⟨revision 1a⟩ ≡

```
$Revision: 3.30 $◇
```

Macro referenced in scraps 1c, 16.

⟨date 1b⟩ ≡

```
$Date: 2000/01/17 11:13:19 $◇
```

Macro referenced in scraps 1c, 16.

⟨revision information 1c⟩ ≡

```
 * Code derived from $Source: /home/fisher/wsk/ms/Moeller/CPrograms/MH.w,v $
 *                        ⟨revision 1a⟩
 * ⟨date 1b⟩.◇
```

Macro referenced in scraps 4ab, 6ab, 7ab, 9ab, 11a, 12ac, 13b, 22ab, 42de, 48bc, 49ab, 50a, 51b, 56ab, 59, 60, 63, 64a.

– 348 (revised) –

Perfect implementation of a Metropolis-Hastings simulation of Markov point processes

by

W.S. Kendall and J. Møller

January 2000

This is `/home/fisher/wsk/ms/Moeller/CPrograms/MH.w` (`unix`) version 3.30.
Last edited: 11:13:19, 17/01/2000.

Perfect implementation of a Metropolis-Hastings simulation of Markov point processes

Wilfrid S Kendall[1]
(Statistics, University of Warwick)
and Jesper Møller[2]
(Mathematics, Aalborg University)

[1]Research supported by EPSRC grant GR/L56831 and EU TMR network ERB-FMRX-CT96-0095.
[2]Research supported by EU TMR network ERB-FMRX-CT96-0095.

Abstract

This document describes a `C` implementation of a perfect algorithm for a Metropolis-Hastings simulation of a point process, based on

- a decomposition into types relating to different square sub-regions, and
- a dominating process which is composed of independent discrete-time $M/M/1$ queues, one for each square sub-region, each in equilibrium.

Mathematical details are to be found in [6], which is the scientific paper corresponding to this research report. The differences between the current document and [6] are that

(a) this document gives a full listing of the program,

(b) this document does not discuss the underlying theory, which is covered in depth in [6].

Contents

Introduction

This document describes a `C` implementation of a perfect algorithm for a Metropolis-Hastings simulation of a point process, based on

- a decomposition into types relating to different square sub-regions, and
- a dominating process which is composed of independent discrete-time $M/M/1$ queues, one for each square sub-region, each in equilibrium.

Mathematical details are to be found in [6], which is the scientific paper corresponding to this research report. The differences between the current document and [6] are that:

(a) the current document gives a full listing of the program,

(b) [6] gives a proper exposition of the underlying theory (almost completely missing below!).

Here is version information for this document:

This is `/home/fisher/wsk/ms/Moeller/CPrograms/MH.w` (`unix`) version 3.30.
Last edited: 11:13:19, 17/01/2000.

Version 1 developed a first attempt at the basic data structures, to the point where a crude *CFTP* implementation was operating as it should. Version 2 was an abortive attempt at simplification, flawed by a mistaken assumption about possibilities for sandwiching. Version 3 is a rewrite based on experience gained with Version 1. The current level of version 3 implements three different interactions:

- `null` of no particular interest, but the base for everything else: numbers of points in different cells are independent but *not* Poisson;
- `poisson` producing a Poisson process;
- `strauss` producing a Strauss repulsion process.

Further work will produce an *area-interaction* option, either using the discrete-mesh approximation of [5] or the option proposed in [4], which uses marks which are Poisson clusters of points.

This document uses Briggs' `NuWeB` program [1], a small, free and portable `C` program based on Knuth's idea of *literate programming* [8, 9]. This enable the production of the LaTeX source `MH.tex` and the various source code and test files from a single `NuWeB` file `MH.w`. The insertions of numbered scraps of computer code at various places in the text of this report comprise the actual source code and various test files - what you see is what NuWeB makes sure the computer gets. The name of the relevant file is to be found at the head of the scrap. An index of identifiers is given at the end of the document.

The NuWeB source (in gzipped tarfile form, together with figures) can be found as a file of size 76k at

```
http://www.warwick.ac.uk/statsdept/Staff/WSK/perfect.html
```

Here is a brief guide to use of the program. The command

```
nuweb MH
```

will produce a Makefile (among several other files!), assuming NuWeB is installed on your system as nuweb. The Makefile is configured to use gcc under unix, so you may have to edit this accordingly. You are advised to edit the relevant part of the NuWeB source, which will otherwise overwrite your edits each time you run NuWeB! The invocation

```
make
```

will then compile and link the C code and run examples of the three main interactions. Here are slightly simplified versions:

```
mh-cftp -i null              -p 0.5 -n 16 -m 4
mh-cftp -i poisson           -p 0.5 -n 16 -m 4
mh-cftp -i "strauss(1,0.5)" -p 0.5 -n 16 -m 4
```

Here -i sets the interaction (in the case of the Strauss interaction with parameters set at interaction radius 1, γ parameter $\gamma = 0.5$); -p controls the underlying intensity of (typically) a reference Poisson process, -n sets the number of cells (defaulting to unit side-length), -m sets the side-length of the rectangular window in terms of number of cells. Other flags set the seed for the random number generator (-s), the time-duration of the first *CFTP* pass (-t), the side-length of a cell (-z), and various output options including in particular

- -P   postscript output; for example, to create a postscript file tmp.ps in unix

  ```
  mh-cftp ... -P > tmp.ps
  ```

- -x   cryptic output for statistical tests of null and poisson (see §7 below);
- -d   extremely long-winded output for debugging purposes.

The program is still in a relatively early stage of development. If you discover bugs then please do report them to WSK at W.S.Kendall@warwick.ac.uk. However, in order to make the correction process as rapid as possible, please try to isolate the problem to the least involved case (smallest number of points, smallest number of cells, poisson preferred to strauss, null to poisson, and so forth ...), and enclose in your email a line giving the simplest invocation leading to a problem, together with a description of what goes wrong.

