```
//
// Purpose:
//    to implement a programs concerning the regression-PPM
//
// Authors:
//    Frederico R. B. Cruz
//    Rosangela H. Loschi
//    Departamento de Estatistica
//    Universidade Federal de Minas Gerais
//    E-mail: {fcruz,loschi}@est.ufmg.br
//
// Version:
//    1.00
//
// Date:
//    Jan/2003
//
//

// inclusion of standard libraries
#include <stdio.h>
#include <math.h>

// inclusion of change point library
#include "chreg.hpp"

// main program
int main(int argc, char *argv[]) {
   // checking input
   if (argc < 2) {
      fprintf(stderr,"Usage: %s <operation> <script file name>\n", argv[0]);
      fprintf(stderr,"Operation\t");
      fprintf(stderr,"\t1 -> Loschi et al.'s\t[p=k]\n");
      exit(0);
   }
   if (argc < 3) {
      fprintf(stderr, "Usage: %s %s <script_file>\n", argv[0],argv[1]);
      exit(0);
   }
   int aux = 0;
   while ((argv[2][aux]!='\n')&&(argv[2][aux]!='\0')) aux++;
   argv[2][aux] = '\0';
   FILE *inputFile = fopen(argv[2],"r");
   if (inputFile == NULL) {
      fprintf(stderr, "%s: No such file\n", argv[2]);
      exit(0);
   }
   fclose(inputFile);
//
// create object
//
   ChangeReg myChReg;
//
// perform operation
//
   if (argv[1][0]=='1') {
      // read data
      myChReg.ReadInputScript(argv[2]);
      myChReg.LoGibbs();
   }
   return 0;
}
```

```
//
// Purpose:
//    This library implements a procedure
//    to obtain product estimates to a regression
//
// Authors:
//    Frederico R. B. Cruz
//    Rosangela H. Loschi
//    Departamento de Estatistica
//    Universidade Federal de Minas Gerais
//    E-mail: {fcruz,loschi}@est.ufmg.br
//
// Version:
//    1.00
//
// Date:
//    Jan/2003
//

#ifndef CHREG_H
#define CHREG_H
//
// inclusion of standard libraries
//
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
//
#ifndef PI
#define PI 3.14159265358979323846
#endif
//
// random number generators
//
#include "randx.c"
//
// seeds for the random number generators
//
unsigned long runifSeed1=362436069, runifSeed2=521288629;
int rnormSeed=13579;
int rbetaSeed=35791;
//
// gamma function
//
#include "gamma.cpp"

#define float double

// prior parameters of alpha
#define aPrior(i,j) (0.0)
#define tau02 (1.0)

// prior parameters of psi
#define zPrior(i,j) (0.0)
#define gamma02 (1.0)

// prior parameters of sigma^2
#define nPrior(i,j) (1.00)
#define dPrior(i,j) (4.00)

float NormEuclid(float *vec1, float *vec2, int size) {
  static int i;
```



```
  static float norm;
  norm=0.0;
  for (i=1;i<=size;i++) {
      norm+=pow((vec1[i]-vec2[i]),2);
//      fprintf(stdout,"NormEuclid: norm=%f\n",norm);
  }
  return sqrt(norm);
}

// class definition
class ChangeReg {
  FILE *fileDataX;
  FILE *fileDataY;
  FILE *fileBurnIn;
  FILE *fileAlpPsiSig;
  FILE *fileNumBlo;
  FILE *filePartic;
  // maximum number of partition to be printed
  int numPartic;
  // time series and their sizes
  float *X;
  int tamX;
  float *Y;
  int tamY;
  // probability p
  float pProb;
  // posterior parameters of alpha
  float **aPost, **dPost, **vPostAlp, **nPostAlp;
  // posterior parameters of psi
  float **zPost, **vPostPsi, **nPostPsi;
  // posterior parameters of sigma^2
  float **nPostSig;
  // predictive distribution
  float **fYijXij, **vPostY, **nPostY;
  // posterior cohesions
  float **cPost;
  // vector of u samples
  int  **vecU;
  // size of each sample u
  int tamU;
  // number of samples u
  int numAmost;
  // burn-in period
  int burnIn;
  // lag to be taken into account
  int lag;
  // "net" number of samples u which is int((numAmost-burnIn)/lag)
  int numAmostU;
  // number of occurences of the i-th partition
  int *partic;
  // number of blocks of the i-th sample
  int *numBlocks;
  // number of occurences of this block
  int *blocks;
  // posterior relevance of the block [ij]
  float **relevPost;
  // posterior estimates of alpha, psi, and sigma^2
  float *alpPost, *psiPost, *sigma2Post;
  // private methods
  int ComputeParamPost(void);
  int ComputeCohesPost(void);
  int ComputeVecU(void);
```

```
  int ComputeNuBloc(void);
  int PrintNumBloc(void);
  int ComputeRelevPost(void);
  int ComputeAlpPsiSigPost(void);
  int PrintAlpPsiSigPost(void);
  int ComputeRhoPost(void);
  int PrintRhoPost(void);
  int ComputePbPost(void);
  int PrintPbPost(void);
public:
  ChangeReg(void);    // default constructor
  ~ChangeReg(void);   // destructor
  int ReadInputScript(char *scriptFileName);
  int LoGibbs(void);
  int PrintResults();
};

// class implementation

ChangeReg::ChangeReg(void) {
    fprintf(stdout, "Object ChangeReg created()\n");
    numPartic = 10;
}

ChangeReg::~ChangeReg(void) {
    fprintf(stdout, "Object ChangeReg destroyed()\n");
}

int ChangeReg::ReadInputScript(char *scriptFileName) {
    fprintf(stdout,"ChangeReg::ReadInputScript(char)\n");

    const int LLENGTH = 256;
    char Line[LLENGTH];
    int i;
    FILE *scriptFile;
    char fileName[LLENGTH];
    float dummy;

//    open script file
    fprintf(stdout, "Script file name\n\t%s\n", scriptFileName);
    scriptFile = fopen(scriptFileName, "r");
//    read script
