Perfect Implementation of Simulation for Conditioned Boolean Model via Correlated Poisson Random Variables

by

Y. Cai and W.S. Kendall

July 1999

This is paper1/nuweb/bl.w (unix). Last edited: 13:26:58, 23/06/1999.

Perfect Implementation of Simulation for Conditioned Boolean Model via Correlated Poisson Random Variables

Yuzhi Cai and Wilfrid S. Kendall

Department of Statistics
University of Warwick
Coventry CV4 7AL
UK

Abstract

This document describes a `C` implementation of an algorithm for a perfect simulation of a Boolean model conditioned to cover a set of points $S = \{x_1, x_2, ..., x_k\}$ in a bounded window W. This is done by using perfect simulation for a set of correlated Poisson random variables conditioned all to be non-zero. Mathematical details are to be found in [1]. In this document, full listing of the programs are given.

Contents

Here is version information for this document:

This is `paper1/nuweb/bl.w` (`unix`). Last edited: 13:26:58, 23/06/1999.

⟨revision information 2⟩ ≡

```
 * Code derived from $Source: /home/bach/yc/papers/perfect/paper1/nuweb/bl.w,v $
 *                    $Revision: 1.1 $
 * $Date: 1999/06/23 13:26:58 $.◇
```

Macro referenced in scraps 4, 5, 11, 12, 14, 17ab, 19ab, 21, 22, 24, 25, 26, 29, 30, 34, 37, 38, 42, 44, 46, 49, 50, 53, 55, 58, 60, 61, 63, 65, 68, 69.

Chapter 1

Introduction

This document describes a C implementation of a perfect algorithm for a conditional Boolean model simulation via correlated Poisson random variables conditioned to be positive.

The main steps for the simulation are as follows:

1. Simulate an ordinary Poisson process of disk of intensity λ in a bounded window in W.

2. Record discs which do not cover any conditioning points and throw away those which cover the conditioning points.

3. Get incident times of Y_A and Z_A for all $A \subseteq S$.

4. Do perfect simulation for correlated Poisson random variables

$$(X_{\{x_1\}}, \dots, X_{\{x_k\}})$$

conditioned to be positive.

5. Place Y_A number of germs in the region E_A uniformly at random, where

$$E_A \quad = \quad \left(\bigcap_{x \in A} x \oplus \check{G} \right) \setminus \left(\bigcup_{y \notin A} y \oplus \check{G} \right)$$

In the following, we present the programming details.

Chapter 2

Programming

2.1 Obtain index set for each $Y_{A_i} \subseteq S$

Since $X_{\{x\}} = \bigcup_{A:x\in A} Y_A$, we need to know which Y_A will make any contributions to $X_{\{x\}}$. In order to do so, for each Y_{A_i}, we generate a binary vector, corresponding to each i. For example, if $i = 2$, then the binary vector will be the k dimentional vector $(0, 1, 0, \ldots, 0)$, which shows that $A_2 = \{x_2\}$. Note that this mapping is $1 - 1$.

However, if k is large, $2^k - 1$ will be very large. But among all these processes Y_{A_i}, most of them will have $\mu_i = 0$, which occurs if at least two points in A_i are apart from each other with distance $2r$. In this case, this process will never contribute to any $X_{\{x_i\}}$, $i = 1, \ldots, k$. Therefore, although the number of Y_{A_i} is very large, we only need to store the binary vector for a small number of the processes, saving a large amount of storage space.

In the following, program `distas.c` calculates the distance between two points; `subset2.c` stores the necessary information about the processes Y_{A_i} with $\mu_{A_i} \neq 0$; `indx_sxi.c` obtains the information about each $X_{\{x_i\}}$ which can be used directly in deciding whether we should perpetuate a point.

"distas.c" 4 $\equiv$

```
/*
 * File distas.c ⟨revision information 2⟩
 */



/**********************************************************************
 * distance.c --- return the distance between two points in W.        *
 **********************************************************************/
```

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

double
dist (double x1, double x2, double y1, double y2)
{
  return (sqrt (pow (x1 - x2, 2) + pow (y1 - y2, 2)));
}
```

◇

"subset2.c" 5 ≡

```
/*
 * File subset2.c ⟨revision information 2⟩
 */


/*********************************************************************
 * subset.c --- return the binary index for each subset A in S.        *

 * subset[][] --- store the binary index for each A in S.              *
 *********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>

typedef struct SNewarray
  {
    struct SNewarray *next_ptr;
    double ua;
    int yi;
    int subset[300];
  }
Newarray;

extern long idum;
extern int old_number_yi;
extern double **condist;
extern double radius;
extern int MAX_K;
```

```
int **newsubset, number_yi, *newyi;
double *newua;

int bit ();
int comp ();
void qqsort ();

void
bisubset ()
{
  int i, k, i1, ii, i2, index;
  double m, n;
  int *bisub, *jin, *temvec, **newsubset1;
  double *newua1;
  Newarray *first, *current, *tem_first;

  if (MAX_K + 1 > 300)
    {
      printf ("MAX_K+1 > 300 in  subset2.c, exit!\n");
      exit (8);
    }

  bisub = (int *) malloc ((MAX_K + 1) * sizeof (int));
  jin = (int *) malloc ((MAX_K + 1) * sizeof (int));

  i1 = 1;
  n = 1;
  first = (Newarray *) malloc (sizeof (Newarray));
  first->next_ptr = NULL;
  first->ua = 2;
  m = n;
  k = 1;
  for (i = 1; i <= MAX_K; i++)
    bisub[i] = 0;

  while (k != 0)
    k = bit (bisub, &m);
  for (i = 1; i <= MAX_K; i++)
    first->subset[i] = bisub[i];
  first->yi = n;
  ++i1;

  for (i2 = 2; i2 <= MAX_K; i2++)
    {
      for (i = 1; i <= MAX_K; i++)
        jin[i] = 0;
      tem_first = first;
```

```
while (tem_first != NULL)
  {
    ii = 1;
    for (i = 1; i <= MAX_K; i++)
      {
        if (tem_first->subset[i] != 0)
          {
            jin[ii] = i - 1;
            ++ii;
          }
      }

    index = 1;
    n = pow (2, (i2 - 1));
    for (i = 1; i < ii; i++)
      {
        n += pow (2, jin[i]);
        if (condist[i2 - 1][jin[i]] < 2 * radius)
          ++index;
      }

    if (index == ii)
      {
        current = (Newarray *) malloc (sizeof (Newarray));
        current->ua = 2;
        current->yi = n;
        for (i = 1; i <= MAX_K; i++)
          bisub[i] = 0;
        m = n;
        k = 1;
        while (k != 0)
          k = bit (bisub, &m);
        for (i = 1; i <= MAX_K; i++)
          current->subset[i] = bisub[i];
        current->yi = n;
        current->next_ptr = first;
        first = current;
        ++i1;
      }
    tem_first = tem_first->next_ptr;
  }

n = pow (2, (i2 - 1));
current = (Newarray *) malloc (sizeof (Newarray));
current->ua = 2;
current->yi = n;
for (i = 1; i <= MAX_K; i++)
```

```
      bisub[i] = 0;

    m = n;
    k = 1;
    while (k != 0)
      k = bit (bisub, &m);
    for (i = 1; i <= MAX_K; i++)
      current->subset[i] = bisub[i];
    current->next_ptr = first;
    first = current;
    ++i1;
  }

number_yi = i1 - 1;
free (bisub);
free (jin);

/* put the binary index etc in newarrays */

newsubset1 = (int **) malloc ((number_yi + 1) * sizeof (int *));
if (newsubset1 == NULL)
  {
    printf ("newsubset1 is NULL in subset.c\n");
    exit (8);
  }
for (i = 1; i <= number_yi; i++)
  {
    newsubset1[i] = (int *) malloc ((MAX_K + 1) * sizeof (int));
    if (newsubset1[i] == NULL)
      {
        printf ("newsubset1[i] is NULL in subset.c\n");
        exit (8);
      }
  }

newyi = (int *) malloc ((number_yi + 1) * sizeof (int));
if (newyi == NULL)
  {
    printf ("newyi is NULL in subset.c\n");
    exit (8);
  }

newua1 = (double *) malloc ((number_yi + 1) * sizeof (double));
if (newua1 == NULL)
  {
    printf ("newua1 is NULL in subset.c\n");
    exit (8);
```

```
  }

i = 1;
tem_first = first;
while (tem_first != NULL)
  {
    newua1[i] = tem_first->ua;
    newyi[i] = tem_first->yi;
    for (k = 1; k <= MAX_K; k++)
      newsubset1[i][k] = tem_first->subset[k];
    ++i;
    tem_first = tem_first->next_ptr;
  }

newsubset = (int **) malloc ((number_yi + 1) * sizeof (int *));
if (newsubset == NULL)
  {
    printf ("newsubset is NULL in subset.c\n");
    exit (8);
  }

for (i = 1; i <= number_yi; i++)
  {
    newsubset[i] = (int *) malloc ((MAX_K + 1) * sizeof (int));
    if (newsubset == NULL)
      {
        printf ("newsubset[i] is NULL in subset.c\n");
        exit (8);
      }
  }

newua = (double *) malloc ((number_yi + 1) * sizeof (double));
if (newua == NULL)
  {
    printf ("newua is NULL in subset.c\n");
    exit (8);
  }

temvec = (int *) malloc ((number_yi + 1) * sizeof (int));
if (temvec == NULL)
  {
    printf ("temvec is NULL in subset.c\n");
    exit (8);
  }

for (i = 1; i <= number_yi; i++)
  temvec[i] = i;
```

```
  qqsort (newyi, temvec, 1, number_yi, comp);

  for (i = 1; i <= number_yi; i++)
    {
      newua[i] = newua1[temvec[i]];
      for (k = 1; k <= MAX_K; k++)
        newsubset[i][k] = newsubset1[temvec[i]][k];
    }

  free (newsubset1);
  free (temvec);
  free (newua1);

  tem_first = first->next_ptr;
  while (tem_first != NULL)
    {
      free (first);
      first = tem_first;
      tem_first = tem_first->next_ptr;
    }
  free (first);

}


int
bit (int *bisub, double *mm)
{
  int i;
  double m, x;

  m = *mm;

  for (i = 1; i <= MAX_K; i++)
    {
      x = pow (2, i);
      if (x <= m)
        continue;
      else
        {
          bisub[i] = 1;
          m -= pow (2, (i - 1));
          *mm = m;
          if (m == 1)
            {
              bisub[1] = 1;
```

```
            return (0);
          }

        if (m == 0)
          return (0);

        if (m > 1)
          return (1);
      }
  }
  return (0);
}
```

◇

where we use a quicksort program qqsort.c to sort the data. Note that we use our qqsort.c rather than the C library function qsort because we need to make changes on the array temvec corresponding to the changes in newyi while it is been sorted.

"qqsort.c" 11 ≡

```
/*
 * File qqsort.c ⟨revision information 2⟩
 */


#include <stdio.h>
#include <stdlib.h>

#define swap(x,y) temp=(x); (x)=(y); (y)=temp;

int
comp (int v1, int v2)
{
  if (v1 < v2)

    return (-1);

  else if (v1 > v2)

    return (1);

  else
    return (0);
}
```

```
void
qqsort (int *a, int *b, int left, int right, int comp (int, int))
{
  int i, last;

  double temp;

  if (left >= right)

    return;

  swap (a[left], a[(left + right) / 2])
    swap (b[left], b[(left + right) / 2])

    last = left;

  for (i = left + 1; i <= right; i++)

    if (comp (a[i], a[left]) < 0)
      {
        ++last;

        swap (a[last], a[i])
          swap (b[last], b[i])
      }

  swap (a[left], a[last])
    swap (b[left], b[last])

    qqsort (a, b, left, last - 1, comp);

  qqsort (a, b, last + 1, right, comp);
}
```

◇

"indx_sxi.c" 12 ≡

```
/*
 * File indx_sxi.c ⟨revision information 2⟩
 */


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

```
 * initial_sxi --- return the sets {n_1,...,n_m), where n_i is relavent *
 * to   x_{x_j}.                                                       *
 *                                                                     *
 *                                                                     *
 **********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>

extern int **sxi;
extern int **newsubset;
extern int number_yi;
extern int number_subset_xi;
extern int *newyi;
extern int MAX_K;

void
index_in_sxi ()
{
  int i1, k, i;

  for (k = 0; k < MAX_K; k++)
    {
      i1 = 0;
      for (i = 1; i <= number_yi; i++)
        {
          if (newsubset[i][k + 1] != 0)
            {
              sxi[k][i1] = i;
              ++i1;
            }
        }
      sxi[k][i1] = (-1);
    }
}
```

◇

Now, we are able to obtain all the incident times of processes Y_{A_i} with $\mu_{A_i} \neq 0$ and independent Poisson process Z_{A_i} with unite rate. The details are shown in the next subsection.

2.2 Obtain all the incident times of Y_A and $\mathcal{Z}_A$

According to the analysis in [1] (section 4.2) we can obtain all the incident times of Y_{A_i} and Z_{A_i} as shown in Table 2.1

Specifically, the values of Y_{A_j} at time 0 for $j = 1, 2, \cdots, N$ are obtained by `iniy0.c`.