Chapter 1

Parameters and data structures

In this chapter we describe the header files containing default parameters and basic data structures. The various structures allow us to build a sequence of marked incidents, which may be used to derive the *CFTP* sequences of coupled simulations of upper- and lower-sandwich processes. Table 1 sets forth these structures in the header files used to define them, together with associated functions. (Bracketed functions are private to the corresponding `C` files.) There is a logical dependence running down the table: structures require all the header files below them as well as their corresponding header file!

header file	*structure*	*functions*
`sim.h`	`Sim`	`make_initial_incident`
		`make_incident`
`inc.h`	`Inc`	`make_null_incident`
`cand.h`	`BP`	`interaction_choice`
`pop.h`	`Pop`	`pick_random`
		`add_to_population`
		`add_to_populations`
		`take_from_population`
		`take_from_populations`
		`copy_population`
		`copy_populations`
		`free_population`
		`free_populations`
		`belong`
		`(belong2)`
`indiv.h`	`Indiv`	
`point.h`	`Pt`	

1.1 Default parameters

The header file `mh-cftp.h` contains global definitions.

"mh-cftp.h" 4a ≡

```
/*
 * File mh-cftp.h ⟨revision information 1c⟩
 */
◇
```

File defined by scraps 4acde, 5abcd.

Note that `mh-cftp.c` includes several other header files!

"mh-cftp.c" 4b ≡

```
/*
 * File mh-cftp.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "sim.h"
#include "CFTP.h"
#include "display.h"
#include "rand.h"
#include "sched.h"
#include "loc.h"
◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

The header file defines global macros holding default values for the total number of types (typically one type per square sub-region in the cell) and cell geometry,

"mh-cftp.h" 4c ≡

```

#define NTYPES 1
#define SCALE 1
#define XGRID 1
◇
```

File defined by scraps 4acde, 5abcd.

default mean values for the underlying Geometric equilibrium distributions,

"mh-cftp.h" 4d ≡

```

#define GEOMETRIC_PROB 0.5
◇
```

File defined by scraps 4acde, 5abcd.

and a default value for the initial time span of CFTP simulation.

"mh-cftp.h" 4e ≡

```

#define INITIAL_TIME_SPAN -2
◇
```

File defined by scraps 4acde, 5abcd.

It also defines global macros holding strings identifying various implemented interactions,

"mh-cftp.h" 5a ≡

```
#define DEFAULT_INTERACTION "null"
#define POISSON_INTERACTION "poisson"
#define STRAUSS_INTERACTION "strauss"
◇
```

File defined by scraps 4acde, 5abcd.

values for truth and falsity,

"mh-cftp.h" 5b ≡

```
#define TRUE 1
#define FALSE 0
◇
```

File defined by scraps 4acde, 5abcd.

and pre- and post-ambles for `verbatim` display of diagnostic output.

"mh-cftp.h" 5c ≡

```
#define BEGIN_VERBATIM puts("\\begin{verbatim}")
#define END_VERBATIM   puts("\\end{verbatim}")
#define SEPARATOR1 puts("==================")
#define SEPARATOR2 puts("------------------")

◇
```

File defined by scraps 4acde, 5abcd.

Finally we need `ntypes` (the number of different types or cells within which individuals can lie) and `debug` (which controls whether or not the program outputs diagnostics).

"mh-cftp.h" 5d ≡

```
extern int ntypes;
extern int debug;

◇
```

File defined by scraps 4acde, 5abcd.

1.2 Simulation data structure

This structure is a utility structure allowing us to keep track of relevant features of the simulation through the backwards and forwards sweeps of the *CFTP* algorithm. We include the header file describing the `Inc` structure, which in turn refers to the header file describing the `Pop` structure. (The `Inc` structure is described below in §1.3. The `Pop` structure is described below in §1.4. We use `*Pop` here because we deal with collections of sub-populations distinguished by different types . . . .)

"sim.h" 6a ≡
```
/*
 * File sim.h ⟨revision information 1c⟩
 */

#include "inc.h"
◇
```
File defined by scraps 6acd.

"sim.c" 6b ≡
```
/*
 * File sim.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "sim.h"
#include "death.h"
#include "birth.h"
#include "rand.h"
◇
```
File defined by scraps 6b, 45ab, 48a.

It is denoted by `Sim`, and points to an `Inc` or incident of the simulation, as well as two sandwiching populations. Figure 1.1 illustrates this structure.

"sim.h" 6c ≡
```

typedef struct Simulation
  {
    struct Incident *inc;
    struct Population **L;
    struct Population **U;
  } Sim;

◇
```
File defined by scraps 6acd.

Public functions for the `Sim` structure (actually strongly related to the `Inc` structure!) are as follows:

- A creator function `make_initial_incident` to make an incident which creates an individual at the initialization stage (which, because of time reversal, corresponds to an individual dying in the simulation!),
- and another creator function `make_incident` to make an incident to extend the simulation, in which an individual is born or dies in the upper-sandwich population but nothing happens in the lower-sandwich pop.

  "sim.h" 6d ≡
  ```

  extern Sim *make_initial_incident (int, Sim *, double);
  extern Sim *make_incident (int, double, Sim *, double);
  ◇
  ```
  File defined by scraps 6acd.

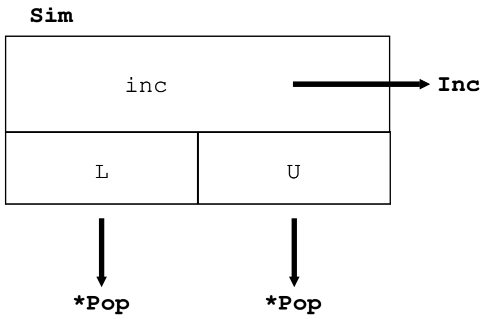

Figure 1.1: The structure `Sim`, with a pointer to an `Inc` and pointers for two sandwiching populations.

1.3 Incident data structure

The basic structure determines an *incident* of the simulation. This is denoted by `Inc`, and occurs as the typical node of an *incident list* which records the details of the CFTP simulation. As the CFTP algorithm progresses so the incident list is extended backwards in time and updated to reflect the increasing coalescence of the sandwiching processes. We include the header file describing the `Pop` structure.

"inc.h" 7a ≡