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    fscanf(scriptFile, "%d\n", &numAmost);
    fprintf(stdout, "numAmost\n\t%d\n", numAmost);
    fgets(Line, LLENGTH, scriptFile);
    fscanf(scriptFile, "%d\n", &burnIn);
    fprintf(stdout, "burnIn\n\t%d\n", burnIn);
    fgets(Line, LLENGTH, scriptFile);
    fscanf(scriptFile, "%d\n", &lag);
    fprintf(stdout, "Lag\n\t%d\n", lag);
    fgets(Line, LLENGTH, scriptFile);
    fscanf(scriptFile, "%lf\n", &pProb);
    fprintf(stdout, "probability p\n\t%f\n", pProb);
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    i = 0;
    while ((fileName[i]=Line[i]) != '\n' ) i++;
    fileName[i] = '\0';
    fprintf(stdout, "File name for data X\n\t%s\n",fileName);
```



```
    fileDataX = fopen(fileName, "r");
    fgets(Line, LLENGTH, scriptFile);
    i = 0;
    while ((fileName[i]=Line[i]) != '\n' ) i++;
    fileName[i] = '\0';
    fprintf(stdout, "File name for data Y\n\t%s\n",fileName);
    fileDataY = fopen(fileName, "r");
//
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    i = 0;
    while ((fileName[i]=Line[i]) != '\n' ) i++;
    fileName[i] = '\0';
    fprintf(stdout, "File name for number of blocks\n\t%s\n",fileName);
    fileBurnIn = fopen(fileName, "w");
//
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    i = 0;
    while ((fileName[i]=Line[i]) != '\n' ) i++;
    fileName[i] = '\0';
    fprintf(stdout, "File name for posterior estimates\n\t%s\n",fileName);
    fileAlpPsiSig = fopen(fileName, "w");
//
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    i = 0;
    while ((fileName[i]=Line[i]) != '\n' ) i++;
    fileName[i] = '\0';
    fprintf(stdout, "File name for posterior distribution of number of blocks\n\t%s\n",fileName);
    fileNumBlo = fopen(fileName, "w");
//
    fgets(Line, LLENGTH, scriptFile);
    fgets(Line, LLENGTH, scriptFile);
    i = 0;
    while ((fileName[i]=Line[i]) != '\n' ) i++;
    fileName[i] = '\0';
    fprintf(stdout, "File name for the most probable partitions\n\t%s\n",fileName);
    filePartic = fopen(fileName, "w");
    fclose(scriptFile);
//  count time series X size
    tamX = 0;
    while (fscanf(fileDataX,"%lf\n", &dummy) == 1) {
          tamX++;
//          fprintf(stdout, "dummy = %lf\t tamX = %d\n", dummy, tamX);
    }
    // read time series
    rewind(fileDataX);
    fprintf(stdout, "Time series X size\n\t%d\n", tamX);
    X = new float[tamX+1];
    tamU = tamX-1;
    for (i=1; i <= tamX; i++) {
        fscanf(fileDataX, "%lf\n", &X[i]);
//        fprintf(stdout, "X%d = %lf\n", i, X[i]);
    }
//  count time series Y size
    tamY = 0;
    while (fscanf(fileDataY,"%lf\n", &dummy) == 1) {
          tamY++;
//          fprintf(stdout, "dummy = %lf\t tamY = %d\n", dummy, tamY);
    }
    // read time series
```

```
    rewind(fileDataY);
    fprintf(stdout, "Time series Y size\n\t%d\n", tamY);
    Y = new float[tamY+1];
    tamU = tamY-1;
    for (i=1; i <= tamY; i++) {
        fscanf(fileDataY, "%lf\n", &Y[i]);
//        fprintf(stdout, "Y%d = %lf\n", i, Y[i]);
    }
    return 0;
}

int ChangeReg::ComputeParamPost(void) {
    fprintf(stdout,"ChangeReg::ComputeParamPost()\n");

    int i, j, k;
    float sumX, sumY, sumXY, sumXX, sumYY, W;
    aPost = new float*[tamX];
    dPost = new float*[tamX];
    vPostAlp = new float*[tamX];
    nPostAlp = new float*[tamX];
    zPost = new float*[tamX];
    vPostPsi = new float*[tamX];
    nPostPsi = new float*[tamX];
    nPostSig = new float*[tamX];
    fYijXij = new float*[tamX];
    vPostY = new float*[tamX];
    nPostY = new float*[tamX];

    for (i=0; i<tamX; i++) {
      // create matrixes
      aPost[i] = new float[tamX+1];
      dPost[i] = new float[tamX+1];
      vPostAlp[i] = new float[tamX+1];
      nPostAlp[i] = new float[tamX+1];
      zPost[i] = new float[tamX+1];
      vPostPsi[i] = new float[tamX+1];
      nPostPsi[i] = new float[tamX+1];
      nPostSig[i] = new float[tamX+1];
      fYijXij[i] = new float[tamX+1];
      vPostY[i] = new float[tamX+1];
      nPostY[i] = new float[tamX+1];
      for (j=i+1; j<=tamX; j++) {
          // compute average within block [ij]
          sumX = 0.0; sumY = 0.0; sumXY = 0.0; sumXX = 0.0; sumYY = 0.0;
          for (k=i+1; k<=j; k++) {
              sumX += X[k];
              sumY += Y[k];
              sumXY += X[k]*Y[k];
              sumXX += X[k]*X[k];
              sumYY += Y[k]*Y[k];
          }
          // compute posterior parameters
          aPost[i][j] = (
                         (gamma02*sumXX+1)*(aPrior(i,j)+tau02*sumY)-
                         tau02*sumX*(zPrior(i,j)+gamma02*sumXY)
                        )/(
                          gamma02*sumXX+1+(j-i)*tau02*(gamma02*sumXX+1)-
                          tau02*gamma02*sumX*sumX
                        );
          dPost[i][j] = dPrior(i,j) + (j-i);
          vPostAlp[i][j] = (
                             tau02*(1+gamma02*sumXX)
```



```
                           )/(
                             (1+tau02*(j-i))*(gamma02*sumXX+1)-
                             gamma02*tau02*sumX*sumX
                           );
          nPostAlp[i][j] = nPrior(i,j) + (
