"]
  nu,a1,b1,a2,b2,aS,bS = priorDict["nu"],priorDict["a1"],priorDict["b1"],priorDict["a2"],
                         priorDict["b2"],priorDict["aS"],priorDict["bS"]
  tauLog = tf.cumsum(deltaLog)
#  print("Option 1")
#  iV = tf.matrix_inverse(V)
#  log_prior_V = tfd.Wishart(f0,V0).log_prob(iV)
#  log_prior_V = 0
  LV = tf.cholesky(V)
  logDetV = 2*tf.reduce_sum(tf.math.log(tf.matrix_diag_part(LV)))
  log_prior_V = -0.5*(f0+nc+1)*logDetV - 0.5*tf.linalg.trace(tf.cholesky_solve(LV,V0))
  delta = tfb.Exp().forward(deltaLog)
  log_prior_delta1 = tfd.Gamma(a1,b1).log_prob(delta)
  log_prior_delta2 = tf.reduce_sum(tfd.Gamma(a2,b2).log_prob(delta))
  log_prior_delta = log_prior_delta1 + log_prior_delta2
  log_prior_Lambda0 = tf.reduce_sum(tfd.StudentT(nu,0,1).log_prob(Lambda0))
  log_prior_sigma2 = tf.reduce_sum(tfd.InverseGamma(aS,bS).log_prob(sigma2))
  Lambda = Lambda0 / tf.expand_dims(tf.exp(0.5*tauLog),1)
  logPrior = log_prior_V + log_prior_delta+log_prior_Lambda0 + log_prior_sigma2
  if ny > 0:
    log_like_list = [None,None,None,None,log_like4]
    log_like = log_like_list[likeInd]
    logLike = log_like(rho,V,U,Lambda,sigma2, Y,X,Pi,T,C, dtype)
    logProb = logLike + logPrior
  else:
    logProb = logPrior
  return logProb


def sample_Beta(rho,V,U,Lambda,sigma2, Y,X,Pi,T,C, dtype):
  ny,ns = Y.shape
  nc,nf = X.shape[1], int(Lambda.shape[0])
  nq = np.size(np.unique(Pi))
  Yo = (~np.isnan(Y)).astype(dtype)
  iD = Yo*tf.math.reciprocal(sigma2)
  iD05 = Yo*tf.rsqrt(sigma2)
  CM = rho*C + (1-rho)*tf.eye(ns,dtype=dtype)
  M1 = tf.reshape(tf.transpose(tf.reshape(V,[nc,nc,1,1])*tf.reshape(CM,[1,1,ns,ns]),
                              [0,2,1,3]),[nc*ns,nc*ns])
  LU = tf.cholesky(U)
  TLU = tf.matmul(T,LU)
  TUT = tf.matmul(TLU,TLU,transpose_b=True)
  M2 = tf.reshape(tf.transpose(tf.matrix_diag(tf.tile(tf.expand_dims(TUT,-1),
                                                    [1,1,nc])),[2,0,3,1]), [nc*ns,nc*ns])
  M = M1 + M2
  LM = tf.cholesky(M)
  iM = tf.cholesky_solve(LM, tf.eye(nc*ns,dtype=dtype))
  XiD05 = tf.expand_dims(tf.transpose(iD05),2)*X
  XTiDX = tf.matmul(XiD05,XiD05,transpose_a=True)
  A11 = tf.reshape(tf.transpose(tf.matrix_diag(tf.transpose(XTiDX,[1,2,0])),
                               [0,2,1,3]), [nc*ns,nc*ns])
  XiD = tf.expand_dims(tf.transpose(iD),2)*X
  if np.array_equal(Pi, np.arange(ny)):
    XTiDP = XiD
  else:
    XTiDP = tf.transpose(tf.unsorted_segment_sum(tf.transpose(XiD,[1,0,2]), Pi, nq),
                        [1,0,2])
  A12S = tf.reshape(Lambda,[nf,ns,1,1]) * XTiDP
  LambdaiD05 = Lambda * tf.expand_dims(iD05,1)
  LambdaiDLambdaT = tf.matmul(LambdaiD05, LambdaiD05, transpose_b=True)
  if np.array_equal(Pi, np.arange(ny)):
    A22St = LambdaiDLambdaT
  else:
    A22St = tf.unsorted_segment_sum(LambdaiDLambdaT, Pi, nq)
  B = tf.eye(nf,dtype=dtype) + A22St
  LB = tf.cholesky(B)
  A21St = tf.reshape(tf.transpose(A12S, [2,0,3,1]), [nq,nf,nc*ns])
  iLBA21St = tf.matrix_triangular_solve(LB, A21St)
  iLBA21 = tf.reshape(iLBA21St, [nq*nf,nc*ns])
  A12iBA21 = tf.matmul(iLBA21, iLBA21, transpose_a=True)
  Q = iM + A11 - A12iBA21
  LQ = tf.cholesky(Q)
  iDY = Y / sigma2
  Z1 = tf.matmul(X,iDY,transpose_a=True)
  if np.array_equal(Pi, np.arange(ny)):
    Z2 = tf.matmul(iDY,Lambda,transpose_b=True)
  else:
    Z2 = tf.unsorted_segment_sum(tf.matmul(iDY,Lambda,transpose_b=True), Pi, nq)
  z1 = tf.reshape(Z1,[nc*ns,1])
  iLBz2 = tf.reshape(tf.matrix_triangular_solve(LB, tf.expand_dims(Z2,2)),[nq*nf,1])
  z = z1 - tf.matmul(iLBA21, iLBz2, transpose_a=True)
  mu = tf.cholesky_solve(LQ, z)
  res = mu + tf.matrix_triangular_solve(LQ, tf.random_normal([nc*ns,1],
                                        dtype=dtype,seed=0), adjoint=True)
  return res