```
/*
 * File inc.h ⟨revision information 1c⟩
 */

#include "cand.h"
◇
```

File defined by scraps 7a, 8ab.

"inc.c" 7b ≡

```
/*
 * File inc.c ⟨revision information 1c⟩
 */

#include <stdlib.h>

#include "inc.h"
◇
```

File defined by scraps 7b, 44.

Each node of the list points to a further `Inc` structure `next`. It also points to two pairs of `Indiv` structures `Lbirth` and `Ldeath`, `Ubirth` and `Udeath`, which record individuals to be born or to die in each of the two current sandwiching populations at this incident. (The `Indiv` structure is described below in §1.5.) Any of these pointers may be `NULL`, corresponding to end of the incident list (for `next`) or to a birth or death not being required.

Finally the `Inc` structure has three fields `type` (an `int`), `time` (a `double`), `flag` (a `char`) recording the type of the population to be affected, the time of

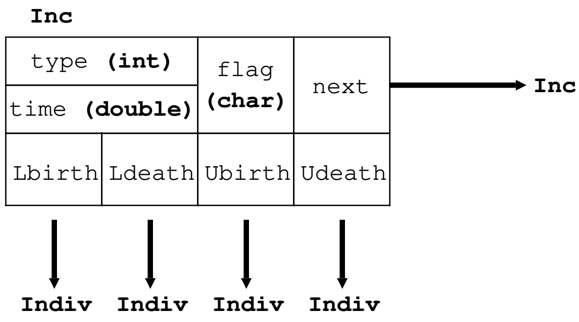

Figure 1.2: The structure `Inc`, with pointers to another `Inc` and pointers for `birth` and `death` for both of the two current sandwiching populations. It also contains fields `type` (integer) and `time` (floating point).

the incident, and whether the incident has yet been updated by a *CFTP* pass. Figure 1.2 illustrates this structure.

"inc.h" 8a ≡

```
typedef struct Incident
  {
    int type;
    double time;
    char flag;
    struct Incident *next;
    struct Individual *Lbirth;
    struct Individual *Ldeath;
    struct Individual *Ubirth;
    struct Individual *Udeath;
  } Inc;
```

◇

File defined by scraps 7a, 8ab.

An assertion must be maintained to guarantee correctness for this structure. If `inc` points to an `Inc` structure then:

- If `inc->Lbirth` exists it must equal `inc->Ubirth`;

We will refer to this assertion elsewhere in the code.

We will use a function which delivers a null incident:

"inc.h" 8b ≡

```
Inc * make_null_incident (int, Inc *, double);
```

◇

File defined by scraps 7a, 8ab.

1.4 Population data structure

The `Sim` data structure points to collections of *populations* distinguished by types. The number of types is held in the global variable `ntypes`. Thus for example if `sim` points to an `Sim` structure then `sim->U` points to a `*Pop` which can be viewed as an array of `ntypes` different `Pop` structures, one for each different type. Figure 1.3 illustrates this structure. We include the header file describing the `BP` structure.

"pop.h" 9a ≡
```
/*
 * File pop.h ⟨revision information 1c⟩
 */

#include "indiv.h"
◇
```
File defined by scraps 9ac, 10b.

"pop.c" 9b ≡
```
/*
 * File pop.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "pop.h"
#include "rand.h"

int belong2 (Indiv *person, Pop *p);
◇
```
File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

So the data structure `Pop` serves as a node on a list and points to an `Indiv` as represented in Figure 1.3. (The `Indiv` structure is described below in §1.5.) A further `int` field `n` keeps tally of the number of individuals in the pop.

"pop.h" 9c ≡
```

typedef struct Population
  {
    struct Individual *subj;
    int n;
    struct Population *next;
  } Pop;

◇
```
File defined by scraps 9ac, 10b.

Again we will in due course assert some invariants: if `pop` points to a `*Pop` structure then for each `i` in $0, \ldots,$ `ntypes` $- 1$ we must have

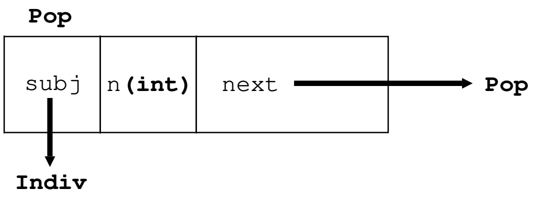

Figure 1.3: The structure `Pop`, with pointers to another `Pop` and to an `Indiv`. The `int` field `n` counts the number of individuals in this `Pop`. Note that typically structures point to an array `*Pop` rather than a single `Pop` structure, thus allowing us to consider populations structured into `ntypes` different types.

- If `(pop[i])->next` is not `NULL` then `(pop[i])->n` is obtained by adding 1 to `(pop[i])->next->n`;
- otherwise `(pop[i])->n` equals 1.

The function `n` is used to access the number, so we don't have to concern ourselves with the exceptional case of an empty population:

"pop.c" 10a ≡

```
int
n (Pop * p)
{
  return (p ? p->n : 0);
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Various functions are associated to this structure as indicated below:

"pop.h" 10b ≡

```
extern int n (Pop *);
extern int equal_populations (Pop **, Pop **);
extern int belong (Indiv *, Pop **);

extern Indiv *pick_random (Pop *);

extern Pop * add_to_population(Pop *, Indiv *);
extern void add_to_populations(Pop **, Indiv *);

extern Pop * take_from_population(Pop *, Indiv *);
extern void take_from_populations(Pop **, Indiv *);

extern Pop * copy_population(Pop *);
extern Pop ** copy_populations(Pop **);

extern void free_population(Pop *);
extern void free_populations(Pop **);
◇
```

File defined by scraps 9ac, 10b.

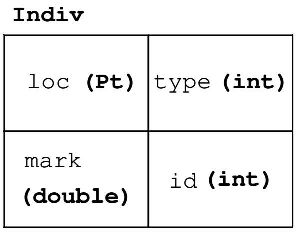

Figure 1.4: The structure `Indiv`, containing fields `location` (a `Pt` structure), `type` (an integer which should be positive), `id` (an integer), and `mark` (a floating-point number).

