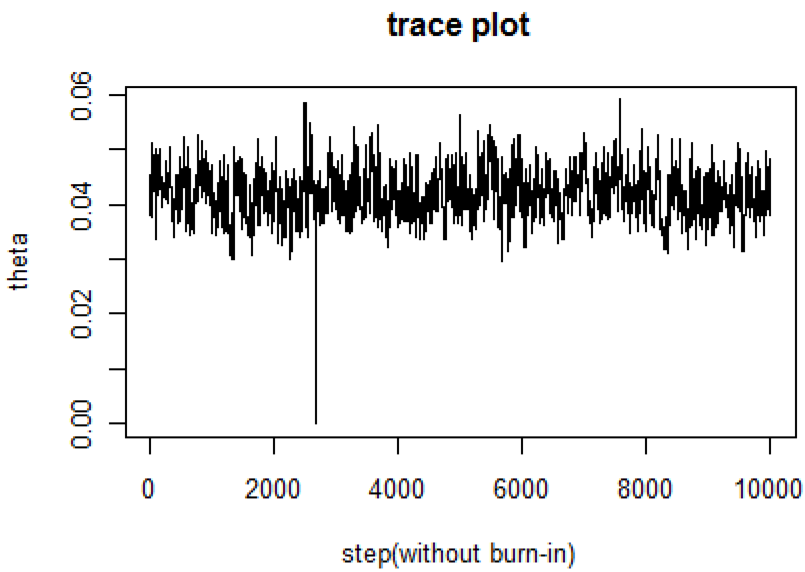
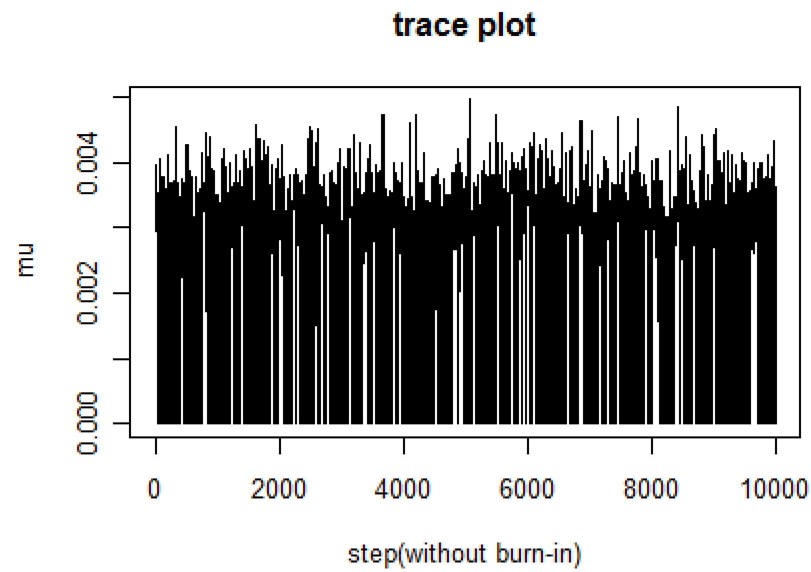
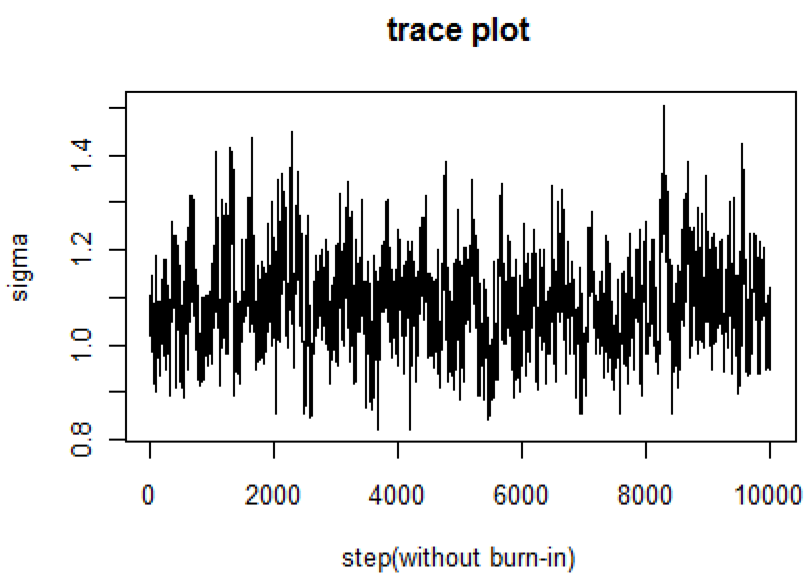
Appendix 1

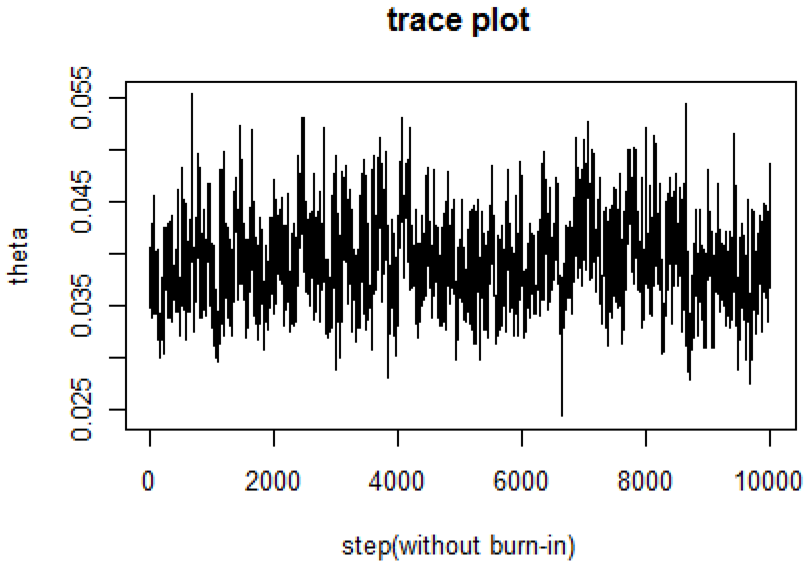
Trace Plots for Theta, Mu, Sigma for Metropolis within Gibbs applied to Discrete Time Model

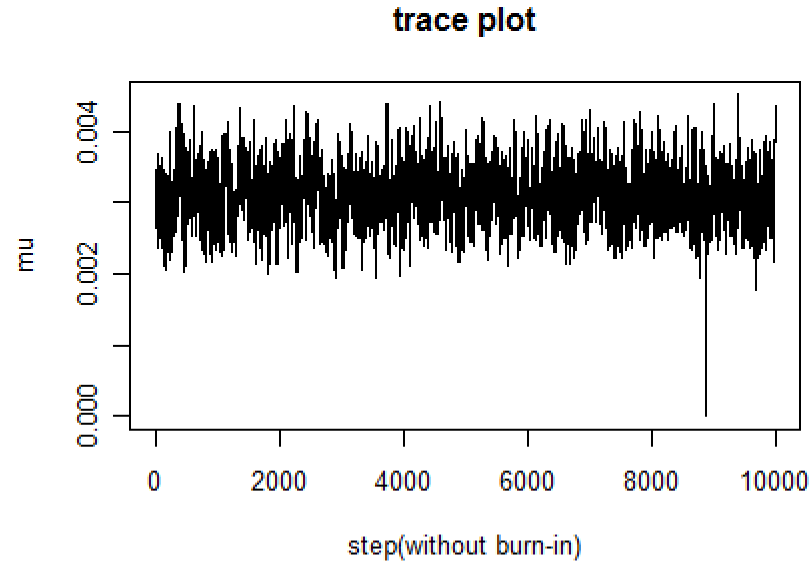


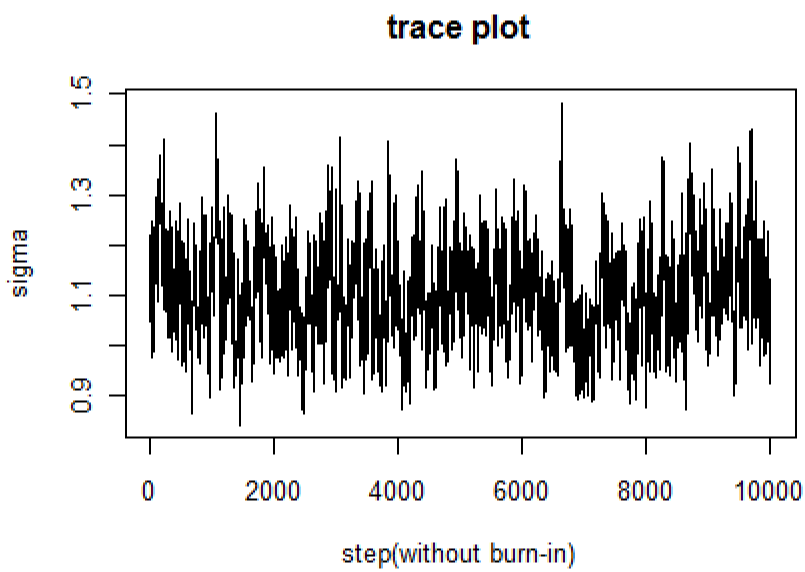




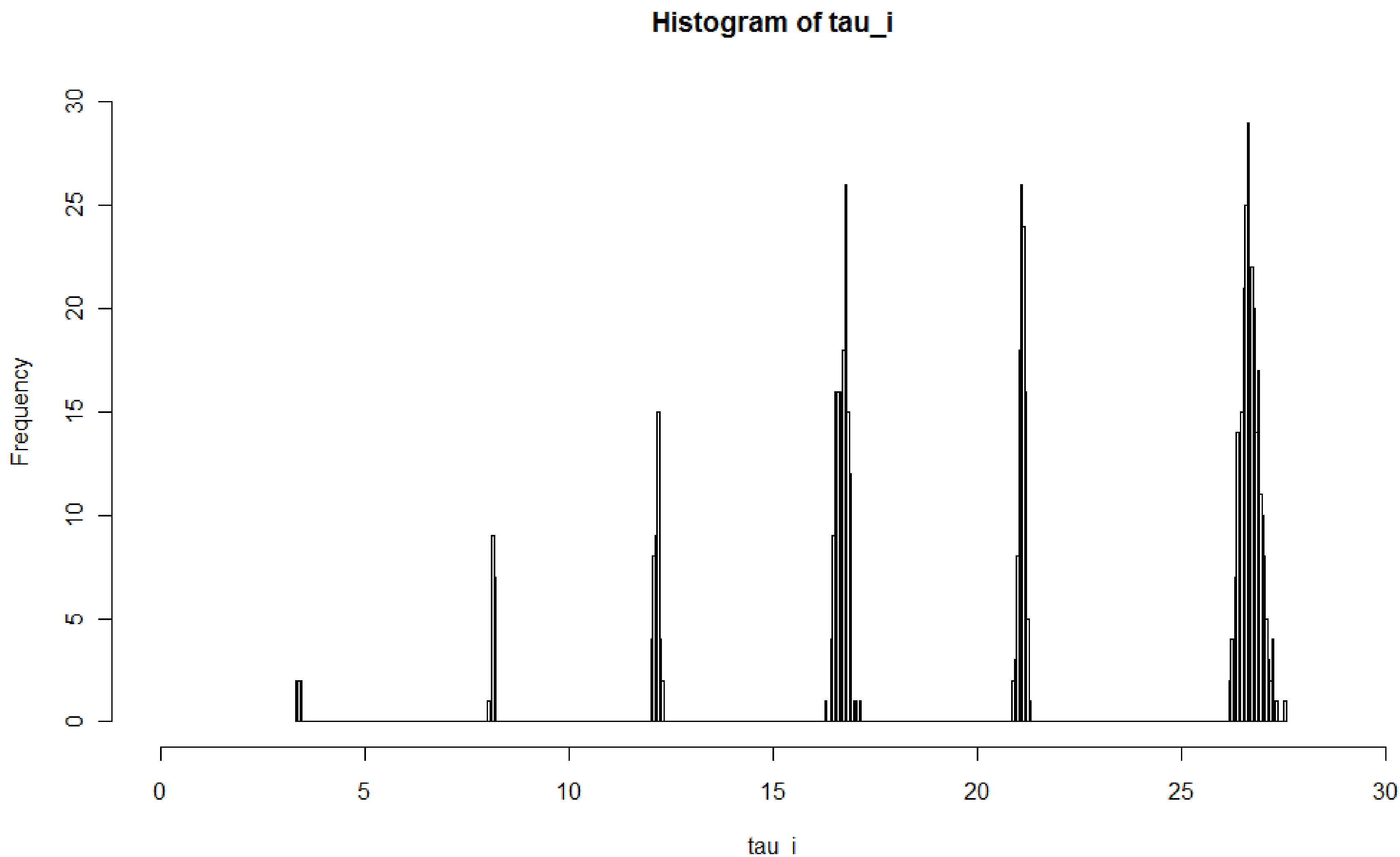
Trace Plots for Theta, Mu, Sigma for Metropolis within Gibbs applied to Continuous Time Model







Histogram of Tau_i Estimates for Metropolis within Gibbs applied to Continuous Time Model