def sample_Gamma(rho,V,U,Beta, T,C):
  ns,nt = T.shape
  nc = int(Beta.shape[0])
  CM = rho*C + (1-rho)*np.eye(ns)
  LCM = np.linalg.cholesky(CM)
  iLCMT = np.linalg.solve(LCM, T)
  iV = np.linalg.inv(V)
  TiCMT = np.matmul(iLCMT.T,iLCMT)
  iSigma = np.kron(np.eye(nc),np.linalg.inv(U)) + np.kron(iV,TiCMT)
  Sigma = np.linalg.inv(iSigma)
  Z = np.matmul(iV, np.matmul(np.linalg.solve(CM,Beta.T).T,T))
  mu = np.linalg.solve(iSigma, np.reshape(Z,[nc*nt]))
  res = mu + np.matmul(np.linalg.cholesky(Sigma), np.random.normal(size=[nc*nt]))
  return res

####################### - Likelihood test - #######################
np.random.seed(cInd)
rho_init = np.random.uniform(size=1).astype(dtype)
deltaLog_init = np.log(np.random.gamma(np.concatenate(([a1],np.repeat(a2,nf-1))),
                       np.concatenate(([b1],np.repeat(b2,nf-1))))).astype(dtype)
V_init = invwishart.rvs(f0,V0).astype(dtype)
V_init = (0.3/2)**2 * np.eye(nc,dtype=dtype)
Lambda0_init = np.random.normal(size=[nf,ns]).astype(dtype)
sigma2_init = np.minimum(np.random.gamma(1,5,size=[ns])**-1, 2).astype(dtype)

rhoT = tf.constant(rho_init)
```

```
VT = tf.constant(V_init)
Lambda0T = tf.constant(Lambda0_init)
sigma2T = tf.constant(sigma2_init)

vT = log_prob(rhoT,VT,U,deltaLog_init,Lambda0T,sigma2T, Y,X,Pi,T,C, likeInd,priorDict,dtype)
#dvT = tf.gradients(vT, [rhoT, VT, UT, LambdaT, sigma2T])
with tf.Session() as sess:
  #  v, dv = sess.run([vT,dvT])
  v = sess.run([vT])
print(v)
#print(dv)

def step_size_setter_fn(kernel_results, new_step_size):
  return kernel_results._replace(
      inner_results=kernel_results.inner_results._replace(
          step_size=new_step_size))

def step_size_getter_fn(kernel_results):
  return [tf.cast(ss,dtype) for ss in kernel_results.inner_results.step_size]

####################### - Parameter learning - #######################
tf.reset_default_graph()
np.random.seed(cInd)
iterN = samN*thinN

burnIterN = int(iterN*0.5)
adaptIterN = int(iterN*0.4)
dirName = "fm"
fileName = "fm15_case%d_ny%.4d_nf%.2d_sam%.5d_thin%.5d_mtd%.3d_like%d_c%.2d.pkl"
% (caseNumber,ny,nf,samN,thinN,maxTreeDepth,likeInd,cInd)
print(fileName)
if calcFlag:
  log_probEff = lambda rho,V,delta,Lambda0,sigma2: log_prob(rho,V,U, delta,Lambda0,
                                          sigma2, Y,X,Pi,T,C, likeInd,priorDict,dtype)
  initialState = [rho_init,V_init, deltaLog_init,Lambda0_init, sigma2_init]
#  adaptive_hmc = tfp.mcmc.SimpleStepSizeAdaptation(
#   tfp.mcmc.NoUTurnSampler(
#        target_log_prob_fn=unnormalized_log_prob,
#        step_size=10.),
#   step_size_setter_fn = step_size_setter_fn,
#   step_size_getter_fn = step_size_getter_fn,
#   num_adaptation_steps=int(num_burnin_steps * 0.8))