1.5 Individual data structure

The data structure `Indiv` carries information about a given individual: it has fields `loc` (a `Pt` structure: see §1.6 below), `type`, `id` (these two being `int` fields: `type` should be positive), and `mark`. We `include` the header file describing the `Pt` structure.

"indiv.h" 11a ≡

```
/*
 * File indiv.h ⟨revision information 1c⟩
 */

#include "point.h"
◇
```

File defined by scraps 11abc.

Figure 1.4 illustrates this structure.

"indiv.h" 11b ≡

```

typedef struct Individual
  {
    struct Point loc;
    int type;
    int id;
    double mark;
  } Indiv;
◇
```

File defined by scraps 11abc.

We also add a macro for the null `Indiv` pointer.

"indiv.h" 11c ≡

```

#define NOBODY (Indiv *) NULL
◇
```

File defined by scraps 11abc.

1.6 Point data structure

The data structure `Pt` has two `double` fields `x` and `y`.

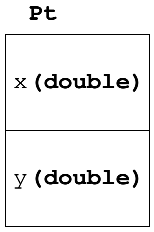

Figure 1.5: The structure `Pt`, containing `double` fields `x` and `y`.

"point.h" 12a ≡

```
/*
 * File point.h ⟨revision information 1c⟩
 */
◇
```

File defined by scraps 12ab.

Figure 1.5 illustrates this structure.

"point.h" 12b ≡

```

typedef struct Point
  {
    double x, y;
  } Pt;

◇
```

File defined by scraps 12ab.

1.7 Birth pair data structure

The data structure `BP` has two `Indiv` pointers as fields: one for the upper birth candidate, one for the lower. We include the header file describing the `Indiv` structure.

"cand.h" 12c ≡

```
/*
 * File cand.h ⟨revision information 1c⟩
 */

#include "pop.h"

#define MAXARGNO 1

typedef struct BirthPair
  {
    Indiv *ubi;
    Indiv *lbi;
  } BP;

◇
```

File defined by scraps 12c, 13a.

We export a function `interaction_choice` which creates and returns the appropriate birth candidates.

"cand.h" 13a ≡

```
extern BP
(* interaction_choice (char *, int))
    (Indiv *, Indiv *, Pop **, Pop **, double *);
◇
```

File defined by scraps 12c, 13a.

Here is the header file.

"cand.c" 13b ≡

```
/*
 * File cand.c ⟨revision information 1c⟩
 */
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "mh-cftp.h"
#include "cand.h"
#include "loc.h"
#include "rand.h"

◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

Chapter 2

Basic structure of algorithm

2.1 Main program

Here is the main program for the CFTP simulation, as contained in `mh-cftp.c`. We begin with some default initializations of global variables.

"mh-cftp.c" 14a ≡

```
int ntypes  = NTYPES;
int debug   = FALSE;
◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

The main program has command-line input, the processing of which is our first concern.

"mh-cftp.c" 14b ≡

```
int
main (int argc, char *argv[])
{
  Sim *sim;
  double *prob, defaultp = GEOMETRIC_PROB,
       scale=SCALE;
  int *schedule, seed = 1,
       concise = FALSE, postscript = FALSE,
       i, k, xgrid=XGRID, wordnumber,
       t0 = 0, t1 = INITIAL_TIME_SPAN;
  char *interaction, **word;
  BP (* interaction_fn)
        (Indiv *, Indiv *,
         Pop **, Pop **, double *);
 ◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

We set the default interaction to be `poisson`:

"mh-cftp.c" 14c ≡

```
  interaction = POISSON_INTERACTION;
 ◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

The next step is to read in command-line arguments, and to use them to alter defaults for some basic parameters (`seed`, `t0`, `t1`, ...). Note that most options are validated with error-trapping! Also note that `concise` is automatically disabled if `debug` is set. We save the pointers to the arguments for later use.

"mh-cftp.c" 15 ≡

```
    wordnumber = argc;
    word = (char **) malloc (wordnumber*sizeof(char *));
    for (i=0; i<wordnumber; i++) word[i] = argv[i];

    while (--argc > 0 && (*++argv)[0] == '-')
      {
        switch (*++argv[0])
          {
          case 's':
            seed = atoi ((++argv)[0]);
            argc--;
            break;
          case 't':
            t1 = -atoi ((++argv)[0]);
            if (!(t1 < 0))
              {
                puts ("switch (-t) value must be positive!");
                exit (11);
              };
            argc--;
            break;
          case 'n':
            ntypes = atoi ((++argv)[0]);
            if (!(ntypes > 0))
              {
                puts ("switch (-n) value must be positive!");
                exit (12);
              };
            argc--;
            break;
          case 'p':
            defaultp = atof ((++argv)[0]);
            argc--;
            if (defaultp <= 0)
              {
                puts ("probability (-p) should lie in (0, 1)");
                exit (13);
              };
            break;
          case 'm':
            xgrid = atoi ((++argv)[0]);
            if (xgrid<=0)
              {
                puts("xgrid value (-m) should be non-positive.");
                exit (15);
              };
            argc--;
```

```
            break;
          case 'z':
            scale = atof ((++argv)[0]);
            if (scale<=0)
              {
                puts("Switch -s should be positive");
                exit (16);
              };
            argc--;
            break;
          case 'd':
            debug = TRUE;
            break;
          case 'P':
            postscript = TRUE;
            break;
          case 'x':
            concise = TRUE;
            break;
          case 'i':
            interaction =
              (char *) malloc ((1+strlen ((++argv)[0]) ) * sizeof (char));
            argc--;
            strcpy (interaction, argv[0]);
            break;
          case 'l':
            open_logfile((++argv)[0]);
            argc--;
            break;
          default:
            printf ("Unknown switch -%c\n", *argv[0]);
            puts ("Useage:");
            printf ("mh-cftp -x -d -P -l [STRING] -s [INT] -t [INT] ");
            puts ("-n [INT] -p [FLOAT] -i [STRING]");
            puts ("-m [INT] -z [FLOAT]\n");
            exit (10);
          };
      };
    set_xgrid(xgrid);
    set_scale(scale);
    if (debug)
      concise = FALSE;
    if (postscript) {
        concise = TRUE;
        debug = FALSE;
      };
◇
```
File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

We first report on the parameters (if in verbose mode!).