"iniy0.c" 14 $\equiv$

```
/*
 * File iniy0.c ⟨revision information 2⟩
 */


/**********************************************************************
 *  iniY0.c :  obtain the starting values of the processes Y_{A_i}(0).  *
 *             the starting value follows the equilibrium distribution. *

 * *initial_y0 --- store the initial values of the process  Y_{A_i}.    *
 *                     for i=1,2,...,number_yi+1.                       *
 * initial_y0[number_yi+1] store the starting value of the germ         *
 * process which is Poisson of mean \lambda area(W).                    *
 **********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>


#define RANDOM ran2(&idum)

extern int number_yi;
extern long idum;
extern double intensity;
extern double window;
extern int **newsubset;
extern double **XcondAbs;
extern double radius;
extern int *newyi;
extern int MAX_K;

int number_point;
double poidev ();
double ran2 ();
double dist ();
```

```
Incidents(λ, N):
    L ← ∅
    t_i ← 0
    Y_{A_j} ← Poisson(μ_{A_j}) for j = 1, ···, N
    repeat
        q = λarea(W) + Σ_{j=1}^{N} Y_{A_j}(t_i) + N
        t_{i+1} ← t_i − exponential(q)
        u ← U[0,1]
        if uq ≤ λarea(W)
            immigration = 1
        else if λarea(W) < uq ≤ λarea(W) + Σ_{j=1}^{N} Y_{A_j}
            death = 1
        else
            perpetuation = 1
        if immigration = 1
            generate x ∈ W
            if there is A_j ⊆ S such that x ∈ A_j
                L ← L ⋃{(t_{i+1}, Y_{A_j}−immigration)}
                Y_{A_j}(t_{i+1}) = Y_{A_j}(t_i) + 1
        if death = 1
            u ← U[0,1]
            if Σ_{j=1}^{ℓ−1} Y_{A_j} < uq ≤ Σ_{j=1}^{ℓ} Y_{A_j}
                L ← L ⋃{(t_{i+1}, Y_{A_ℓ}−death)}
                Y_{A_ℓ}(t_{i+1}) = Y_{A_ℓ}(t_i) − 1
        if perpetuation = 1
            u ← U[0,1]
            if j − 1 < uq ≤ j
                L ← L ⋃{(t_{i+1}, Z_{A_j}−perpetuation)}
        t_i ← t_{i+1}
        until t_i < −T
    return L
```

Table 2.1: `Incidents`: Algorithm for obtaining incident times of unconditioned immigration-death processes $\{Y_A\}$ and associated perpetuation Poisson processes $\{\mathcal{Z}_A\}$.

```
void
init_yo (int *initial_y0)
{
  int i, j, tem, j1, j2;
  double mean, distance, x, y;
  int *ntem;

  printf ("\n");
  printf ("the following are the germs whose grains \n do not cover any points in S\n");
  printf ("\n");

  ntem = (int *) malloc (MAX_K * sizeof (int));

  for (i = 1; i <= number_yi + 1; i++)
    initial_y0[i] = 0;

  mean = window * window * intensity;

  initial_y0[number_yi + 1] = poidev (mean, &idum);

  number_point = 0;

  for (i = 0; i < initial_y0[number_yi + 1]; i++)
    {
      x = RANDOM * window;
      y = RANDOM * window;

      for (j = 0; j < MAX_K; j++)
        {
          distance = dist (x, XcondAbs[j][1], y, XcondAbs[j][2]);

          if (distance < radius)
            ntem[j] = 1;
          else
            ntem[j] = 0;
        }

      j2 = 0;
      for (j = 1; j <= number_yi; j++)
        {
          tem = 0;
          for (j1 = 0; j1 < MAX_K; j1++)
            tem += abs ((ntem[j1] - newsubset[j][j1 + 1]));

          if (tem == 0)
            {
```

```
              ++initial_y0[j];
              ++number_point;
              ++j2;
              break;
            }
        }

      if (j2 == 0)
        printf ("%f\t%f \n", x, y);
    }
  free (ntem);
}
```

◇

where `initial_y0[number_yi + 1]` is obtained by `poide.c`,

"poide.h" 17a ≡

```
/*
 * File poide.h ⟨revision information 2⟩
 */

/****************************************************************
 * poide.h --- the header file of poide.c                       *
 ****************************************************************/

#define PI 3.141592654
```

◇

"poide.c" 17b ≡

```
/*
 * File poide.c ⟨revision information 2⟩
 */

/****************************************************************
 * poide.c --- Returns as a floating-point number an integer    *
 * value that is a random deviate drawn from a Poisson          *
 * distributin of mean xm, using ran2(&idum) as a source of     *
 * uniform random deviates.                                     *
 ****************************************************************/
```

```c
#include <math.h>

#include "poide.h"

double gammln ();
double ran2 ();

double
poidev (double xm, long *idum)
{
  static double sq, alxm, g, oldm = (-1.0);
  double em, t, y;

  if (xm < 12.0)
    {
      if (xm != oldm)
        {
          oldm = xm;
          g = exp (-xm);
        }

      em = (-1);
      t = 1.0;
      do
        {
          ++em;
          t *= ran2 (idum);
        }
      while (t > g);
    }
  else
    {
      if (xm != oldm)
        {
          oldm = xm;
          sq = sqrt (2.0 * xm);
          alxm = log (xm);
          g = xm * alxm - gammln (xm + 1.0);
        }

      do
        {
          do
            {
              y = tan (PI * ran2 (idum));
```

```
              em = sq * y + xm;
            }
          while (em < 0.0);

          em = floor (em);
          t = 0.9 * (1.0 + y * y) * exp (em * alxm - gammln (em + 1.0) - g);
        }
      while (ran2 (idum) > t);
    }

  return em;
}
```

◇

RANDOM is obtained by ran2.c,

"ran2.h" 19a ≡

```
/*
 * File ran2.h ⟨revision information 2⟩
 */


/*********************************************************************
 * ran2.h --- the header file of ran2.c                              *
 *********************************************************************/

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1 - 1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
```

◇

"ran2.c" 19b ≡

```
/*
 * File ran2.c ⟨revision information 2⟩
 */


/*********************************************************************
 * Long period (> 2 x 10 ^18) random number generantor of L'Ecuyer      *
 * with Bays-Durham shuffle                                              *
 * and added safeguards. Returns a uniform random deviate between 0.0   *
 * and 1.0 (exclusive of                                                 *
 * the endpoint values). Call with idum a negative integer to           *
 * initialize;  thereafter, do not alter                                 *
 * idum between successive deviates in sequence. RNMX should            *
 * approximate the largest doubling                                      *
 * value that is less than 1.                                            *
 *********************************************************************/

#include "ran2.h"

double
ran2 (long *idum)
{
  int j;
  long k;
  static long idum2 = 123456789;
  static long iy = 0;
  static long iv[NTAB];
  float temp;

  if (*idum <= 0)
    {
      if (-(*idum) < 1)
        *idum = 1;
      else
        *idum = -(*idum);
      idum2 = (*idum);

      for (j = NTAB + 7; j >= 0; j--)
        {
          k = (*idum) / IQ1;
          *idum = IA1 * (*idum - k * IQ1) - k * IR1;
          if (*idum < 0)
            *idum += IM1;
          if (j < NTAB)
            iv[j] = *idum;
```

```
        }

      iy = iv[0];
    }

  k = (*idum) / IQ1;
  *idum = IA1 * (*idum - k * IQ1) - k * IR1;
  if (*idum < 0)
    *idum += IM1;
  k = idum2 / IQ2;
  idum2 = IA2 * (idum2 - k * IQ2) - k * IR2;
  if (idum2 < 0)
    idum2 += IM2;
  j = iy / NDIV;
  iy = iv[j] - idum2;
  iv[j] = *idum;
  if (iy < 1)
    iy += IMM1;
  if ((temp = AM * iy) > RNMX)
    return RNMX;
  else
    return (temp);
}
```

◇

`gammar.c` is a function for calculating the value of log $gamar(x)$ for $x > 0$

"gammar.c" 21 ≡

```
/*
 * File gammar.c ⟨revision information 2⟩
 */

/*******************************************************
 * gammar.c --- Return the value ln[gamar(xx)] for xx>0 *
 *******************************************************/

#include <math.h>

double
gammln (double xx)
{
  double x, y, tmp, ser;
  static double cof[6] =
```

```
  {76.18009172947146, -86.50532032941677,
   24.01409824083091, -1.231739572450155,
   0.1208650973866179e-2, -0.5395239384953e-5};
  int j;

  y = x = xx;
  tmp = x + 5.5;
  tmp -= (x + 0.5) * log (tmp);
  ser = 1.000000000190015;
  for (j = 0; j <= 5; j++)
    ser += cof[j] / ++y;

  return (-tmp + log (2.5066282746310005 * ser / x));
}
```

◇

Now consider the incident times t_i in Table 2.1. If t_i is a birth time, we need to know whether there is $A \subseteq S$ such that t_i is a birth time of Y_A. Program `ybirth.c` will be used to carry out this job.

"ybirth.c" 22 ≡

```
/*
 * File ybirth.c ⟨revision information 2⟩
 */


/*********************************************************************
 * ybirth.c --- this program is used to decide if an incident time in   *
 * Table 2.1 is a birth time of the process  Y_{A_i}.   *
 *********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>

#define RANDOM ran2(&idum)

extern long idum;
extern double **XcondAbs;
extern double radius;
extern int number_yi;
extern int **newsubset;
```

```
extern double window;
extern int *newyi;
extern int number_point;
extern int MAX_K;

double ran2 ();
double dist ();

int
yindex_birth (int ya)
{
  double distance, x, y;
  int j, i, *ntem, tem, jj;

  x = RANDOM * window;
  y = RANDOM * window;

  ntem = (int *) malloc (MAX_K * sizeof (int));

  for (j = 0; j < MAX_K; j++)
    {
      distance = dist (x, XcondAbs[j][1], y, XcondAbs[j][2]);
      if (distance < radius)
        ntem[j] = 1;
      else
        ntem[j] = 0;
    }

  for (jj = 1; jj <= number_yi; jj++)
    {
      tem = 0;
      for (i = 0; i < MAX_K; i++)
        {
          tem += abs ((ntem[i] - newsubset[jj][i + 1]));
        }

      if (tem == 0)
        {
          ++number_point;
          free (ntem);
          return (-jj);
        }
    }

  free (ntem);
  return (0);
}
```

◇

Once an incident time has been accepted as an incident of some process Y_A, we create an ordered linked list for all the incident times of all the processes, which is done by `enter.c`.

"condpois.h" 24 ≡

```
/*
 * File condpois.h ⟨revision information 2⟩
 */

/*********************************************************************
 * condpois.h --- define a structure for each incident time.          *
 *                                                                    *
 * type --- the type of the incident: birth, death of y_Ai, incident of *
 *           z_Ai, perpetuate of the minimum and maximum or perpetuate  *
 *           time of the maximum process.                             *

 * type = -j --- death time of Y_{A_j}                                *

 * type = j  --- birth time of Y_{A_j}                                *

 * type = j, 2*k1 < j < 3*k1  --- incident of Z_{A_{3*k1-j}}          *
 *                                                                    *

 * type = j, 3*k1 < j < 4*k1  --- perpetuate time of                  *
 *        Y_{A_{4*k1-j}} ^{min} and Y_{A_{4*k1-j}} ^{max}             *

 * type = j, 4*k1 < j < 5*k1  --- perpetuate time of                  *
 *        Y_{A_{5*k1-j}} ^{max} only.                                 *

 * where  k1 = number_yi + 1.                                         *

 *********************************************************************/


typedef struct SIncident
  {
    struct SIncident *next;
    int type;
    double incident;
  }
Incident;
```

◇

"enter.c" 25 ≡

```
/*
 * File enter.c ⟨revision information 2⟩
 */


/**********************************************************************
 * enter.c --- order the link the list for the incident times of all    *
 *              y_A and \Z_A.                                            *
 **********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>

#include "condpois.h"

#define swap(a,b) swaptem=a; a=b; b=swaptem;
#define swapp(a,b) swaptem1=a; a=b; b=swaptem1;

void
enter (Incident * first, int index, double time)
{
  Incident *current;
  Incident *before;
  Incident *after;
  double swaptem;
  int swaptem1;

  current = (Incident *) malloc (sizeof (Incident));

  if (current == NULL)
    {
      printf ("current is NULL, and exit from enter\n");
      exit (8);
    }

  current->type = index;
  current->incident = time;

  if (current->incident < first->incident)
```

```
    {
      current->next = first->next;
      first->next = current;
      swapp (current->type, first->type);
      swap (current->incident, first->incident);
    }
  else
    {
      before = first;
      after = before->next;

      while (1)
        {
          if (after == NULL)
            break;

          if (after->incident > current->incident)
            break;

          after = after->next;
          before = before->next;
        }

      before->next = current;
      current->next = after;
    }
}
```

◇

If t_i is a death time (or perpetuation time), Table 2.1 has already indicated a way to decide whether there exist $A \subseteq S$ such that t_i is a death (or perpetuation) time of Y_A (or $\mathcal{Z}_A$).

The program `alltim1.c` sumarize all the above procedures.

