

```

/*
* Code is of the file DepdenetPartitionsAR1LikeAR1Theta_sPPM2.c
* that one since the R function drpm_fit calls the function which
* is defined there:
    initial_partition <- 0
    C.out <- .C("drpm_ar1_sppm", // defined in that file
                as.integer(draws), as.integer(burn), as.integer(thin),
                as.integer(nsubject), as.i)
* so it should be the "main" file
*/
*****Copyright (c) 2018 Garrett Leland Page*****
*
* This file contains C code for an MCMC algorithm constructed
* to fit a hierarchical model that incorporates the idea of
* temporally dependent partitions.
*
* I will include model details at a later date
*
*****#include "matrix.h"
#include "Rutil.h"
#include <R_ext/Lapack.h>
#include <R.h>
#include <Rmath.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
*****
* The following are the inputs of the function that are read from R
*
* draws = total number of MCMC draws
* burn = number of MCMC draws discarded as burn-in
* thin = indicates how much MCMC chain should be thinned
*
* nsubject = integer for the number of subjects in data set
* ntime = integer for the number of time points
* y = double nsubject x ntime matrix containing response for each
subject at time t
* s1 = nsubject x 1 vector containing spatial coordinate one
* s2 = nsubject x 1 vector containing spatial coordinate two
*
* M = double indicating value of M associated with cohesion (scale
parameter of DP).
* alpha = double - prior probability of being pegged, starting value
*****  

extreme longest well look  

value t1 t2 t3 ...  

# of time points: ntime  

# of draws: draws
```

```

only if update_alpha is TRUE
* priorvals = vector containing values for prior distributions as
follows
*
* time_specific_alpha = integer - logical indicating whether to make
alpha time-specific or one global alpha.
* update_alpha = integer - logical indicating whether to update alpha
or not.
* update_eta1 = integer - logical indicating whether to update eta1
or set it to zero for all subjects.
* update_phi1 = integer - logical indicating whether to update phi1
or set it to zero.
* sppm = integer - logical indicating whether to use spatial
information or not
*
* SpatialCohesion = integer indication which cohesion to use
* 1 -Auxiliary
* 2- Double dipper
*
* cParms - vector holding values employed in the cohesion
*
* OUTPUT
* Si -
* mu -
* sig2 -
* eta1
* theta -
* tau2 -
* phi0 -
* phi1
* gamma -
* alpha.out -
* like
* lpml -
* waic -
*****
void drpm_ar1_sppm(int *draws, int *burn, int *thin, int *nsubject,
int *ntime,
    double *y, double *s1, double *s2, double *M,
    double *alpha, double *modelPriors, double *alphaPriors,
    int *time_specific_alpha,
    int *update_alpha, int *update_eta1, int *update_phi1,
    int *sppm, int *SpatialCohesion, double *cParms, double *mh,
    int *space_1, int *simpleModel, double *theta_tau2,
    int *Si, double *mu, double *sig2, double *eta1, double
*theta, double *tau2,
    double *phi0, double *phi1, double *lam2, int *gamma, double
*alpha_out,

```

```

double *fitted, double *llike, double *lpm1, double *waic){

// i - MCMC iterate
// ii - MCMC iterate that is saved
// j - subject iterate
// jj - second subject iterate
// t - time iterate
// k - cluster iterate
// kk - second cluster iterate
// p - prediction iterate

int i, ii, j, jj, t, k, kk;
ii = 0;

int nout = (*draws - *burn) / (*thin);
Rprintf("nsubject = %d\n", *nsubject);
Rprintf("ntime = %d\n", *ntime);
Rprintf("nout = %d\n", nout);
Rprintf("update_alpha = %d\n", *update_alpha);
Rprintf("update_eta1 = %d\n", *update_eta1);
Rprintf("update_phi1 = %d\n", *update_phi1);

// =====
// Memory vectors to hold MCMC iterates for non cluster specific
parameters
// =====

// This variable is used to create a "buffer" zone of memory so
that updating
// things on time boundary do not need special attention in the
algorithm since
// I have to look at time period before and after when updating
partitions
int ntime1 = *ntime + 1;
// I am adding one more year as an empty vector
// so that the C program does not crash.
int gamma_iter[(*nsubject)*(ntime1)];
int si_iter[(*nsubject)*(ntime1)];
int nclus_iter[ntime1];
double *eta1_iter = R_VectorInit(*nsubject, runif(0,1));
double *theta1_iter = R_VectorInit(ntime1, rnorm(0,3));
double *tau2_iter = R_VectorInit(ntime1, runif(0,
modelPriors[3]*modelPriors[3]));

```

*Annotations:*

- Introducing `ntime1` as `ntime + 1` to handle the boundary case.
- Creating a buffer vector `gamma_iter` of size `nsubject * ntime1`.
- Initializing `si_iter` and `nclus_iter` vectors.
- Defining `modelPriors[3]*modelPriors[3]` as `53`.
- Explaining the use of `gamma_iter` to store subject-level covariates for each time point and each subject.
- Notes on memory allocation and vector operations.

```

double phi0_iter = rnorm(0,3);
double phi1_iter = runif(0,1);
double lam2_iter = runif(0, modelPriors[4]*modelPriors[4]);
double *alpha_iter = R_VectorInit(ntime1, *alpha);

//
=====

//
// Memory vectors to hold MCMC iterates for cluster specific
parameters
//
=====

double *muh = R_VectorInit((*nsubject)*(ntime1), 0.0);
double *sig2h = R_VectorInit((*nsubject)*(ntime1), 1.0);
if(*simpleModel==1){
    for(t = 0; t < ntime1; t++){
        theta1_iter[t] = theta_tau2[0];
        tau2_iter[t] = theta_tau2[1];
    }
}
int nh[(*nsubject)*(ntime1)];
```