"mh-cftp.c" 16 ≡

```
    if (!concise)
      {
```

```
      if (debug)
        BEGIN_VERBATIM;
      puts ("$Source: /home/fisher/wsk/ms/Moeller/CPrograms/MH.w,v $ ⟨revision
1a⟩");
      puts ("       ⟨date 1b⟩");
      printf ("   seed = %d\n", seed);
      printf ("   t1 = %d\n", t1);
      printf ("   ntypes = %d\n", ntypes);
      printf ("   default probability = %f\n", defaultp);
      printf ("   interaction is %s\n", interaction);
    };
  state_geometry (concise);
  if (debug)
    {
      printf("Debugging output is attached at level %d:\n",
        debug);
      SEPARATOR1;
      END_VERBATIM;
    };
◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

We now print out the command-line, and cell size and geometry, in case it is needed for future reference. To avoid printing it in LaTeX output the command-line is prefixed by `%`. Note, we avoid printing anything in `postscript` because of the way in which `malloc` is implemented in some versions of `C`.

"mh-cftp.c" 17a ≡

```
  if (!postscript)
    {
      printf ("%%Output produced by: ");
      i = wordnumber;
      while (i)
        {
          printf (" %s", word[wordnumber - i]);
          i--;
        };
      putchar ('\n');
    };
◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

The choice of interaction is now implemented. Notice that the details of the interaction implementation are all concealed in `cand.h` and `cand.c`!

"mh-cftp.c" 17b ≡

```
  interaction_fn = (* interaction_choice)(interaction,postscript);
◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

We now explicitly initialize the components of the `Sim` structure, the schedule (which will specify how we should run through the various types), and set the seed for the random number generator.

"mh-cftp.c" 18a ≡

```
    sim      = (Sim *) malloc (sizeof(Sim));
    sim->inc = (Inc *) NULL;
    sim->L   = (Pop **) malloc (ntypes * sizeof(Pop *));
    sim->U   = (Pop **) malloc (ntypes * sizeof(Pop *));
    for (k = 0; k < ntypes; k++)
      {
        (sim->L)[k]  = (Pop *) NULL;
        (sim->U)[k]  = (Pop *) NULL;
      };

    schedule = (int *) malloc ((2*ntypes)*sizeof(int));

    INITIALIZE_RANDOM (seed);
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

Now we discuss the Geometric equilibrium distributions for the various sub-populations of the "dominating process" (the initial upper-sandwich process) at time zero.

The discrete-time $M/M/1$ queueing process which serves as the dominating process for this perfect simulation has the following transition probabilities: for a fixed positive parameter p set by the switch `-p` in the command-line,

$$\begin{aligned} N \to N+1 & \qquad \text{with probability} \qquad \frac{p}{1+p} \\ N+1 \to N & \qquad \text{with probability} \qquad \frac{1}{1+p} \end{aligned} \tag{2.1}$$

Therefore its equilibrium distribution $\{\pi_n : n \geq 0\}$ can be obtained using the equations of detailed balance: for $n \geq 0$ we find

$$\pi_n \cdot \frac{p}{1+p} \quad = \quad \pi_{n+1} \cdot \frac{1}{1+p} \tag{2.2}$$

and so

$$\pi_{n+1} \quad = \quad p \cdot \pi_n \quad = \quad p^{n+1} \cdot \pi_0$$

We recognize this as a Geometric distribution with parameter p:

$$\pi_n \quad = \quad p^n(1-p) \quad \text{for } n \geq 0\,. \tag{2.3}$$

It is of course vital that this equilibrium distribution can be simulated with great ease, as this allows us to simulate the dominating process in reverse-time.

Implementing this in C, we obtain:

"mh-cftp.c" 18b ≡

```
    prob = (double *) malloc (ntypes * sizeof (double));
    for (k = 0; k < ntypes; k++)
      prob[k] = defaultp;
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

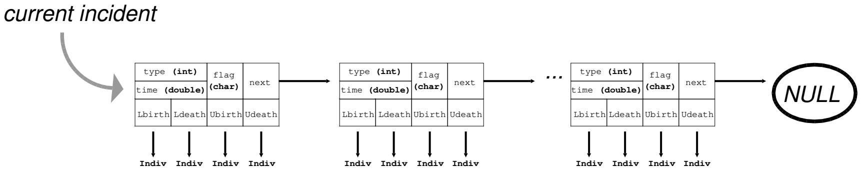

Figure 2.1: The incident list is built out of a series of `Inc` structures, each pointing to the next. Each structure is set up using the function `make_initial_incident`.

The initialization phase sets up the sequence of incidents downstream of time zero, corresponding to a steady state of upper-sandwich of Geometric distributions with the same parameter p. See Figure 2.1 for a schematic illustration of the resulting structure. The incidents are set up using `make_initial_incident` (declared in `inc.h`). Notice that `make_initial_incident` actually uses the `Sim` structure which points to the current incident!

"mh-cftp.c" 19a ≡

```
    if (debug>1) BEGIN_VERBATIM;
    for (k = 0; k < ntypes; k++)
      {
        i = Geometric (prob[k]);
        while (i > 0)
          {
            sim = make_initial_incident (k, sim, 0.0);
            i--;
          };
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

We interpose higher-level debugging code to trace the detailed behaviour of the populations:

"mh-cftp.c" 19b ≡

```
    if(debug>1 && sim->inc)
      print_diagnostic(sim->inc, (sim->U)[k]);
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

"mh-cftp.c" 19c ≡

```
      };
    if (debug>1) END_VERBATIM;
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

Finally the *CFTP* phase itself repeatedly extends the incident list by generating incidents over redoubled ranges until the `CFTP` function returns a `TRUE` value. We set the schedule repeatedly within the extension phase to allow for the possibility of the schedule being rand. (Notice in this case that the schedule at a fixed time does not vary over repeated *CFTP* runs, being recorded in the appropriate `Inc` structures!) The second argument of `make_incident` is the birth probability. *However* the generation of incidents is being carried out using the time-reversed version of the underlying $M/M/1$ queue, so we have to interchange birth and death probabilities, since a birth in reverse-time corresponds to a death in forwards-time. Consequently we use $1/(1+\texttt{prob}[\texttt{k}])$ for the birth probability here rather than $\texttt{prob}[\texttt{k}]/(1+\texttt{prob}[\texttt{k}])$.