"alltim1.c" 26 ≡

```
/*
 * File alltim1.c ⟨revision information 2⟩
 */


/**************************************************************
 * alltime1.c --- this program is used to generate all the     *
 *  incident time of all the processes Y_{A_i} and  the        *
 *   processes z_{A_i} by using ordered linked list.           *
```

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

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

#define RANDOM ran2(&idum)

extern int T_old, T_new;
extern int total_number;
extern int number_yi;
extern int number_subset_xi;
extern double *newua;
extern int *initial_y0;
extern int *last_ya;
extern int MAX_K;
extern int restart_index;
extern long idum;
extern double intensity;
extern double window;
extern *newyi;
extern Incident *first;
extern int sample_size;

double exponentialRV ();
double ran2 ();
void enter ();
int yindex_birth ();
int yindex_death ();

void
all_incident_time_one ()
{
  int j, k1, l;
  double t1, t2, u;
  int ya1, ya2;
  int doindex, index;
  double rate, a1, *prob;

  k1 = number_yi + 1;
  prob = (double *) malloc ((number_yi + 1) * sizeof (double));

  if (restart_index == 0)
```

```
  {
    for (j = 1; j <= number_yi + 1; j++)
      last_ya[j] = initial_y0[j];
    ya1 = 0;
    for (l = 1; l <= number_yi; l++)
      ya1 += last_ya[l];
  }
else
  ya1 = last_ya[number_yi + 1];

ya2 = ya1;
t1 = (-T_old);
doindex = 1;
a1 = intensity * window * window;

while (doindex == 1)
  {
    rate = a1 + (double) ya1 + (double) number_yi;
    t2 = t1 - exponentialRV (rate);
    if (t2 < (-T_new))
      {
        doindex = 0;
      }
    else
      {
        u = rate * RANDOM;
        if (u <= a1)
          {
            index = yindex_birth (ya1);
            if (index != 0)
              {
                enter (first, index, t2);
                ++last_ya[-index];
                ya2 = ya1 + 1;
                ++total_number;
              }
          }
        else if ((u > a1) && (u <= a1 + (double) ya1))
          {
            for (l = 1; l <= number_yi; l++)
              prob[l] = 0;

            prob[1] = (last_ya[1] + a1);
            for (l = 2; l <= number_yi; l++)
              prob[l] = prob[l - 1] + last_ya[l];

            for (l = 1; l <= number_yi; l++)
```

```
          {
            if (u <= prob[l])
              {
                --last_ya[l];
                ya2 = ya1 - 1;
                index = l;
                enter (first, index, t2);
                ++total_number;
                break;
              }
          }
      }
    else
      {
        index = k1 + (RANDOM * number_yi + 1);
        enter (first, index, t2);
        ++total_number;
      }
    ya1 = ya2;
    t1 = t2;
  }
}

  last_ya[number_yi + 1] = ya1;
  free (prob);
}
```

◇

where `exponentialRV (rate)` is obtained from `exponent.c`.

`"exponent.c"` 29 ≡

```
/*
 * File exponent.c ⟨revision information 2⟩
 */


/***************************************************************
 * exponentialRV: Return an exponentially distributedl positive,*
 *  random deviate of mean 1/rate                               *
 ***************************************************************/

#include <stdio.h>
#include <math.h>
```

```
#include <stdlib.h>

#define RANDOM ran2(&idum)

extern long idum;

double ran2 ();

double
exponentialRV (double rate)
{
  return (-log (RANDOM) / rate);
}
```

◇

The above method deals with the following case: if we only know the intensity λ of the germ process $\bar{Y}$ on a bounded window which is Poisson of mean $\lambda\,\mathrm{area}(W)$, then how to produce incident times of all Y_A and $\mathcal{Z}_A$. On the other hand if we know the rates of all the process Y_{A_i} then we can use the following method to obtaining all the incident times which are performed by `alltim2.c`.

"alltim2.c" 30 ≡

```
/*
 * File alltim2.c ⟨revision information 2⟩
 */


/**************************************************************
 * allTime.c --- this program is used to generate all the      *
 *  incident times of all the processes Y_{A_i} and  the       *
 *   processes z_{A_i}, at the same time assign an index       *
 *   to each incident time and sorted the incident times       *
 *   in order.                                                 *

 * total_incident[] --- store all the incident times of        *
 *      Y_{A_i}  and Z_{A_i}.                                  *

 * total_b_d_p_index[] --- store all the indexes for the       *
 *      incident time:                                         *

 * total_b_d_p_index[i] = -j --- death time of Y_{A_j}         *

 * total_b_d_p_index[i] = j  --- birth time of Y_{A_j}         *
```

```
 * total_b_d_p_index[i] = j, 2*k1 < j < 3*k1  --- incident of   *
 *                       Z_{A_{3*k1-j}}                          *

 * total_b_d_p_index[i] = j, 3*k1 < j < 4*k1  --- perpetuate    *
 *      time of Y_{A_{4*k1-j}} ^{min} and Y_{A_{4*k1-j}} ^{max} *

 * total_b_d_p_index[i] = j, 4*k1 < j < 5*k1  --- perpetuate    *
 *      time of Y_{A_{5*k1-j}} ^{max} only.                      *

 * last_ya[i] --- store the   value of ya[sub_total - 1]        *

 * last_ya[i] --- store the incident time of                    *
 *              ya_incident[sub_total - 1]                      *
 ***************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

#define UNIFORM ran2(&idum)

extern int T_old, T_new;
extern int total_number;
extern int number_yi;
extern int number_subset_xi;
extern double *newua;
extern int *initial_y0;
extern int *last_ya;
extern int restart_index;
extern long idum;
extern Incident *first;
extern int MAX_K;

double exponentialRV ();
double ran2 ();
void enter ();

void
all_incident_time_two ()
{
  int j;
  double t1, t2, t3;
  int ya1, ya2;
  int doindex, num, index;
```

```
double rate;

if (restart_index == 0)
  {
    for (j = 1; j <= number_yi; j++)
      {
        last_ya[j] = initial_y0[j];
      }
  }

for (j = 1; j <= number_yi; j++)
  {
    ya1 = last_ya[j];
    t1 = (-T_old);
    if (newua[j] == 0)
      {
        last_ya[j] = 0;
        continue;
      }
    doindex = 1;
    while (doindex == 1)
      {
        rate = newua[j] + (double) ya1;
        t2 = t1 - exponentialRV (rate);
        if (t2 < (-T_new))
          {
            doindex = 0;
          }
        else
          {
            if ((UNIFORM * rate <= newua[j]) || (ya1 == 0))
              {
                ya2 = ya1 + 1;
                index = (-j);
              }
            else
              {
                ya2 = ya1 - 1;
                index = j;
              }

            enter (first, index, t2);

            ya1 = ya2;
            t1 = t2;
            ++total_number;
          }
```

```
        }

      last_ya[j] = ya1;
    }


  /* generate incident of z_A
   */

    for (j = 1; j <= number_yi; j++)
      {
        if (newua[j] == 0)
          continue;

        t1 = (-T_old);
        rate = 1.;
        doindex = 1;
        num = 0;
        while (doindex == 1)
          {
            if (num == 0)
              {
                t2 = t1 - exponentialRV (rate);
                ++num;
              }

            if (t2 < (-T_new))
              doindex = 0;
            else
              {
                t3 = t2 - exponentialRV (rate);
                if ((t3 < -(T_new)) && (newua[j] > 0))
                  index = 3 * (number_yi + 1) + j;
                else
                  index = (number_yi + 1) + j;
                enter (first, index, t2);
                t2 = t3;
                ++total_number;
              }
          }
      }
  }
```

◇

where $*newua$ etc., in this case, can be obtained from the following program `rate_ua.c`.

"rate_ua.c" 34 ≡

```
/*
 * File rate_ua.c ⟨revision information 2⟩
 */

/*****************************************************************
 * rate_ua:  from the provided values of rates for each process    *
 *                      yi to construct *newua, number_yi, *newyi, *
 *                      **newsubset.                               *
 *****************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

extern int number_yi, *newyi, **newsubset;
extern double *newua;
extern int *bisub;
extern int MAX_K;

int bit ();

void
rate_ua ()
{
  int i, ii, k, i1, m;
  int *bisub;
  double rate;
  float rat;
  char line[50], *line_ptr;
  FILE *infile;

  infile = fopen ("rate.dat", "r");
  if (infile == NULL)
    {
      printf ("can not open rate.dat\n");
      exit (8);
    }

  i1 = 1;
  while (1)
    {
      line_ptr = fgets (line, sizeof (line), infile);
```

```
    if (line_ptr == NULL)
      break;
    else
      {
        sscanf (line, "%f", &rat);
        rate = (double) rat;
        if (rate != 0)
          ++i1;
      }
  }

number_yi = i1 - 1;
fclose (infile);
bisub = (int *) malloc ((MAX_K + 1) * sizeof (int));
if (bisub == NULL)
  {
    printf ("bisut is NULL in rate_ua.c.\n");
    exit (8);
  }

newsubset = (int **) malloc ((number_yi + 1) * sizeof (int *));
if (newsubset == NULL)
  {
    printf ("newsubset is NULL in rate_ua.c.\n");
    exit (8);
  }

for (i = 1; i <= number_yi; i++)
  {
    newsubset[i] = (int *) malloc ((MAX_K + 1) * sizeof (int));
    if (newsubset[i] == NULL)
      {
        printf ("newsubset[i] is NULL in rate_ua.c.\n");
        exit (8);
      }
  }

newyi = (int *) malloc ((number_yi + 1) * sizeof (int));
if (newyi == NULL)
  {
    printf ("newyi is NULL in rate_ua.c.\n");
    exit (8);
  }

newua = (double *) malloc ((number_yi + 1) * sizeof (double));
if (newua == NULL)
  {
```

```
      printf ("newua is NULL in rate_ua.c.\n");
      exit (8);
    }

  infile = fopen ("rate.dat", "r");
  if (infile == NULL)
    {
      printf ("can not open rate.dat second time. \n");
      exit (8);
    }

  i1 = 1;
  ii = 1;
  while (1)
    {
      printf ("i1=%d\n", i1);
      line_ptr = fgets (line, sizeof (line), infile);
      if (line_ptr == NULL)
        break;
      else
        {
          sscanf (line, "%f", &rat);
          rate = (double) rat;
          if (rate != 0)
            {
              {
                for (k = 1; k <= MAX_K; k++)
                  bisub[k] = 0;
              }
              newua[ii] = rate;
              newyi[ii] = i1;
              m = i1;
              k = 1;
              while (k != 0)
                k = bit (bisub, &m);
              for (k = 1; k <= MAX_K; k++)
                {
                  newsubset[ii][k] = bisub[k];
                  printf ("%d", newsubset[ii][k]);
                }
              printf ("\n");
              ++ii;
            }
        }
      ++i1;
    }
}
```

$\diamond$

We have now two different ways to obtain ordered incident times of Y_A and $\mathcal{Z}_A$ for all $A \subseteq S$, depending on the way in which the problem is presented. This is the information we need in order to construct $\tilde{X}_{\{x_i\}}$, hence, the Boolean model. In the next sections, we will describe how to construct $\tilde{Y}_A^{min}(t)$ and $\tilde{Y}_A^{max}(t)$ on $[-T, 0]$.

2.3 Construction of $\tilde{Y}_A^{min}(t)$ and $\tilde{Y}_A^{max}(t)$

The initial value of $\tilde{Y}_A^{min}(-T)$ and $\tilde{Y}_A^{max}(-T)$ with $\mu_A \neq 0$ are given as follows:

$$\begin{aligned} \tilde{Y}_A^{min}(-T) &= Y_A(-T)\,, \\ \tilde{Y}_A^{max}(-T) &= Y_A(-T) + 1\,. \end{aligned}$$

The `yinitial.c` carries out the above task.

"yinitial.c" 37 ≡

```
/*
 * File yinitial.c ⟨revision information 2⟩
 */


/**********************************************************************
 * yInitial.c --- this program is used to initialize the processes      *
 *      Y^{min}_{A_{death_index}} and Y^{max}_{A_{death_index}} at -T.  *
 *                                                                      *
 *  min_ny[i][j] --- store the value of nY_{A_i} ^ {min} (t_j).         *
 *                                                                      *
 *  min_py[i][j] --- store the value of pY_{A_i} ^ {min} (t_j).         *
 *                                                                      *
 *  max_ny[i][j] --- store the value of nY_{A_i} ^ {max} (t_j).         *
 *                                                                      *
 *  max_py[i][j] --- store the value of pY_{A_i} ^ {max} (t_j).         *
 *                                                                      *
 **********************************************************************/


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

```
#include <assert.h>

extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_yi;
extern double *newua;
extern int *last_ya;
extern int MAX_K;

void
YiInitial ()
{
  int i;

  for (i = 1; i <= number_yi; i++)
    {
      min_ny[i][0] = last_ya[i];
      min_py[i][0] = 0;
      max_ny[i][0] = last_ya[i];

      if (newua[i] > 0.)
        max_py[i][0] = 1;
      else
        max_py[i][0] = 0;
    }
}
```

◇

The evolution of $\tilde{Y}_A^{min}(t)$ and $\tilde{Y}_A^{max}(t)$ on $[-T,\ 0]$ are obtained by `yievolu.c`.