*Annotations:*

- Defining `phi0_iter`, `phi1_iter`, `lam2_iter`, and `alpha_iter`.
- Memory vectors for cluster-specific parameters: `muh` and `sig2h`.
- Condition for simple model (value 1).
- Initialization of `theta1_iter` and `tau2_iter` from `theta_tau2`.
- Definition of `nh` as a matrix of size `nsubject * ntime1`.
- Notes on memory management and buffer zones.

```

// Initialize a few parameter vectors
// =====

// Initialize Si according to covariates
// I am adding one time period here to have
// scratch memory (never filled in) so that
// I can avoid dealing with boundaries in algorithm
for(j = 0; j < *nsubject; j++){
    for(t = 0; t < ntime1; t++) { // Note I am not initializing the
        // added time memory"
        Si_iter[j*(ntime1) + t] = 1;
        gamma_iter[j*(ntime1) + t] = 0;
        nh[j*(ntime1) + t] = 0;
        if(t==1) Si_iter[j*(ntime1) + t] = 1;
        if(t==ntime) Si_iter[j*(ntime1) + t] = 0;
    }
}

// Initial enumeration of number of subjects per cluster;
for(j = 0; j < *nsubject; j++){
    for(t = 0; t < *ntime; t++){
        nh[(Si_iter[j*(ntime1)+t]-1)*(ntime1) + t] = nh[(Si_iter[j*
(ntime1)+t]-1)*(ntime1) + t + 1];
    }
}
```

*Annotations:*

- Initialization of `Si_iter`, `gamma_iter`, and `nh` vectors.
- Handling boundary conditions for `nh`.
- Notes on subject-level covariates and cluster assignment.

*Annotations:*

- Notes on the `nh` matrix and its relationship to `gamma_iter`.
- Explanation of the `nh` matrix structure and its use in the algorithm.

$t=1 \Rightarrow$  two > values  
 $t=2 \Rightarrow$  2 clusters

$t=2 \Rightarrow$  three > values  
 $t=3 \Rightarrow$  3 clusters

$n_{\text{cl}} = \begin{pmatrix} 8 & 6 & 6 \\ 2 & 3 & 2 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

```

} // Initialize the number of clusters
for(t = 0; t < *ntime; t++){
    nclus_iter[t] = 0;
    for(j = 0; j < *nsubject; j++){
        if(nh[j*(ntime1) + t] > 0) nclus_iter[t] = nclus_iter[t] + 1;
    }
    nclus_iter[t] += (me[j, t] > 0);
}
nclus_iter[*ntime] = 0;

// =====
// scratch vectors of memory needed to update parameters
// =====
// stuff needed to update gamma vectors
int nclus_red=0, nh_red[*nsubject], n_red=0, gt, n_red_1=0, cit_1;
int nh_redtmp[*nsubject], nh_tmp[*nsubject];
int nh_redtmp_no_zero[*nsubject],
nh_tmp_no_zero[*nsubject], nh_red_no_zero[*nsubject];
int nh_red_1[*nsubject];
// int nclus_red_1;
int nh_redtmp_1[*nsubject], nh_tmp_1[*nsubject];
int nh_redtmp_no_zero_1[*nsubject],
nh_red_no_zero_1[*nsubject], nh_tmp_no_zero_1[*nsubject];
double *s1_red = R_VectorInit(*nsubject, 0.0);
double *s2_red = R_VectorInit(*nsubject, 0.0);
for(j=0; j<*nsubject; j++){
    nh_tmp[j] = 0; nh_red[j] = 0; nh_redtmp[j] = 0;
    nh_redtmp_no_zero[j] = 0; nh_tmp_no_zero[j] = 0;
    nh_red_no_zero[j] = 0; nh_red_no_zero_1[j] = 0;
    nh_tmp_1[j] = 0; nh_red_1[j] = 0; nh_redtmp_1[j] = 0;
    nh_redtmp_no_zero_1[j] = 0; nh_red_no_zero_1[j] = 0;
    nh_tmp_no_zero_1[j] = 0;
}
// stuff that I need to update Si (the partition);
int compit[(*nsubject)], comptm1[(*nsubject)], comp2t[(*nsubject)], comptp1[(*
int rho_tmp[*nsubject], Si_tmp[*nsubject], Si_tmp2[*nsubject];
int Si_red[*nsubject], Si_red_1[*nsubject];
int oldLab[*nsubject], reorder[*nsubject];
int iaux, Rindx1, Rindx2, n_tmp, nclus_tmp, rho_comp, indx;
double auxm, auxs, mudraw, sigdraw, maxph, denph, cprobh, uu, lCo,
lCn, lCn_1, lpp;
    
```

score sums in t (more rows)  
 for all subjects (the columns)  
 re) ywands revalue curce de  
 convenience to sumis nclu  
 stutine sw mbr

```

double *ph = R_VectorInit(*nsubject, 0.0);
double *phtmp = R_VectorInit(*nsubject, 0.0);
double *probh = R_VectorInit(*nsubject, 0.0);
double *lgweight = R_VectorInit(*nsubject, 0.0);
double *s1o = R_Vector(*nsubject);
double *s2o = R_Vector(*nsubject);
double *s1n = R_Vector(*nsubject);
double *s2n = R_Vector(*nsubject);
for(j=0; j<(*nsubject); j++){
    comp1t[j] = 0; comptm1[j] = 0, comp2t[j]=0, comptp1[j]=0;
}
// stuff I need to update eta1
double e1o, e1n, logito, logitn, one_phisq;
// stuff I need to update muh and sig2h
double mstar, s2star, sumy, sume2;
double nsig, osig, llo, lln, llr;
double *mu_tmp = R_VectorInit(*nsubject, 0.0);
double *sig2_tmp = R_VectorInit(*nsubject, 1.0);

// stuff that I need for theta and lam2
double summu, nt, ot, lam2tmp, phi1sq, sumt, op1, np1, ol, nl;
// double ssq;

// stuff that I need to update alpha
int sumg;
double astar, bstar, alpha_tmp;

// Stuff to compute lpml, likelihood, and WAIC
int like0, nout_0=0;
double lpml_iter, elppdWAIC;
double *CPO = R_VectorInit((*nsubject)*(ntime1), 0.0);
double *like_iter = R_VectorInit((*nsubject)*(ntime1), 0.0);
double *fitted_iter = R_VectorInit((*nsubject)*(ntime1), 0.0);
double *mnlike = R_VectorInit((*nsubject)*(ntime1), 0.0);
double *mnllike = R_VectorInit((*nsubject)*(ntime1), 0.0);
// stuff to predict
// int gpred[*nsubject], nh_pred[*nsubject];
// =====
// Prior parameter values
// =====
// prior values for sig2
double Asig=modelPriors[2];
double Atau=modelPriors[3];
double Alam=modelPriors[4];
    
```

```

// priors for phi0
double m0 = modelPriors[0], s20 = modelPriors[1];
// priors for alpha
double a = alphaPriors[0], b = alphaPriors[1];
//priors for eta1
double b_eta1 = modelPriors[5];

// DP weight parameter
double Mdp = *M;
Rprintf("Prior values: Asig = %.2f, Atau = %.2f, Alam = %.2f, \n
m0 = %.2f, s20 = %.2f\n", Asig, Atau, Alam, m0, s20);
// Cohesion auxiliary model parameters for Cohesions 3 and 4
double k0=cParms[1], v0=cParms[2];
double *mu0 = R_VectorInit(2,cParms[0]);
double *L0 = R_VectorInit(2*2,0.0);
L0[0] = cParms[3]; L0[3] = cParms[3];

if(*sPPM==1){
  RprintVecAsMat("mu0", mu0, 1, 2);
  Rprintf("k0 = %f\n", k0);
  Rprintf("v0 = %f\n", v0);
  RprintVecAsMat("L0", L0, 2, 2);
}

// M-H step tuning parameter
double csigSIG=mh[0], csigTAU=mh[1], csigLAM=mh[2], csigETA1=mh[3],
csigPHI1=mh[4];
GetRNGstate();
// =====
// start of the mcmc algorithm;
// =====
for(i = 0; i < *draws; i++){
  if((i+1) % 10000 == 0){
    time_t now;
    time(&now);
    Rprintf("mcmc iter = %d
===== \n", i+1);
    Rprintf("%s", ctime(&now));
  }
  // Start updating gamma and partition for each time period
  for(t = 0; t < *ntime; t++){
    // =====