Appendix 2

C Program for Metropolis within Gibbs for Discrete Time Model, Original Version

```
/* Metropolis-within-Gibbs algorithm for Guadeloupe sugar cane data */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>

#define MAXN 1742
#define K 30
#define infinity 999999999.0
#define PI 3.1415926536

double drand48();
int L[MAXN], U[MAXN];
double lambda[MAXN][K], D[MAXN][MAXN], sq();
int N;

/* BEGIN MAIN PROGRAM. */

int main(int argc, char **argv)
{

/* Initial declarations. */
FILE *fp;
int M, B, i, j, k, t;
char tmpstring[20];
double x[MAXN], y[MAXN];
int S6[MAXN], S10[MAXN], S14[MAXN], S19[MAXN], S23[MAXN], S30[MAXN];
double uniform(), exponential(), normal(), logpi();
void seedrand();
double curtheta, curmu, curlogpi, newlogpi, newmu, newtheta, summu, sumtheta;
int curtau[MAXN], newtau[MAXN], sumtau[MAXN];
int propmu, propth, proptau, accmu, accth, acctau;

/* Seed the random number generator. */
seedrand();

/* Read in the data. */
printf("Reading data ...\n");
if ((fp = fopen("canedata","r")) == NULL) {
```

```
    fprintf(stderr, "Unable to read file 'canedata'.\n");
    exit(1);
}
N = 0;
for (i=0; i<MAXN; i++) {
    fscanf(fp, "%s", &tmpstring);
    x[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    y[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S6[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S10[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S14[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S19[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S23[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S30[N] = atoi(tmpstring);
    /* Do "cheat" of restricting to 10 x 10 grid. */
    if ( (x[N] < 10.0) && (y[N] < 10.0) ) {
      /* printf("Keeping site %d.\n", N); */
      N++;
    }
}
fclose(fp);
printf("Number of sites studied: %d\n", N);

/* OUTPUT SOME TEST VALUES -- NO.
printf("%f  %f  %d  %d  %d\n", x[2], y[2], S6[2], S10[2], S14[2]);
for (i=0; i<N; i++)
  printf("i=%d: x=%f, y=%f\n", i, x[i], y[i]);
*/

/* Determine the L_x and U_x values, etc. */
printf("Computing infection time ranges ...\n");
for (i=0; i<N; i++) {
  if (S6[i]) {
    L[i] = 0;
    U[i] = 6;
  } else if (S10[i]) {
    L[i] = 6;
    U[i] = 10;
  } else if (S14[i]) {
    L[i] = 10;
    U[i] = 14;
  } else if (S19[i]) {
```

```
    L[i] = 14;
    U[i] = 19;
  } else if (S23[i]) {
    L[i] = 19;
    U[i] = 23;
  } else if (S30[i]) {
    L[i] = 23;
    U[i] = 30;
  } else {
    L[i] = 30;
    U[i] = K+10;
  }
}

/* Compute the cane-cane distances. */
printf("Computing pairwise distances ... ");
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    D[i][j] = sqrt( sq(x[i]-x[j]) + sq(y[i]-y[j]) );
printf("done.\n");
printf("L[0]=%d\n", L[0]);

/* Test the logpi function ... */
for (i=0; i<N; i++)
  curtau[i] = U[i];
printf("Test logpi value: %f\n", logpi(2.0,3.0,curtau));

/* Initialise the Markov chain values. */
M = 110000;  /* run length */
B = 10000;  /* burn-in */
M = 11000;
B = 1000;
/* NO, TOO UNSTABLE:
curtheta = 10 * exponential();
curmu = 10 * normal();
*/
curtheta = 2.0;
curmu = 3.0;
sumtheta = summu = 0.0;
for (i=0; i<N; i++) {
  curtau[i] = iround((L[i]+U[i])/2.0);
  sumtau[i] = 0;
}
propth = propmu = proptau = accth = accmu = acctau = 0;
curlogpi = logpi(curtheta,curmu,curtau);
printf("Initial logpi value: %f\n", curlogpi);

/* Run the Markov chain! */
printf("Running chain, for %d iterations (burn-in %d) ...\n", M, B);
```

```
for (t=1; t<=M; t++) {
  printf(" t=%d", t);
  fflush(stdout);

  /* Propose new theta value. */
  propth++;
  newtheta = curtheta + 2.0 * normal();
  newlogpi = logpi(newtheta,curmu,curtau);
  if ( log(uniform()) < newlogpi - curlogpi ) {
    /* Accept proposal. */
    accth++;
    curtheta = newtheta;
    curlogpi = newlogpi;
  }

  /* Propose new mu value. */
  propmu++;
  newmu = curmu + 0.1 * normal();
  newlogpi = logpi(curtheta,newmu,curtau);
  if ( log(uniform()) < newlogpi - curlogpi ) {
    /* Accept proposal. */
    accmu++;
    curmu = newmu;
    curlogpi = newlogpi;
  }

  /* Loop through the cane sites. */
  for (i=0; i<N; i++) {
    /* printf(" i=%d", i);
    fflush(stdout); */
    if (curtau[i] <= K) {
      /* Propose new tau[i] value. */
      proptau++;
      for (j=0; j<N; j++) {
        if (j==i)
          newtau[j] = curtau[j] + pmo();
        else
          newtau[j] = curtau[j];
      }
      newlogpi = logpi(curtheta,curmu,newtau);
/* printf("t=%d, i=%d; logpi diff = %f\n", t, i, newlogpi - curlogpi); */
      if ( log(uniform()) < newlogpi - curlogpi ) {
        /* Accept proposal. */
           acctau++;
/* printf("Accepted tau proposal at i=%d; acctau=%d.\n", i, acctau); */
        curtau[i] = newtau[i];
        curlogpi = newlogpi;
      }
    }
```

```
    }

    /* Update our running sums. */
    if (t > B) {
      sumtheta = sumtheta + curtheta;
      summu = summu + curmu;
      for (j=0; j<N; j++)
        sumtau[j] = sumtau[j] + curtau[j];
    }

  } /* End of Markov chain run. */

  /* Output the results. */
  printf("\n\nARtheta=%f,  ARmu=%f,  ARtau=%f\n",
    ((double)accth)/propth, ((double)accmu)/propmu, ((double)acctau)/proptau );
  printf("Mean theta: %f\n", sumtheta/(M-B));
  printf("Mean mu: %f\n", summu/(M-B));
  if ((fp = fopen("tauvals","w")) == NULL) {
      fprintf(stderr, "Unable to write file 'tauvals'.\n");
      exit(1);
  }
  for (i=0; i<N; i++)
    fprintf(fp, "%f  %f  %f\n", x[i], y[i], ((double)sumtau[i])/(M-B));
  fclose(fp);

  return(0);

  }  /* End of Main Program. */


  /* pmo: function which returns +1 or -1, with probability 1/2 each. */
  int pmo() {
    if (drand48() < 0.5)
      return(+1);
    return(-1);
  }

  /* Specify the target log density. */
  double logpi(double thetheta, double themu, int thetau[]) {
    int ii, jj, kk;
    double tmpsum;
    /* Compute the lambda_{x,k} values. */
    for (ii=0; ii<N; ii++) {
      if ( (thetau[ii] <= L[ii]) || (thetau[ii] > U[ii]) ) {
        /* printf("out of range: ii=%d, L=%d, tau=%d, U=%d\n",
                                   ii, L[ii], thetau[ii], U[ii]); */
        return(-infinity);
      }
      for (kk=0; kk<K; kk++) {
```

```
      tmpsum = themu;
      for (jj=0; jj<N; jj++) {
        if (thetau[jj] <= kk-1)
             tmpsum = tmpsum + exp(-thetheta*D[ii][jj]);
      }
      lambda[ii][kk] = exp(tmpsum);
    }
  }
  /* Compute the sums. */
  tmpsum = -thetheta/10 - sq(themu)/200;
  for (ii=0; ii<N; ii++) {
    if (thetau[ii] <= K)
      tmpsum = tmpsum + log(1-exp(-lambda[ii][thetau[ii]]));
    for (kk=0; kk < imin(thetau[ii]-1,K); kk++)
      tmpsum = tmpsum - lambda[ii][kk];
  }
  return(tmpsum);
}


/* SEEDRAND: SEED RANDOM NUMBER GENERATOR. */
void seedrand()
{
    int i, seed;
    struct timeval tmptv;
    gettimeofday (&tmptv, (struct timezone *)NULL);
    seed = (int) tmptv.tv_usec;
    srand48(seed);
    (void)drand48();  /* Spin it once. */
}


double sq(double xxx)
{
  return(xxx*xxx);
}


double uniform()
{
    double drand48();
    return( drand48() );
}

double exponential()
{
    double uniform();
    return( -log(uniform()) );
}
```

```
/* NORMAL:  return a standard normal random number. */
double normal()
{
    double RRR, ttt, uniform();

    RRR = - log(uniform());
    ttt = 2 * PI * uniform();

    return( sqrt(2*RRR) * cos(ttt));
}

/* IFLOOR */
int ifloor(double xxx)  /* returns floor(xxx) as an integer */
{
    return((int)floor(xxx));
}

int iround(double xxx)
{
    return( ifloor(xxx+0.5) );
}

int imin(int iii, int jjj)
{
    if (iii < jjj)
      return(iii);
    return(jjj);
}
```

PMMH Algorithm for Toy Example 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include   <stdio.h>
```

```
#include   <dos.h>
#include <stdio.h>
#include <conio.h>


#define N 200
#define T 30
#define infinity 999999999.0
#define PI 3.1415926536

/* M stores the sampled values, margLike the marginal likelihood
corresponding to each row */
double M[T][N], SMCSample[T], logMarg;
    double W[T][N];
        int A[T-1][N];
double x[T], y[T];

/* BEGIN MAIN PROGRAM. */

int main(int argc, char **argv)
{

/* Initial declarations.
M - number of PMMH iterations
B - burn-in
T - length of Markov Chain
N - number of particles for SMC sampling
*/
int MRuns, B;
int i, k, j;
/* Rsig1, Rsig2 - real values of sig1, sig2 */
double Rsig1, Rsig2;
double Curr[T], Prop[T];
double cSig1, cSig2, pSig1, pSig2;
double sig1sum, sig2sum;
double currLike, propLike;
double helper, conv;
double uniform(), exponential(), normal(), imin();
void seedrand(), SMCSampler(), SMCSampler2();
int accCounter;

/* Seed the random number generator. */
seedrand();

/* Set values of Rsig1, Rsig2. */
Rsig1 = 2;
Rsig2 = 0.25;

/* Generate Hidden Markov Chain and the Observed one. */
```

```
x[1] <- normal();
for (i=1; i<T; i++) {
    x[i+1] = x[i] + Rsig1* normal();
}
for (i=1; i<T+1; i++) {
    y[i] = (x[i]/2000.0)*(x[i]/2000.0) + Rsig2 * normal();
}

/* Initialise the Markov chain values. */
MRuns = 11000;  /* run length */
B = 1000;  /* burn-in */

/* Start PMMH Algorithm
Set "arbitrary" values for parameters */
cSig1 = 1 + 1* uniform();
cSig2 = 0.25 + 0.5* uniform();
sig1sum = sig2sum = 0;

SMCSampler(cSig1, cSig2);
currLike = logMarg;

for (j=1; j<T+1; j++){
Curr[j] = SMCSample[j];
}

// used to determine acceptance rate
accCounter = 0;

/* Run remaining steps in Markov Chain */
for (k=2; k<MRuns+1; k++)
{
           /* Propose new parameter values */
           pSig1 = fabs(cSig1+0.001*normal());
           //printf("%f\n", pSig1);
           pSig2= fabs(cSig2+0.001*normal());

           // Obtain new sample using SMC
           SMCSampler(pSig1, pSig2);
           propLike = logMarg;
           for (j=1; j<T+1; j++){
        Prop[j] = SMCSample[j];
    }

           // Accept/reject
           helper = uniform();

           conv = imin(0, propLike - currLike);
           // Accept/reject
           if (log(helper) < conv){
```

```
                 for (j=1; j<T+1; j++){
            Prop[j] = Curr[j];
        }
                 currLike = propLike;
                 cSig1 = pSig1;
                 cSig2 = pSig2;
                 accCounter = accCounter + 1;
           }
           /* Update our running sums. */
             if (k > B) {
    sig1sum = sig1sum + cSig1;
           sig2sum = sig2sum + cSig2;
  }
  printf("%f ", (double)k);
}

/* Output the results. */
printf("Mean sig1: %f\n",  sig1sum/(double)(MRuns-B));
printf("Mean sig2: %f\n", sig2sum /(double)(MRuns-B));
printf("Acceptance ratio: %f\n", (double)accCounter  /(double)(MRuns-1));
sleep(10000000);
return(0);

}  /* End of Main Program. */


/* pmo: function which returns +1 or -1, with probability 1/2 each. */
int pmo() {
  if ((double)rand()/RAND_MAX < 0.5)
    return(+1);
  return(-1);
}


/* SEEDRAND: SEED RANDOM NUMBER GENERATOR. */
void seedrand()
{
     time_t t;
     t = time (NULL);
     srand(t);
    (double)rand()/RAND_MAX;  /* Spin it once. */
}



double sq(double xxx)
{
  return(xxx*xxx);
}
```

```
double uniform()
{
    return((double)rand()/RAND_MAX);
}

double exponential()
{
    double uniform();
    return( -log(uniform()) );
}

/* NORMAL:  return a standard normal random number. */
double normal()
{
    double RRR, ttt, uniform();

    RRR = - log(uniform());
    ttt = 2 * PI * uniform();

    return( sqrt(2*RRR) * cos(ttt));
}

double imin(double iii, double jjj)
{
    if (iii < jjj)
      return(iii);
    return(jjj);
}