"mh-cftp.c" 20a ≡

```
    assert (t0 == 0 && t1 < 0);
    do
      {
        if (debug)
          {
            BEGIN_VERBATIM;
            SEPARATOR1;
            printf ("Simulating over [%d,%d]\n", t1, t0);
          };
        do
          {
            t0--;
            schedule = make_schedule (ntypes);
            for (i = 0; i < 2 * ntypes; i++)
              {
                k = schedule[i];
                sim = make_incident (k, 1 / (1 + prob[k]),
                                     sim, (double) t0);
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

Here again we interpose higher diagnostic code to trace what is going on!

"mh-cftp.c" 20b ≡

```
  if (debug>1)
    {
      if (sim->inc->Ubirth)
        {
          printf("%2db ",sim->inc->Ubirth->id);
        }
      else if (sim->inc->Udeath)
        {
          printf("%2dd ",sim->inc->Udeath->id);
        }
      else
        {
          printf("    ");
        };
    print_diagnostic (sim->inc, (sim->U)[k]);
    };
```
◇

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

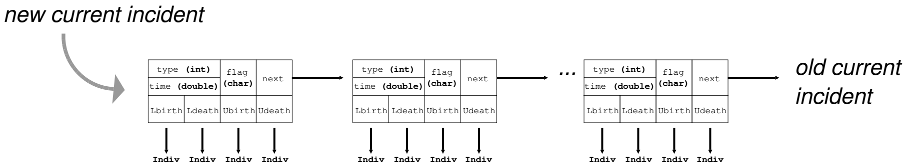

Figure 2.2: The loop involving `CFTP` replaces `sim->incident` by a list of `Inc` structures pointing to `inc` and then performs a "coupling-from-the-past" simulation run.

"mh-cftp.c" 21 ≡

```
                };
            }
          while (t0 > t1);
          t1 *= 2;
          if (debug) END_VERBATIM;
        }
      while (!CFTP (sim, prob,
                    interaction, interaction_fn,
                    concise, postscript));

      close_logfile();
      if (postscript) close_postscript(wordnumber,word,defaultp);

      exit (0);
    }
    ◇
```

File defined by scraps 4b, 14abc, 15, 16, 17ab, 18ab, 19abc, 20ab, 21.

In the course of the main loop above we progressively replace `sim->incident` by a further list of `Inc` structures pointing to the old `Inc` (see Figure 2.2) and then use this new structure to check for coalescence of an extended backwards simulation at time zero.

2.2 The CFTP construction

Once the incident list is constructed, we may carry out the CFTP procedure to see whether we get coalescence. Here is where the process of interest is simulated! We add functions to open and close log files/

"CFTP.h" 22a ≡

```
/*
 * File CFTP.h⟨revision information 1c⟩
 */

#define NENTRY 4

extern int CFTP (Sim *, double *,
                 char *,
                 BP (*)
                       (Indiv *, Indiv *,
                        Pop **, Pop **, double *),
                 int, int);

extern void open_logfile (char *);
extern void close_logfile (void);
◇
```

Notice that we must include `stdlib.h` in order to obtain access to memory allocation routines!

"CFTP.c" 22b ≡

```
/*
 * File  CFTP.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "sim.h"
#include "CFTP.h"
#include "display.h"
#include "check.h"
#include "rand.h"
◇
```

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

We begin the *CFTP* forward pass by setting up and checking the incident and populations referred to by the `sim` argument. Notice that we can assume `sim->L` is empty! Notice we add a global `FILE` pointer for the log file.

"CFTP.c" 22c ≡

```

static FILE *diagfile = (FILE *) NULL;

int
CFTP (Sim * sim, double *prob,
      char *interaction,
      BP (* interaction_fn)
        (Indiv *, Indiv *,
```

```
            Pop **, Pop **, double *),
        int concise, int postscript)
  {
    Inc * inc;
    Pop **LL, **L, **U, **UU;
    Indiv *lbi, *ldi, *ubi, *udi, *dom;
    BP bcand;
    double t, cprob;
    int k, test;

    inc = sim->inc;

    LL = copy_populations (sim->L);
    L  = copy_populations (sim->L);
    U  = copy_populations (sim->U);
    UU = copy_populations (sim->U);

    assert (check_populations (LL));
    assert (check_populations (L));
    assert (check_populations (U));
    assert (check_populations (UU));

    assert (check_equality (LL, L));
    assert (check_inclusion (L, U));
    assert (check_equality (U, UU));
  ◇
```

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

We now record and output the time `t` of the starting incident `inc`, and output initial values of the two sandwiching populations `L`, `U` together with the history of the original population derived from the current earliest incident `sim->inc`. We also add a title for the log file if used.

"CFTP.c" 23 ≡

```
    t = inc->time;

    if(debug)
      {
        BEGIN_VERBATIM;
        printf("CFTP simulation using %s interaction\n",
          interaction);
        printf("Initial populations at time %f\n",t);
        printf("Lower bound on population:");
        print_populations (L);
        SEPARATOR2;
        printf("Upper bound on population:");
        print_populations (U);
        SEPARATOR2;
        END_VERBATIM;
        print_history (inc, U);
      };

    if (diagfile)
      fprintf(diagfile, "ldi lbi udi ubi nL nU Type\n");
```

◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

We now carry out the forward pass of the *CFTP* algorithm, looping till `t` is zero or till we run out of simulation incidents. At each stage we report the current time.

"CFTP.c" 24 ≡