```

```

    // begin by updating gamma (pegged) parameters
    //

    /////////////////////////////////
    for(j = 0; j < *nsubject; j++){
      // at time period one, all gammas are zero (none are
      ``pegged'')
      if(t == 0){
        gamma_iter[j*(ntime1) + t] = 0; vector = [0 0 0] vector vector vector vector vector vector
      } else {
        /////////////////////////////////
        // find the reduced partition information
        // i.e., vector of cluster labels;
        /////////////////////////////////

        Rindx1 = 0;
        for(jj = 0; jj < *nsubject; jj++){
          if(gamma_iter[jj*ntime1 + (t)] == 1){ cluster the units which cross not = 1, we take units containing Rt
            if(jj != j){ that because we treat here regardless they are about the current object j (not w)
              Si_tmp[Rindx1] = Si_iter[jj*ntime1 + (t)];
              Si_tmp2[Rindx1] = Si_iter[jj*ntime1 + (t)];
              comptm1[Rindx1] = Si_iter[jj*ntime1 + (t-1)];
              // Also get the reduced spatial coordinates if
              // space is included
              if(*sPPM==1){
                if((*space_1==1 & t == 0) | (*space_1==0)){
                  s1_red[Rindx1] = s1[jj];
                  s2_red[Rindx1] = s2[jj];
                }
              }
              Rindx1 = Rindx1 + 1;
            }
          }
        }
        Si_tmp2[Rindx1] = Si_iter[j*ntime1 + (t)];
        comptm1[Rindx1] = Si_iter[j*ntime1 + (t-1)];
        n_red = Rindx1; # objects in the reduced partition
        n_red_1 = Rindx1 + 1;
        relabel(Si_tmp, *nsubject, Si_red, reorder, oldLab);
        relabel(Si_tmp2, *nsubject, Si_red_1, reorder, oldLab);
        // I need to keep the relabeled cluster label for the
        // individual so that I know what lgweight to keep in the
        // full conditional.
        cit_1 = Si_red_1[Rindx1];
      }
    }
  }
}

pegged

```

**Definition 1.** We say that partitions  $\rho_{t-1}$  and  $\rho_t$  are compatible with respect to  $\gamma_t$ , if  $\rho_t$  may be obtained from  $\rho_{t-1}$  by reallocating items as indicated by  $\gamma_t$ , that is, those items  $i$  such that  $\gamma_{it} = 0$  for  $i = 1, \dots, m$ . Note that the compatibility relation is an equivalence relation.

There is a simple way to check if  $\rho_{t-1}$  is compatible with  $\rho_t$  with respect to  $\gamma_t$ . Let  $\mathfrak{R}_t = \{i : \gamma_{it} = 1\}$  be the collection of units that remain fixed when moving from time  $t - 1$  to time  $t$ , and  $\mathfrak{R}_t^C = \{i : \gamma_{it} = 0\}$  is the collection of units that do not. Next denote with  $\rho_t^{\mathfrak{R}_t}$  the “reduced” partition at time  $t$  that remains after removing all items in  $\mathfrak{R}_t^C$  from the subsets of  $\rho_t$ . Similarly, let  $\rho_{t-1}^{\mathfrak{R}_t}$  be the reduced partition at time  $t - 1$  based on  $\gamma_t$ . Then  $\rho_{t-1}^{\mathfrak{R}_t}$  and  $\rho_t$  are compatible with respect to  $\gamma_t$  if and only if  $\rho_{t-1}^{\mathfrak{R}_t} = \rho_t^{\mathfrak{R}_t}$ .

```

for(jj = 0; jj < n_red_1; jj++){
    nh_red[jj]=0; nh_red_1[jj]=0;
}
nclus_red = 0;
for(jj = 0; jj < n_red; jj++){
    nh_red[Si_red[jj]-1] = nh_red[Si_red[jj]-1] + 1;
    nh_red_1[Si_red_1[jj]-1] = nh_red_1[Si_red_1[jj]-1] + 1;
    if(Si_red[jj] > nclus_red) nclus_red = Si_red[jj];
}

nh_red_1[Si_red_1[n_red]-1] = nh_red_1[Si_red_1[n_red]-1] +
1;
// this may need to be updated depending on if the value of
gamma changes
//nclus_red_1 = nclus_red;
//if(Si_red_1[n_red] > nclus_red) nclus_red_1 =
Si_red_1[n_red];
lCo=0.0, lCn=0.0;
for(k = 0; k < nclus_red; k++){
    if(*sPPM==1){
        // Note that if space is only included for first time
point
        // then it does not inform gamma.
        if((*space_1==1 & t == 0) | (*space_1==0)){
            indx = 0;
            for(jj = 0; jj < n_red; jj++){
                if(Si_red[jj] == k+1){
                    s1o[indx] = s1_red[jj];
                    s2o[indx] = s2_red[jj];

                    s1n[indx] = s1_red[jj];
                    s2n[indx] = s2_red[jj];
                    indx = indx+1;
                }
            }
            s1n[indx] = s1[j];
            s2n[indx] = s2[j];

            lCo = Cohesion3_4(s1o, s2o, mu0, k0, v0, L0,
nh_red[k], *SpatialCohesion, 1);
            lCn = Cohesion3_4(s1n, s2n, mu0, k0, v0, L0,
nh_red[k]+1,*SpatialCohesion, 1);
        }
    }

    lgweight[k] = log(nh_red[k]) + lCn - lCo;
    Si_red_1[Rindx1] = k+1;
    rho_comp = compatibility(Si_red_1, comptm1, Rindx1+1);
}