/* Produces SMC sample based on observed data y, an array of length
T, using N particles.
fsig1, fsig2 - values of the parameters
Returns log Marginal Likelihood of the sample
*/
void SMCSampler(double fsig1, double fsig2)
{
    //double uniform(), normal();
    double weightSum, helper;
    double partSum;
    double uniform(), normal();
    int i, j, k, t;
    double counter;
    logMarg = 0;
    counter = 0;

    /* Step 1 */
    for (i=1; i<N+1; i++)
```

```
        {
              // set proposal density to be mu_theta
              M[1][i] = normal();
              // Since we are sampling from the correct distribution,
              // non-normalized weights are all equal to 1
              W[1][i] = 1/(double)N;
        }

        /* Steps 2, 3, ..., N. */
        for (t=2; t<T+1; t++)
        {

     partSum = 0;
              // Sample each subsequent particle
              for (j=1; j<N+1; j++)
              {
                    // Sample the parents

                    // used to sample from a descrete distribution
                    // http://frank.itlab.us/datamodel/node86.html
                    helper = uniform();
                    double total = 0;
                    int index = 1;
                    for (k=1; k<N+1; k++)
                    {
                          total = total + W[t-1][k];
                          if (helper < total){
                            index = k;
                            break;
                          }
                    }
                    // record the value of sampled ancestor index in matrix A
                    A[t-1][j]= index;
                    // sample next value
               M[t][j] = M[t-1][index]+(fsig1)*normal();

      /* compute non-normalized weights
             note the cancellation for the case when ftheta(xn|xn-1,yn)=ftheta(xn|xn-1)
             W[t, j] = 1/(sqrt(2pi)*sig1)*exp(-(M[t,j]-M[t-1, A[t-1, j]])^2/(2*(sig1)^2)))
               * 1/(sqrt(2pi)*sig2)*exp(-(y[t]-(M[t,j])^2)^2/(2*(sig2)^2)))
               /( 1/(sqrt(2pi)*sig11)*exp(-(y[t]-(M[t,j])^2)^2/(2*(sig1)^2)))  ) */

                    W[t][j] = (long double)1/(sqrt(2*PI)*fsig2)*(long double)exp(-(y[t]-(M[t][j]/2000)*(M[t]
[j]/2000))*(y[t]-(M[t][j]/2000)*(M[t][j]/2000))/(2*fsig2*fsig2));


                    partSum = partSum + W[t][j];
              }
              counter = counter +1;
```

```
                /* add log marginal likelihood */
                logMarg = logMarg + log(partSum/(double)N);

                /* normalize the weights */
                weightSum = 0.0;
                for (k=1; k<N+1; k++)
                {
                      weightSum = weightSum + W[t][k];
                }

                for (j=1; j<N+1; j++)
                {
                      W[t][j] = W[t][j] / weightSum;
                }

          }
}
```

PMMH Algorithm for Toy Example 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include   <stdio.h>
#include   <dos.h>
#include <sys/time.h>

#define N 100
#define T 20
#define infinity 999999999.0
#define PI 3.1415926536

/* M stores the sampled values, margLike the marginal likelihood
corresponding to each row */
double M[T][N], SMCSample[T], logMarg;
    double W[T][N];
        int A[T-1][N];
double x[T], y[T];

/* BEGIN MAIN PROGRAM. */

int main(int argc, char **argv)
{

/* Initial declarations.
M - number of PMMH iterations
B - burn-in
T - length of Markov Chain
N - number of particles for SMC sampling
*/
int MRuns, B;
int i, k, j;
/* Rmean1, Rmean2 - real values of mean1, mean2 */
double RMean1, RMean2, inMean1, inMean2;
double Curr[T], Prop[T];
double cMean1, cMean2, pMean1, pMean2;
double mean1sum ,mean2sum ;
double currLike, propLike;
double helper, conv;
double uniform(), exponential(), normal(), imin();
double accCounter;
void seedrand(), SMCSampler(), SMCSampler2();
 time_t t;
t = time (NULL);
```

```
srand(t);

/* Seed the random number generator. */
seedrand();

/* Set values of Rmean1, Rmean2. */
RMean1 = 2.0;
RMean2 = 0.5;

/* Generate Hidden Markov Chain and the Observed one. */
x[1] <- normal();
for (i=1; i<T; i++) {
    x[i+1] = x[i] + RMean1 + 0.01*normal();
}
for (i=1; i<T+1; i++) {
    y[i] = x[i]/10.0 + RMean2 + 0.01*normal();
}

/* Initialise the Markov chain values. */
MRuns = 11000;  /* run length */
B = 1000;  /* burn-in */


/* Start PMMH Algorithm
Set "arbitrary" values for parameters */
cMean1 = 1.5 + 1.0* uniform();
cMean2 = 0.25 + 0.5 * uniform();
inMean1 = cMean1;
inMean2 = cMean2;

mean1sum = mean2sum = 0;

SMCSampler(cMean1, cMean2);

currLike = logMarg;
for (j=1; j<T+1; j++){
Curr[j] = SMCSample[j];
}
accCounter = 0;

/* Run remaining steps in Markov Chain */
for (k=2; k<MRuns+1; k++)
{
           /* Propose new parameter values */
           pMean1 = cMean1+0.001*normal();
           //printf("%f\n", pSig1);
           pMean2= cMean2+0.001*normal();

           // Obtain new sample using SMC
```

```
           SMCSampler(pMean1, pMean2);
           propLike = logMarg;
           for (j=1; j<T+1; j++){
        Prop[j] = SMCSample[j];
    }

           // Accept/reject
           helper = uniform();

           conv = imin(0, propLike - currLike);

           // Accept/reject
           if (log(helper) < conv){
                 for (j=1; j<T+1; j++){
            Prop[j] = Curr[j];
        }
                 currLike = propLike;
                 cMean1 = pMean1;
                 cMean2 = pMean2;
                 accCounter = accCounter + 1;
           }
           /* Update our running sums. */
             if (k > B) {
    mean1sum  = mean1sum + cMean1;
           mean2sum  = mean2sum + cMean2;
  }
  printf("%f\n", (double)k);
}

/* Output the results. */
//Print initial values
printf("Init mean1: %f\n",  inMean1);
printf("Init mean2: %f\n", inMean2);

printf("Mean mean1: %f\n",  mean1sum/(double)(MRuns-B));
printf("Mean mean2: %f\n", mean2sum  /(double)(MRuns-B));
printf("Acceptance ratio: %f\n", accCounter  /(double)(MRuns-1));
sleep(10000000);
return(0);

}  /* End of Main Program. */


/* pmo: function which returns +1 or -1, with probability 1/2 each. */
int pmo() {
  if ((double)rand()/RAND_MAX < 0.5)
    return(+1);
  return(-1);
}
```

```
/* SEEDRAND: SEED RANDOM NUMBER GENERATOR. */
void seedrand()
{
    (double)rand()/RAND_MAX;  /* Spin it once. */
}



double sq(double xxx)
{
  return(xxx*xxx);
}


double uniform()
{
    return((double)rand()/RAND_MAX);
}

double exponential()
{
    double uniform();
    return( -log(uniform()) );
}

/* NORMAL:  return a standard normal random number. */
double normal()
{
    double RRR, ttt, uniform();

    RRR = - log(uniform());
    ttt = 2 * PI * uniform();

    return( sqrt(2*RRR) * cos(ttt));
}

double imin(double iii, double jjj)
{
    if (iii < jjj)
      return(iii);
    return(jjj);
}

/* Produces SMC sample based on observed data y, an array of length
T, using N particles.
fMean1, fMean2 - values of the parameters
```

```
Returns log Marginal Likelihood of the sample
*/
//double SMCSampler(double fsig1, double fsig2){
//       return(uniform());
//       }
void SMCSampler(double fMean1, double fMean2)
{
    //double uniform(), normal();
    double weightSum, helper;
    double partSum;
    double uniform(), normal();
    int i, j, k, t;
    double counter;
    logMarg = 0;
    counter = 0;

    /* Step 1 */
    for (i=1; i<N+1; i++)
           {
                 // set proposal density to be mu_theta
                 M[1][i] = normal();
                 W[1][i] = (long double)1/(sqrt(2*PI)*0.1)*(long double)exp(-(y[1]-M[1][i]-fMean2)*(y[1]-M[1][i]-
fMean2)/(2*0.01));
                 // Since we are sampling from the correct distribution,
                 // non-normalized weights are all equal to 1
        // W[1][i] = 1/(double)N;
           }
           weightSum = 0.0;
                 for (k=1; k<N+1; k++)
                 {
                       weightSum = weightSum + W[1][k];
                 }

                 for (j=1; j<N+1; j++)
                 {
                       W[1][j] = W[1][j] / weightSum;
                 }

           /* Steps 2, 3, ..., N. */
           for (t=2; t<T+1; t++)
           {

        partSum = 0;
                 // Sample each subsequent particle
                 for (j=1; j<N+1; j++)
                 {
                       // Sample the parents

                       // used to sample from a descrete distribution
```

```
        // http://frank.itlab.us/datamodel/node86.html
        helper = uniform();
        double total = 0;
        int index = 1;
        for (k=1; k<N+1; k++)
        {
              total = total + W[t-1][k];
              if (helper < total){
                index = k;
                break;
              }
        }
        // record the value of sampled ancestor index in matrix A
        A[t-1][j]= index;

        // sample next value
   M[t][j] = M[t-1][index]+fMean1+ 0.01*normal();


   /* compute non-normalized weights
  note the cancellation for the case when ftheta(xn|xn-1,yn)=ftheta(xn|xn-1)
  W[t, j] = 1/(sqrt(2pi)*sig1)*exp(-(M[t,j]-M[t-1, A[t-1, j]])^2/(2*(sig1)^2)))
    * 1/(sqrt(2pi)*sig2)*exp(-(y[t]-(M[t,j])^2)^2/(2*(sig2)^2)))
    /( 1/(sqrt(2pi)*sig11)*exp(-(y[t]-(M[t,j])^2)^2/(2*(sig1)^2)))  ) */

        W[t][j] = (long double)1/(sqrt(2*PI)*0.1)*(long double)exp(-(y[t]-M[t][j]-fMean2)*(y[t]-M[t][j]-
fMean2)/(2*0.01));
        //W[t][j] = log(1/sqrt(2*PI)*fsig2)- pow(y[t]-pow(M[t][j]/2000, 2), 2)/2/pow(fsig2, 2);
        //W[t][j]=25.0;


        partSum = partSum + W[t][j];
  }
  counter = counter +1;

  /* add log marginal likelihood */
  logMarg = logMarg + log(partSum/(double)N);

  /* normalize the weights */
  weightSum = 0.0;
  for (k=1; k<N+1; k++)
  {
        weightSum = weightSum + W[t][k];
  }

  for (j=1; j<N+1; j++)
  {
        W[t][j] = W[t][j] / weightSum;
  }
```

```
        }
}
```