                            (gamma02*sumXX+1)*(
                              aPrior(i,j)*aPrior(i,j)*gamma02 +
                              tau02*gamma02*sumYY +
                              zPrior(i,j)*zPrior(i,j)*tau02
                            )
                           )/(
                             tau02*gamma02*(gamma02*sumXX+1)) - (
                                gamma02*sumXY*sumXY +
                                2*zPrior(i,j)*sumXY +
                                zPrior(i,j)*zPrior(i,j)
                             )/(gamma02*sumXX+1) -
                             pow((
                               (gamma02*sumXX+1)*(aPrior(i,j)+tau02*sumY) -
                               tau02*sumX*(gamma02*sumXY+zPrior(i,j)
                             )),2)/(
                               ((gamma02*sumXX+1)*(1+tau02*(j-i)) -
                                 tau02*gamma02*sumX*sumX
                               )*
                               (tau02*(gamma02*sumXX+1))
                             );
          zPost[i][j] = ((
                           1+tau02*(j-i))*(gamma02*sumXY+zPrior(i,j)) -
                           gamma02*sumX*(tau02*sumY+aPrior(i,j))
                         )/(
                           (1+tau02*(j-i))*(gamma02*sumXX+1) -
                           gamma02*tau02*sumX*sumX
                         );
          vPostPsi[i][j] = (
                             gamma02*(1+tau02*(j-i))
                            )/(
                             (1+tau02*(j-i))*(gamma02*sumXX+1) -
                             gamma02*tau02*sumX*sumX
                            );
          W = (
               (1+tau02*(j-i))*(gamma02*sumXX+1)-gamma02*tau02*sumX*sumX
              )/(
               gamma02*(1+tau02*(j-i))
              );
          nPostPsi[i][j] = nPrior(i,j) +
                           W*(
                             (1+tau02*(j-i))*(tau02*gamma02*sumYY +
                              aPrior(i,j)*aPrior(i,j)*gamma02 +
                              zPrior(i,j)*zPrior(i,j)*tau02
                             ) - (
                              gamma02*pow((tau02*sumY+aPrior(i,j)),2)
                             )
                           )/(
                             tau02*(1+tau02*(j-i))*(gamma02*sumXX+1)-
                             gamma02*tau02*sumX*sumX
                           ) -
                           W*pow((
                              (
                                (1+tau02*(j-i))*(gamma02*sumXY+zPrior(i,j)) -
                                gamma02*sumX*(tau02*sumY+aPrior(i,j))
                              )/(
                                (1+tau02*(j-i))*(gamma02*sumXX+1)-
                                gamma02*tau02*sumX*sumX
```

```
                          )
                        ),2);
        nPostSig[i][j] = nPrior(i,j) + (
                         (1+tau02*(j-i))*(tau02*zPrior(i,j)*zPrior(i,j)+
                          tau02*gamma02*sumYY+
                          gamma02*aPrior(i,j)*aPrior(i,j)
                         )/(
                          tau02*gamma02*(1+tau02*(j-i))
                        )
                        ) - (
                          tau02*tau02*gamma02*sumY*sumY+
                          gamma02*aPrior(i,j)*aPrior(i,j)+
                          2*aPrior(i,j)*tau02*gamma02*sumY
                        )/(
                          tau02*gamma02*(1+tau02*(j-i))
                        ) - (
                          pow((
                             (1+tau02*(j-i))*(zPrior(i,j)+gamma02*sumXY)-
                             gamma02*tau02*sumY*sumX-
                             gamma02*aPrior(i,j)*sumX
                          ),2)
                        )/(
                          gamma02*(1+tau02*(j-i))*((1+tau02*(j-i))*
                            (gamma02*sumXX+1)-
                            gamma02*tau02*sumX*sumX
                          )
                        );
        // compute predictive distribution
        vPostY[i][j] = (
                         tau02*(1+gamma02*sumXX)*gamma02*
                         (1+(j-i)*tau02)-
                         tau02*tau02*gamma02*gamma02*sumX*sumX
                       )/(
                         pow((
                           (1+(j-i)*tau02)*(1+gamma02*sumXX)-
                           tau02*gamma02*sumX*sumX
                         ),2)
                       );
        nPostY[i][j] = nPrior(i,j) +
                       aPrior(i,j)*aPrior(i,j)/tau02 +
                       zPrior(i,j)*zPrior(i,j)/gamma02 +
                       sumYY -
                       aPost[i][j]*aPost[i][j]*(1+(j-i)*tau02)/tau02 -
                       2*aPost[i][j]*zPost[i][j]*sumX -
                       zPost[i][j]*zPost[i][j]*(1+gamma02*sumXX)/gamma02;
        fYijXij[i][j] = (
                         sqrt(vPostY[i][j])*pow(nPrior(i,j),(dPrior(i,j)/2))*
                         Gamma(dPost[i][j]/2)*
                         pow(nPostY[i][j],(-dPost[i][j]/2))
                        )/(
                          sqrt(gamma02*tau02)*pow(PI,((j-i)/2))*