```
    while (inc && t<0)
      {
        if (debug)
          {
            BEGIN_VERBATIM;
            printf("Evolution at time %f\n", t);
          };
        while (inc && (inc->time) <=t)
          {
```

◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

Now we must sandwich on the sub-population `inc->type` at time `inc->time`. We must take great care to maintain the sandwiching relationship

$$\mathtt{LL} \subseteq \mathtt{L} \subseteq \mathtt{U} \subseteq \mathtt{LL} \tag{2.4}$$

while using the correct conditional probabilities for the evolution of `L` and `U`. (See Fill and Machida [3] for an indication of just how non-trivial such monotonicity issues can be!) We then update `LL` and `UU` using old `inc` data, place new lower/upper birth/death candidates in `inc`, and finally update `L` and `U` using new `inc` data.

The required conditional probabilities have been obtained using an inhomogeneous Markov chain and checked by conditional probability calculations carried out in *Mathematica* (see [6] for details). The results are as follows, in the particular example where we condition on the deaths or absence of deaths in populations `LL` and `UU`, and calculate the conditional probability of a death or absence thereof in `U`. Note that we suppress dependence on the sub-population `k` and further abbreviate `prob[k]` by `p`.

`UU` event	`LL` event	`U` event	$\mathbb{P}$[Uevent|other two events]
no d	no d	no d	1
d in `U-LL`	no d	d in `U-LL`	1
d in `UU-U`	no d	no d	$(\mathtt{n(LL)}+\mathtt{p})/(\mathtt{n(U)}+\mathtt{p})$
d in `UU-U`	no d	d in `U-LL`	$(\mathtt{n(U)}-\mathtt{n(LL)})/(\mathtt{n(U)}+\mathtt{p})$
no d	d in `LL`	d in `LL`	1
d in `U-LL`	d in `LL`	d in `U-LL`	1
d in `UU-U`	d in `LL`	d in `LL`	$(\mathtt{n(LL)}+\mathtt{p})/(\mathtt{n(U)}+\mathtt{p})$
d in `UU-U`	d in `LL`	d in `U-LL`	$(\mathtt{n(U)}-\mathtt{n(LL)})/(\mathtt{n(U)}+\mathtt{p})$

This must be further adapted to ensure that the *dominating* process, the relevant $M/M/1$ queue, is respected, in the sense that if it experiences a death then so do all non-empty processes in the sandwiching. But this is trivial! Death in the $M/M/1$ queue is signalled by `inc->flag=='d'` (after processing by a code fragment below which reduces the flag to lower case). This has probability

$\mathtt{p}/(1+\mathtt{p})$, and so we require this event to coincide with the event that the underlying inhomogeneous Markov chain referred to above makes its first step from 0 to 1. Hence conditioning on `inc->flag=='d'` corresponds exactly to conditioning on there being a death in the minimal non-zero lower sandwiching population: the Markov property now allows us to deal with this with essentially no further calculation.

We find that we can sandwich deaths to maintain the monotonicity of overall death probabilities, so that

$$\mathtt{LL}\text{death} \Rightarrow \mathtt{L}\text{death} \Rightarrow \mathtt{U}\text{death} \Rightarrow \mathtt{UU}\text{death}\,, \tag{2.5}$$

while the monotonicity of death rates (running in the *reverse* direction!) allows us to fulfil the relationships

$$\begin{aligned} \mathtt{UU}\text{death in } \mathtt{U} &\Rightarrow \mathtt{U}\text{death} && (2.6)\\ \mathtt{U}\text{death in } \mathtt{L} &\Rightarrow \mathtt{L}\text{death} && (2.7)\\ \mathtt{L}\text{death in } \mathtt{LL} &\Rightarrow \mathtt{LL}\text{death}\,. && (2.8) \end{aligned}$$

Finally we implement birth proposals very simply, by requiring that in a sandwiching population $\mathtt{L} \subseteq \mathtt{U}$ there is a $\mathtt{L}$-birth if and only if there is no $\mathtt{U}$-death, and requiring the analogous relation between $\mathtt{U}$-births and $\mathtt{L}$-deaths, except that in both cases we require that if the alternate sub-population is *empty* then we reduce the birth probability from 1 to $\mathtt{p}/(1+\mathtt{p})$.

We can now compute the equilibrium distribution for the coalesced Markov chain: the transition probabilities are

$$\begin{array}{rll} x \to x+1 & \text{with probability} & \dfrac{p}{x+p} \ \text{ if } x>0 \\ 0 \to 1 & \text{with probability} & \dfrac{p}{1+p} \\ x+1 \to x & \text{with probability} & \dfrac{x+1}{x+1+p}\,. \end{array}$$

Hence for $x > 0$ we have equations of detailed balance

$$\pi_x \frac{p}{x+p} \quad = \quad \pi_{x+1} \frac{x+1}{x+1+p} \tag{2.9}$$

which can be solved (for $x > 0$) to give

$$\begin{aligned} \pi_{x+1} &= \frac{x+1+p}{x+p}\frac{p}{x+1}\pi_x \;=\; \frac{x+1+p}{1+p}\frac{p^x}{(x+1)!}\pi_1 \\ &= \pi_1 \times \frac{1}{1+p}\left(\frac{p^x}{x!} + \frac{p^{x+1}}{(x+1)!}\right)\,. \end{aligned}$$

Of course for $x = 1$ we have $\pi_1 = p\pi_0$. We can therefore sum the probabilities over $x = 0,1,\ldots$, and telescope terms to obtain

$$\begin{aligned} \sum_{x=0}^{\infty} \pi_x \;=\; 1 &= \pi_1 \times \left(\frac{1}{p} + \frac{1}{1+p} \times \sum_{x=1}^{\infty}\left[\frac{p^x}{x!} + \frac{p^{x-1}}{(x-1)!}\right]\right) \\ &= \pi_1 \times \left[\frac{1}{p} + \frac{1}{1+p} \times (2e^p - 1)\right] \;=\; \pi_1 \times \frac{2p\,e^p + 1}{p(1+p)} \end{aligned}$$

Thus the equilibrium distribution is given by the probability mass function

$$\begin{aligned}
\pi_x &= \frac{p}{2p+e^{-p}} \times \left(\frac{p^x}{x!}e^{-p} + \frac{p^{x-1}}{(x-1)!}e^{-p}\right) \qquad \text{if } x>0 \\
\pi_0 &= \frac{(1+p)e^{-p}}{2p+e^{-p}}
\end{aligned} \tag{2.10}$$

Hence we find the equilibrium distribution can be related to that of a Poisson(p) random variable in the following way: X has the equilibrium distribution if with probability $p/(2p+e^{-p})$ it is distributed as $1+\text{Poisson}(p)$, and with probability $p/(2p+e^{-p})$ it is distributed as Poisson(p) itself, and otherwise it is zero. Hence its mean value is $p(2p+1)/(2p+e^{-p})$. We use these facts when testing the program (see Chapter 7).

We begin the simulation by recording the current births and deaths in the previous sandwiching populations and the current sub-population type, and by making a number of assertions reflecting the above monotonicity relationships.

"CFTP.c" 26a ≡