PMMH Algorithm for Toy Example 3, code written by Marco Läubli (2011)

```
# Toy function definitions
# Assume S = 0,1
# density for x_1
toy.mu <- function(theta, S){
  c(0.01, 0.99)
 }

# density for x_{n+1}|x_{n}
toy.f<- function( theta ,S , prevVal){
if (prevVal == 0) ans1 = c(1-theta[1], theta[1])
else ans1 = c(theta[2], 1-theta[2])
ans1
}

# density for y_{n}|x_{n}
toy.g<- function( theta ,yval , prevVal){
ans1 = rep(1, length(prevVal))
for (i in (1:length(prevVal))){
if (prevVal[i] == yval) ans1[i] = 0.88
else ans1[i] = 0.12
}
ans1
}


# ===============================================================
 # helper function : SMC algorithm with naive proposal
 # ===============================================================
 algSMCnaive <- function ( theta ,y ,N)
 {
 T <- length (y) # end time
 X <- matrix ( nrow =T , ncol = N) # particle matrix
 S <- c (0 ,1) # the sample space
 # -----------------------------------------------------------
 # step 1 at time n =1:
 # -----------------------------------------------------------
 # (a) sample X1 ^k ~ q (.| y1 )
 X1 <- sample (S ,N , TRUE , toy.mu ( theta ,S ))
 X [1 ,] <- X1
 # (b) compute and normalize the weights
 w <- toy.g( theta ,y [1] , X1 )
 sw <- sum (w)
 W <- w/ sw
 py <- sw /N
 # -----------------------------------------------------------
```

```
# step 2 at times n =2 ,... , T:
# ------------------------------------------------------------
for (n in 2: T){
# (a) sample An -1^ k ~ F (.| Wn -1)
A <- sample (N ,N , replace = TRUE , prob =W)
# (b) sample Xn ^k ~ q (.| yn ,Xn -1^ An -1^ k ) and
# set X1 : n^k = ( X1 :n -1^ An -1^ k , Xn ^k )
X1A <- X[n -1 , A]
for (k in 1: N){
X[n ,k] <- sample (S ,1 , prob = toy.f( theta ,S , X1A [k ]) )
}
X [1:( n -1) ,] <- X [1:( n -1) ,A]
# compute and normalize the weights
w <- toy.g( theta ,y [n],X [n ,])
sw <- sum (w)
W <- w/ sw
py <- py * sw /N
}
return ( list ( X=X ,W =W , py = py ) ) # returning values
}
# ==============================================================
# The PMMH algortihm with naive SMC proposal
# =============================================================
algPMMHnaive <- function (y ,N ,N.PMMH , sigma =1)
{
# sigma is the std dev . of the random walk on the logit scale
t0 <- proc.time () [3] # starting time
nparam <- 2 # number of parameters
cat (" \n algPMMHnaive : algorithm started with N=" ,N ,
" , T=" , length (y) ,
" , N.PMMH =" ,N.PMMH ,
" , sigma =" , sigma ," :\ n" , sep ="")
# ------------------------------------------------------------
# the prior of theta
# ------------------------------------------------------------
algPMMH.p <- function ( theta ){
return (1) # uniform prior on (0 ,1) ^2
}
# ------------------------------------------------------------
# the proposal densities q( theta | theta (i -1) )
# ------------------------------------------------------------
algPMMH.q <- function ( theta , theta.i1 ) {
origin <- log ( theta.i1 / (1 - theta.i1 )) # origin of r. walk
g <- function ( x){ # logit scale -> (0 ,1)
exp (x )/ (1+ exp (x ))
}
g.prime <- function (x){ # derivative of g
g(x )* (1 - g(x) )
}
```

```
 g.inv <- function ( y){ # inverse of g
 log (y / (1 - y ))
 }
 # see W ' keit und Statistik script K n s c h , F l l m e r page 40
 return ( prod ( 1/ g.prime (g.inv ( theta ))*
 dnorm (g.inv ( theta ) , mean = origin , sd = sigma ) ))
 }
 # ------------------------------------------------------------
 # function to sample from the proposal density
 # q (.| theta (i -1) ) for theta
 # ------------------------------------------------------------
 algPMMH.q.sample <- function ( theta.i1 ){
 # a random walk on the logit scale log ( theta / (1 - theta ))
 origin <- log ( theta.i1 / (1 - theta.i1 ))
 jump <- rnorm (2 , sd = sigma )
 destiny <- origin + jump
 return ( exp ( destiny )/ (1+ exp ( destiny ) ))
 }
 # ============================================================
 # main part of the PMMH algorithm
 # ============================================================
 theta <- matrix ( NA , nparam ,N.PMMH +1)
 X <- matrix (NA , length (y) ,N.PMMH +1)
 py <- rep (NA , N.PMMH +1)
 # -------------------------------------------------------------
 # step 1: initialization i =0
 # -------------------------------------------------------------
 # (a) set theta (1) arbitrarly
 theta [ ,1] <- rep (1 /2, nparam )
 # (b) run an smc algorithm targeting p_ theta (1) ( x1 :T| y1 :T)
 smc <- algSMCnaive ( theta [ ,1] ,y ,N)
 # sample X1 :T (1) ~p^ _ theta (1) (.| y1 :T)
 k <- sample (N ,1 , prob = smc $W) # draw an index k
 X [ ,1] <- smc $X [,k] # corresponds to X_ 1: T^ k
 # the corresponding marginal likelihood estimate
 py [1] <- smc $ py
 # ------------------------------------------------------------
 # step 2: iteration i >=1
 # ------------------------------------------------------------
acc.rate <- 0 # acceptance rate init
 acc <- rep (0 , N.PMMH ) # accepted in which iter .
 for (i in 2:( N.PMMH +1) )
 {
print(i)
#print(theta[,i-1])
 # (a) sample theta *~q {.| theta (i -1) }
 theta. <- algPMMH.q.sample ( theta [,i -1])
 # (b) run an SMC algorithm targeting p_ theta *( x1 : T| y1 : T)
 smc <- algSMCnaive ( theta.,y ,N)
```

```
 # sample X1 :T* ~ p^_ theta * (.| y1 :T)
 k <- sample (1: N ,1 , prob = smc $W ) # draw an index k
 X. <- smc$X[,k ] # corresponds to X_ 1: T^k
 # the corresponding marginal likelihood estimate
 py. <- smc$py
 # (c) with probability
 # min {1 , a} accept
 a <- py.* algPMMH.p( theta.) /( py [i -1] * algPMMH.p( theta [ ,i -1]) )*
 algPMMH.q( theta [ ,i -1] , theta.) /
 algPMMH.q( theta. , theta [,i -1])
 if ( runif (1) <a ){ # accepted
 acc [i -1] <- 1 # accepted in this iter .
 theta [,i] <- theta.
 X[, i] <- X.
 py [i] <- py.
 acc.rate <- acc.rate +1
 }
 else { # rejected
 theta [,i] <- theta [,i -1]
 X[, i] <- X[,i -1]
 py [i] <- py [i -1]
 }
 if ((i -1) %% 10==0) {
 progress <- (i -1) /N.PMMH # progress of algPMMH
 ti <- proc.time () [3] - t0 # elapsed time
 timeleft <- (1 - progress )/ progress * ti # remaining time
 cat (" algPMMHnaive : " , progress * 100 , "% completed " ,
 " in " , signif (ti ,3) ,"s , " ,
 signif ( timeleft ,3) ,"s remaining \n" , sep ="" )
 }
 }
 acc.rate <- acc.rate / N.PMMH
 cat (" algPMMHnaive : average acceptance rate =" ,
 100 * acc.rate ,"% \n ")
 ind <- 2:( N.PMMH +1)
 cputime <- proc.time () [3] - t0
print(mean(theta[1,100:1000]))
print(mean(theta[2,100:1000]))
print(mean(theta[1,]))
print(mean(theta[2,]))
 return ( list ( theta = theta [, ind ],X =X[, ind ], py = py [ ind ],
 acc.rate = acc.rate , acc = acc ,y=y ,N=N ,
 N.PMMH =N.PMMH , cputime = cputime )) # returning values
 }
```

THIS IS ANOTHER FILE THAT SHOULD BE INCLUDED IN THE SAME FOLDER AS THE ABOVE FILE; THIS FILE SHOULD BE EXECUTED

```
source("BD1.r")
```

```
T = 100
# Real theta
theta2 = c(0.3, 0.3)

set.seed(19)
# Generate Sample values
x <- rep(1, T)
y <- rep(1, T)
S = c(0,1) # sample space


x[1] = sample(S ,1, TRUE , toy.mu ( theta2 ,S ))
for (i in (2:T)){
x[i]=sample (S ,1 , prob = toy.f( theta2 ,S , x[i-1]) )

}


toy.g2<- function( theta ,S , prevVal){
if (prevVal == 0) ans1 = c(0.88, 0.12)
else ans1 = c(0.12, 0.88)
ans1
}
```

Final C Program for Metropolis within Gibbs for Discrete Time Model, including Simplifications

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <Windows.h>
#include <unistd.h>

#define MAXN 1742
#define K 30
#define M 11000
#define B 1000
#define infinity 999999999.0
#define PI 3.1415926536

int L[MAXN], U[MAXN],N, sortedplant[MAXN][MAXN],tau[MAXN][M];
double lambda[MAXN][K], D[MAXN][MAXN],x[MAXN], y[MAXN],d[MAXN][MAXN],sortedd[MAXN][MAXN];
double sq(),logpi(),logpi2(),logpi3(), uniform(),exponential(),normal(),f(),dmax(),ftr(), Distance(), distance();
int pmo(),ifloor(),iround(),imin();
void seedrand();
double a1 = 1.0, b1 = 0.05, a2 = 1.0, b2 = 0.01, a3 = 1.0, b3 = 1.0 ;

int main(int argc, char **argv)
{
int i, j, k, l, h, t, propmu, propth, proptau, accmu, accth, acctau, propsigma, accsigma, tmpi;
double curtheta, curmu, newmu, newtheta, summu, sumtheta, logPiDiff, newlogpi, curlogpi, cursigma, newsigma, sumsigma,
tmpd, tmpsum;
int S6[MAXN], S10[MAXN], S14[MAXN], S19[MAXN], S23[MAXN], S30[MAXN],curtau[MAXN], newtau[MAXN], sumtau[MAXN];
double theta[M],mu[M],sigma[M];
char tmpstring[20];
FILE *fp;
clock_t  clock1, clock2, clock3, clock4, clock5;
double t1 = 0.0, t2 = 0.0, t3 = 0.0, t4 = 0.0, t5 = 0.0;

/*set the proposal scale for theta and mu*/
double sigma1 = 0.01, sigma2 = 0.002, sigma3 = 0.1;

/* Seed the random number generator. */
seedrand();

/* Read in the data. */
printf("Reading data ...");
if ((fp = fopen("C:/Alexander/University/2011-2012/STA496/Programs/canedata.txt","r")) == NULL) {
    fprintf( stderr, "Unable to read file 'canedata'.\n");
    exit(1);
```

```
}
N = 0;
for (i=0; i<MAXN; i++) {
    fscanf(fp, "%s", &tmpstring);
    x[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    y[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S6[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S10[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S14[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S19[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S23[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S30[N] = atoi(tmpstring);
           //if( x[N]<4 && y[N]<4 )
             N++;
}
fclose(fp);
printf("done. \n");
printf("Number of sites studied: %d\n", N);

/* Determine the L_x and U_x values, etc. */
printf("Computing infection time ranges ...");
for (i=0; i<N; i++) {
  if (S6[i]) {
    L[i] = 0;
    U[i] = 6;
  } else if (S10[i]) {
    L[i] = 6;
    U[i] = 10;
  } else if (S14[i]) {
    L[i] = 10;
    U[i] = 14;
  } else if (S19[i]) {
    L[i] = 14;
    U[i] = 19;
  } else if (S23[i]) {
    L[i] = 19;
    U[i] = 23;
  } else if (S30[i]) {
    L[i] = 23;
    U[i] = 30;
  } else {
    L[i] = 30;
```

```
    U[i] = K+10;
  }
}
printf("done. \n");

/* Compute the cane-cane distances. */
printf("Computing pairwise distances ... ");
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    D[i][j] = Distance(i,j);
printf("done.\n");

clock1 = clock();
/*Compute neighbor lists. */
printf("Sort plants by distance ... ");
// initialize
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    sortedd[i][j] = d[i][j] = distance(i,j);
    sortedplant[i][j] = j;
  }
}

// sort plants by distance
for (i=0; i<N; i++){
  for (j=0; j<(N-1); j++){
           for(k=j+1;k<N;k++){
                 if(sortedd[i][k] < sortedd[i][j]){
                     tmpd = sortedd[i][k];
                     sortedd[i][k] = sortedd[i][j];
                     sortedd[i][j] = tmpd;
                       tmpi = sortedplant[i][k];
                       sortedplant[i][k] = sortedplant[i][j];
                       sortedplant[i][j] = tmpi;
                 }
           }
  }
}
printf("done. \n");
clock2 = clock();
t1 = (double) (clock2 - clock1)/CLOCKS_PER_SEC;

curtheta = 0.04;
curmu = 0.003;
cursigma = 1.0;
for (i=0; i<N; i++) {
  curtau[i] = iround((L[i]+U[i])/2.0);
  //curtau[i] = U[i];
  sumtau[i] = 0;
```

```
  //printf("if saved in float, tau is %f, the correct way is %d",curtau[i],curtau[i]);
}

sumtheta = summu = sumsigma = 0.0;
propth = propmu = proptau = accth = accmu = acctau = propsigma =  accsigma = 0;
curlogpi = logpi3(curtheta,curmu,curtau,cursigma);

/* Run the Markov chain! */
printf("Running chain, for %d iterations (burn-in %d) ...\n", M, B);
for (t=1; t<=M; t++) {
  printf(" t=%d:", t);
  fflush(stdout);

  clock1 = clock();
  /* Propose new theta value. */
  propth++;
  newtheta = curtheta + sigma1 * normal();
  if(newtheta > 0){
	 newlogpi = logpi3(newtheta,curmu,curtau,cursigma);
	 if ( log(uniform()) < (newlogpi - curlogpi)) {
		accth++;
		curtheta = newtheta;
		curlogpi = newlogpi;
	  }
	  theta[t-1] = curtheta;
  }

  clock2 = clock();
  /* Propose new mu value. */
  propmu++;
  newmu = curmu + sigma2 * normal();
  if(newmu > 0) {
	 logPiDiff = -(a2+1.0)*(newmu-curmu) - b2*(1.0/newmu-1.0/curmu) ;
	 for (k = 1; k <= K; k++){
      for (i = 0; i < N; i++){
         if (curtau[i] == k){
            tmpsum = 0.0;
            for (j=0; j<N && sortedd[i][j]<=4*cursigma; j++) {
                 h = sortedplant[i][j];
                if (curtau[h] < k)
               tmpsum += f(D[i][h],cursigma);
            }
            logPiDiff += log(1-exp(-(newmu+curtheta*tmpsum))) - log(1-exp(-(curmu+curtheta*tmpsum)));
         }
         if (curtau[i] > k)
            logPiDiff -= newmu-curmu;
       }
  }
	  if ( log(uniform()) < logPiDiff ) {
```