```

```

//           if(rho_comp==0) lgweight[k] = log(0);
}

// What if pegged subject creates a singleton in the
reduced partition?
lCn_1=0.0;
if(*sPPM==1){
    if((*space_1==1 & t == 0) | (*space_1==0)){
        s1o[0] = s1[j];
        s2o[0] = s2[j];
        lCn_1 = Cohesion3_4(s1o, s2o, mu0, k0, v0, L0,
1,*SpatialCohesion, 1);
    }
}
lgweight[nclus_red] = log(Mdp) + lCn_1;

Si_red_1[Rindx1] = nclus_red+1;
rho_comp = compatibility(Si_red_1, comptm1, Rindx1+1);
//           if(rho_comp == 0) lgweight[nclus_red] = log(0);

denph = 0.0;
for(k = 0; k < nclus_red + 1; k++){
    phtmp[k] = lgweight[k] ;
}

R_rsort(phtmp, nclus_red + 1) ;

maxph = phtmp[nclus_red];
denph = 0.0;
for(k = 0; k < nclus_red + 1; k++){

    lgweight[k] = exp(lgweight[k] - maxph);
    denph = denph + lgweight[k];
}

for(k = 0; k < nclus_red + 1; k++){
    lgweight[k] = lgweight[k]/denph;
}

probh[1] = alpha_iter[t]/(alpha_iter[t] + (1-
alpha_iter[t])*lgweight[cit_1-1]);
// If gamma is 1 at current MCMC iterate, then there are no
// concerns about partitions being incompatible as gamma

```

```

changes
    // from 1 to 0.
    //
    // However, if gamma's current value is 0, then care must
be taken when
    // trying to change from gamma=0 to gamma=1 as the
partitions may
        // no longer be compatible
        if(gamma_iter[j*(ntime1) + t] == 0){

            // To determine compatibility, I need to make sure that
            // comparison of the reduced partitions is being made with
            // correct cluster labeling. I try to do this by
identifying
            // the sets of units and sequentially assigning "cluster
labels"
            // starting with set that contains the first unit. I
wonder if
            // there is a way to do this in C without using loops?
Who
            // can I ask about this?
            // Get rho_t | gamma_t = 1 and rho_{t-1} | gamma_t = 1
            // when gamma_{it} = 1;
Rindx1 = 0;
for(jj = 0; jj < *nsubject; jj++){
    if(gamma_iter[jj*ntime1 + (t)] == 1){
        comptm1[Rindx1] = Si_iter[jj*ntime1 + (t-1)];
        comp1t[Rindx1] = Si_iter[jj*ntime1 + (t)];
        Rindx1 = Rindx1 + 1;
    }
    // I need to include this because determine what
happens when
        // gamma goes from 0 to 1;
        if(jj == j){
            comptm1[Rindx1] = Si_iter[jj*ntime1 + (t-1)];
            comp1t[Rindx1] = Si_iter[jj*ntime1 + (t)];
            Rindx1 = Rindx1 + 1;
        }
    rho_comp = compatibility(comptm1, comp1t, Rindx1);
    // If rho_comp = 0 then not compatible and probability of
    // pegging subject needs to be set to 0;
    if(rho_comp==0){
        probh[1] = 0;
    }
}

```

```

gt = rbinom(1,probh[1]);
gamma_iter[j*(ntime1) + t] = gt;
}

///////////////////////////////
//
// update partition
//

///////////////////////////////
// The cluster probabilities depend on four partition
probabilities
//
// rho_t
// rho_t.R
// rho_t+1
// rho_t+1.R
//
// I have switched a number of times on which of these needs to
be computed
// and which one can be absorbed in the normalizing constant.
Right now I am
    // leaning towards Pr(rho_t+1) and Pr(rho_t+1.R) can be
absorbed. But I need
    // to use rho_t.R and rho_t+1.R to check compatibility as I
update rho_t.
//

///////////////////////////////
for(jj = 0; jj < *nsubject; jj++){
    rho_tmp[jj] = Si_iter[jj*(ntime1) + t];
}
// It seems to me that I can use some of the structure used to
carry
    // out Algorithm 8 from previous code to keep track of empty
clusters
    // etc.
    for(j = 0; j < *nsubject; j++){
        // Only need to update partition relative to units that are
not pegged
        if(gamma_iter[j*(ntime1) + t] == 0){
            if(nh[(Si_iter[j*(ntime1) + t]-1)*(ntime1) + t] > 1){

                // Observation belongs to a non-singleton ...
                nh[(Si_iter[j*(ntime1) + t]-1)*(ntime1) + t] =