                          Gamma(dPrior(i,j)/2)
                        );
      }
    }
    // print results
//    fprintf(stdout, "aPost:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", aPost[i][j]);
```



```
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "dPost:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", dPost[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "vPostAlp:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", vPostAlp[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "nPostAlp:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", nPostAlp[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "zPost:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", zPost[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "vPostPsi:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", vPostPsi[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "nPostPsi:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", nPostPsi[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "nPostSig:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", nPostSig[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "vPosY:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
```

```
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", vPostY[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "nPostY:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", nPostY[i][j]);
//        fprintf(stdout, "\n");
//    }
//    fprintf(stdout, "fYijXij:\n");
//    for (i=0; i<tamX; i++) {
//       for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", fYijXij[i][j]);
//        fprintf(stdout, "\n");
//    }
    return 0;
};

int ChangeReg::ComputeCohesPost(void) {
    fprintf(stdout,"ChangeReg::ComputeCohesPost\n");
    int i, j;
    cPost = new float*[tamX];
    for (i=0; i<tamX; i++) {
      // create matrixes
      cPost[i] = new float[tamX+1];
      for (j=i+1; j<tamX; j++) {
          // compute cohesion of block [ij]
          cPost[i][j] = log(pProb) + (j-i-1)*log(1-pProb) + log(fYijXij[i][j]);
          cPost[i][j] = exp(cPost[i][j]);
      }
      cPost[i][tamX] = (tamX-i-1)*log(1-pProb) + log(fYijXij[i][tamX]);
      cPost[i][tamX] = exp(cPost[i][tamX]);
    }
    // print results
//    fprintf(stdout, "cPost:\n");
//    for (i=0; i<tamX; i++) {
//        for (j=1; j<=i; j++)
//            fprintf(stdout, "         ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", cPost[i][j]);
//        fprintf(stdout, "\n");
//    }
    return 0;
};

int ChangeReg::ComputeVecU(void) {
    fprintf(stdout, "ChangeReg::ComputeVecU\n");
    int k, i, l, x, y;
    float Ri, u;

    vecU = new int*[numAmost+1];
    for (k=0; k<=numAmost; k++) {
        vecU[k] = new int[tamU+1];
    }
    // create seed
    for (i=1; i<=tamU; i++) {
        vecU[0][i] = 0;
```



```
    }
//    fprintf(stdout, "vecU[%d]:\n", 0);
//    for (i=1; i<=tamU; i++) {
//        fprintf(stdout, "%d ", vecU[0][i]);
//    }
//    fprintf(stdout, "\n");
    // create kth sample
    for (k=1; k<=numAmost; k++) {
        for (i=1; i<=tamU; i++) {
            vecU[k][i] = vecU[k-1][i];
        }
        for (i=1; i<=tamU; i++) {
            // find x
            x = 0;
            for (l=1; l<i; l++) {
                if ( vecU[k][l] == 0 ) x = l;
            }
            // find y
            y = tamX;
            for (l=tamU; l>i; l--) {
                if ( vecU[k-1][l] == 0 ) y = l;
            }
            // compute Ri
            Ri = log(cPost[x][y]) - log(cPost[x][i]) - log(cPost[i][y]);
            Ri = exp(Ri);
            // generate Uki
//            u = UNI;
            u = rUnif2(&runifSeed1,&runifSeed2);
            if ( Ri >= (1-u)/u ) {
               vecU[k][i] = 1;
            } else {
               vecU[k][i] = 0;
            }
        }
//        fprintf(stdout, "vecU[%d]:\n", k);
//        for (i=1; i<=tamU; i++) {
//            fprintf(stdout, "%d ", vecU[k][i]);
//        }
//        fprintf(stdout, "\n");
    }
    return 0;
}

int ChangeReg::ComputeNuBloc(void) {
   fprintf(stdout, "ChangeReg::ComputeNuBloc\n");
    int k,i;
    numBlocks = new int[numAmost+1];
    for (k=1; k<=numAmost; k++) {
        numBlocks[k] = 1;
        for (i=1; i<=tamU; i++) {
            if (vecU[k][i] == 0) {
               numBlocks[k]++;
            }
        }
    }
   return 0;
}

int ChangeReg::PrintNumBloc() {
    fprintf(stdout, "ChangeReg::PrintNumBloc()\n");
    for (int k=1; k<=numAmost; k++) {
        fprintf(fileBurnIn,"%d\n", numBlocks[k]);
```

```
    }
    return 0;
}

int ChangeReg::ComputeRelevPost(void) {
    fprintf(stdout, "ChangeReg::ComputeRelevPost()\n");
    int i, j, inicBloc, finalBloc;
    relevPost = new float*[tamX];
    // creat matrix
    for (i=0; i<tamX; i++) {
        relevPost[i] = new float[tamX+1];
        for (j=i+1; j<=tamX; j++)
            relevPost[i][j] = 0;
    }
    // make counts
    numAmostU = 0;
    for (i=burnIn+1; i<=numAmost; i+=lag) {
        numAmostU++;
        inicBloc = 0;
        for (j=1; j<tamX; j++) {
            if (vecU[i][j] == 0) {
               finalBloc = j;