```
                  accmu++;
                  curmu = newmu;
                  curlogpi += logPiDiff;
            }
            mu[t-1] = curmu;
 }

 clock3 = clock();
/* Loop through the cane sites. */
for (i=0; i<N; i++) {
   if (curtau[i] <= K) {
                proptau++;
          for (j=0; j<N; j++) {
                if (j==i)
                            newtau[j] = curtau[j] + pmo();
                      else
                      newtau[j] = curtau[j];
                }
                if ( (newtau[i] > L[i]) && (newtau[i] <= U[i]) ) {
                      logPiDiff = 0;
           if ( newtau[i] > curtau[i]) {
               tmpsum = 0.0;
               for (h=0; h<N && sortedd[i][h]<=4*cursigma; h++) {
                j = sortedplant[i][h];
                if (curtau[j] < curtau[i])
                  tmpsum += f(D[i][j],cursigma);
               }
               tmpsum = curmu+curtheta*tmpsum;
               logPiDiff -= tmpsum + log(1-exp(-tmpsum));
               for (k=0; k<N && sortedd[i][k]<=4*cursigma;k++) {
                l = sortedplant[i][k];
                   if (curtau[l] > newtau[i]){
                       logPiDiff += curtheta * f(D[i][l],cursigma);
                }
                   if (curtau[l] == newtau[i] && l != i){
                      tmpsum = 0.0;
                      for (h=0; h<N && sortedd[l][h]<=4*cursigma; h++) {
                            j = sortedplant[l][h];
                           if (newtau[j] < newtau[i])
                             tmpsum += f(D[l][j],cursigma);
                      }
                      tmpsum = curmu+curtheta*tmpsum;
                      logPiDiff += log(1-exp(-tmpsum))-log(1-exp(-tmpsum-curtheta*f(D[i][l],cursigma)));
                            }
               }
               tmpsum = 0.0;
               for (h=0; h<N && sortedd[i][h]<=4*cursigma; h++) {
                j = sortedplant[i][h];
                   if (newtau[j] < newtau[i])
```

```
                        tmpsum += f(D[i][j],cursigma);
                 }
                 tmpsum = curmu+curtheta*tmpsum;
                 logPiDiff += log(1-exp(-tmpsum));
             }
             else{
                 tmpsum = 0.0;
                 for (h=0; h<N && sortedd[i][h]<=4*cursigma; h++) {
                  j = sortedplant[i][h];
                        if (curtau[j] < newtau[i])
                    tmpsum += f(D[i][j],cursigma);
                 }
                 tmpsum = curmu+curtheta*tmpsum;
                 logPiDiff += tmpsum + log(1-exp(-tmpsum));
                 for (k=0; k<N && sortedd[i][k]<=4*cursigma;k++) {
                  l = sortedplant[i][k];
                     if (curtau[l] > curtau[i]){
                         logPiDiff -= curtheta * f(D[i][l],cursigma);
                     }
                     if (curtau[l] == curtau[i] && l != i){
                        tmpsum = 0.0;
                        for (h=0; h<N && sortedd[l][h]<=4*cursigma; h++) {
                              j = sortedplant[l][h];
                             if (newtau[j] < curtau[i])
                               tmpsum += f(D[l][j],cursigma);
                        }
                        tmpsum = curmu+curtheta*tmpsum;
                        logPiDiff += log(1-exp(-tmpsum)) - log(1-exp(-tmpsum+curtheta*f(D[i][l],cursigma)));
                     }
                 }
                 tmpsum = 0.0;
                 for (h=0;h<N && sortedd[i][h]<=4*cursigma; h++) {
                  j = sortedplant[i][h];
                     if (curtau[j] < curtau[i])
                        tmpsum = tmpsum + f(D[i][j],cursigma);
                 }
                 tmpsum = curmu+curtheta*tmpsum;
                 logPiDiff -= log(1-exp(-tmpsum));
             }
     if ( log(uniform()) < logPiDiff ) {
         acctau++;
         curtau[i] = newtau[i];
         curlogpi = curlogpi + logPiDiff;
     }
    }
   }
   tau[i][t-1] = curtau[i];
}
```

```
  clock4 = clock();
  /* Propose new sigma value. */
  propsigma++;
  newsigma = cursigma+ sigma3 * normal();
  if(newsigma > 0){
           newlogpi = logpi3(curtheta,curmu,curtau,newsigma);
           if ( log(uniform()) < (newlogpi-curlogpi)) {
                   accsigma++;
                   cursigma = newsigma;
                   curlogpi = newlogpi;
            }
            sigma[t-1] = cursigma;
  }

  clock5 = clock();
  /* Update our running sums. */
  if (t > B) {
    sumtheta = sumtheta + curtheta;
    summu = summu + curmu;
    sumsigma = sumsigma + cursigma;
    for (j=0; j<N; j++)
      sumtau[j] = sumtau[j] + curtau[j];
  }
  t2 = t2 + (double) (clock2 - clock1)/CLOCKS_PER_SEC;
  t3 = t3 + (double) (clock3 - clock2)/CLOCKS_PER_SEC;
  t4 = t4 + (double) (clock4 - clock3)/CLOCKS_PER_SEC;
  t5 = t5 + (double) (clock5 - clock4)/CLOCKS_PER_SEC;
}

/* Output the results. */
if ((fp = fopen("theta","w")) == NULL){
           fprintf(stderr,"Unable to write file 'theta'.\n");
           exit(1);
}
fprintf(fp,"theta=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",theta[i]);
fprintf(fp,"%f ) \n",theta[M-1]);
fclose(fp);

if ((fp = fopen("mu","w")) == NULL){
           fprintf(stderr,"Unable to write file 'mu'.\n");
           exit(1);
}
fprintf(fp,"mu=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",mu[i]);
fprintf(fp,"%f ) \n",mu[M-1]);
fclose(fp);
```

```
if ((fp = fopen("tau_infected","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'tau_infected'.\n");
    exit(1);
}
for(i=0;i<N;i++){
           if(curtau[i] <= K){
                 fprintf(fp,"tau[%d,] = c(",(i+1));
                 for (j=0;j<(M-1);j++){
                       fprintf(fp,"%d,",tau[i][j]);
                       //printf("%d,",tau[i][j]);
                 }
                 fprintf(fp,"%d); \n ",tau[i][M-1]);
           }
}
fclose(fp);

if ((fp = fopen("sigma","w")) == NULL){
           fprintf(stderr,"Unable to write file 'sigma'.\n");
           exit(1);
}
fprintf(fp,"sigma=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",sigma[i]);
fprintf(fp,"%f ) \n",sigma[M-1]);
fclose(fp);

if ((fp = fopen("tau_est","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'tau.est'.\n");
    exit(1);
}
fprintf(fp,"tau_est=c(");
for (i=0; i<(N-1); i++){
           fprintf(fp, "%f,\n",((double)sumtau[i])/(M-B));
}
fprintf(fp,"%f); \n",(double)sumtau[N-1]/(M-B));
fclose(fp);

if ((fp = fopen("out","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'out'.\n");
    exit(1);
}
fprintf(fp,"M=%d;\n",M);
fprintf(fp,"B=%d;\n",B);
fprintf(fp,"N=%d;\n",N);
fprintf(fp,"ARtheta=%f;\n",((double)accth)/propth);
fprintf(fp,"ARmu=%f;\n",  ((double)accmu)/propmu);
fprintf(fp,"ARtau=%f;\n", ((double)acctau)/proptau );
fprintf(fp,"ARsigma=%f;\n", ((double)accsigma)/propsigma );
```

```
fprintf(fp,"Mean_theta=%f;\n", sumtheta/(M-B));
fprintf(fp,"Mean_mu=%f;\n", summu/(M-B));
fprintf(fp,"Mean_sigma=%f;\n", sumsigma/(M-B));
fprintf(fp,"it takes %f seconds to sort plants; \n", t1);
fprintf(fp,"it takes %f seconds to update theta; \n", t2);
fprintf(fp,"it takes %f seconds to update mu; \n", t3);
fprintf(fp,"it takes %f seconds to update tau; \n", t4);
fprintf(fp,"it takes %f seconds to update sigma; \n", t5);
fclose(fp);

printf("\n done.\n");
return(0);
}

/* pmo: function which returns +1 or -1, with probability 1/2 each. */
int pmo() {
  if (uniform() < 0.5){
          return(-1);
  }
  return (1);
}

/* Specify the target log density. */
double logpi(double thetheta, double themu, int thetau[],double thesigma) {
  int ii, jj, kk;
  double tmpsum;
  if(thetheta <= 0.0 || themu <= 0.0 || thesigma <= 0.0)
           return (-infinity);
  for (ii=0; ii<N; ii++) {
    if ( (thetau[ii] <= L[ii]) || (thetau[ii] > U[ii]) ) {
          return(-infinity);
    }
    for (kk=0; kk<K; kk++) {
      tmpsum = themu;
      for (jj=0; jj<N; jj++) {
        if (thetau[jj] <= kk)
            tmpsum += thetheta*ftr(ii,jj,thesigma);
      }
      lambda[ii][kk] = tmpsum;
    }
  }
  tmpsum = -(a1+1.0)*thetheta - b1/thetheta -(a2+1.0)*themu - b2/themu -(a3+1.0)*thesigma - b3/thesigma;
  for (ii=0; ii<N; ii++) {
    if (thetau[ii] <= K)
      tmpsum += log(1-exp(-lambda[ii][thetau[ii]-1]));
    for (kk=1; kk <= imin(thetau[ii]-1,K); kk++)
      tmpsum -= lambda[ii][kk-1];
  }
  return(tmpsum);
```

```
}

/* Specify the target log density. */
double logpi2(double thetheta, double themu, int thetau[], double thesigma) {
  int ii, jj, kk;
  double tmpsum, tmpsum2;
  for (ii=0; ii<N; ii++) {
    if ( (thetau[ii] <= L[ii]) || (thetau[ii] > U[ii]) ){
             return(-infinity);
    }
  }
  if(thetheta <= 0.0 || themu <= 0.0 || thesigma <= 0.0)
            return (-infinity);
  tmpsum2 = -(a1+1.0)*thetheta - b1/thetheta -(a2+1.0)*themu - b2/themu -(a3+1.0)*thesigma - b3/thesigma;
  for (kk = 1; kk <= K; kk++){
      for (ii = 0; ii < N; ii++){
         if (thetau[ii] == kk){
            tmpsum = 0.0;
            for (jj=0; jj<N; jj++) {
                if (thetau[jj] < kk)
               tmpsum += ftr(ii,jj,thesigma);
            }
            tmpsum = themu+thetheta*tmpsum;
            tmpsum2 += log(1-exp(-tmpsum));
         }
         if (thetau[ii] > kk){
           tmpsum = 0.0;
           for (jj=0; jj<N; jj++) {
                        if (thetau[jj] < kk)
                 tmpsum += ftr(ii,jj,thesigma);
            }
            tmpsum = themu+thetheta*tmpsum;
            tmpsum2 -= tmpsum;
         }
       }
  }
  return(tmpsum2);
}

/*the target log density: not update lamda for already infected plants, advanced truncation.*/
double logpi3(double thetheta, double themu, int thetau[], double thesigma) {
  int ii, jj, kk,hh;
  double tmpsum, tmpsum2;
  for (ii=0; ii<N; ii++) {
    if ( (thetau[ii] <= L[ii]) || (thetau[ii] > U[ii]) ) {
           return(-infinity);
    }
  }
   if(thetheta <= 0.0 || themu <= 0.0 || thesigma <= 0.0)
```

```
            return (-infinity);
  tmpsum2 = -(a1+1.0)*thetheta - b1/thetheta -(a2+1.0)*themu - b2/themu -(a3+1.0)*thesigma - b3/thesigma;
  for (kk = 1; kk <= K; kk++){
      for (ii = 0; ii < N; ii++){
         if (thetau[ii] == kk){
            tmpsum = 0.0;
            for (jj=0; jj<N && sortedd[ii][jj]<=4*thesigma; jj++) {
                 hh = sortedplant[ii][jj];
                if (thetau[hh] < kk)
               tmpsum += f(D[ii][hh],thesigma);
            }
            tmpsum = themu+thetheta*tmpsum;
            tmpsum2 += log(1-exp(-tmpsum));
         }
         if (thetau[ii] > kk){
           tmpsum = 0.0;
           for (jj=0; jj<N && sortedd[ii][jj]<=4*thesigma; jj++) {
                 hh = sortedplant[ii][jj];
                if (thetau[hh] < kk)
               tmpsum += f(D[ii][hh],thesigma);
            }
            tmpsum = themu+thetheta*tmpsum;
            tmpsum2 -= tmpsum;
         }
       }
  }
  return(tmpsum2);
}

/* SEEDRAND: SEED RANDOM NUMBER GENERATOR. */
void seedrand()
{
    SYSTEMTIME str_t;
    double helper;
    GetSystemTime(&str_t);
    int seed;
    seed = (int) str_t.wMilliseconds;
    srand(seed);
}

double sq(double xxx)
{
  return(xxx*xxx);
}

double uniform()
{
    return((double)rand()/RAND_MAX);
}
```

```
double exponential()
{
    double uniform();
    return( -log(uniform()) );
}

/* NORMAL:  return a standard normal random number. */
double normal()
{
    double RRR, ttt, uniform();
    RRR = - log(uniform());
    ttt = 2 * PI * uniform();
    return( sqrt(2*RRR) * cos(ttt));
}

int ifloor(double xxx)
{
    return((int)floor(xxx));
}

int iround(double xxx)
{
    return( ifloor(xxx+0.5) );
}

int imin(int iii, int jjj)
{
    if (iii < jjj)
      return(iii);
    return(jjj);
}

double dmax(double xxx,double yyy)
{
  if(xxx<yyy) return (yyy);
  return (xxx);
}

double f(double xxx,double thesigma)
{
  return (exp(-sq(xxx)/(2*sq(thesigma)))/pow(2*PI*sq(thesigma),0.5));
}

double ftr(int ii, int jj, double thesigma)
{
  double Dist = D[ii][jj], dist = d[ii][jj];
  if( dist > (4.0*thesigma) ) {
           return (0.0);
```