```

```

nh[(Si_iter[j*(ntime1) + t]-1)*(ntime1) + t] - 1;

} else{

    // Observation is a member of a singleton cluster ...

    iaux = Si_iter[j*(ntime1) + t];
    if(iaux < nclus_iter[t]){

        // Need to relabel clusters. I will do this by swapping
        cluster labels
        // Si_iter[j] and nclus_iter along with cluster
        specific parameters;

        // All members of last cluster will be assigned subject
        j's label
        for(jj = 0; jj < *nsubject; jj++){
            if(Si_iter[jj*(ntime1) + t] == nclus_iter[t]){
                Si_iter[jj*(ntime1) + t] = iaux;
            }
        }

        Si_iter[j*(ntime1) + t] = nclus_iter[t];
        // The following steps swaps order of cluster specific
        parameters
        // so that the newly labeled subjects from previous
        step retain
        // their correct cluster specific parameters
        auxs = sig2h[(iaux-1)*ntime1 + t];
        sig2h[(iaux-1)*ntime1 + t] = sig2h[(nclus_iter[t]-1)*
        (ntime1)+t];
        sig2h[(nclus_iter[t]-1)*(ntime1)+t] = auxs;
        auxm = muh[(iaux-1)*ntime1 + t];
        muh[(iaux-1)*ntime1 + t] = muh[(nclus_iter[t]-1)*
        (ntime1)+t];
        muh[(nclus_iter[t]-1)*(ntime1)+t] = auxm;
        // the number of members in cluster is also swapped
        with the last
        nh[(iaux-1)*(ntime1)+t] = nh[(nclus_iter[t]-1)*
        (ntime1)+t];
        nh[(nclus_iter[t]-1)*(ntime1)+t] = 1;
    }
}

```

```

}

// Now remove the ith obs and last cluster;
nh[(nclus_iter[t]-1)*(ntime1)+t] = nh[(nclus_iter[t]-1)*
(ntime1)+t] - 1;
nclus_iter[t] = nclus_iter[t] - 1;

}

for(jj = 0; jj < *nsubject; jj++){

    rho_tmp[jj] = Si_iter[jj*(ntime1) + t];
}

for(k = 0; k < nclus_iter[t]; k++){
    rho_tmp[j] = k+1;

    // First need to check compatibility
    Rindx2=0;
    for(jj = 0; jj < *nsubject; jj++){
        if(gamma_iter[jj*ntime1 + (t+1)] == 1){
            comp2t[Rindx2] = rho_tmp[jj];
            comptp1[Rindx2] = Si_iter[jj*ntime1 + (t+1)];
            Rindx2 = Rindx2 + 1;
        }
    }
    // check for compatibility
    rho_comp = compatibility(comp2t, comptp1, Rindx2);
    if(rho_comp != 1){
        ph[k] = Log(0); // Not compatible
    } else {
        // Need to compute Pr(rhot), Pr(rhot.R), Pr(rhot+1),
        Pr(rhot+1.R)

        for(jj = 0; jj < *nsubject; jj++){
            nh_tmp[jj] = 0;
        }
        n_tmp = 0;
        for(jj = 0; jj < *nsubject; jj++){
            nh_tmp[rho_tmp[jj]-1] = nh_tmp[rho_tmp[jj]-1]+1;
            n_tmp=n_tmp+1;
        }

        nclus_tmp=0;
        for(jj = 0; jj < *nsubject; jj++){
            if(nh_tmp[jj] > 0) nclus_tmp = nclus_tmp + 1;
        }
    }
}

```

```

}

lpp = 0.0;
for(kk = 0; kk < nclus_tmp; kk++){
    // Beginning of spatial part
    lCn = 0.0;
    if(*sPPM==1){
        if((*space_1==1 & t == 0) | (*space_1==0)){
            indx = 0;
            for(jj = 0; jj < *nsubject; jj++){
                if(rho_tmp[jj] == kk+1){

                    s1n[indx] = s1[jj];
                    s2n[indx] = s2[jj];
                    indx = indx+1;
                }
            }
        }
        nh_tmp[kk]*SpatialCohesion, 1);
    }
    // End of spatial part
    lpp = lpp + nclus_tmp*log(Mdp) + lgamma((double)
nh_tmp[kk]) + lCn;
    lpp = lpp + nh_tmp[kk]*log(Mdp) + lgamma((double)
nh_tmp[kk]) + lCn;
    lpp = lpp + (Log(Mdp) + lgamma((double) nh_tmp[kk]) +
lCn);

    if(t==0){
        // //
        ph[k] = dnorm(y[j*ntime] + t,
                        sqrt(sig2h[k*(ntime1) + t]), 1));
        lpp;
    }
    if(t > 0){
        muh[k*(ntime1) + t] +
            eta1_iter[j]*y[j*(ntime) + t-1],
            sqrt(sig2h[k*(ntime1) + t]*
//                                         (1-eta1_iter[j])*eta1_iter[j])),

ph[k] = dnorm(y[j*ntime] + t,
               muh[k*(ntime1) + t] +
                   eta1_iter[j]*y[j*(ntime) + t-1],
                   sqrt(sig2h[k*(ntime1) + t]*
(1-eta1_iter[j])*eta1_iter[j])), 1)+

lpp;
}
// use this to test if MCMC draws from prior are
correct
// ph[k] = lpp;

}

// Determine if E.U. gets allocated to a new cluster
// Need to check compatibility first

rho_tmp[j] = nclus_iter[t]+1;

// First need to check compatibility
Rindx1 = 0, Rindx2=0;
for(jj = 0; jj < *nsubject; jj++){
    if(gamma_iter[jj*ntime1 + (t+1)] == 1){
        comp2t[Rindx2] = rho_tmp[jj];
        comptp1[Rindx2] = Si_iter[jj*ntime1 + (t+1)];
        Rindx2 = Rindx2 + 1;
    }
}
// check for compatibility
rho_comp = compatibility(comp2t, comptp1, Rindx2);
if(rho_comp != 1){
    ph[nclus_iter[t]] = Log(0); // going to own cluster is
not compatible;
} else {

    mudraw = rnorm(theta_iter[t], sqrt(tau2_iter[t]));
    sigdraw = runif(0, Asig);

    for(jj = 0; jj < *nsubject; jj++){
        nh_tmp[jj] = 0;
    }
    n_tmp = 0;
    for(jj = 0; jj < *nsubject; jj++){
        nh_tmp[rho_tmp[jj]-1] = nh_tmp[rho_tmp[jj]-1]+1;
        n_tmp=n_tmp+1;
    }
}