               relevPost[inicBloc][finalBloc]++;
               inicBloc = j;
            }
        }
        relevPost[inicBloc][tamX]++;
    }
    fprintf(stdout, "Number of net samples: %d\n",numAmostU);
    // print results
//    for (i=0; i<tamX; i++) {
//        for (j=1; j<=i; j++)
//            fprintf(stdout, "  ");
//        for (j=i+1; j<=tamX; j++)
//            fprintf(stdout, "%f ", relevPost[i][j]);
//        fprintf(stdout, "\n");
//    }
    return 0;
}

int ChangeReg::ComputeAlpPsiSigPost(void) {
    fprintf(stdout, "ChangeReg::ComputeAlpPsiSigPost()\n");
    int k, i, j;
    alpPost = new float[tamX+1];
    psiPost = new float[tamX+1];
    sigma2Post = new float[tamX+1];
    for (k=1; k<=tamX; k++) {
        alpPost[k] = 0.0;
        psiPost[k] = 0.0;
        sigma2Post[k] = 0.0;
        for (i=0; i<=k-1; i++) {
            for (j=k; j<=tamX; j++) {
                alpPost[k] += relevPost[i][j] * aPost[i][j] / numAmostU;
                psiPost[k] += relevPost[i][j] * zPost[i][j] / numAmostU;
                sigma2Post[k] += relevPost[i][j]*nPostSig[i][j]/
                                 ((dPost[i][j] - 2)*numAmostU);
            }
        }
    }
    return 0;
}
```



```
int ChangeReg::PrintAlpPsiSigPost(void) {
    fprintf(stdout, "ChangeReg::PrintAlpPsiSigPost()\n");
    fprintf(fileAlpPsiSig, "alpPost\t psiPost\t sigma2Post\n");
    for (int i=1; i<=tamX; i++) {
        fprintf(fileAlpPsiSig, "%f\t %f\t %f\n", alpPost[i], psiPost[i], sigma2Post[
i]);
    }
    return 0;
}

int ChangeReg::ComputeRhoPost(void) {
    fprintf(stdout, "ChangeReg::ComputePostRho()\n");
    int i, j, k;
    partic = new int[numAmost+1];
    for (i=0; i<=numAmost; i++) {
        partic[i] = 1;
    }
    for (i=burnIn+1; i<=numAmost; i+=lag) {
        if ( partic[i] != -1 ) {
           for (j=i+lag; j<=numAmost; j+=lag) {
               // comparar i-esima amostra com j-esima amostra
               k = 1;
               while ( (vecU[i][k]==vecU[j][k])&&(k<tamU)) {
                   k++;
               }
               if ( vecU[i][k]==vecU[j][k] ) {
                  partic[i]++;
                  partic[j] = -1;
               }
           }
        }
    }
    return 0;
}

int ChangeReg::PrintRhoPost(void) {
    fprintf(stdout, "ChangeReg::PrintRhoPost()\n");
    int i, j, maxJ, max;
    fprintf(filePartic, "partition \t probability\t #\n");
    for (i=1; i<=numPartic; i++) {
        max = -1;
        for (j=burnIn+1; j<=numAmost; j+=lag) {
            if (partic[j]>max) {
               max = partic[j];
               maxJ = j;
            }
        }
//      imprima particoes mais provaveis (ignorando as de ocorrencia)
        if ( partic[maxJ] > 0 ) {
           for (j=1; j<=tamU; j++) {
               fprintf(filePartic, "%d ", vecU[maxJ][j]);
           }
           fprintf(filePartic, "\t %f\t %d\n",
                    float(partic[maxJ])/numAmostU, partic[maxJ] );
           partic[maxJ] = -1;
        }
    }
    return 0;
};

int ChangeReg::ComputePbPost(void) {
    fprintf(stdout, "ChangeReg::ComputePbPost()\n");
```

```
    int i, j;
    blocks = new int[numAmost+1];
    for (i=1; i<=numAmost; i++) {
        blocks[i] = 1;
    }
    for (i=burnIn+1; i<=numAmost; i+=lag) {
        if ( blocks[i] != -1 ) {
           for (j=i+lag; j<=numAmost; j+=lag) {
               // comparar i-esima amostra com j-esima amostra
               if ( numBlocks[i]==numBlocks[j] ) {
                  blocks[i]++;
                  blocks[j] = -1;
               }
           }
        }
    }
//    for (i=burnIn+1; i<=numAmost; i+=lag) {
//        fprintf(arqNumBlo, "%d\n", blocks[i]);
//    }
    return 0;
};

int ChangeReg::PrintPbPost(void) {
    fprintf(stdout, "ChangeReg::PrintPbPost()\n");
    int i, j, minJ, min;
    fprintf(fileNumBlo, "#blocks\t #occur\t prob\n" );
    for (i=burnIn+1; i<=numAmost; i+=lag) {
        // find minimum
        min = tamX+1;
        for (j=burnIn+1; j<=numAmost; j+=lag) {
            if ( (numBlocks[j]<min)&&(blocks[j]!=-1)) {
               min = numBlocks[j];
               minJ = j;
            }
        }
        // print it
        if ((blocks[minJ]!=-1)&&(numBlocks[minJ]!=(tamX+1))) {
           fprintf(fileNumBlo, "%d\t %d\t %f\n",
                 numBlocks[minJ], blocks[minJ], float(blocks[minJ])/numAmostU )
;
        }
        numBlocks[minJ] = tamX+1;
    }
    return 0;
}

int ChangeReg::LoGibbs(void) {
   fprintf(stdout, "ChangeReg::LoGibbs()\n");
   // compute aij*, dij*, VijAlp*, nijAlp*,
   // zij*, VijPsi*, nijPsi*, nijSig*, VijY*, nijY*, fYijXij
   ComputeParamPost();
   // compute posterior cohesions cij*
   ComputeCohesPost();
   // generate samples u
   ComputeVecU();
   // compute number of blocks for each sample
   ComputeNuBloc();
   // compute posterior relevance for each block [ij]
   ComputeRelevPost();
   // compute the product estimates of alpha, psi, sigma^2
   ComputeAlpPsiSigPost();
   // compute probability of each partition
```



```
   ComputeRhoPost();
   // compute the posterior distribution of number of blocks
   ComputePbPost();
   // print results
   PrintResults();