"yievolu.c" 38 ≡

```
/*
 * File yievolu.c ⟨revision information 2⟩
 */



/*********************************************************************
 * yiEvolu.c --- this program is used to decide the values of          *
 * Y^{min}_{A_{death_index}} and   Y^{max}_{A_{death_index}} for       *
 *  t in [-T,0]                                                        *
```

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


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_yi;
extern int number_restart;
extern int total_number;
extern int MAX_K;
extern Incident *first;

void death ();
void birth ();
void perpetuate_min ();
void perpetuate_max ();


void
YiEvolution ()
{
  int i, j, tem_index, k1, tem, l;
  int death_index, birth_index, perp_index;
  Incident *value;
  int *firstperp;

  k1 = number_yi + 1;

/*  assign a temporal index to birth, death times of Y_{A_i}, incident
 *  times of Z_{A_i}, and perpetuated times of Y_{A_i}.
 *
 * tem_index = 0 --- death of  Y_{A_i}.
 * tem_index = 1 --- birth of  Y_{A_i}.
 * tem_index = 2 --- incident of  Z_{A_i}.
 * tem_index = 3 --- perpetuate time in both  Y_{A_i}^{min} and  Y_{A_i}^{max}.
 * tem_index = 4 --- perpetuate time in  Y_{A_i}^{max}.
 * tem_index = 5 --- the index for time t=0;.
 */

  firstperp = (int *) malloc ((number_yi + 1) * sizeof (int));
```

```
for (l = 1; l <= number_yi; l++)
  firstperp[l] = 0;

value = first;

i = 1;
while (value != NULL)
  {
    tem = (value->type);
    if (tem < 0)
      {
        tem_index = 0;
        j = (-tem);
      }
    else if ((tem > 0) && (tem < k1))
      {
        tem_index = 1;
        j = tem;
      }
    else if ((tem > k1) && (tem < 2 * k1))
      {
        tem_index = 2;
        j = tem - k1;
        if (firstperp[j] == 0)
          {
            tem_index = 4;
            tem = j + 3 * k1;
            firstperp[j] = 1;
          }
      }
    else if ((tem > 2 * k1) && (tem < 3 * k1))
      {
        tem_index = 3;
        j = tem - 2 * k1;
      }
    else if ((tem > 3 * k1) && (tem < 4 * k1))
      {
        tem_index = 4;
        j = tem - 3 * k1;
      }
    else
      {
        tem_index = 5;
        j = 1;
      }
```

```
switch (tem_index)
  {
  case 0:
    death_index = (-tem);
    death (death_index, i, tem, value);
    break;

  case 1:
    birth_index = tem;
    birth (birth_index, i);
    break;

  case 2:
    for (l = 1; l <= number_yi; l++)
      {
        min_ny[l][1] = min_ny[l][0];
        min_py[l][1] = min_py[l][0];
        max_ny[l][1] = max_ny[l][0];
        max_py[l][1] = max_py[l][0];
      }
    break;

  case 3:
    perp_index = tem - 2 * k1;
    perpetuate_min (perp_index, i, tem, value);
    value->type = k1 + perp_index;
    break;

  case 4:
    perp_index = tem - 3 * k1;
    perpetuate_max (perp_index, i, value);
    value->type = k1 + perp_index;
    break;

  case 5:
    for (l = 1; l <= number_yi; l++)
      {
        min_ny[l][1] = min_ny[l][0];
        min_py[l][1] = min_py[l][0];
        max_ny[l][1] = max_ny[l][0];
        max_py[l][1] = max_py[l][0];
      }
    break;

  default:
    printf ("Impossible case happens in function YiEvolution() \n");
    break;
```

```
        }

      assert (min_ny[j][1] >= 0);
      assert (min_py[j][1] >= 0);
      assert (max_ny[j][1] >= 0);
      assert (max_py[j][1] >= 0);

      for (l = 1; l <= number_yi; l++)
        {
          min_ny[l][0] = min_ny[l][1];
          min_py[l][0] = min_py[l][1];
          max_ny[l][0] = max_ny[l][1];
          max_py[l][0] = max_py[l][1];
        }

      ++i;

      value = value->next;
    }

  free (firstperp);

}
```

$\diamond$

The details about the algorithm are as follows.

Let $-T = t_0 < t_1 < ... < t_n = 0$ be the ordered incident times of Y_A and $\mathcal{Z}_A$ for all $A \subseteq S$. Then

(1). If t_i is a birth time of process Y_A, we have

$$\tilde{Y}_A^{min}(t_i) = \tilde{Y}_A^{min}(t_i-) + 1\,,$$
$$\tilde{Y}_A^{max}(t_i) = \tilde{Y}_A^{max}(t_i-) + 1\,,$$

while

$$\tilde{Y}_B^{min}(t_i) = \tilde{Y}_B^{min}(t_i-)\,,$$
$$\tilde{Y}_B^{max}(t_i) = \tilde{Y}_B^{max}(t_i-)\,,$$

for all $B \neq A, \quad B \subseteq S$.

The above is done by `birth.c`.

`"birth.c"` 42 $\equiv$

```
/*
 * File birth.c ⟨revision information 2⟩
 */


/**********************************************************************
 * birth.c --- this program is used to decide what to do if             *
 *      the current time is a birth  time of Y_{A_{birth_index}},        *
 **********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>

extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_yi;
extern int MAX_K;

void
birth (int birth_index, int time)
{
  int i;

  for (i = 1; i <= number_yi; i++)
    {
      if (birth_index != i)
        {
          min_ny[i][1] = min_ny[i][0];
          max_ny[i][1] = max_ny[i][0];
        }
      else
        {
          min_ny[i][1] = min_ny[i][0] + 1;
          max_ny[i][1] = max_ny[i][0] + 1;
        }

      min_py[i][1] = min_py[i][0];
      max_py[i][1] = max_py[i][0];
    }
}
```

◇

(2). If t_i is a death time of Y_A, we use `death.c` to deal with this case.

"death.c" 44 ≡

```
/*
 * File death.c ⟨revision information 2⟩
 */


/*********************************************************************
 * death.c --- this program is used to decide what to do if          *
 *      t_{time} is a death time of Y_{A_{death_index}},             *
 *********************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_yi;
extern int MAX_K;

int max_pdeath ();
int min_pdeath ();
void new_perp_time ();

void
death (int death_index, int time, int time_type, Incident * value)
{
  int i;
  int pdeath_max, pdeath_min;

  for (i = 1; i <= number_yi; i++)
    {
      pdeath_max = 0;
      pdeath_min = 0;

      if (death_index != i)
```

```
  {
    min_ny[i][1] = min_ny[i][0];
    min_py[i][1] = min_py[i][0];
    max_ny[i][1] = max_ny[i][0];
    max_py[i][1] = max_py[i][0];
  }
else
  {
    if ((min_ny[i][0] + min_py[i][0]) > 1)
      {
        max_ny[i][1] = max_ny[i][0] - 1;
        max_py[i][1] = max_py[i][0];
      }
    else
      {
        pdeath_max = max_pdeath (death_index, time, time_type);

        if (pdeath_max == 0)
          {
            max_ny[i][1] = max_ny[i][0] - 1;
            max_py[i][1] = max_py[i][0];
          }
        else
          {
            max_ny[i][1] = max_ny[i][0] - 1;
            max_py[i][1] = max_py[i][0] + 1;
          }
      }

    if ((max_ny[i][0] + max_py[i][0]) > 1)
      {
        min_ny[i][1] = min_ny[i][0] - 1;
        min_py[i][1] = min_py[i][0];
      }
    else
      {
        pdeath_min = min_pdeath (death_index, time, time_type);

        if (pdeath_min == 0)
          {
            min_ny[i][1] = min_ny[i][0] - 1;
            min_py[i][1] = min_py[i][0];
          }
        else
          {
            min_ny[i][1] = min_ny[i][0] - 1;
            min_py[i][1] = min_py[i][0] + 1;
```

```
            }
          }
        }

      if ((pdeath_min == 1) || (pdeath_max == 1))
        new_perp_time (death_index, pdeath_min, value);
    }
}
```

◇

The above program considered several cases. If $B \neq A$, where for all $B \subseteq S$, we have

$$\tilde{Y}_B^{min}(t_i) = \tilde{Y}_B^{min}(t_i-)\,,$$
$$\tilde{Y}_B^{max}(t_i) = \tilde{Y}_B^{max}(t_i-)\,,$$

For the processes $\tilde{Y}_A^{min}(t)$ and $\tilde{Y}_A^{max}(t)$, if $\tilde{Y}_A^{min}(t_i-) > 1$, we let the particle die in $\tilde{Y}_A^{max}(t)$, that is

$$\tilde{Y}_A^{max}(t_i) = \tilde{Y}_A^{max}(t_i-) - 1\,.$$

Otherwise, we have to decide whether we need to perpetuate the point. If $\tilde{X}_{\{x_i\}}^{min} = 0$ for some $x_i \in A$, then if there is already a perpetuated point in $\tilde{Y}_A^{max}$, we know x_i is already covered so we set

$$\tilde{Y}_A^{max}(t_i) = \tilde{Y}_A^{max}(t_i-) - 1\,,$$

otherwise, we perpetuate the point. This part of the work is done by `maxpdeth.c`.

"maxpdeth.c" 46 ≡

```
/*
 * File maxpdeth.c ⟨revision information 2⟩
 */


/****************************************************************
 * maxpdeth.c --- this program is used to decide whether        *
 *      perpetuate  happens in the process Y^{max}_{A_i}.       *

 * death_index, time --- indicate that the potential perpetuate *
 *               point  is in the process                       *
 *               Y^{max}_{A_{death_index}}                      *
 *               and at the time point t_{time}.                *
```

```
 * Note:  This program deals with two cases:                    *
 *  (1)  if t_i is a death time of Y_A.                          *
 *  (2)  if t_i is a perpetuated time of Y_{A_i}^{min} and       *
 *              Y_{A_i}^{max}.                                   *
 ****************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

extern int **sxi;
extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_subset_xi;
extern int number_yi;
extern int *newyi;
extern int MAX_K;

int
max_pdeath (int death_index, int time, int time_type)
{
  int i, j, k, *tem, pdeath_max, tem_index;
  int k1, k2, ii;

  tem = (int *) malloc (MAX_K * sizeof (int));

  pdeath_max = 0;

  k1 = number_yi + 1;
  k2 = time_type;

/*
 * calculate X_{x_i}^{min} (t_i -)
 */

  for (i = 0; i < MAX_K; i++)
    {
      tem[i] = 0;
      tem_index = 0;

      ii = 0;
      j = 0;
```

```
      while ((ii == 0) && (sxi[i][j] != (-1)))
        {
          if ((death_index) != sxi[i][j])
            {
              ii = 0;
              ++j;
            }
          else
            {
              ii = 1;
              tem_index = 1;
              k = 0;
              while (sxi[i][k] != (-1))
                {
                  tem[i] += (min_ny[sxi[i][k]][0] + min_py[sxi[i][k]][0]);
                  ++k;
                }
            }
        }

      if (tem_index == 0)
        tem[i] = -1;
    }

  tem_index = 0;
  for (k = 0; k < MAX_K; k++)
    {
      if (tem[k] < 0)
        continue;

      else if (tem[k] == 1)
        ++tem_index;
    }

/*
 * consider two different cases:
 */

  if (tem_index != 0)
    {
      if ((k2 > 2 * k1) && (k2 < 3 * k1))
        pdeath_max = 1;

      else if (max_py[death_index][0] == 0)
        pdeath_max = 1;
    }
```

```
  free (tem);
  return (pdeath_max);
}
```

◇

The new death time of the perpetuated point is provided by the next incident time of $\mathcal{Z}_A$, which is given by `newptime.c`.

"newptime.c" 49 ≡

```
/*
 * File newptime.c ⟨revision information 2⟩
 */


/*************************************************************
 * newPtime.c --- this program is used to assign a new death    *
 *                time to the perpetuate point.                 *

 * death_index, time, pdeath_min, pdeath_max --- indicate that  *
 *      the potential perpetuate point  is in the process       *
 *      Y^{min}_{A_{death_index}} if "pdeath_min"=1             *
 *      or in the process Y^{max}_{A_{death_index}}             *
 *      if "pdeath_max"=1                                       *
 *      and at the time point t_{time}.                         *

 * Note: if a point is perpetuated in Y^{min}_{A_{death_index}} *
 *       then it will perpetuated in Y^{max}_{A_{death_index}}  *
 *************************************************************/



#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

extern int number_yi;
extern int MAX_K;

void
new_perp_time (int death_index, int pdeath_min, Incident * value)
{
  int i1, j1, k1;
```

```
  Incident *temvalue;

  k1 = number_yi + 1;
  i1 = death_index;
  j1 = i1 + k1;

  temvalue = value->next;

  while (temvalue != NULL)
    {
      if (temvalue->type != j1)
        temvalue = temvalue->next;
      else
        {
          if (pdeath_min == 1)
            temvalue->type = 2 * k1 + i1;
          else
            temvalue->type = 3 * k1 + i1;
          break;
        }
    }
}
```

◇

On the other hand, if $\tilde{Y}_A^{max}(t_i-) > 1$, then

$$\tilde{Y}_A^{min}(t_i) = \tilde{Y}_A^{min}(t_i-) - 1\,,$$

otherwise, if $\tilde{X}_{\{x_i\}}^{max} = 0$ for some $x_i \in A$, we perpetuate the point. The code `minpdeth.c` carries out the above task.

"minpdeth.c" 50 ≡