```

```

}

nclus_tmp=0;
for(jj = 0; jj < *nsubject; jj++){
    if(nh_tmp[jj] > 0) nclus_tmp = nclus_tmp + 1;
}

lpp = 0.0;
for(kk = 0; kk < nclus_tmp; kk++){
    // Beginning of spatial part
    lCn = 0.0;
    if(*sPPM==1){
        if((*space_1==1 & t == 0) | (*space_1==0)){
            indx = 0;
            for(jj = 0; jj < *nsubject; jj++){

                if(rho_tmp[jj] == kk+1){

                    s1n[indx] = s1[jj];
                    s2n[indx] = s2[jj];
                    indx = indx+1;
                }
            }
            lCn = Cohesion3_4(s1n, s2n, mu0, k0, v0, L0,
nh_tmp[kk],*SpatialCohesion, 1);
        }
    }
    // End of spatial part

    lpp = lpp + (Log(Mdp) + Lgamma((double) nh_tmp[kk]) +
lCn);
    lpp = lpp + nh_tmp[kk]*log(Mdp) + lgamma((double)
nh_tmp[kk]) + lCn;
}

if(t==0){
    ph[nclus_iter[t]] = dnorm(y[j*(*ntime) + t], mudraw,
sigdraw, 1) +
                                lpp;
}
if(t > 0){

    ph[nclus_iter[t]] = dnorm(y[j*(*ntime) + t],
mudraw + eta1_iter[j]*y[j*(*ntime) + t-1],
sigdraw*sqrt(1-

```

```

eta1_iter[j]*eta1_iter[j]), 1) +
lpp;
}

ph[nclus_iter[t]] = lpp;

}

// Now compute the probabilities
for(k = 0; k < nclus_iter[t]+1; k++) phtmp[k] = ph[k];

R_rsort(phtmp, nclus_iter[t]+1);

maxph = phtmp[nclus_iter[t]];

denph = 0.0;
for(k = 0; k < nclus_iter[t]+1; k++){

    ph[k] = exp(ph[k] - maxph);
    denph = denph + ph[k];
}

for(k = 0; k < nclus_iter[t]+1; k++){
    probh[k] = ph[k]/denph;
}

uu = runif(0.0,1.0);
cprobh= 0.0;;
for(k = 0; k < nclus_iter[t]+1; k++){
    cprobh = cprobh + probh[k];
    if (uu < cprobh){

        iaux = k+1;
        break;
    }
}

if(iaux <= nclus_iter[t]){
    Si_iter[j*(ntime1) + t] = iaux;
    nh[(Si_iter[j*(ntime1) + t]-1)*(ntime1)+t] =
nh[(Si_iter[j*(ntime1) + t]-1)*(ntime1)+t] + 1;
    rho_tmp[j] = iaux;
} else{

    nclus_iter[t] = nclus_iter[t] + 1;
    Si_iter[j*(ntime1) + t] = nclus_iter[t];
}
```

```

nh[(Si_iter[j*(ntime1) + t]-1)*(ntime1)+t] = 1;
rho_tmp[j] = nclus_iter[t];

muh[(Si_iter[j*(ntime1) + t]-1)*(ntime1) + t] = mudraw;
sig2h[(Si_iter[j*(ntime1) + t]-1)*(ntime1) + t] =
sigdraw*sigdraw;
    if(*simpleModel==1) sig2h[(Si_iter[j*(ntime1) +
t]-1)*(ntime1) + t] = 1.0;
}
}

for(jj = 0; jj < *nsubject; jj++){
    Si_tmp[jj] = Si_iter[jj*(ntime1) + t];
    Si_tmp2[jj] = 0;
    reorder[jj] = 0;
}
// I believe that I have to make sure that groups are order
so that
// EU one is always in the group one, and then the smallest
index not
// with group 1 anchors group 2 etc.

relabel(Si_tmp, *nsubject, Si_tmp2, reorder, oldLab);

for(jj=0; jj<*nsubject; jj++){
    Si_iter[jj*(ntime1) + t] = Si_tmp2[jj];
}
for(k = 0; k < nclus_iter[t]; k++){
    mu_tmp[k] = muh[k*(ntime1)+t];
    sig2_tmp[k] = sig2h[k*(ntime1)+t];
}
for(k = 0; k < nclus_iter[t]; k++){
    nh[k*(ntime1)+t] = reorder[k];
    muh[k*(ntime1)+t] = mu_tmp[(oldLab[k]-1)];
    sig2h[k*(ntime1)+t] = sig2h[(oldLab[k]-1)];
}
}
for(j = 0; j < *nsubject; j++){
    Si_tmp[j] = Si_iter[j*(ntime1) + t];
    Si_tmp2[j] = 0;
    reorder[j] = 0;
}
// I believe that I have to make sure that groups are order so
that
// EU one is always in the group one, and then the smallest
index not
// with group 1 anchors group 2 etc.

relabel(Si tmp, *nsubject, Si tmp2, reorder, oldLab);

```

```

for(j=0; j<*nsubject; j++){
    Si_iter[j*(ntime1) + t] = Si_tmp2[j];
}
for(k = 0; k < nclus_iter[t]; k++){
    mu_tmp[k] = muh[k*(ntime1)+t];
    sig2_tmp[k] = sig2h[k*(ntime1)+t];
}
for(k = 0; k < nclus_iter[t]; k++){
    nh[k*(ntime1)+t] = reorder[k];
    muh[k*(ntime1)+t] = mu_tmp[(oldLab[k]-1)];
    sig2h[k*(ntime1)+t] = sig2_tmp[(oldLab[k]-1)];
}

// for(k = 0; k < nclus_iter[t]; k++) sig2h[k*(ntime1)+t] = 1.0;
for(k = 0; k < nclus_iter[t]; k++){
    ///////////////////////////////////////////////////////////////////
    // update yt ∈ Rntime1, but to one / store the
    // result in one certain yt to each
    // iteration in case the will be obs under
    // they have yt ∈ to the same cluster
    ///////////////////////////////////////////////////////////////////  $y^t = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \Rightarrow y^t = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$ 
    if(t==0){
        sumy = 0.0;
        for(j = 0; j < *nsubject; j++){
            if(Si_iter[j*(ntime1) + t] == k+1){
                sumy = sumy + y[j*(ntime1)+t];
            }
        }
        s2star = 1/((double) nh[k*(ntime1)+t]/sig2h[k*(ntime1) + t]
+ 1/tau2_iter[t]);
        mstar = s2star*( (1/sig2h[k*(ntime1) + t])*sumy +
(1/tau2_iter[t])*theta_iter[t]);
    }
    if(t > 0){
        sumy = 0.0;
        sume2 = 0.0;
        for(j = 0; j < *nsubject; j++){
            if(Si_iter[j*(ntime1) + t] == k+1){
                sume2 = sume2 + 1.0/(1-eta1_iter[j]*eta1_iter[j]);
                sumy = sumy + (y[j*(ntime1)+t] - eta1_iter[j]*y[j*
(ntime1)+t-1])/
(1-eta1_iter[j]*eta1_iter[j]);
            }
        }
        s2star = 1/( (1.0/sig2h[k*(ntime1) + t])*sume2 +
1/tau2_iter[t]);
    }
}