```
  }
  return (exp(-sq(Dist)/(2*sq(thesigma)))/pow(2*PI*sq(thesigma),0.5));
}

double Distance(int ii, int jj){
           return ( sqrt( sq(x[ii]-x[jj]) + sq(y[ii]-y[jj]) ) );
}

double distance(int ii, int jj){
           return ( dmax(fabs(x[ii]-x[jj]), fabs(y[ii]-y[jj])) );
}
```

Final C Program for Metropolis within Gibbs for Continuous Time Model, including Simplifications

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <Windows.h>
#include <unistd.h>

#define MAXN 1742
#define K 30
#define M 11000
#define B 1000
#define infinity 999999999.0
#define PI 3.1415926536

int N, sortedplant[MAXN][MAXN];
double L[MAXN], U[MAXN];
double lambda[MAXN][K], D[MAXN][MAXN],tau[MAXN][M],x[MAXN], y[MAXN],d[MAXN][MAXN],sortedd[MAXN][MAXN];
double sq(),logpi(),logpi2(),logpi3(), uniform(),exponential(),normal(),f(),dmax(),ftr(), Distance(), distance();
int pmo(),ifloor(),iround(),imin();
double a1 = 1.0, b1 = 0.05, a2 = 1.0, b2 = 0.01, a3 = 1.0, b3 = 1.0 ;
void seedrand();

int main(int argc, char **argv)
{
int i, j, k, l, h, t, r, propmu, propth, proptau, accmu, accth, acctau, propsigma, accsigma, tmpi;
double curtheta, curmu, newmu, newtheta, summu, sumtheta, logPiDiff, newlogpi, curlogpi, cursigma, newsigma, sumsigma,
tmpd, tmpsum, tmpsum2, tmpsum3;
int S6[MAXN], S10[MAXN], S14[MAXN], S19[MAXN], S23[MAXN], S30[MAXN];
double curtau[MAXN], newtau[MAXN], sumtau[MAXN];
double a1, a2, a3, b1, b2, b3;
double theta[M],mu[M],sigma[M], tauVal[34840];
char tmpstring[20];
FILE *fp;
clock_t  clock1, clock2, clock3, clock4, clock5;
double t1 = 0.0, t2 = 0.0, t3 = 0.0, t4 = 0.0, t5 = 0.0;

/*set the proposal scale for theta and mu*/
double sigma1 = 0.005, sigma2 = 0.0005, sigma3 = 0.07, sigma4 = 1.0;
/* set prior distribution parameters */


/* Seed the random number generator. */
seedrand();

/* Read in the data. */
```

```
printf("Reading data ...");
if ((fp = fopen("C:/Alexander/University/2011-2012/STA496/Programs/canedata.txt","r")) == NULL) {
    fprintf( stderr, "Unable to read file 'canedata'.\n");
    exit(1);
}
N = 0;
for (i=0; i<MAXN; i++) {
    fscanf(fp, "%s", &tmpstring);
    x[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    y[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S6[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S10[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S14[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S19[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S23[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S30[N] = atoi(tmpstring);
//    if( x[N]<10.0 && y[N]<10.0 )
             N++;
}
fclose(fp);
printf("done. \n");
printf("Number of sites studied: %d\n", N);

/* Determine the L_x and U_x values, etc. */
printf("Computing infection time ranges ...");
for (i=0; i<N; i++) {
  if (S6[i]) {
    L[i] = 0.0;
    U[i] = 6.0;
  } else if (S10[i]) {
    L[i] = 6.0;
    U[i] = 10.0;
  } else if (S14[i]) {
    L[i] = 10.0;
    U[i] = 14.0;
  } else if (S19[i]) {
    L[i] = 14.0;
    U[i] = 19.0;
  } else if (S23[i]) {
    L[i] = 19.0;
    U[i] = 23.0;
  } else if (S30[i]) {
```

```
    L[i] = 23.0;
    U[i] = 30.0;
  } else {
    L[i] = 30.0;
    U[i] = 30.0+10.0;
  }
}
printf("done. \n");

/* Compute the cane-cane distances. */
printf("Computing pairwise distances ... ");
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    D[i][j] = Distance(i,j);
printf("done.\n");

clock1 = clock();
/*Compute neighbor lists. */
printf("Sort plants by distance ... ");
// initialize
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    sortedd[i][j] = d[i][j] = distance(i,j);
    sortedplant[i][j] = j;
  }
}

// sort plants by distance
for (i=0; i<N; i++){
  for (j=0; j<(N-1); j++){
           for(k=j+1;k<N;k++){
                 if(sortedd[i][k] < sortedd[i][j]){
                     tmpd = sortedd[i][k];
                     sortedd[i][k] = sortedd[i][j];
                     sortedd[i][j] = tmpd;
                       tmpi = sortedplant[i][k];
                       sortedplant[i][k] = sortedplant[i][j];
                       sortedplant[i][j] = tmpi;
                 }
           }
  }
}

printf("done. \n");
clock2 = clock();
t1 = (double) (clock2 - clock1)/CLOCKS_PER_SEC;

/* intial value for theta, mu and tau*/
curtheta = 0.04;
```

```
curmu = 0.003;
cursigma = 1.0;
for (i=0; i<N; i++) {
  curtau[i] = (L[i]+U[i])/2.0;
  sumtau[i] = 0;
}

sumtheta = summu = sumsigma = 0.0;
propth = propmu = proptau = accth = accmu = acctau = propsigma =  accsigma = 0;
curlogpi = logpi3(curtheta,curmu,curtau,cursigma);

/* Run the Markov chain! */
printf("Running chain, for %d iterations (burn-in %d) ...\n", M, B);
for (t=1; t<=M; t++) {
  printf(" t=%d: \n", t);
  fflush(stdout);

  clock1 = clock();
  /* Propose new theta value. */
  propth++;
  newtheta = curtheta + sigma1 * normal();
  if(newtheta > 0){
           newlogpi = logpi3(newtheta,curmu,curtau,cursigma);
           if ( log(uniform()) < (newlogpi - curlogpi)) {
                   accth++;
                   curtheta = newtheta;
                   curlogpi = newlogpi;
            }
            theta[t-1] = curtheta;
  }

  clock2 = clock();
  /* Propose new mu value. */
  propmu++;
  newmu = curmu + sigma2 * normal();
  if(newmu > 0) {
           newlogpi = logpi3(curtheta,newmu,curtau,cursigma);
            if ( log(uniform()) < (newlogpi - curlogpi)){
                   accmu++;
                   curmu = newmu;
                   curlogpi += logPiDiff;
             }
             mu[t-1] = curmu;
  }

  clock3 = clock();

 for (i=0; i<N; i++) {
    if (curtau[i] <= K) {
```

```
                proptau++;
              for (j=0; j<N; j++) {
                if (j==i){
                            newtau[j] = curtau[j] + sigma4 * normal();
        }
                      else
                      newtau[j] = curtau[j];
                }
      if ( (newtau[i] > L[i]) && (newtau[i] <= U[i]) ) {

                      logPiDiff = 0;
            if ( newtau[i] >= curtau[i]) {
                logPiDiff -= curmu * (newtau[i] - curtau[i]);
               tmpsum = 0.0;
                tmpsum2 = 0.0;
                for (h=1; h<N && sortedd[i][h]<=4*cursigma; h++) {
                      k = sortedplant[i][h];
                      if (curtau[k] >= curtau[i]){
                            if (curtau[k] > newtau[i]){
                      /* change in log likelihood for plant k; no contribution to change in log likelihood for plant i
*/
                                  logPiDiff += curtheta * (newtau[i] - curtau[i]) * f(D[i][k], cursigma);
                            }
                            else{

                                  /* change in log likelihood for plant k uninfected*/
                                    logPiDiff += curtheta * (curtau[k] - curtau[i]) * f(D[i][k], cursigma);
                                  /* change in log likelihood for plant k infected*/
                  tmpsum3 = 0.0;
                  if (curtau[k] > curtau[i]){
                                  for (r = 0; r<N&& sortedd[k][r]<=4*cursigma; r++) {
                            l = sortedplant[k][r];
                             if (curtau[l] < curtau[k])
                                              tmpsum3 += f(D[l][k],cursigma);
                            }
                            tmpsum3 = curmu + curtheta*tmpsum3;
                          logPiDiff += log(tmpsum3-curtheta*f(D[i][k],cursigma)) - log(tmpsum3);
                   }
                                  /* contribution to change in log likelikelihood for plant i uninfected*/
                                  logPiDiff -= curtheta * (newtau[i] - curtau[k]) * f(D[i][k], cursigma);
           if (curtau[k] < newtau[i]){
                                  /* contribution to change in log likelikelihood for plant i infected*/
                                  tmpsum2 += f(D[i][k], cursigma);
    }
                            }
                      }
                      else {
                      /* contribution to change in log likelikelihood for plant i infected*/
                      tmpsum += f(D[i][k], cursigma);
```

```
                        }
                  }
                  /* contribution to change in log likelikelihood for plant i uninfected by plants with infection times
less than curtau_i*/
                  logPiDiff -= curtheta * (newtau[i] - curtau[i]) * tmpsum;

                  /* change in log likelikelihood for plant i infected*/
                  logPiDiff = logPiDiff + log(curmu+curtheta*(tmpsum2+tmpsum)) - log(curmu+curtheta*tmpsum);
            }
            else{
                  logPiDiff += curmu * (curtau[i] - newtau[i]);

                  tmpsum = 0.0;
                  tmpsum2 = 0.0;

                  for (h=1; h<N && sortedd[i][h]<=4*cursigma; h++) {
                        k = sortedplant[i][h];
                        if (curtau[k] >= newtau[i]){
                              if (curtau[k] > curtau[i]){
                                    /* change in log likelihood for plant k; no contribution to change in log likelihood
for plant i */
                                    logPiDiff -= curtheta * (curtau[i] - newtau[i]) * f(D[i][k], cursigma);
                              }
                              else{
                                    /* change in log likelihood for plant k */
                                      logPiDiff -= curtheta * (curtau[k] - newtau[i]) * f(D[i][k], cursigma);

                                    tmpsum3 = 0.0;
                                        if (curtau[k] > newtau[i]){
                                       for (r = 0; r<N&& sortedd[k][r]<=4*cursigma; r++) {
                                  l = sortedplant[k][r];
                                   if (curtau[l] < curtau[k])
                                                tmpsum3 += f(D[l][k],cursigma);
                               }
                                 tmpsum3 = curmu + curtheta*tmpsum3;
                               logPiDiff += log(tmpsum3+curtheta*f(D[i][k],cursigma)) - log(tmpsum3);
                         }
                                    /* contribution to change in log likelikelihood for plant i uninfected*/
                        logPiDiff += curtheta * (curtau[i] - curtau[k]) * f(D[i][k], cursigma);
                              if (curtau[k] < curtau[i]){
                                    /* contribution to change in log likelikelihood for plant i infected*/
                                       tmpsum2 += f(D[i][k], cursigma);
                         }
                              }
                        }
                        else{
                        /* contribution to change in log likelikelihood for plant i infected*/
                        tmpsum += f(D[i][k], cursigma);
                        }
```

```
                 }

                 /* contribution to change in log likelikelihood for plant i uninfected by plants with infection times
less than newtau_i*/
                 logPiDiff += curtheta * (curtau[i] - newtau[i]) * tmpsum;

                 /* change in log likelikelihood for plant i infected*/
                 logPiDiff = logPiDiff + log(curmu+curtheta*tmpsum) - log(curmu+curtheta*(tmpsum2+tmpsum));
            }
    if ( log(uniform()) < logPiDiff ) {
        acctau++;
        curtau[i] = newtau[i];
        curlogpi = curlogpi + logPiDiff;
    }
   }
  }
  tau[i][t-1] = curtau[i];
  if (((t % 500) == 0) && t > B){
        tauVal[N*(t/500-3)+i] = curtau[i];
  }
}

  clock4 = clock();
  /* Propose new sigma value. */
  propsigma++;
  newsigma = cursigma+ sigma3 * normal();
  if(newsigma > 0){
           newlogpi = logpi3(curtheta,curmu,curtau,newsigma);
           if ( log(uniform()) < (newlogpi-curlogpi)) {
                   accsigma++;
                   cursigma = newsigma;
                   curlogpi = newlogpi;
            }
            sigma[t-1] = cursigma;
  }

  clock5 = clock();
  /* Update our running sums. */
  if (t > B) {
    sumtheta = sumtheta + curtheta;
    summu = summu + curmu;
    sumsigma = sumsigma + cursigma;
    for (j=0; j<N; j++)
      sumtau[j] = sumtau[j] + curtau[j];
  }
  t2 = t2 + (double) (clock2 - clock1)/CLOCKS_PER_SEC;
  t3 = t3 + (double) (clock3 - clock2)/CLOCKS_PER_SEC;
  t4 = t4 + (double) (clock4 - clock3)/CLOCKS_PER_SEC;
  t5 = t5 + (double) (clock5 - clock4)/CLOCKS_PER_SEC;
```

```
}

/* Output the results. */
if ((fp = fopen("theta","w")) == NULL){
           fprintf(stderr,"Unable to write file 'theta'.\n");
           exit(1);
}
fprintf(fp,"theta=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",theta[i]);
fprintf(fp,"%f ) \n",theta[M-1]);
fclose(fp);

if ((fp = fopen("mu","w")) == NULL){
           fprintf(stderr,"Unable to write file 'mu'.\n");
           exit(1);
}
fprintf(fp,"mu=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",mu[i]);
fprintf(fp,"%f ) \n",mu[M-1]);
fclose(fp);

if ((fp = fopen("tau_infected","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'tau_infected'.\n");
    exit(1);
}
for(i=0;i<N;i++){
           if(curtau[i] <= K){
                 fprintf(fp,"tau[%d,] = c(",(i+1));
                 for (j=0;j<(M-1);j++){
                       fprintf(fp,"%d,",tau[i][j]);
                 }
                 fprintf(fp,"%d); \n ",tau[i][M-1]);
           }
}
fclose(fp);

if ((fp = fopen("sigma","w")) == NULL){
           fprintf(stderr,"Unable to write file 'sigma'.\n");
           exit(1);
}
fprintf(fp,"sigma=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",sigma[i]);
fprintf(fp,"%f ) \n",sigma[M-1]);
fclose(fp);

if ((fp = fopen("tauVal","w")) == NULL){
```

```
           fprintf(stderr,"Unable to write file 'tauVal'.\n");
           exit(1);
}
fprintf(fp,"tauVal=c(");
for(i=0;i<N;i++){
        for (j = 21; j < 221; j++){
                 if (tau[i][50*j] != 35.0){
                    fprintf(fp,"%f,\n",tau[i][50*j]);
        }
    }
}
fprintf(fp,"%f ) \n",0.000);
fclose(fp);

if ((fp = fopen("tau_est","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'tau.est'.\n");
    exit(1);
}
fprintf(fp,"tau_est=c(");
for (i=0; i<(N-1); i++){
           fprintf(fp, "%f,\n",((double)sumtau[i])/(M-B));
}
fprintf(fp,"%f); \n",(double)sumtau[N-1]/(M-B));
fclose(fp);

if ((fp = fopen("out","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'out'.\n");
    exit(1);
}
fprintf(fp,"M=%d;\n",M);
fprintf(fp,"B=%d;\n",B);
fprintf(fp,"N=%d;\n",N);
fprintf(fp,"ARtheta=%f;\n",((double)accth)/propth);
fprintf(fp,"ARmu=%f;\n",  ((double)accmu)/propmu);
fprintf(fp,"ARtau=%f;\n", ((double)acctau)/proptau );
fprintf(fp,"ARsigma=%f;\n", ((double)accsigma)/propsigma );
fprintf(fp,"Mean_theta=%f;\n", sumtheta/(M-B));
fprintf(fp,"Mean_mu=%f;\n", summu/(M-B));
fprintf(fp,"Mean_sigma=%f;\n", sumsigma/(M-B));
fprintf(fp,"it takes %f seconds to sort plants; \n", t1);
fprintf(fp,"it takes %f seconds to update theta; \n", t2);
fprintf(fp,"it takes %f seconds to update mu; \n", t3);
fprintf(fp,"it takes %f seconds to update tau; \n", t4);
fprintf(fp,"it takes %f seconds to update sigma; \n", t5);
fclose(fp);

printf("\n done.\n");
return(0);
}
```

```
/* pmo: function which returns +1 or -1, with probability 1/2 each. */
int pmo() {
  if (uniform() < 0.5){
           return(-1);
  }
  return (1);
}
```

```
/*the target log density: not update lamda for already infected plants, advanced truncation.*/
double logpi3(double thetheta, double themu, double thetau[], double thesigma) {
  int ii, jj, kk,hh;
  double tmpsum, tmpsum2, tmpsum3;
  for (ii=0; ii<N; ii++) {
    if ( (thetau[ii] <= L[ii]) || (thetau[ii] > U[ii]) ) {
           return(-infinity);
    }
  }
  if(thetheta <= 0.0 || thesigma <= 0.0)
            return (-infinity);
  tmpsum2 = -(a1+1.0)*thetheta - b1/thetheta -(a2+1.0)*themu - b2/themu -(a3+1.0)*thesigma - b3/thesigma;
      for (ii = 0; ii < N; ii++){
                 /* sum over plants uninfected throughout whole period*/
                 tmpsum = 0.0;
                 if (thetau[ii] > K){
                        for (jj=0; jj<N && sortedd[ii][jj]<=4*thesigma; jj++) {
                                   hh = sortedplant[ii][jj];
                              if (thetau[hh] < K+1)
                       tmpsum += (K-thetau[hh])*f(D[ii][hh],thesigma);
                              }
                        tmpsum = themu*K + thetheta*tmpsum;
                        if(tmpsum <= 0.0) return (-infinity);
                          tmpsum2 = tmpsum2 - tmpsum;
                 }
                 /* sum over plants infected throughout time period */
```

```
                else{
                      /* log likelihood that the plant was not infected before time tau_i */
                      tmpsum = 0.0;
                      tmpsum3 = 0.0; /* to keep track of all plants infected before time tau_u */
                      for (jj=0; jj<N && sortedd[ii][jj]<=4*thesigma; jj++) {
                                  hh = sortedplant[ii][jj];
                             if (thetau[hh] < thetau[ii]){
                      tmpsum += (thetau[ii]-thetau[hh])*f(D[ii][hh],thesigma);
                            tmpsum3 += f(D[ii][hh],thesigma);
                            }
                             }
                       tmpsum = themu*thetau[ii] + thetheta*tmpsum;
                       if(tmpsum <= 0.0) return (-infinity);
                         tmpsum2 = tmpsum2 - tmpsum;
                       /* log likelihood that the plant was infected at time tau_i */
                       tmpsum3 = themu + thetheta*tmpsum3;
                       if(tmpsum3 <= 0.0) return (-infinity);
                         tmpsum2 = tmpsum2 + log(tmpsum3);
                }
          }
  return(tmpsum2);
}


/* SEEDRAND: SEED RANDOM NUMBER GENERATOR. */
void seedrand()
{
    SYSTEMTIME str_t;
    double helper;
    GetSystemTime(&str_t);
    int seed;
    seed = (int) str_t.wMilliseconds;
    srand(seed);
}

double sq(double xxx)
{
  return(xxx*xxx);
}

double uniform()
{
    return((double)rand()/RAND_MAX);
}

double exponential()
{
    double uniform();
    return( -log(uniform()) );
```

```
}

/* NORMAL:  return a standard normal random number. */
double normal()
{
    double RRR, ttt, uniform();
    RRR = - log(uniform());
    ttt = 2 * PI * uniform();
    return( sqrt(2*RRR) * cos(ttt));
}

int ifloor(double xxx)
{
    return((int)floor(xxx));
}

int iround(double xxx)
{
    return( ifloor(xxx+0.5) );
}

int imin(int iii, int jjj)
{
    if (iii < jjj)
      return(iii);
    return(jjj);
}

double dmax(double xxx,double yyy)
{
  if(xxx<yyy) return (yyy);
  return (xxx);
}

double f(double xxx,double thesigma)
{
  return (exp(-sq(xxx)/(2*sq(thesigma)))/pow(2*PI*sq(thesigma),0.5));
}

double ftr(int ii, int jj, double thesigma)
{
  double Dist = D[ii][jj], dist = d[ii][jj];
  if( dist > (4.0*thesigma) ) {
           return (0.0);
  }
  return (exp(-sq(Dist)/(2*sq(thesigma)))/pow(2*PI*sq(thesigma),0.5));
}

double Distance(int ii, int jj){
```

```
            return ( sqrt( sq(x[ii]-x[jj]) + sq(y[ii]-y[jj]) ) );
}

double distance(int ii, int jj){
            return ( dmax(fabs(x[ii]-x[jj]), fabs(y[ii]-y[jj])) );
}
```