```
/*
 * File minpdeth.c ⟨revision information 2⟩
 */



/*****************************************************************
 * minpdeth.c --- this program is used to decide whether        *
 *      perpetuate  happens in the process Y^{min}_{A_i}.       *

 * death_index, time --- indicate that the potential perpetuate *
```

```
 *              point  is in the process                    *
 *              Y^{min}_{A_{death_index}}                   *
 *              and at the time point t_{time}.             *

 * Note:  This program deals with two cases:                *
 *  (1)  if t_i is a death time of Y_A.                     *
 *  (2)  if t_i is a perpetuated time of Y_{A_i}^{min} and  *
 *              Y_{A_i}^{max}.                              *
 ***********************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

extern int **sxi;
extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_subset_xi;
extern int number_yi;
extern int *newyi;
extern int MAX_K;

int
min_pdeath (int death_index, int time, int time_type)
{
  int i, j, k, *tem, pdeath_min, tem_index;
  int k1, k2, ii;

  pdeath_min = 0;

  tem = (int *) malloc (MAX_K * sizeof (int));

  k1 = number_yi + 1;
  k2 = time_type;

/*
 * calculate X_{x_i}^{max} (t_i -)
 */

  for (i = 0; i < MAX_K; i++)
    {
      tem[i] = 0;
```

```
      tem_index = 0;

      ii = 0;
      j = 0;
      while ((ii == 0) && (sxi[i][j] != (-1)))
        {
          if (death_index != sxi[i][j])
            {
              ii = 0;
              ++j;
            }
          else
            {
              tem_index = 1;
              ii = 1;
              k = 0;
              while (sxi[i][k] != (-1))
                {
                  tem[i] += (max_ny[sxi[i][k]][0] + max_py[sxi[i][k]][0]);
                  ++k;
                }
            }
        }

      if (tem_index == 0)
        tem[i] = -1;
    }

  tem_index = 0;
  for (k = 0; k < MAX_K; k++)
    {
      if (tem[k] < 0)
        continue;

      else if (tem[k] == 1)
        ++tem_index;
    }


/*
 * consider two different cases:
 */

  if (tem_index != 0)
    {
      if ((k2 > 2 * k1) && (k2 < 3 * k1))
        pdeath_min = 1;
```

```
      else if (min_py[death_index][0] == 0)
        pdeath_min = 1;
    }

  free (tem);
  return (pdeath_min);
}
```

◇

The new death time is again provided by newptime.c.

(3). If t_i is an incident time of $\mathcal{Z}_A$, then we do nothing, i.e.

$$\tilde{Y}_B^{min}(t_i) = \tilde{Y}_B^{min}(t_i-)\,,$$
$$\tilde{Y}_B^{max}(t_i) = \tilde{Y}_B^{max}(t_i-)\,,$$

for all $B \subseteq S$.

(4). If t_i is a perpetuation time for $\tilde{Y}_A^{min}$, then it should also be a perpetuation time for $\tilde{Y}_A^{max}$. Since we treat t_i as a death time, we decide whether the point should die or not in a similar way to case (2). The code perpmin.c carries out this task.

"perpmin.c" 53 ≡

```
/*
 * File perpmin.c ⟨revision information 2⟩
 */


/************************************************************************
 * perpmin.c --- this program is used to decide what to do if           *
 *      t_{time} is a perpetuation time of Y^{max}_{A_{death_index}},   *
 *       and of Y^{min}_{A_{death_index}}                               *

 ************************************************************************/



#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"
```

```
extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_yi;
extern int *newyi;
extern int MAX_K;

int max_pdeath ();
int min_pdeath ();
void new_perp_time ();

void
perpetuate_min (int perp_index, int time, int time_type, Incident * value)
{
  int i;
  int pdeath_min, pdeath_max;

  for (i = 1; i <= number_yi; i++)
    {
      pdeath_min = 0;
      pdeath_max = 0;

      if (i != perp_index)
        {
          min_ny[i][1] = min_ny[i][0];
          min_py[i][1] = min_py[i][0];
          max_ny[i][1] = max_ny[i][0];
          max_py[i][1] = max_py[i][0];
        }
      else
        {
          if ((min_ny[i][0] + min_py[i][0]) > 1)
            {
              max_ny[i][1] = max_ny[i][0];
              max_py[i][1] = max_py[i][0] - 1;
            }
          else
            {
              pdeath_max = max_pdeath (perp_index, time, time_type);

              if (pdeath_max == 0)
                {
                  max_ny[i][1] = max_ny[i][0];
                  max_py[i][1] = max_py[i][0] - 1;
                }
              else
```

```
            {
              max_ny[i][1] = max_ny[i][0];
              max_py[i][1] = max_py[i][0];
            }
        }

      if ((max_ny[i][0] + max_py[i][0]) > 1)
        {
          min_ny[i][1] = min_ny[i][0];
          min_py[i][1] = min_py[i][0] - 1;
        }
      else
        {
          pdeath_min = min_pdeath (perp_index, time, time_type);
          if (pdeath_min == 0)
            {
              min_ny[i][1] = min_ny[i][0];
              min_py[i][1] = min_py[i][0] - 1;
            }
          else
            {
              min_ny[i][1] = min_ny[i][0];
              min_py[i][1] = min_py[i][0];
            }
        }
    }

  if ((pdeath_min == 1) || (pdeath_max == 1))
    new_perp_time (perp_index, pdeath_min, value);
  }
}
```

◇

(5). If t_i is a perpetuation time of $\tilde{Y}_A^{max}$, but not $\tilde{Y}_A^{min}$, then

$$\begin{aligned}
\tilde{Y}_B^{min}(t_i) &= \tilde{Y}_B^{min}(t_i-) && \text{for all } B \subseteq S, \\
\tilde{Y}_C^{max}(t_i) &= \tilde{Y}_C^{max}(t_i-) && \text{for all } C \neq A,\ C \subseteq S.
\end{aligned}$$

For the process $\tilde{Y}_A^{max}$, if there exist $\tilde{X}_{\{x_i\}}^{min}(t_i-) = 0$ for some $x_i \in A$, we have to perpetuate the point again. Otherwise, let it die. This task is done by `perpmax.c`.

"perpmax.c" 55 $\equiv$

```
/*
```

```
 * File perpmax.c ⟨revision information 2⟩
 */


/***********************************************************************
 * perpmax.c --- this program is used to decide what to do if          *
 *      t_{time} is a perpetuation time of   Y^{max}_{A_{perp_index}},  *
 *       but not of Y^{min}_{A_{perp_index}}                          *

 ***********************************************************************/



#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"

extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int number_yi;
extern int **sxi;
extern int number_subset_xi;
extern int *newyi;
extern int MAX_K;

void new_perp_time ();

void
perpetuate_max (int perp_index, int time, Incident * value)
{
  int i, ii;
  int k, j, k1, *tem_min, *tem_max, tem_index;
  int pdeath_min, pdeath_max;

  tem_min = (int *) malloc (MAX_K * sizeof (int));
  tem_max = (int *) malloc (MAX_K * sizeof (int));

  for (i = 1; i <= number_yi; i++)
    {
      min_ny[i][1] = min_ny[i][0];
      min_py[i][1] = min_py[i][0];

      if (i != perp_index)
```

```
  {
    max_ny[i][1] = max_ny[i][0];
    max_py[i][1] = max_py[i][0];
  }
else
  {
    for (k = 0; k < MAX_K; k++)
      {
        tem_min[k] = 0;
        tem_max[k] = 0;

        tem_index = 0;

        j = 0;
        ii = 0;
        while ((ii == 0) && (sxi[k][j] != (-1)))
          {
            if (perp_index != sxi[k][j])
              {
                ii = 0;
                ++j;
              }
            else
              {
                ii = 1;
                tem_index = 1;
                k1 = 0;
                while (sxi[k][k1] != (-1))
                  {
                    tem_min[k] += (min_ny[sxi[k][k1]][0] +
                                   min_py[sxi[k][k1]][0]);

                    tem_max[k] += (max_ny[sxi[k][k1]][0] +
                                   max_py[sxi[k][k1]][0]);
                    ++k1;
                  }
              }
          }

        if (tem_index == 0)
          {
            tem_min[k] = -1;
            tem_max[k] = -1;
          }
      }

    tem_index = 0;
```

```
          for (k = 0; k < MAX_K; k++)
            {
              if (tem_min[k] < 0)
                continue;
              else
                {

/*
 * The following condition is used to decide if we should further
 * perpetuate the point again.
 */

                  if (tem_min[k] == 0 )
                    ++tem_index;
                }
            }

          if (tem_index != 0)
            {
              max_ny[i][1] = max_ny[i][0];
              max_py[i][1] = max_py[i][0];
              pdeath_min = 0;
              pdeath_max = 1;

              new_perp_time (perp_index, pdeath_min, value);
            }
          else
            {
              max_ny[i][1] = max_ny[i][0];
              max_py[i][1] = max_py[i][0] - 1;
            }
        }
    }
  free (tem_min);
  free (tem_max);
}
```

$\diamond$

2.4 Checking for Coalescence

If $\tilde{X}^{max}_{\{x_i\}}(0) = \tilde{X}^{min}_{\{x_i\}}(0),\ i = 1, ..., k$, then $(\tilde{X}^{max}_{\{x_1\}}(0), ..., \tilde{X}^{max}_{\{x_k\}}(0))$ is a perfect sample from the desired distribution. Otherwise let $-T = -2 * T$, and repeat the above procedure, until coalescence. This part is performed by `checkxxi.c` and `cftp.c`.

"checkxxi.c" 58 ≡

```
/*
 * File checkxxi.c ⟨revision information 2⟩
 */


/**********************************************************************
 * checkXxi.c --- this program is used to check the coalesence at      *
 *             time t=0;                                               *
 * restart_index --- if the returned value of  restart_index =1        *
 *      then, set T=2*T, if restart_index =0, then we get the perfect  *
 *      sample.                                                        *

 * min_xbi[i] --- store the value of X_{x_i} ^{min} (0),i=1,...,k;      *
 * max_xbi[i] --- store the value of X_{x_i} ^{max} (0) ,i=1,...,k;     *
 **********************************************************************/



#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>

extern int *min_xbi;
extern int *max_xbi;
extern int MAX_K;
extern int **sxi;
extern int number_subset_xi;
extern int total_number;
extern int **min_ny;
extern int **min_py;
extern int **max_ny;
extern int **max_py;
extern int *newyi;
extern int number_yi;

int
check_xxi ()
{
  int i, j, k;
  int restart;

  restart = 1;

  k = 0;
```

```
  for (i = 0; i < MAX_K; i++)
    {
      min_xbi[i] = 0;
      max_xbi[i] = 0;
      j = 0;
      while (sxi[i][j] != (-1))
        {
          min_xbi[i] += (min_ny[sxi[i][j]][0] +
                         min_py[sxi[i][j]][0]);

          max_xbi[i] += (max_ny[sxi[i][j]][0] +
                         max_py[sxi[i][j]][0]);
          ++j;
        }

      if (min_xbi[i] == max_xbi[i])
        {
          k = k + 1;
          continue;
        }
      else
        break;
    }

  if (k == MAX_K)
    {
      restart = 0;
      return (restart);
    }

  return (restart);
}
```

◇

"cftp.h" 60 ≡

```
/*
 * File cftp.h ⟨revision information 2⟩
 */


/***************************************************************
 * cftp.h --- the header file of cftp.c.                       *
 *                                                             *
 *  **min_ny  --- store the value of nY_{A_i} ^ {min} (t_j).   *
```

```
 *                                                             *
 *  **min_py  --- store the value of pY_{A_i} ^ {min} (t_j).    *
 *                                                             *
 *  **max_ny  --- store the value of nY_{A_i} ^ {max} (t_j).    *
 *                                                             *
 *  **max_py  --- store the value of pY_{A_i} ^ {max} (t_j).    *

 ****************************************************************/


int **min_ny;
int **min_py;
int **max_ny;
int **max_py;
int *min_xbi;
int *max_xbi;

extern int T_old, T_new;
extern int total_number;
extern int restart_index;
extern int number_restart;
extern int number_yi;
extern int *initial_y0;
extern int method;
extern Incident *first;
extern int MAX_K;

void all_incident_time_one ();
void all_incident_time_two ();
void YiInitial ();
void YiEvolution ();
int check_xxi ();
```

◇

"cftp.c" 61 ≡

```
/*
 * File cftp.c ⟨revision information 2⟩
 */


/****************************************************************
 * cftp.c --- this program is used to return the perfect       *
 *      sample or set T=2*T and go back to cfpt anain.          *
 *                                                             *
```

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

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"
#include "cftp.h"


void
CCP_CFTP (int method)
{
  int i;

  printf ("number_restart= %d \n", number_restart);

  min_xbi = (int *) malloc (MAX_K * sizeof (int));
  max_xbi = (int *) malloc (MAX_K * sizeof (int));

  if (method == 1)
    all_incident_time_one ();
  else
    all_incident_time_two ();

  min_ny = (int **) malloc ((number_yi + 1) * sizeof (int *));
  min_py = (int **) malloc ((number_yi + 1) * sizeof (int *));
  max_ny = (int **) malloc ((number_yi + 1) * sizeof (int *));
  max_py = (int **) malloc ((number_yi + 1) * sizeof (int *));

  for (i = 1; i <= number_yi; i++)
    {
      min_ny[i] = (int *) malloc (2 * sizeof (int));
      min_py[i] = (int *) malloc (2 * sizeof (int));
      max_ny[i] = (int *) malloc (2 * sizeof (int));
      max_py[i] = (int *) malloc (2 * sizeof (int));
    }

  YiInitial ();

  YiEvolution ();

  restart_index = check_xxi ();

  free (min_ny);
  free (min_py);
```

```
  free (max_ny);
  free (max_py);
  free (min_xbi);
  free (max_xbi);

  while (restart_index == 1)
    {
      T_old = T_new;
      T_new = 2 * T_old;

      ++number_restart;

      CCP_CFTP (method);
    }
}
```

◇

2.5 The Configuration of the Conditional Points and the Perfect sample of the Boolean Model

The configuration of the conditional points on the bounded window W is obtained by `abscond.c`.