```

```

1/tau2_iter[t]);
    mstar = s2star*( 1.0/sig2h[k*(ntime1) + t])*sumy +
(1/tau2_iter[t])*theta_iter[t]);
}

muh[k*(ntime1) + t] = rnorm(mstar, sqrt(s2star));
muh[k] = 0.0;
///////////////////////////////
// update sig2h           //
// /////////////////////////
osig = sqrt(sig2h[k*(ntime1) + t]);
nsig = rnorm(osig,csigSIG);
if(nsig > 0.0 & nsig < Asig){

lln = 0.0;
llo = 0.0;
if(t == 0){
    for(j = 0; j < *nsubject; j++){
        if(Si_iter[j*(ntime1) + t] == k+1){
            llo = llo + dnorm(y[j*(*ntime)+t], muh[k*(ntime1) +
t], osig,1);
            lln = lln + dnorm(y[j*(*ntime)+t], muh[k*(ntime1) +
t], nsig,1);
        }
    }
}
if(t > 0){
    for(j = 0; j < *nsubject; j++){
        if(Si_iter[j*(ntime1) + t] == k+1){
            llo = llo + dnorm(y[j*(*ntime)+t], muh[k*(ntime1) +
t] +
eta1_iter[j]*y[j*(*ntime) + t-1],
osig*sqrt(1-
eta1_iter[j]*eta1_iter[j]),1);
            lln = lln + dnorm(y[j*(*ntime)+t], muh[k*(ntime1) +
t] +
eta1_iter[j]*y[j*(*ntime) + t-1],
nsig*sqrt(1-
eta1_iter[j]*eta1_iter[j]),1);
        }
    }
}
llo = llo + dunif(osig, 0.0, Asig, 1);
lln = lln + dunif(nsig, 0.0, Asig, 1);
}

```

```

llr = lln - llo;
uu = runif(0,1);

if(Log(uu) < llr){
    sig2h[k*(ntime1) + t] = nsig*nsig;
}
if(*simpleModel==1) sig2h[k*(ntime1) + t] = 1.0;
}

/////////////////////////////
// update theta (mean of mh) //
/////////////////////////////
summu = 0.0;
for(k = 0; k < nclus_iter[t]; k++){
    summu = summu + muh[k*(ntime1) + t];
}

phi1sq = phi1_iter*phi1_iter;
lam2tmp = lam2_iter*(1.0 - phi1sq);

if(t==0){
    s2star = 1.0/((double) nclus_iter[t]/tau2_iter[t] +
1.0/lam2_iter + phi1sq/lam2tmp);
    mstar = s2star*( (1.0/tau2_iter[t])*summu +
(1.0/lam2_iter)*phi0_iter +
(1.0/lam2tmp)*phi1_iter*(theta_iter[t+1]-
phi0_iter*(1-phi1_iter)));
}
else if(t==(*ntime-1)){
    s2star = 1.0/((double) nclus_iter[t]/tau2_iter[t] +
1.0/lam2tmp);
    mstar = s2star*((1.0/tau2_iter[t])*summu +
(1.0/lam2tmp)*(phi0_iter*(1-phi1_iter) +
phi1_iter*theta_iter[t-1]));
}
else {
    s2star = 1.0/((double) nclus_iter[t]/tau2_iter[t] + (1.0 +
phi1sq)/lam2tmp);
    mstar = s2star*( (1.0/tau2_iter[t])*summu +

```

```

(1.0/lam2tmp)*(phi1_iter*(theta_iter[t-1] +
theta_iter[t+1]) +
phi0_iter*(1.0 - phi1_iter)*
(1.0 - phi1_iter)));
}

theta_iter[t] = rnorm(mstar, sqrt(s2star));
if(*simpleModel==1) theta_iter[t] = 0.0;

////////////////////////////// // ///////////////////////
// update tau2 (variance of mh) // //
// // // //

////////////////////////////// // ///////////////////////
ot = sqrt(tau2_iter[t]);
nt = rnorm(ot,csigTAU);
if(nt > 0){

lln = 0.0;
llo = 0.0;
for(k = 0; k < nclus_iter[t]; k++){
    llo = llo + dnorm(mu[k*(ntime1) + t], theta_iter[t],
ot,1);
    lln = lln + dnorm(mu[k*(ntime1) + t], theta_iter[t],
nt,1);
}

llo = llo + dunif(ot, 0.0, Atau, 1);
lln = lln + dunif(nt, 0.0, Atau, 1);

llr = lln - llo;
uu = runif(0,1);

if(Log(uu) < llr){
    tau2_iter[t] = nt*nt;
    tau2_iter[t] = 5*5;
}
if(*simpleModel==1) tau2_iter[t] = theta_tau2[1];
}

}

```

```

//                                         //
// update alpha                         //           //
//                                         //

////////////////////////////////////////////////////////////////

if(*update_alpha == 1){
    if(*time_specific_alpha != 1){
        sumg = 0;
        for(j = 0; j < *nsubject; j++){
            for(t = 1; t < *ntime; t++){
                sumg = sumg + gamma_iter[j*ntime1 + t];
            }
        }
        astar = (double) sumg + a;
        bstar = (double) ((*nsubject)*(*ntime-1) - sumg) + b;

        alpha_tmp = rbeta(astar, bstar);
        for(t=0;t<*ntime;t++){alpha_iter[t] = alpha_tmp;}
    } else {
        for(t = 0; t < *ntime; t++){
            sumg = 0;
            for(j = 0; j < *nsubject; j++){
                sumg = sumg + gamma_iter[j*ntime1 + t];
            }
            astar = (double) sumg + a;
            bstar = (double) ((*nsubject) - sumg) + b;
            alpha_iter[t] = rbeta(astar, bstar);
        }
    }
    alpha_iter[0] = 0.0;
}

////////////////////////////////////////////////////////////////
//                                         //
// update phi0                          //           //
//                                         //