   return 0;
}

int ChangeReg::PrintResults(void) {
   fprintf(stdout, "ChangeReg::PrintResults()\n");
   // print results
   PrintNumBloc();
   PrintAlpPsiSigPost();
   PrintRhoPost();
   PrintPbPost();
   return 0;
}

#endif
```

```
/*****************************************
*           FUNCTION  GAMMA(X)            *
* ------------------------------------- *
* Returns the value of Gamma(x) in double *
* precision as EXP(LN(GAMMA(X))) for X>0. *
*****************************************/
#ifndef GAMMA_CPP
#define GAMMA_CPP
#include <stdio.h>
#include <math.h>

#define  half 0.5
#define  one  1.0
#define  fpf  5.5
#define  zero 0.0

double Gamma(double xx)  {
  double cof[7],stp,x,tmp,ser;
  int j;
  cof[1]= 76.18009173;
  cof[2]=-86.50532033;
  cof[3]= 24.01409822;
  cof[4]= -1.231739516;
  cof[5]=  0.120858003e-2;
  cof[6]= -0.536382e-5;
  stp=2.50662827465;

  x=xx-one;
  tmp=x+fpf;
  tmp=(x+half)*log(tmp)-tmp;
  ser=one;
  for (j=1; j<7; j++) {
   x=x+one;
        ser=ser+cof[j]/x;
  }
  return (exp(tmp+log(stp*ser)));
}

/*****************************************
* Program to demonstrate the Gamma Function *
*                                           *
*               C++ version by J-P Moreau   *
* ----------------------------------------- *
* Reference:                                *
* "Numerical Recipes, by W.H. Press, B.P.   *
*  Flannery, S.A. Teukolsky and T. Vetter-  *
*  ling, Cambridge University Press, 1986"  *
* ----------------------------------------- *
* SAMPLE RUN:                               *
*                                           *
*       X        Gamma(X)                   *
*  ------------------------                 *
*    0.500000    1.772454                   *
*    1.000000    1.000000                   *
*    1.500000    0.886227                   *
*    2.000000    1.000000                   *
*    2.500000    1.329340                   *
*    3.000000    2.000000                   *
*    3.500000    3.323351                   *
*    4.000000    6.000000                   *
*    4.500000    11.631728                  *
*    5.000000    24.000000                  *
```



```
*                                           *
*****************************************/

/*****************************************
void main()  {
  double x, y;
  int i;
  printf("\n      X        Gamma(X)   \n");
  printf(" ------------------------  \n");
  x=zero;
  for (i=1; i<11; i++) {
    x += half;
    y = Gamma(x);
    printf("   %f    %f\n", x, y);
  }
  printf("\n");
}
*****************************************/
/* end of file gamma.cpp */
#endif
```

```
/*********************************************************************
*                                                                    *
* Purpose:                                                           *
*     randX is a library for random number generation.               *
*     The integer randXseed should be initialized to an arbitrary    *
* integer prior to the first call to the desired function. The calling *
* program should not alter the value of randXseed between subsequent  *
* calls.                                                             *
*                                                                    *
* Author:                                                            *
*    Frederico R. B. Cruz                                            *
*    Departamento de Estatistica                                     *
*    Universidade Federal de Minas Gerais                            *
*    Brazil                                                          *
*    E-mail: fcruz@est.ufmg.br                                       *
*                                                                    *
* Version:                                                           *
*    1.00                                                            *
*                                                                    *
* Date:                                                              *
*    March/2002                                                      *
*                                                                    *
*********************************************************************/

#ifndef RANDX_C
#define RANDX_C
#include <math.h>
#define RANDX_PI 3.14159265358979323846
#define RANDX_EE 2.71828182845904523536

/* uniform [0,1]                                                    */
float  rUnif(int *randXseed);
float  rUnif2(unsigned long *randXseed1, unsigned long *randXseed2);

/* normal(mu=0,sd=1)                                                */
float  rNorm(int *randXseed);

/* exponetial(lambda)                                               */
float  rExpo(int *randXseed, float lambda);

/* gamma(alpha)                                                     */
double rGamma(int *randXseed, double alpha);

/* beta(alpha,beta)                                                 */
double rBeta(int *randXseed, double alpha, double beta);


/*********************************************************************