"abscond.c" 63 ≡

```
/*
 * File abscond.c ⟨revision information 2⟩
 */


/**********************************************************************
 * abscond.c --- return the abscissae of the conditional point in S.    *
 *               and the disteances between any two points.             *

 * XcondAbs[][ ] ---  store the abscissae of conditional point x_i.     *

 * condist[][] --- store the distance between 2 points in S.            *
 **********************************************************************/


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

```c
#include <assert.h>

#define RANDOM ran2(&idum)

extern double radius, window, intensity, *newua;;
extern long idum;
extern int *uewyi;

/*extern FILE *pf;
 */

double **XcondAbs;
double **condist;

extern int MAX_K;

double ran2 ();
double dist ();

void
abscissaCondpt ()
{
  int i, j;
  double x1, x2, y1, y2;

  printf ("\n");
  printf ("The number of conditional points: \t %d\n", MAX_K);
  printf ("\n");
  printf ("Radius of the grain: \t %f \n", radius);
  printf ("\n");
  printf ("Intensity of the germs: \t %f \n", intensity);
  printf ("\n");
  printf ("The window width : \t %f \n", window);
  printf ("\n");
  printf ("The conditional points are: \n");
  printf ("\n");


  XcondAbs = (double **) malloc (MAX_K * sizeof (double *));
  condist = (double **) malloc (MAX_K * sizeof (double *));

  for (i = 0; i < MAX_K; i++)
    XcondAbs[i] = (double *) malloc (3 * sizeof (double));

  for (i = 0; i < MAX_K; i++)
    condist[i] = (double *) malloc (MAX_K * sizeof (double));
```

```
  for (i = 0; i < MAX_K; i++)
    {
      XcondAbs[i][1] = RANDOM * window;
      XcondAbs[i][2] = RANDOM * window;

      printf ("%f \t %f \n", XcondAbs[i][1], XcondAbs[i][2]);

/*      fprintf (pf, "%f \t %f \n", XcondAbs[i][1], XcondAbs[i][2]);
 */
    }

  for (i = 0; i < MAX_K - 1; i++)
    {
      for (j = i + 1; j < MAX_K; j++)
        {
          x1 = XcondAbs[i][1];
          x2 = XcondAbs[j][1];
          y1 = XcondAbs[i][2];
          y2 = XcondAbs[j][2];

          condist[i][j] = dist (x1, x2, y1, y2);
          condist[j][i] = condist[i][j];
        }
    }
}
```

◇

The configuration of a perfect sample of the Boolean model is generated by `absya.c`.

"absya.c" 65 ≡

```
/*
 * File absya.c ⟨revision information 2⟩
 */


/***********************************************************************
 * absya.c --- simulate the configuration of y_A (0).                 *
 ***********************************************************************/


#include <stdio.h>
#include <math.h>
```

```
#include <stdlib.h>
#include <assert.h>

#define RANDOM ran2(&idum)

extern int **min_ny;
extern int **min_py;
extern int number_yi;
extern int **newsubset;
extern int *newyi;
extern int total_number;
extern double radius, window;
extern long idum;
extern double **XcondAbs;
extern int **max_ny;
extern int **max_py;
extern int MAX_K;

/*extern FILE *pf;
 */

double ran2 ();
double dist ();

void
abscissaya ()
{
  int tem, k, index, l, ithya;
  double y1, y2, x1, x2;
  double dis;

  for (ithya = 1; ithya <= number_yi; ithya++)
    {
      printf (" \n");
      printf ("************\n");
      printf ("configuration of Y_ %d \n", ithya);
      printf ("************\n");

      for (l = 1; l <= MAX_K; l++)
        if (newsubset[ithya][l] != 0)
          printf ("%f \t %f \n", XcondAbs[l - 1][1], XcondAbs[l - 1][2]);

      printf ("\n");

      tem = min_py[ithya][0] + min_ny[ithya][0];

      if (tem == 0)
```

```
          printf ("no germs whose grains cover the above points.\n");
        else
          {
            printf ("the following are the germs whose grains should \n ");
            printf (" cover the above points. \n ");
          }

        for (k = 0; k < tem; k++)
          {
            index = 1;
            while (index == 1)
              {
                index = 0;
                x1 = RANDOM * window;
                x2 = RANDOM * window;

                for (l = 0; l < MAX_K; l++)
                  {
                    if (newsubset[ithya][l + 1] != 0)
                      {
                        y1 = XcondAbs[l][1];
                        y2 = XcondAbs[l][2];
                        dis = dist (x1, y1, x2, y2);

                        if (dis > radius)
                          {
                            index = 1;
                          }
                      }
                    else
                      continue;
                  }
              }
            printf ("%f \t %f \n", x1, x2);
          }
      }
}
```

◇

2.6 The Main Program

The program `mainprog.c` describes the main structure of the implementation about the perfect simulation for the conditional Boolean model. The perfect samples of the conditional correlated Poisson processes can be stored in a result file CCP.res. The format of the file is as follows: The values of

$\tilde{X}^{min}_{\{x_1\}}(0), ..., \tilde{X}^{min}_{\{x_k\}}(0)$ are stored in the corresponding columns. The perfect samples of conditional Boolean model can be saved in a file which will be mentioned in the next chapter.

"mainprog.h" 68 ≡

```
/*
 * File mainprog.h ⟨revision information 2⟩
 */


/************************************************************************
 * mainprog.h --- the header file of mainprog.c.                        *

 * number_yi --- total number of process Y_{A_i}                        *
 *                                                                      *
 * number_subset_xi --- total number of the processes Y_{A_i}           *
 *      which are used to form X_{x_i}.                                 *
 *                                                                      *
 *                                                                      *
 * T_old --- store the value of -2^{i-1} T.                             *
 *                                                                      *
 * T_new --- store the value of -2^{i} T.                               *
 *                                                                      *
 * total_number --- the total number of incident times of               *
 *      Y_{A_i} and Z_{A_i} on the interval (-T_new, 0).                *
 *                                                                      *
 * number_restart --- number of times we have to go back                *
 *              further in time to get perfect samples.                 *

 * restart_index --- indicate if we need to go back in time             *
 *                      to get the coalescence.                         *

 * **sxi --- is used to store the indexes of the processes              *
 *              Y_{A_i} which are used for X_{x_i}.                     *

 * **subset  --- store the binary index for each A in S.                *

 * *initial_y0 --- store the initial values of the process  Y_{A_i}.    *
 *                      for i=1,2,...,MAX_K.                            *

 * *last_ya --- store the   value of y_Ai at time -T                    *

 *                                                                      *
 * *ua --- store the rate of the processes Y_{A_i}                      *
 *                                                                      *
 * idum --- the seed for starting the random number generator ran2().   *
```

```
 * radius --- the radius of the grains of the Boolean model.          *

 * window --- the width of the window W.                               *

 * Incident *first --- define a point first which points to the first  *
 *                     incident time from -T to 0.                     *

 **********************************************************************/



int number_subset_xi;
int T_old, T_new;
int total_number;
int number_restart;
int restart_index;
int **sxi;
int **subset;
int *initial_y0;
int *last_ya;
double *ua;
long idum;
int method;
double radius, window;
double intensity;

Incident *first;

FILE *pf, *popen ();

extern int *min_xbi;
extern int *max_xbi;
extern int **min_ny;
extern int **min_py;

void CCP_CFTP ();
void index_in_sxi ();
void rate_ua ();
void abscissaCondpt ();
void bisubset ();
void init_yo ();
void abscissaya ();
```

◇

"mainprog.c" 69 ≡

```
/*
 * File mainprog.c ⟨revision information 2⟩
 */



/*************************************************************
 * mainprog.c --- this program describes the main structure     *
 *       of the implementation about the perfect simulation     *
 *       for conditioning Boolean model.                        *
 *************************************************************/



#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "condpois.h"
#include "mainprog.h"
#include "poide.h"

void newarray ();

int old_number_yi;
int **newsubset, *newyi, number_yi;
double *newua;
int MAX_K;

extern int number_point;

int
main (int argc, char **argv)
{
  FILE *pp;

  int i, T, j;
  double uaa;
  int sample_size;
  char c, c1;
  float radiu, windo, intensit;

  printf ("%% Output from  \n");
```

```
i = argc;

while (i)
  {
    printf ("%s", argv[argc - i]);
    i--;
  }

while ((--argc > 0) && ((*++argv)[0] == '-'))
  {
    c = *++argv[0];

    switch (c)
      {
      case 's':
        idum = -atoi (++argv[0]);
        break;

      case 'm':
        c1 = *++argv[0];
        switch (c1)
          {
          case 'p':
            method = 1;
            break;

          case 'r':
            method = 2;
            break;
          }
        break;

      default:
        printf ("unknown switch -%c \n", c);
        puts ("Usage:");
        puts ("cboolean -s [positive integer] -m [p or r] \n");
        exit (8);
      }
  }

printf ("\n");
printf ("method= %d \n", method);

scanf ("%d", &MAX_K);
scanf ("%d", &T);
scanf ("%f", &radiu);
scanf ("%f", &windo);
```

```
scanf ("%f", &intensit);

radius = (double) radiu;
window = (double) windo;
intensity = (double) intensit;

printf ("MAX_K, T, radius, window, intensity = \n");
printf ("%d %d %f %f %f \n ", MAX_K, T, radius, window, intensity);

pp = fopen ("CCP.res", "w");

sample_size = 1;

fprintf (pp, "%d\t %d\n", (int) (intensity * 1000), (int) (radius * 1000));

old_number_yi = pow (2., (double) (MAX_K));

old_number_yi -= 1;

abscissaCondpt ();

printf ("finish abscissaCondpt()\n");

if (method == 1)
  bisubset ();
else
  rate_ua ();

printf ("finish bisubset()  or rate_ua ().\n");

if (method != 1)
  {
    printf ("the rate of yi: \n");
    printf ("\n");
    for (i = 1; i <= number_yi; i++)
      printf ("ua[%d] = %f \n", newyi[i], newua[i]);
  }

printf ("number_yi=%d\n", number_yi);

sxi = (int **) malloc (MAX_K * sizeof (int *));
if (sxi == NULL)
  {
    printf ("sxi is NULL, and exit from mainprog\n");
    exit (8);
  }
```

```
for (i = 0; i < MAX_K; i++)
  {
    sxi[i] = (int *) malloc ((number_yi + 1) * sizeof (int));
    if (sxi[i] == NULL)
      {
        printf ("sxi[i] is NULL, and exit from mainprog\n");
        exit (8);
      }
  }

for (i = 0; i < MAX_K; i++)
  for (j = 0; j <= number_yi; j++)
    sxi[i][j] = 0;

index_in_sxi ();

printf ("finish index_in_sxi()\n");

uaa = intensity * PI * radius * radius;

for (i = 0; i < sample_size; i++)
  {
    printf ("sample size = %d \n", i);

    T_old = 0;
    T_new = T;

    total_number = 0;
    restart_index = 0;
    number_restart = 0;

    first = (Incident *) malloc (sizeof (Incident));
    if (first == NULL)
      {
        printf ("no enough memory for *first\n");
        exit (8);
      }

    first->next = NULL;
    first->incident = 0;
    first->type = 0;

    initial_y0 = (int *) malloc ((number_yi + 2) * sizeof (int));
    if (initial_y0 == NULL)
      {
        printf ("no enough memory for *initial_y0\n");
        exit (8);
```

```
        }

      last_ya = (int *) malloc ((number_yi + 2) * sizeof (int));
      if (last_ya == NULL)
        {
          printf ("no enough memory for *last_ya\n");
          exit (8);
        }

      init_yo (initial_y0);

      printf ("finish init_yo()\n");

      CCP_CFTP (method);

      printf ("finish CCP_CFTP()\n");