```

```

    llr = lln - llo;
    if(llr > log(runif(0,1))) phi1_iter = np1;
}
}

//////////////////////////////          //
// update lam2          //
//          //

//////////////////////////////          //
// Update lambda with a MH step
phi1sq = phi1_iter*phi1_iter;
ol = sqrt(lam2_iter);
nl = rnorm(ol, csigLAM);
if(nl > 0.0){
    lln = 0.0;
    llo = 0.0;
    for(t=1; t<*ntime; t++){
        llo = llo + dnorm(theta_iter[t],
                            phi0_iter*(1-phi1_iter) +
                            phi1_iter*theta_iter[t-1], ol*sqrt(1-phi1sq),1);
        lln = lln + dnorm(theta_iter[t],
                            phi0_iter*(1-phi1_iter) +
                            phi1_iter*theta_iter[t-1], nl*sqrt(1-phi1sq),1);
    }
    llo = llo + dnorm(theta_iter[0], phi0_iter, ol, 1) + dunif(ol,
0.0, Alam, 1);
    lln = lln + dnorm(theta_iter[0], phi0_iter, nl, 1) + dunif(nl,
0.0, Alam, 1);
    llr = lln - llo;
    uu = runif(0,1);
    if(log(uu) < llr){
        lam2_iter = nl*nl;
    }
}
/*
phi1sq = phi1_iter*phi1_iter;
ssq = 0.0;
for(t=1; t<*ntime; t++){
    ssq = ssq + (theta_iter[t] - (phi0_iter*(1-phi1_iter) +
phi1_iter*theta_iter[t-1]))*
(theta_iter[t] - (phi0_iter*(1-phi1_iter) +
phi1_iter*theta_iter[t-1]));
}
ssq = 1.0/(1.0 - phi1sq)*ssq + (theta_iter[0]-phi0_iter)*
(theta_iter[0]-phi0_iter);
astar = 0.5*(*ntime) + 1;

```

```

bstar = 0.5*ssq + 1/1;
lam2_iter = 1.0/rgamma(astar, 1/bstar);
*/
//////////////////////////////          //
// predict partition for new time period          //
//          //

//////////////////////////////          //
/*          //
for(p = 0; p < *npred; p++){

    for(j=0; j<*nsubject; j++){
        nh_pred[j] = 0;
        predSi_iter[j*(*npred) + p] = 0;
    }
    if(*update_alpha == 0){
        n_red = 0;
        for(j=0;j<*nsubject;j++){
            gpred[j] = rbinom(1,*alpha);

            if(gpred[j] == 1){
                nh_pred[Si_iter[j*(ntime1)+(*ntime)-1] - 1] =
nh_pred[Si_iter[j*(ntime1)+(*ntime)-1] - 1] + 1;
                n_red = n_red + 1;
                predSi_iter[j*(*npred) + p] = Si_iter[j*(ntime1)+
(*ntime)-1];
            }
        }
    }
    if(*update_alpha == 1){
        if(*time_specific_alpha == 1){

            n_red = 0;
            for(j=0;j<*nsubject;j++){

                gpred[j] = rbinom(1,alpha_iter[1]);
                if(gpred[j] == 1){
                    nh_pred[Si_iter[j*(ntime1)+(*ntime)-1] - 1] =
nh_pred[Si_iter[j*(ntime1)+(*ntime)-1] - 1] + 1;
                    n_red = n_red + 1;
                    predSi_iter[j*(*npred) + p] = Si_iter[j*(ntime1)+
(*ntime)-1];
                }
            }
        }
    }
}

```



```

if(like0==0){
    for(j = 0; j < *nsubject; j++){
        for(t = 0; t < *ntime; t++){
            CPO[j*(*ntime)+t] = CPO[j*(*ntime)+t] +
(1/exp(like_iter[j*(*ntime)+t]));
        }
    }
}

///////////////////////////////
//                                //
// Save MCMC iterates          //
//                                //
///////////////////////////////

if((i > (*burn-1)) & ((i+1) % *thin == 0)){
    for(t = 0; t < *ntime; t++){
        alpha_out[ii*(*ntime) + t] = alpha_iter[t];
        theta[ii*(*ntime) + t] = theta_iter[t];
        tau2[ii*(*ntime) + t] = tau2_iter[t];
        for(j = 0; j < *nsubject; j++){
            sig2[(ii*(*nsubject) + j)*(*ntime) + t] = sig2h[(Si_iter[j*
(ntime1) + t]-1)*(ntime1) + t];
            mu[(ii*(*nsubject) + j)*(*ntime) + t] = muh[(Si_iter[j*
(ntime1) + t]-1)*(ntime1) + t];
            Si[(ii*(*nsubject) + j)*(*ntime) + t] = Si_iter[j*ntime1 +
t];
            gamma[(ii*(*nsubject) + j)*(*ntime) + t] =
gamma_iter[j*ntime1 + t];
            llike[(ii*(*nsubject) + j)*(*ntime) + t] = like_iter[j*
(*ntime)+t];
            fitted[(ii*(*nsubject) + j)*(*ntime) + t] = fitted_iter[j*
(*ntime)+t];
        }
    }
    for(j=0; j<*nsubject; j++){

        eta1[ii*(*nsubject) + j] = eta1_iter[j];
    }
    phi1[ii] = phi1_iter;
    phi0[ii] = phi0_iter;
    lam2[ii] = lam2_iter;
    ii = ii+1;
}
/**/
}

```

```

lpml_iter=0.0;
for(t = 0; t < *ntime; t++){
    for(j = 0; j < *nsubject; j++){

        lpml_iter = lpml_iter - Log((1/(double) nout-nout_0)*CPO[j*(*ntime)+t]);
    }
}
lpml[0] = lpml_iter;
elppdWAIC = 0.0;

for(j = 0; j < *nsubject; j++){
    for(t = 0; t < *ntime; t++){
        elppdWAIC = elppdWAIC + (2*mnllike[j*(*ntime)+t] -
Log(mnllike[j*(*ntime)+t]));
    }
}
waic[0] = -2*elppdWAIC;
PutRNGstate();

}

```