  nuts = tfp.mcmc.NoUTurnSampler(
    target_log_prob_fn=log_probEff,
    seed=0, step_size=step_size_init)
  VBijector = tfb.Chain([tfb.CholeskyOuterProduct(),
                         tfb.ScaleTriL(diag_shift=dtype(1e-05))])
  bijectorList = [tfb.Sigmoid(),VBijector,tfb.Identity(),
                  tfb.Identity(),tfb.Softplus()]
  unbounded_nuts = tfp.mcmc.TransformedTransitionKernel(inner_kernel=nuts,
                                                        bijector=bijectorList)
  adaptive_unbounded_nuts = tfp.mcmc.SimpleStepSizeAdaptation(unbounded_nuts,
                                                    num_adaptation_steps=adaptIterN,
                                                    target_accept_prob=dtype(0.75),
                                                    adaptation_rate=dtype(0.01),
                                                    step_size_getter_fn=step_size_getter_fn,
                                                    step_size_setter_fn=step_size_setter_fn)
  samples, [step_size, log_accept_ratio] = tfp.mcmc.sample_chain(
      num_results=samN,
      num_burnin_steps=burnIterN,
      current_state=initialState,
      kernel=adaptive_unbounded_nuts,
      num_steps_between_results=np.max(thinN-1,0),
      trace_fn=lambda _, pkr: [pkr.inner_results.inner_results.step_size,
                              pkr.inner_results.inner_results.log_accept_ratio])
  print("Initialization", strftime("%Y-%m-%d %H:%M:%S", localtime()))
  with tf.Session() as sess:
    tf.global_variables_initializer().run()
    print("Sampling started", flush=True)
    startTime = time.time()
    samplesVal, step_sizeVal, log_accept_ratioVal =
      sess.run([samples, step_size, log_accept_ratio])
  endTime = time.time()
  elapsedTime = endTime-startTime
  print("Accept ratio", np.mean(np.exp(np.minimum(log_accept_ratioVal,0))),
        "   step size", np.mean(step_sizeVal))
  print(elapsedTime)
  os.makedirs(dirName,exist_ok=True)
  with open('%s/%s' % (dirName,fileName), 'wb') as f:
    pickle.dump(samplesVal, f)
else:
  with open('%s/%s' % (dirName,fileName), 'rb') as f:
    samplesVal = pickle.load(f)
  elapsedTime = 0

rhoPost, VPost, deltaLogPost, Lambda0Post, sigma2Post = samplesVal[:5]
tauLogPost = np.cumsum(deltaLogPost, axis=1)
LambdaPost = Lambda0Post / np.exp(0.5*tauLogPost)[:,:,None]

print("Sampling Beta")
BetaPost = np.zeros([samN,nc,ns])
tf.reset_default_graph()
rhoP = tf.placeholder(dtype, shape=(1))
VP = tf.placeholder(dtype, shape=(nc,nc))
LambdaP = tf.placeholder(dtype, shape=(nf,ns))
sigma2P = tf.placeholder(dtype, shape=(ns))
BetaT = sample_Beta(rhoP,VP,U, LambdaP, sigma2P, Y,X,Pi,T,C, dtype)
with tf.Session() as sess:
  for i in tqdm(range(samN)):
    feed_dict = {rhoP: rhoPost[i], VP: VPost[i], LambdaP: LambdaPost[i],
      sigma2P: sigma2Post[i]}
    BetaPost[i] = np.reshape(sess.run(BetaT, feed_dict=feed_dict), [nc,ns])

print("Sampling Gamma")
GammaPost = np.zeros([samN,nc,nt])
```

```
for i in tqdm(range(samN)):
  GammaPost[i] = np.reshape(sample_Gamma(rhoPost[i],VPost[i],U,BetaPost[i], T,C),
                            [nc,nt])

postList = samplesVal + [BetaPost, GammaPost, elapsedTime]
with open('%s/%s' % (dirName,fileName), 'wb') as f:
  pickle.dump(postList, f)

if plotFlag:
  OmegaPost = np.matmul(np.transpose(LambdaPost,[0,2,1]), LambdaPost)
  OmegaEss = tfp.mcmc.effective_sample_size(OmegaPost).eval(session=tf.Session())
  sigma2Ess = tfp.mcmc.effective_sample_size(sigma2Post).eval(session=tf.Session())
  bins = np.linspace(0, samN, 50)
  plt.hist(OmegaEss.flatten(),bins,density=True,alpha=0.5)
  plt.hist(sigma2Ess.flatten(),bins,density=True,alpha=0.5)
  plt.xlim((0,samN))
  plt.show()
  OmegaMean = np.mean(OmegaPost+0*np.array([np.diag(sigma2Post[s,:]) for s in range(samN)]),0)
  vMaxAbs = np.max(np.abs(OmegaMean))
  plt.figure(figsize=(10,8))
  plt.imshow(OmegaMean, cmap="bwr", vmin=-vMaxAbs,vmax=vMaxAbs)
  plt.colorbar()
  plt.show()
```

## References

Carpenter, Bob, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. "Stan: A Probabilistic Programming Language." *Journal of Statistical Software* 76 (1). doi:10.18637/jss.v076.i01.

Neal, R. M. 2012. "MCMC Using Hamiltonian Dynamics." *arXiv* https://arxiv.org/abs/1206.1901v1.

Ovaskainen, O., G. Tikhonov, A. Norberg, F. Guillaume Blanchet, L. Duan, D. Dunson, T. Roslin, and N. Abrego. 2017. "How to Make More Out of Community Data? A Conceptual Framework and Its Implementation as Models and Software." *Ecol Lett* 20: 561–76. doi:10.1111/ele.12757.

Team, Stan Development. 2019. "RStan: The R Interface to Stan. R Package Version 2.19.2. Http://Mc-Stan.org/."