/*
 * generate the configurations for Y_A;
 */

      abscissaya ();

      printf ("finish abscissaya()\n");

      free (first);
      free (initial_y0);
      free (last_ya);

      for (j = 0; j < MAX_K; j++)
         fprintf(pp, "%d \t", min_xbi[j]);
    }

  return (0);
}
```

◇

Chapter 3

How to Run the Program

To run the program, the command line is

```
filename -s[positive number] -m[p or r] < param.dat > result.res
```

where `filename` is the name of the executable file, `-s[ ]` sets random number generator seed to a positive number, `-m[p]` means to use method 1, i.e. the configuration of the conditional points to start the simulation, while `-m[r]` means to use the method 2, i.e. the rate of each Y_{A_i} to start the perfect simulation. `param.dat` provide the number of conditioning points, i.e. the value of MAX_K, initial T, radius of the grain, window width and the intensity of the germ process. `result.res` stores the conditional points, the perfect sample of the conditional Boolean model etc.

Chapter 4

Checking the Program

We believe any simulation methods should be checked out carefully before putting them into use, because it is so easy to make a mistake in coding the simulation method, even the theory is correct. Therefore, we now consider several cases designed to enable us to check the correctness of the method.

4.1 Case 1

In this case, we have one conditioning point $S = \{x_1\}$ and $x_1 \oplus G \subset W$. Let $A = \{x_1\}$, then $X_{\{x_1\}}(t) = Y_A(t)$. where $Y_A(t)$ is immigration-death process with immigration rate $\mu_A = \lambda\,\text{area}(G)$ and unit death rate. The equilibrium distribution of $X_{\{x_1\}}(t)$ is the unconditional correlated Poisson distribution. That is

$$P\left(X_{\{x_1\}} = n\right) = \frac{\mu_A^n e^{-\mu_A}}{n!}.$$

Now, conditioned on $X_{\{x_1\}} > 0$, we have

$$\begin{aligned} P\left(X_{\{x_1\}} = n \,|\, X_{\{x_1\}} > 0\right) &= \frac{P(X_{\{x_1\}} = n)}{P(X_{\{x_1\}} > 0)} \\ &= \frac{1}{1 - e^{-\mu_A}} \frac{\mu_A^n e^{-\mu_A}}{n!}, \quad n = 1, 2, ... \end{aligned}$$

The initial setting for this simulation is as follows. The simulation interval is initialized at $[-2,\ 0]$, G is a disk with radius $r = 0.5$, window W is a square with width 1, conditioning point is $S = \{(0.5,\ 0.5)\}$, the intensity of the germ process is $\lambda = 5$. We collected 10000 perfect samples and carried out an χ^2 test. The results pass the χ^2 test at 5% level.

The following table shows the test result for this case.

$X_{\{x_1\}}(T)$	o_i	e_i	$(o_i - e_i)^2/e_i$	$(o_i - e_i)/10000$	prob
1	800	789.281118	0.145568	0.001072	0.078928
2	1567	1549.749852	0.192010	0.001725	0.154975
3	2029	2028.617812	0.000072	0.000038	0.202862
4	1923	1991.590880	2.362287	-0.006859	0.199159
5	1543	1564.191820	0.287109	-0.002119	0.156419
≥ 6	2138	2076.568517	1.817338	0.006143	0.207657
	10000	10000	4.804384		

Table 4.1: The goodness-of-fit test with sample size = 10000

4.2 Case 2

In this case, we have $S = \{x_1,\ x_2\}$ and $(x_1 \oplus G) \bigcup (x_2 \oplus G) \subset W$ and $(x_1 \oplus G) \bigcap (x_2 \oplus G) = \emptyset$.

Let $y_1 = Y_{\{x_1\}},\ y_2 = Y_{\{x_2\}},\ y_3 = Y_{\{x_1, x_2\}}$, then

$$\begin{aligned} X_{\{x_1\}} &= y_1 + y_3 \\ X_{\{x_2\}} &= y_2 + y_3 \,. \end{aligned}$$

Generally, the joint distribution of $X_{\{x_1\}}$, $X_{\{x_2\}}$ conditioned to be positive is

$$\begin{aligned} & P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n \mid X_{\{x_1\}} > 0,\, X_{\{x_2\}} > 0\right) \\ & = P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n)/P(X_{\{x_1\}} > 0,\, X_{\{x_2\}} > 0\right), \quad (m > 0,\, n > 0)\,, \end{aligned}$$

where

$$P\left(X_{\{x_1\}} > 0,\, X_{\{x_2\}} > 0\right) \quad = \quad 1 - e^{-\mu_1 - \mu_3} - e^{-\mu_2 - \mu_3} + e^{-\mu_1 - \mu_2 - \mu_3},$$

and if $m \leq n$,

$$\begin{aligned} P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n\right) &= P\left(y_1 + y_3 = m, y_2 + y_3 = n\right) \\ &= \sum_{k=0}^{m} P\left(y_1 = k\right) P\left(y_2 = n - m + k\right) P\left(y_3 = m - k\right) \\ &= \sum_{k=0}^{m} \frac{\mu_1^k e^{-\mu_1}}{k!} \frac{\mu_2^{n-m+k} e^{-\mu_2}}{(n - m + k)!} \frac{\mu_3^{m-k} e^{-\mu_3}}{(m - k)!}\,, \end{aligned}$$

if $m > n$, we have

$$P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n\right) \quad = \quad P\left(y_1 + y_3 = m, y_2 + y_3 = n\right)$$

$$= \sum_{k=0}^{m} P(y_1 = m-n+k) P(y_2 = k) P(y_3 = n-k)$$

$$= \sum_{k=0}^{m} \frac{\mu_1^{n-m+k} e^{-\mu_1}}{(n-m+k)!} \frac{\mu_2^k e^{-\mu_2}}{k!} \frac{\mu_3^{n-k} e^{-\mu_3}}{(n-k)!},$$

However, in this special case, since $(x_1 \oplus G) \bigcap (x_2 \oplus G) = \emptyset$, we have $\mu_3 = 0$, $\mu_1 = \mu_2 = \lambda\,\mathrm{area}(G)$. $X_{\{x_i\}}$, $i = 1, 2$ are independent Poisson of mean $\lambda\,\mathrm{area}(G)$.

In this simulation, G is a disk with radius $r = 0.25$, $\lambda = 5$ and the conditioning points are $x_1 = (0.25, 0.25)$, $x_2 = (0.75, 0.25)$. The χ^2 testing results are shown in the following table.

	o_i	e_i	$(o_i - e_i)^2/e_i$	$(o_i - e_i)/10000$	prob
(1,1)	3521	3459.602724	1.089612	0.006140	0.345960
(1,2)	1730	1698.228516	0.594400	0.003177	0.169823
(1,≥3)	726	724.007524	0.005483	0.000199	0.072401
(2,1)	1605	1698.228516	5.118013	-0.009323	0.169823
(2,2)	850	833.615974	0.322014	0.001638	0.083362
(2,≥3)	368	355.396362	0.446970	0.001260	0.035540
(≥3,1)	707	724.007524	0.399521	-0.001701	0.072401
(≥3,2)	356	355.396362	0.001025	0.000060	0.035540
(≥3,≥3)	137	151.516499	1.390797	-0.001452	0.015152
	10000	10000	9.367837		

Table 4.2: The goodness-of-fit test for $(X_{\{x_1\}}, X_{\{x_2\}})$ with sample size = 10000

4.3 Case 3

The difference between this case and the Case 2 is only that the two grains centred at x_1 and x_2 overlap. Let $E_1 = (x_1 \oplus G) \setminus ((x_1 \oplus G) \bigcap (x_2 \oplus G))$, $E_2 = (x_2 \oplus G) \setminus ((x_1 \oplus G) \bigcap (x_2 \oplus G))$, $E_3 = (x_1 \oplus G) \bigcap (x_2 \oplus G)$. Then in order to carry out the χ^2 test, we need to calculate $\mathrm{area}(E_i)$, $i = 1, 2, 3$. Since $\mathrm{area}(E_1) = \mathrm{area}(E_2) = \pi r^2 - \mathrm{area}(E_3)$, we only need to calculate the $\mathrm{area}(E_3)$.

Let $x_1 = (u_1, v_1), x_2 = (u_2, v_1)$, then

$$E_3 = \left\{ (u,v) : a \le v \le b,\ u_2 - \sqrt{(r^2 - (v-v_1)^2)} \le u \le u_1 + \sqrt{(r^2 - (v-v_1)^2)} \right\}.$$

By some mathematical calculation, we have

$$\mathrm{area}(E_3) = (u_1 - u_2)\sqrt{4r^2 - (u_2 - u_1)^2} + f(b - v_1) - f(a - v_1),$$

where r is the radius of the grain,

$$\begin{aligned} a - v_1 &= -\sqrt{r^2 - (u_2 - u_1)^2/4}\,, \\ b - v_1 &= \sqrt{r^2 - (u_2 - u_1)^2/4)}\,, \end{aligned}$$

and

$$f(u) = u\sqrt{r^2 - u^2} + r^2 \arcsin(u/r)\,.$$

Then it is easy to see that $\mu_1 = \mu_2 = \lambda(\pi r^2 - \mathrm{area}(E_3))$, $\mu_3 = \lambda\,\mathrm{area}(E_3)$. In our simulation, we take $u_1 = 0.25$, $u_2 = 0.4$, $v_1 = 0.25$, $\lambda = 5$, $r = 0.25$. The following table shows the testing results.

	o_i	e_i	$(o_i - e_i)^2/e_i$	$(o_i - e_i)/10000$	prob
((1,1)	3802	3804.992415	0.002353	-0.000299	0.380499
(1,2)	1306	1277.212980	0.648829	0.002879	0.127721
(1,3)	221	227.959442	0.212467	-0.000696	0.022796
(1,≥4)	38	30.292625	1.960993	0.000771	0.003029
(2,1)	1282	1277.212980	0.017942	0.000479	0.127721
(2,2)	1388	1401.021224	0.121020	-0.001302	0.140102
(2,3)	434	433.207621	0.001449	0.000079	0.043321
(2,≥4)	80	84.676874	0.258313	-0.000468	0.008468
(3,1)	237	227.959442	0.358536	0.000904	0.022796
(3,2)	451	433.207621	0.730755	0.001779	0.043321
(3,3)	323	339.346395	0.787410	-0.001635	0.033935
(3,≥4)	121	116.149951	0.202523	0.000485	0.011615
(≥4,1)	28	30.292625	0.173512	-0.000229	0.003029
(≥4,2)	94	84.676874	1.026499	0.000932	0.008468
(≥4,3)	98	116.149951	2.836167	-0.001815	0.011615
(≥4,≥4)	97	115.640983	3.004871	-0.001864	0.011564
	10000	10000	12.343640		

Table 4.3: The goodness-of-fit test for $(X_{\{x_1\}}, X_{\{x_2\}})$ with sample size = 10000

4.4 Case 4

We consider a more complicated case here. Let $S = \{x_1, x_2, x_3\}$ and $(x_1 \oplus G)\bigcup(x_2 \oplus G)\bigcup(x_3 \oplus G) \subset W$ and $(x_1 \oplus G)\bigcap(x_2 \oplus G)\bigcap(x_3 \oplus G) \neq \emptyset$. Let $A_1 = \{x_1\}$, $A_2 = \{x_2\}$, $A_3 = \{x_3\}$, $A_4 = \{x_1, x_2\}$, $A_5 = \{x_1, x_3\}$, $A_6 = \{x_2, x_3\}$, $A_7 = \{x_1, x_2, x_3\}$. Furthermore, let E_i be the region associated with process y_{A_i}, for $(i = 1, \cdots, 7)$, then to obtain the immigration rate of the processes y_{A_i}, we need to calculate the area of E_i. To ease the calculation, let (u_i, v_i) be the abscissae of the conditional points x_i, where $i = 1, 2, 3.$ and given (u_1, v_1) and the radius of the grain r, we construct the other two conditional points by setting $(u_2, v_2) = (u_1 - r/2,\, v_1 + \frac{\sqrt{3}}{2} r)$, $(u_3, v_3) = (u_1 + r/2, v_2)$. Then we have $\mathrm{area}(E_1) = \mathrm{area}(E_2) = \mathrm{area}(E_3)$ and $\mathrm{area}(E_4) = \mathrm{area}(E_5) = \mathrm{area}(E_6)$. Thus we only need to calculate $\mathrm{area}(E_1)$, $\mathrm{area}(E_4)$ and $\mathrm{area}(E_7)$ which are given as follows.

It is easy to see that two of the intersect points of three grains are $(a_1, b_1) = (u_1 - r,\, v_1)$ and $(a_2, b_2) = (u_1 + r,\, v_1)$. Let

$$f(a) \quad = \quad (a\sqrt{(r^2 - a^2)} + r^2 \arcsin(a/r))/2\,,$$

then

$$\begin{aligned}
\mathrm{area}(E_1) &= 2(v_2 - v_1)(u_1 - a_1) + 2(f(0) - f(a_1 - u_1)) - 2(f(u_1 - u_2) - f(a_1 - u_2))\,,\\
\mathrm{area}(E_4) &= (v_1 - v_2)(u_2 - a_1) + (v_3 - v_2)(u_1 - u_2) + f(u_2 - u_1) - f(a_1 - u_1)\\
&\quad + f(0) - f(a_1 - u_2) + f(u_1 - u_2) - f(0) - f(u_1 - u_3) + f(u_2 - u_3)\\
\mathrm{area}(E_7) &= \pi r^2 - \mathrm{area}(E_1) - 2\,\mathrm{area}(E_4)\,.
\end{aligned}$$

So, the immigration rates of y_{A_i} is given by $\mu_i = \lambda\,\mathrm{area}(E_i)$, for $i = 1, \cdots, 7$.