*                                                                    *
*       The rUnif function is a uniform random number generator based *
*  on theory and suggestions given in Forsythe, G.E., Malcolm, M.A., & *
*  Moter, C.B. Computer Methods For Mathematical Computations,       *
*  Prentice-Hall, 1977.                                              *
*       The integer randXseed should be initialized to an arbitrary  *
* integer prior to the first call to rUnif. The calling program should *
* not alter the value of randXseed between subsequent calls to rUnif. *
* Values of rUnif will be returned in the interval (0,1).            *
*                                                                    *
*********************************************************************/

float rUnif(int *randXseed) {
    /* initialized data */
```



```
    static int m2 = 0;
    static int two = 2;
    /* local variables */
    static int m;
    static float s;
    static float halfm;
    static int a, c, mc;
    /* if first entry then ... */
    if (m2 == 0) {
        /* compute machine integer word length */
        m = 1;
        do {
            m2 = m;
            m = two * m2;
        } while( m > m2 );
        halfm = (float) m2;
        /* compute multiplier and increment for linear  */
        /* congruential method                          */
        a = ( (int) ( halfm * atan(1.) / 8.) << 3 ) + 5;
        c = ( (int) ( halfm * ( 0.5 - sqrt(3.) / 6.)) << 1 ) + 1;
        mc = m2 - c + m2;
        /*   s is the scale factor for converting to floating point */
        s = 0.5 / halfm;
    }
    /* compute next random number */
    *randXseed *= a;
    /*  the following statement is for computers which do not */
    /*  allow integer overflow on addition                    */
    if (*randXseed > mc)
       *randXseed = *randXseed - m2 - m2;
    *randXseed += c;
    /* the following statement is for computers where the word */
    /* length for addition is greater than for multiplication  */
    if (*randXseed / 2 > m2)
       *randXseed = *randXseed - m2 - m2;
    /* the following statement is for computers where integer    */
    /* overflow affects the sign bit                             */
    if (*randXseed < 0)
       *randXseed = *randXseed + m2 + m2;
    return ( (float) (*randXseed) * s );
}

/*********************************************************************
*                                                                    *
*       George Marsaglia's uniform random number generator           *
*       The integers randXseed1 and randXseed2 should be initialized  *
* to an arbitrary integer prior to the first call to rUnif2. The     *
* calling program should not alter these values between subsequent   *
* calls to rUnif.                                                    *
*                                                                    *
*********************************************************************/

float  rUnif2(unsigned long *randXseed1, unsigned long *randXseed2) {
  static unsigned long s1new, s2new;
  s1new = ((*randXseed1=36969*(*randXseed1&65535)+(*randXseed1>>16))<<16);
  s2new = ((*randXseed2=18000*(*randXseed2&65535)+(*randXseed2>>16))&65535);
  return ((s1new+s2new)*2.32830643708E-10);
}

/*********************************************************************
#include <stdio.h>
#include "uni.c"
```

```
int main() {
  int replic = 1000;
  int randXseed = 123456;
  unsigned long randXseed1=362436069, randXseed2=521288629;
  int i;
  for (i=0; i<replic; i++){
      fprintf(stdout, "%20.18f\t%20.18f\t%20.18f\n",
              rUnif(&randXseed), rUnif2(&randXseed1,&randXseed2), UNI);
  }
  return 0;
}
*************************************************************************/


/*************************************************************************
*                                                                        *
*        The rNorm function is a normal random number generator.         *
*        The integer randXseed should be initialized to an arbitrary     *
* integer prior to the first call to rNorm. The calling program should   *
* not alter the value of randXseed between subsequent calls to rNorm.    *
* Values of rNorm will be returned following N(0,1).                     *
*                                                                        *
*************************************************************************/

float rNorm(int *randXseed) {
  static float y1, y2;
  while(1) {
      y1=-log(rUnif(randXseed));
      y2=-log(rUnif(randXseed));
      if (y2-(y1-1)*(y1-1)/2 >= 0) {
         if (rUnif(randXseed) > 0.5)
            return -y1;
         else
            return y1;
      }
  }
}

/*************************************************************************
#include <stdio.h>
int main() {
 int replic = 100000;
 float mu = 0;
 float sigma = 1;
 int seed = 123456;
 float numb;
 float sum, sum2;
 int i;
 sum = 0.0;
 sum2 = 0.0;
 for (i=0; i<replic; i++){
   numb = mu + rNorm(&seed)*sigma;
   sum += numb;
   sum2 += (numb*numb);
 }