C Program for Metropolis within Gibbs for Continuous Time Model, including Simplifications, and no Truncated Kernel

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <Windows.h>
#include <unistd.h>

#define MAXN 1742
#define K 30
#define M 1000
#define B 100
#define infinity 999999999.0
#define PI 3.1415926536

int N, sortedplant[MAXN][MAXN];
double L[MAXN], U[MAXN];
double lambda[MAXN][K], D[MAXN][MAXN],tau[MAXN][M],x[MAXN], y[MAXN],d[MAXN][MAXN],sortedd[MAXN][MAXN];
double sq(),logpi(),logpi2(),logpi3(), uniform(),exponential(),normal(),f(),dmax(),ftr(), Distance(), distance();
int pmo(),ifloor(),iround(),imin();
double a1 = 1.0, b1 = 0.05, a2 = 1.0, b2 = 0.01, a3 = 1.0, b3 = 1.0 ;
void seedrand();

int main(int argc, char **argv)
{
int i, j, k, l, h, t, r, propmu, propth, proptau, accmu, accth, acctau, propsigma, accsigma, tmpi;
double curtheta, curmu, newmu, newtheta, summu, sumtheta, logPiDiff, newlogpi, curlogpi, cursigma, newsigma, sumsigma,
tmpd, tmpsum, tmpsum2, tmpsum3;
int S6[MAXN], S10[MAXN], S14[MAXN], S19[MAXN], S23[MAXN], S30[MAXN];
double curtau[MAXN], newtau[MAXN], sumtau[MAXN];
double a1, a2, a3, b1, b2, b3;
double theta[M],mu[M],sigma[M], tauVal[34840];
char tmpstring[20];
FILE *fp;
clock_t  clock1, clock2, clock3, clock4, clock5;
double t1 = 0.0, t2 = 0.0, t3 = 0.0, t4 = 0.0, t5 = 0.0;

/*set the proposal scale for theta and mu*/
double sigma1 = 0.005, sigma2 = 0.0005, sigma3 = 0.07, sigma4 = 1.0;
/* set prior distribution parameters */


/* Seed the random number generator. */
seedrand();
/* Read in the data. */
printf("Reading data ...");
```

```
if ((fp = fopen("C:/Alexander/University/2011-2012/STA496/Programs/canedata.txt","r")) == NULL) {
    fprintf( stderr, "Unable to read file 'canedata'.\n");
    exit(1);
}
N = 0;
for (i=0; i<MAXN; i++) {
    fscanf(fp, "%s", &tmpstring);
    x[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    y[N] = atof(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S6[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S10[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S14[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S19[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S23[N] = atoi(tmpstring);
    fscanf(fp, "%s", &tmpstring);
    S30[N] = atoi(tmpstring);
//    if( x[N]<10.0 && y[N]<10.0 )
             N++;
}
fclose(fp);
printf("done. \n");
printf("Number of sites studied: %d\n", N);

/* Determine the L_x and U_x values, etc. */
printf("Computing infection time ranges ...");
for (i=0; i<N; i++) {
  if (S6[i]) {
    L[i] = 0.0;
    U[i] = 6.0;
  } else if (S10[i]) {
    L[i] = 6.0;
    U[i] = 10.0;
  } else if (S14[i]) {
    L[i] = 10.0;
    U[i] = 14.0;
  } else if (S19[i]) {
    L[i] = 14.0;
    U[i] = 19.0;
  } else if (S23[i]) {
    L[i] = 19.0;
    U[i] = 23.0;
  } else if (S30[i]) {
    L[i] = 23.0;
```

```
    U[i] = 30.0;
  } else {
    L[i] = 30.0;
    U[i] = 30.0+10.0;
  }
}
printf("done. \n");

/* Compute the cane-cane distances. */
printf("Computing pairwise distances ... ");
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    D[i][j] = Distance(i,j);
printf("done.\n");

clock2 = clock();

/* intial value for theta, mu and tau*/
/*curtheta = 10 * exponential();
curmu = 10 * normal();*/
curtheta = 0.04;
curmu = 0.003;
cursigma = 1.0;
for (i=0; i<N; i++) {
  curtau[i] = (L[i]+U[i])/2.0;
  //curtau[i] = U[i];
  sumtau[i] = 0;
  //printf("if saved in float, tau is %f, the correct way is %d",curtau[i],curtau[i]);
}

sumtheta = summu = sumsigma = 0.0;
propth = propmu = proptau = accth = accmu = acctau = propsigma =  accsigma = 0;
curlogpi = logpi3(curtheta,curmu,curtau,cursigma);

/* Run the Markov chain! */
printf("Running chain, for %d iterations (burn-in %d) ...\n", M, B);
for (t=1; t<=M; t++) {
  printf(" t=%d: \n", t);
  printf("VVVVV %f,\n",curtheta);
  fflush(stdout);

  clock1 = clock();
  /* Propose new theta value. */
  propth++;
  newtheta = curtheta + sigma1 * normal();
  if(newtheta > 0){
           newlogpi = logpi3(newtheta,curmu,curtau,cursigma);
//         printf("VVVVV %f,\n",curtheta);
//           Sleep(10);
```

```
        if ( log(uniform()) < (newlogpi - curlogpi)) {
                accth++;
                curtheta = newtheta;
                curlogpi = newlogpi;
         }
         theta[t-1] = curtheta;
}


clock2 = clock();
/* Propose new mu value. */
propmu++;
newmu = curmu + sigma2 * normal();
if(newmu > 0) {
        newlogpi = logpi3(curtheta,newmu,curtau,cursigma);
//      printf("VVVVV %f,\n",curmu);
//        Sleep(10);
         if ( log(uniform()) < (newlogpi - curlogpi)){
                accmu++;
                curmu = newmu;
                curlogpi += logPiDiff;
          }
          mu[t-1] = curmu;
}

clock3 = clock();

for (i=0; i<N; i++) {


  if (curtau[i] <= K) {
              proptau++;
            for (j=0; j<N; j++) {
              if (j==i){
                          newtau[j] = curtau[j] + sigma4 * normal();
      }
                    else
                    newtau[j] = curtau[j];
              }
   if ( (newtau[i] > L[i]) && (newtau[i] <= U[i]) ) {

                    logPiDiff = 0;
         if ( newtau[i] >= curtau[i]) {
              logPiDiff -= curmu * (newtau[i] - curtau[i]);
             tmpsum = 0.0;
              tmpsum2 = 0.0;
              for (k=0; k<N; k++) {
         if (k != i){
```