In order to do χ^2 test, we also need to calculate the probabilities which are given as follows:

$$\begin{aligned}
&P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n, X_{\{x_3\}} = k \,|\, X_{\{x_1\}} > 0, X_{\{x_2\}} > 0, X_{\{x_3\}} > 0\right)\\
&= P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n, X_{\{x_3\}} = k\right) / P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} > 0, X_{\{x_3\}} > 0\right)\,,
\end{aligned}$$

where $m > 0, n > 0, k > 0$, and

$$\begin{aligned}
&P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} > 0, X_{\{x_3\}} > 0\right)\\
= \;&1 - P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} > 0, X_{\{x_3\}} > 0\right) - P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} = 0, X_{\{x_3\}} > 0\right)\\
&- P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} > 0, X_{\{x_3\}} = 0\right) - P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} = 0, X_{\{x_3\}} > 0\right)\\
&- P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} > 0, X_{\{x_3\}} = 0\right) - P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} = 0, X_{\{x_3\}} = 0\right)\\
&- P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} = 0, X_{\{x_3\}} = 0\right)\,,
\end{aligned}$$

$$\begin{aligned}
&P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} > 0, X_{\{x_3\}} > 0\right)\\
= \;& e^{-\mu_1 - \mu_4 - \mu_5 - \mu_7}(1 - e^{-\mu_2 - \mu_6} - e^{-\mu_3 - \mu_6} + e^{-\mu_2 - \mu_3 - \mu_6})\,,
\end{aligned}$$

$$
\begin{aligned}
& P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} = 0, X_{\{x_3\}} > 0\right) \\
= \; & e^{-\mu_2-\mu_4-\mu_6-\mu_7}\left(1 - e^{-\mu_1-\mu_5} - e^{-\mu_3-\mu_5} + e^{-\mu_1-\mu_3-\mu_5}\right),
\end{aligned}
$$

$$
\begin{aligned}
& P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} > 0, X_{\{x_3\}} = 0\right) \\
= \; & e^{-\mu_3-\mu_5-\mu_6-\mu_7}\left(1 - e^{-\mu_1-\mu_4} - e^{-\mu_2-\mu_4} + e^{-\mu_1-\mu_2-\mu_4}\right),
\end{aligned}
$$

$$
\begin{aligned}
& P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} = 0, X_{\{x_3\}} > 0\right) \\
= \; & e^{-\mu_1-\mu_2-\mu_4-\mu_5-\mu_6-\mu_7}\left(1 - e^{-\mu_3}\right),
\end{aligned}
$$

$$
\begin{aligned}
& P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} > 0, X_{\{x_3\}} = 0\right) \\
= \; & e^{-\mu_1-\mu_3-\mu_4-\mu_5-\mu_6-\mu_7}\left(1 - e^{-\mu_2}\right),
\end{aligned}
$$

$$
\begin{aligned}
& P\left(X_{\{x_1\}} > 0, X_{\{x_2\}} = 0, X_{\{x_3\}} = 0\right) \\
= \; & e^{-\mu_2-\mu_3-\mu_4-\mu_5-\mu_6-\mu_7}\left(1 - e^{-\mu_1}\right),
\end{aligned}
$$

$$
\begin{aligned}
& P\left(X_{\{x_1\}} = 0, X_{\{x_2\}} = 0, X_{\{x_3\}} = 0\right) \\
= \; & e^{-\mu_1-\mu_2-\mu_3-\mu_4-\mu_5-\mu_6-\mu_7},
\end{aligned}
$$

and

$$
\begin{aligned}
& P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n, X_{\{x_3\}} = k\right) \\
= \; & \sum_{l_1=0}^{m} \sum_{l_4=0}^{m-l_1} \sum_{l_5=0}^{m-l_1-l_4} \sum_{l_2=0}^{n-m+l_1+l_5} \frac{\mu_1^{l_1} e^{-\mu_1}}{l_1!} \frac{\mu_4^{l_4} e^{-\mu_4}}{l_4!} \frac{\mu_5^{l_5} e^{-\mu_5}}{l_5!} \frac{\mu_7^{m-l_1-l_4-l_5} e^{-\mu_7}}{(m-l_1-l_4-l_5)!} \\
& \frac{\mu_2^{l_2} e^{-\mu_2}}{l_2!} \frac{\mu_6^{n-m+l_1+l_5-l_2} e^{-\mu_6}}{(n-m+l_1+l_5-l_2)!} \frac{\mu_3^{k-n+l_2+l_4-l_5} e^{-\mu_3}}{(k-n+l_2+l_4-l_5)!}.
\end{aligned}
$$

The above is valid when $n - m + l_1 + l_5 \leq k - m + l_1 + l_4$, otherwise, the formula is given by:

$$
\begin{aligned}
& P\left(X_{\{x_1\}} = m, X_{\{x_2\}} = n, X_{\{x_3\}} = k\right) \\
= \quad & \sum_{l_1=0}^{m} \sum_{l_4=0}^{m-l_1} \sum_{l_5=0}^{m-l_1-l_4} \sum_{l_3=0}^{k-m+l_1+l_4} \frac{\mu_1^{l_1} e^{-\mu_1}}{l_1!} \frac{\mu_4^{l_4} e^{-\mu_4}}{l_4!} \frac{\mu_5^{l_5} e^{-\mu_5}}{l_5!} \frac{\mu_7^{m-l_1-l_4-l_5} e^{-\mu_7}}{(m-l_1-l_4-l_5)!} \\
& \frac{\mu_2^{n-k+l_3-l_4+l_5} e^{-\mu_2}}{(n-k+l_3-l_4+l_5)!} \frac{\mu_6^{k-m+l_1+l_4-l_3} e^{-\mu_6}}{(k-m+l_1+l_4-l_3)!} \frac{\mu_3^{l_3} e^{-\mu_3}}{l_3!}\,.
\end{aligned}
$$

Now, by setting $\lambda = 8,\ r = 0.3,\ u_1 = 0.5,\ v_1 = 0.375$, sample size is 10000, we carry out the χ^2 test, the results are shown in the following table.

	o_i	e_i	$(o_i - e_i)^2/e_i$	$(o_i - e_i)/10000$	prob
(1,1,1)	347	337.018781	0.295606	0.000998	0.033702
(1,1,2)	273	267.302967	0.121421	0.000570	0.026730
(1,1,≥3)	202	185.475025	1.472299	0.001652	0.018548
(1,2,1)	286	267.302967	1.307801	0.001870	0.026730
(1,2,2)	261	262.994956	0.015133	-0.000199	0.026299
(1,2,≥3)	227	234.214240	0.222212	-0.000721	0.023421
(1,≥3,1)	163	185.475025	2.723422	-0.002248	0.018548
(1,≥3,2)	229	234.214240	0.116083	-0.000521	0.023421
(1,≥3,≥3)	326	310.586369	0.764940	0.001541	0.031059
(2,1,1)	264	267.302967	0.040814	-0.000330	0.026730
(2,1,2)	264	262.994956	0.003841	0.000101	0.026299
(2,1,≥3)	246	234.214240	0.593064	0.001179	0.023421
(2,2,1)	251	262.994956	0.547079	-0.001199	0.026299
(2,2,2)	314	317.845512	0.046526	-0.000385	0.031785
(2,2,≥3)	360	360.729776	0.001476	-0.000073	0.036073
(2,≥3,1)	261	234.214240	3.063336	0.002679	0.023421
(2,≥3,2)	368	360.729776	0.146526	0.000727	0.036073
(2,≥3,≥3)	631	625.136542	0.054996	0.000586	0.062514
(≥3,1,1)	201	185.475025	1.299500	0.001552	0.018548
(≥3,1,2)	225	234.214240	0.362498	-0.000921	0.023421
(≥3,1,≥3)	294	310.586369	0.885769	-0.001659	0.031059
(≥3,2,1)	241	234.214240	0.196600	0.000679	0.023421
(≥3,2,2)	324	360.729776	3.739853	-0.003673	0.036073
(≥3,2,≥3)	633	625.136542	0.098913	0.000786	0.062514
(≥3,≥3,1)	316	310.586369	0.094362	0.000541	0.031059
(≥3,≥3,2)	641	625.136542	0.402551	0.001586	0.062514
(≥3,≥3,≥3)	1852	1903.172602	1.375932	-0.005117	0.190317
	10000	10000	19.992552		

Table 4.4: The goodness-of-fit test for $(X_{\{x_1\}}, X_{\{x_2\}}, X_{\{x_3\}})$ with sample size = 10000

Bibliography

[1] Y. Cai and W.S. Kendall. Perfect simulation for correlated Poisson random variables conditioned to be positive. Research report 349, Department of Statistics, University of Warwick, 1999.

Other University of Warwick Department of Statistics Research Reports authored or co–authored by W.S. Kendall.

161: The Euclidean diffusion of shape.
162: Probability, convexity, and harmonic maps with small image I: Uniqueness and fine existence.
172: A spatial Markov property for nearest–neighbour Markov point processes.
181: Convexity and the hemisphere.
202: A remark on the proof of Itô's formula for C^2 functions of continuous semimartingales.
203: Computer algebra and stochastic calculus.
212: Convex geometry and nonconfluent Γ-martingales I: Tightness and strict convexity.
213: The Propeller: a counterexample to a conjectured criterion for the existence of certain convex functions.
214: Convex Geometry and nonconfluent Γ-martingales II: Well–posedness and Γ-martingale convergence.
216: *(with E. Hsu)* Limiting angle of Brownian motion in certain two–dimensional Cartan–Hadamard manifolds.
217: Symbolic Itô calculus: an introduction.
218: *(with H. Huang)* Correction note to "Martingales on manifolds and harmonic maps."
222: *(with O.E. Barndorff-Nielsen and P.E. Jupp)* Stochastic calculus, statistical asymptotics, Taylor strings and phyla.
223: Symbolic Itô calculus: an overview.
231: The radial part of a Γ-martingale and a non-implosion theorem.
236: Computer algebra in probability and statistics.
237: Computer algebra and yoke geometry I: When is an expression a tensor?
238: *Itovsn3*: doing stochastic calculus with *Mathematica*.
239: On the empty cells of Poisson histograms.
244: *(with M. Cranston and P. March)* The radial part of Brownian motion II: Its life and times on the cut locus.
247: Brownian motion and computer algebra (Text of talk to BAAS Science Festival '92, Southampton Wednesday 26 August 1992, with screenshots of illustrative animations).
257: Brownian motion and partial differential equations: from the heat equation to harmonic maps (Special invited lecture, 49th session of the ISI, Firenze).
260: Probability, convexity, and harmonic maps II: Smoothness via probabilistic gradient inequalities.
261: *(with G. Ben Arous and M. Cranston)* Coupling constructions for hypoelliptic diffusions: Two examples.
280: *(with M. Cranston and Yu. Kifer)* Gromov's hyperbolicity and Picard's little theorem for harmonic maps.

292: Perfect Simulation for the Area-Interaction Point Process.
293: *(with A.J. Baddeley and M.N.M. van Lieshout)* Quermass-interaction processes.
295: On some weighted Boolean models.
296: A diffusion model for Bookstein triangle shape.
301: COMPUTER ALGEBRA: an encyclopaedia article.
308: Perfect Simulation for Spatial Point Processes.
319: Geometry, statistics, and shape.
321: From Stochastic Parallel Transport to Harmonic Maps.
323: *(with E. Thönnes)* Perfect Simulation in Stochastic Geometry.
325: *(with J.M. Corcuera)* Riemannian barycentres and geodesic convexity.
327: Symbolic Itô calculus in *AXIOM*: an ongoing story.
328: *Itovsn3* in *AXIOM*: modules, algebras and stochastic differentials.
331: *(with K. Burdzy)* Efficient Markovian couplings: examples and counterexamples.
333: Stochastic calculus in *Mathematica*: software and examples.
341: Stationary countable dense random sets.
347: *(with J. Møller)* Perfect Metropolis-Hastings simulation of locally stable point processes.
348: *(with J. Møller)* Perfect implementation of a Metropolis-Hastings simulation of Markov point processes
349: *(with Y. Cai)* Perfect simulation for correlated Poisson random variables conditioned to be positive.
350: *(with Y. Cai)* Perfect implementation of simulation for conditioned Boolean Model via correlated Poisson random variables.

Also see the following related preprints

317: E. Thönnes: Perfect Simulation of some point processes for the impatient user.
334: M.N.M. van Lieshout and E. Thönnes: A Comparative Study on the Power of van Lieshout and Baddeley's J-function.

If you want copies of any of these reports then please mail your requests to the secretary of the Department of Statistics, University of Warwick, Coventry CV4 7AL, UK.