 printf("X ~ N(%f,%f) had E(x)=%f and Var(x)=%f over %d replics.\n",
        mu, sigma*sigma, sum/replic, (sum2-(sum*sum)/replic)/replic, replic);
 return 0;
}
*************************************************************************/
```



```
/*************************************************************************
*                                                                        *
*        The rExpo function is an exponential random number generator.   *
*        The integer randXseed should be initialized to an arbitrary     *
* integer prior to the first call to erand. The calling program should   *
* not alter the value of randXseed between subsequent calls to erand.    *
* Values of erand will be returned following E(lambda).                  *
*                                                                        *
*************************************************************************/

float rExpo(int *randXseed, float lambda) {
  return (-log(rUnif(randXseed))/lambda);
}

/*************************************************************************
#include <stdio.h>
int main() {
 int replic = 10000;
 float lambda = 2;
 int seed = 123456;
 float numb;
 float sum = 0.0;
 float sum2 = 0.0;
 int i;
 int cont;
 for (i=0; i<replic; i++){
   numb  = rExpo(&seed,lambda);
   sum += numb;
   sum2 += (numb*numb);
 }
 printf("X ~ E(%f) had E(x)=%f^-1 and Var(x)=%f^-2 over %d replics.\n",
        lambda, 1/(sum/replic), 1/sqrt((sum2-(sum*sum)/replic)/replic), replic);
 sum = 0.0;
 cont = 0;
 while (sum <= replic){
   numb  = rExpo(&seed,lambda);
   sum += numb;
   cont++;
 }
 printf("There were %d events over %d time units.\n", cont, replic);
 return 0;
}
*************************************************************************/

/*************************************************************************
*                                                                        *
*        The rGamma function is a gamma random number generator.         *
*        The integer randXseed should be initialized to an arbitrary     *
* integer prior to the first call to erand. The calling program should   *
* not alter the value of randXseed between subsequent calls to erand.    *
* Values of erand will be returned following G(alpha).                   *
*                                                                        *
*************************************************************************/

double rGamma(int *randXseed, double alpha) {
   static double r1,r2,aa,x,w,c1,c2,c3,c4,c5;
   if (alpha<=0.)
      return 0.;
   if (alpha == 1.)
      return rExpo(randXseed,1.);
   if (alpha<1) {
      aa=(alpha+RANDX_EE)/RANDX_EE;
```

```
      do {
         r1=rUnif(randXseed);
         r2=rUnif(randXseed);
         if(r1>1./aa) {
            x = -log(aa*(1.-r1)/alpha);
            if (r2<pow(x,(alpha-1.)))
               return x;
         }
         else {
            x = pow((aa*r1),(1./alpha));
            if (r2<exp(-x))
               return x;
         }
      } while(1);
   }
   else {
      c1=alpha-1;
      c2=(alpha-1./(6.*alpha))/c1;
      c3=2./c1;
      c4=c3+2.;
      c5=1./sqrt(alpha);
      do {
         do {
            r1=rUnif(randXseed);
            r2=rUnif(randXseed);
            if (alpha>2.5)
               r1=r2+c5*(1.-1.86*r1);
         } while(r1<=0 || r1 >= 1);
         w=c2*r2/r1;
         if (c3*r1+w+1/w <= c4)
            return c1*w;
         if (c3*log(r1)-log(w)+w<1)
            return c1*w;
      } while (1);
   }
}

/*************************************************************************
#include <stdio.h>
int main() {
 int replic = 10000;
 int seed = 123456;
 int i;
 double alpha = 1.5;
 for (i=0; i<replic; i++){
   fprintf(stdout, "%f\n", rGamma(&seed,alpha));
 }
 return 0;
}
*************************************************************************/

/*************************************************************************
*                                                                        *
*        The rBeta function is a beta random number generator.           *
*        The integer randXseed should be initialized to an arbitrary     *
* integer prior to the first call to erand. The calling program should   *
* not alter the value of randXseed between subsequent calls to erand.    *
* Values of erand will be returned following B(alpha,beta).              *
*                                                                        *
*************************************************************************/
double rBeta(int *randXseed, double alpha, double beta) {
   double r1;
```



```
   if (alpha <=0. || beta <= 0.)
      return 0.;
   r1=rGamma(randXseed,alpha);
   return r1/(r1+rGamma(randXseed,beta));
}

/*************************************************************************
#include <stdio.h>
int main() {
 int replic = 10000;
 int seed = 123456;
 int i;
 double alpha = 2.0;
 double beta = 3.0;
 for (i=0; i<replic; i++){
   fprintf(stdout, "%f\n", rBeta(&seed,alpha,beta));
 }
 return 0;
}
*************************************************************************/

#endif
```