```
                        if (curtau[k] >= curtau[i]){
                              if (curtau[k] > newtau[i]){
                              /* change in log likelihood for plant k; no contribution to change in log likelihood for
plant i */
                                    logPiDiff += curtheta * (newtau[i] - curtau[i]) * f(D[i][k], cursigma);
                              }
                              else{

                                    /* change in log likelihood for plant k uninfected*/
                                      logPiDiff += curtheta * (curtau[k] - curtau[i]) * f(D[i][k], cursigma);
                                    /* change in log likelihood for plant k infected*/
                        tmpsum3 = 0.0;
                        if (curtau[k] > curtau[i]){
                                    for (l = 0; l<N; l++) {
                                     if (curtau[l] < curtau[k])
                                                tmpsum3 += f(D[l][k],cursigma);
                              }
                              tmpsum3 = curmu + curtheta*tmpsum3;
                            logPiDiff += log(tmpsum3-curtheta*f(D[i][k],cursigma)) - log(tmpsum3);
                         }
                                    /* contribution to change in log likelikelihood for plant i uninfected*/
                                    logPiDiff -= curtheta * (newtau[i] - curtau[k]) * f(D[i][k], cursigma);
                        if (curtau[k] < newtau[i]){
                                                /* contribution to change in log likelikelihood for plant i infected*/
                                                tmpsum2 += f(D[i][k], cursigma);
                          }
    //                              printf("k VALUE %f \n", (double)k);
                              }
                            }
                        else {
                        /* contribution to change in log likelikelihood for plant i infected*/
                        tmpsum += f(D[i][k], cursigma);
                        }
                }
             }
                  /* contribution to change in log likelikelihood for plant i uninfected by plants with infection times
less than curtau_i*/
                  logPiDiff -= curtheta * (newtau[i] - curtau[i]) * tmpsum;

                  /* change in log likelikelihood for plant i infected*/
                  logPiDiff = logPiDiff + log(curmu+curtheta*(tmpsum2+tmpsum)) - log(curmu+curtheta*tmpsum);
            }
            else{
                  logPiDiff += curmu * (curtau[i] - newtau[i]);

                  tmpsum = 0.0;
                  tmpsum2 = 0.0;

                  for (k=0; k<N; k++) {
```

```
            if (k!= i){
                        if (curtau[k] >= newtau[i]){
                              if (curtau[k] > curtau[i]){
                                    /* change in log likelihood for plant k; no contribution to change in log likelihood
for plant i */
                                    logPiDiff -= curtheta * (curtau[i] - newtau[i]) * f(D[i][k], cursigma);
                              }
                              else{
                                    /* change in log likelihood for plant k */
                                      logPiDiff -= curtheta * (curtau[k] - newtau[i]) * f(D[i][k], cursigma);

                                    tmpsum3 = 0.0;
                                        if (curtau[k] > newtau[i]){
                                       for (l = 0; l<N; l++) {
                                         if (curtau[l] < curtau[k])
                                                tmpsum3 += f(D[l][k],cursigma);
                               }
                                 tmpsum3 = curmu + curtheta*tmpsum3;
                               logPiDiff += log(tmpsum3+curtheta*f(D[i][k],cursigma)) - log(tmpsum3);
                             }
                                    /* contribution to change in log likelikelihood for plant i uninfected*/
                            logPiDiff += curtheta * (curtau[i] - curtau[k]) * f(D[i][k], cursigma);
                              if (curtau[k] < curtau[i]){
                                    /* contribution to change in log likelikelihood for plant i infected*/
                                       tmpsum2 += f(D[i][k], cursigma);
                             }
                              }
                        }
                        else{
                        /* contribution to change in log likelikelihood for plant i infected*/
                        tmpsum += f(D[i][k], cursigma);
                        }
            }
                 }

                 /* contribution to change in log likelikelihood for plant i uninfected by plants with infection times
less than newtau_i*/
                 logPiDiff += curtheta * (curtau[i] - newtau[i]) * tmpsum;

                 /* change in log likelikelihood for plant i infected*/
                 logPiDiff = logPiDiff + log(curmu+curtheta*tmpsum) - log(curmu+curtheta*(tmpsum2+tmpsum));

            }
    if ( log(uniform()) < logPiDiff ) {
        acctau++;
        curtau[i] = newtau[i];
        curlogpi = curlogpi + logPiDiff;
    }
   }
```

```
  }
  tau[i][t-1] = curtau[i];
}

  clock4 = clock();
  /* Propose new sigma value. */
  propsigma++;
  newsigma = cursigma+ sigma3 * normal();
  if(newsigma > 0){
           newlogpi = logpi3(curtheta,curmu,curtau,newsigma);
                                  //  printf("UUUU %f,\n",curlogpi);
        //         printf("VVVVV %f,\n",cursigma);
        //   Sleep(100);
           if ( log(uniform()) < (newlogpi-curlogpi)) {
                   accsigma++;
                   cursigma = newsigma;
                   curlogpi = newlogpi;
            }
            sigma[t-1] = cursigma;
  }

  clock5 = clock();
  /* Update our running sums. */
  if (t > B) {
    sumtheta = sumtheta + curtheta;
    summu = summu + curmu;
    sumsigma = sumsigma + cursigma;
    for (j=0; j<N; j++)
      sumtau[j] = sumtau[j] + curtau[j];
  }
  t2 = t2 + (double) (clock2 - clock1)/CLOCKS_PER_SEC;
  t3 = t3 + (double) (clock3 - clock2)/CLOCKS_PER_SEC;
  t4 = t4 + (double) (clock4 - clock3)/CLOCKS_PER_SEC;
  t5 = t5 + (double) (clock5 - clock4)/CLOCKS_PER_SEC;
}

/* Output the results. */
if ((fp = fopen("theta","w")) == NULL){
           fprintf(stderr,"Unable to write file 'theta'.\n");
           exit(1);
}
fprintf(fp,"theta=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",theta[i]);
fprintf(fp,"%f ) \n",theta[M-1]);
fclose(fp);

if ((fp = fopen("mu","w")) == NULL){
           fprintf(stderr,"Unable to write file 'mu'.\n");
```

```
            exit(1);
}
fprintf(fp,"mu=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",mu[i]);
fprintf(fp,"%f ) \n",mu[M-1]);
fclose(fp);

if ((fp = fopen("tau_infected","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'tau_infected'.\n");
    exit(1);
}
for(i=0;i<N;i++){
            if(curtau[i] <= K){
                  fprintf(fp,"tau[%d,] = c(",(i+1));
                  for (j=0;j<(M-1);j++){
                        fprintf(fp,"%d,",tau[i][j]);
                  }
                  fprintf(fp,"%d); \n ",tau[i][M-1]);
            }
}
fclose(fp);

if ((fp = fopen("sigma","w")) == NULL){
            fprintf(stderr,"Unable to write file 'sigma'.\n");
            exit(1);
}
fprintf(fp,"sigma=c(");
for(i=0;i<(M-1);i++)
    fprintf(fp,"%f,\n",sigma[i]);
fprintf(fp,"%f ) \n",sigma[M-1]);
fclose(fp);

if ((fp = fopen("tau_est","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'tau.est'.\n");
    exit(1);
}
fprintf(fp,"tau_est=c(");
for (i=0; i<(N-1); i++){
            fprintf(fp, "%f,\n",((double)sumtau[i])/(M-B));
}
fprintf(fp,"%f); \n",(double)sumtau[N-1]/(M-B));
fclose(fp);

if ((fp = fopen("out","w")) == NULL) {
    fprintf(stderr, "Unable to write file 'out'.\n");
    exit(1);
}
fprintf(fp,"M=%d;\n",M);
```

```
fprintf(fp,"B=%d;\n",B);
fprintf(fp,"N=%d;\n",N);
fprintf(fp,"ARtheta=%f;\n",((double)accth)/propth);
fprintf(fp,"ARmu=%f;\n",  ((double)accmu)/propmu);
fprintf(fp,"ARtau=%f;\n", ((double)acctau)/proptau );
fprintf(fp,"ARsigma=%f;\n", ((double)accsigma)/propsigma );
fprintf(fp,"Mean_theta=%f;\n", sumtheta/(M-B));
fprintf(fp,"Mean_mu=%f;\n", summu/(M-B));
fprintf(fp,"Mean_sigma=%f;\n", sumsigma/(M-B));
fprintf(fp,"it takes %f seconds to update theta; \n", t2);
fprintf(fp,"it takes %f seconds to update mu; \n", t3);
fprintf(fp,"it takes %f seconds to update tau; \n", t4);
fprintf(fp,"it takes %f seconds to update sigma; \n", t5);
fclose(fp);

printf("\n done.\n");
return(0);
}

/* pmo: function which returns +1 or -1, with probability 1/2 each. */
int pmo() {
  if (uniform() < 0.5){
           return(-1);
  }
  return (1);
}
```

```
/*the target log density: not update lamda for already infected plants, advanced truncation.*/
double logpi3(double thetheta, double themu, double thetau[], double thesigma) {
  int ii, jj, kk,hh;
  double tmpsum, tmpsum2, tmpsum3;
  for (ii=0; ii<N; ii++) {
    if ( (thetau[ii] <= L[ii]) || (thetau[ii] > U[ii]) ) {
           return(-infinity);
    }
```

```
  }
  if(thetheta <= 0.0 || thesigma <= 0.0)
            return (-infinity);
  tmpsum2 = -(a1+1.0)*thetheta - b1/thetheta -(a2+1.0)*themu - b2/themu -(a3+1.0)*thesigma - b3/thesigma;
      for (ii = 0; ii < N; ii++){
                 /* sum over plants uninfected throughout whole period*/
                 tmpsum = 0.0;
                 if (thetau[ii] > K){
                        for (hh=0; hh<N; hh++) {
                              if (thetau[hh] < K+1)
                       tmpsum += (K-thetau[hh])*f(D[ii][hh],thesigma);
                              }
                        tmpsum = themu*K + thetheta*tmpsum;
                        if(tmpsum <= 0.0) return (-infinity);
                          tmpsum2 = tmpsum2 - tmpsum;
                 }
                 /* sum over plants infected throughout time period */
                 else{
                       /* log likelihood that the plant was not infected before time tau_i */
                       tmpsum = 0.0;
                       tmpsum3 = 0.0; /* to keep track of all plants infected before time tau_u */
                       for (hh=0; hh<N; hh++) {
                              if (thetau[hh] < thetau[ii]){
                       tmpsum += (thetau[ii]-thetau[hh])*f(D[ii][hh],thesigma);
                             tmpsum3 += f(D[ii][hh],thesigma);
                             }
                              }
                        tmpsum = themu*thetau[ii] + thetheta*tmpsum;
                        if(tmpsum <= 0.0) return (-infinity);
                          tmpsum2 = tmpsum2 - tmpsum;
                        /* log likelihood that the plant was infected at time tau_i */
                        tmpsum3 = themu + thetheta*tmpsum3;
                        if(tmpsum3 <= 0.0) return (-infinity);
                          tmpsum2 = tmpsum2 + log(tmpsum3);
                 }
           }
  return(tmpsum2);
}


/* SEEDRAND: SEED RANDOM NUMBER GENERATOR. */
void seedrand()
{
    SYSTEMTIME str_t;
    double helper;
    GetSystemTime(&str_t);
    int seed;
    seed = (int) str_t.wMilliseconds;
    srand(seed);
```

```
}

double sq(double xxx)
{
  return(xxx*xxx);
}

double uniform()
{
    return((double)rand()/RAND_MAX);
}

double exponential()
{
    double uniform();
    return( -log(uniform()) );
}

/* NORMAL:  return a standard normal random number. */
double normal()
{
    double RRR, ttt, uniform();
    RRR = - log(uniform());
    ttt = 2 * PI * uniform();
    return( sqrt(2*RRR) * cos(ttt));
}

int ifloor(double xxx)
{
    return((int)floor(xxx));
}

int iround(double xxx)
{
    return( ifloor(xxx+0.5) );
}

int imin(int iii, int jjj)
{
    if (iii < jjj)
      return(iii);
    return(jjj);
}

double dmax(double xxx,double yyy)
{
  if(xxx<yyy) return (yyy);
  return (xxx);
}
```

```
double f(double xxx,double thesigma)
{
  return (exp(-sq(xxx)/(2*sq(thesigma)))/pow(2*PI*sq(thesigma),0.5));
}

double ftr(int ii, int jj, double thesigma)
{
  double Dist = D[ii][jj], dist = d[ii][jj];
  return (exp(-sq(Dist)/(2*sq(thesigma)))/pow(2*PI*sq(thesigma),0.5));
}

double Distance(int ii, int jj){
           return ( sqrt( sq(x[ii]-x[jj]) + sq(y[ii]-y[jj]) ) );
}

double distance(int ii, int jj){
           return ( dmax(fabs(x[ii]-x[jj]), fabs(y[ii]-y[jj])) );
}
```