```
lbi = inc->Lbirth;
ldi = inc->Ldeath;
ubi = inc->Ubirth;
udi = inc->Udeath;

assert(!ldi || udi);
assert (!lbi || (ubi && lbi->id == ubi->id));
assert (!belong (udi,LL) || (ldi && udi->id == ldi->id));

k = inc->type;

assert(!(ldi && lbi));
```

◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

Now we must implement the sandwiching using the above conditional probabilities. First of all we must apply a correction if the current incident has not yet been processed (which occurs if `inc->flag` is in upper case). For in this case there is a UU-death with probability $1/(1+\texttt{prob}[\mathrm{k}]$ and we need a candidate for UU-death which has the larger probability $\mathtt{n}(\mathtt{UU}[\mathrm{k}])/(\mathtt{n}(\mathtt{UU}[\mathrm{k}])+\texttt{prob}[\mathrm{k}]$. To correct this we need a conditional probability of making `udi` into a UU death (given it is not already the case) as follows:

$$\frac{\frac{\mathtt{n}(\mathtt{UU}[\mathrm{k}])}{\mathtt{n}(\mathtt{UU}[\mathrm{k}])+\texttt{prob}[\mathrm{k}]} - \frac{1}{1+\texttt{prob}[\mathrm{k}]}}{\frac{\texttt{prob}[\mathrm{k}]}{r1+\texttt{prob}[\mathrm{k}]}} = \frac{\mathtt{n}(\mathtt{UU}[\mathrm{k}])-1}{\mathtt{n}(\mathtt{UU}[\mathrm{k}])+\texttt{prob}[\mathrm{k}]}\,. \tag{2.11}$$

"CFTP.c" 26b ≡

```
if (isupper(inc->flag))
  {
    cprob = (n(UU[k])-1)/(n(UU[k])+prob[k]);
    if (!udi && UNIFORM < cprob)
      udi = pick_random(UU[k]);
  };
```

◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

Now we need to alter `ldi` in accordance with the conditional probabilities expressed in the table above. If there is no `udi`, or if `udi` belongs to `L`, then we must use `udi` for `ldi`. If there is a `udi` and it belongs to `UU-L` then we must (with appropriate conditional probabilities) use either `inc->Ldeath` from `LL` or kill a random member of `L-LL`. Fortunately we can implement this conveniently by first picking at random from `L`, then replacing it by `inc->Ldeath` if one of the two following events occur:

- if the random selection belongs to `LL`,
- or if an event occurs of conditional probability

$$\frac{\frac{\mathtt{n(LL[k])+prob[k]}}{\mathtt{n(L[k])+prob[k]}} - \frac{\mathtt{n(LL[k])}}{\mathtt{n(L[k])}}}{\frac{\mathtt{n(L[k])-n(LL[k])}}{\mathtt{n(L[k])}}} \quad = \quad \frac{\mathtt{prob[k]}}{\mathtt{n(L[k])+prob[k]}}\,. \tag{2.12}$$

The only variation to note is that if `n(LL[k])` is zero then we condition on whether or not `inc->flag=='d'`. We alter the conditioning by generating the next sandwiching sub-population up from `LL[k]`, and supposing this singleton sub-population to have a death or not according as to whether or not `inc->flag` is `'d'` or `'D'`.

"CFTP.c" 27 ≡

```
               if (udi && !belong(udi,L))
                 {
                   ldi = (L[k] ? pick_random(L[k]) : NOBODY);
                   if (LL[k])
                     {
                       cprob = prob[k]/(n(L[k])+prob[k]);
                       if (belong(ldi,LL) || UNIFORM < cprob)
                         ldi = inc->Ldeath;
                     }
                   else if (L[k])
                     {
                       dom = pick_random (L[k]);
                       cprob = prob[k]/(n(L[k])-1+prob[k]);
                       if (ldi->id == dom->id || UNIFORM < cprob)
                         ldi = ( tolower(inc->flag) == 'd' ?
                                    dom : NOBODY );
                     }
                   else
                     {
                       ldi = NOBODY;
                     };
                 };


               if (udi && belong(udi,L))
                 ldi = udi;
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

Now we can repeat the previous calculation to find the correct `udi` in the event that the candidate `udi` exists and does not belong to `U`. Again we need to modify this in case `n(L[k])` is zero, to take account of the event `inc->flag` is `'d'` or `'D'`.

"CFTP.c" 28a ≡

```
        if (udi && !belong(udi,U))
          {
            udi = (U[k] ? pick_random(U[k]) : NOBODY);
            if (L[k])
              {
                cprob = prob[k]/(n(U[k])+prob[k]);
                if (belong(udi,L) || UNIFORM < cprob)
                  udi = ldi;
              }
            else if (U[k])
              {
                dom = pick_random (U[k]);
                cprob = prob[k]/(n(U[k])-1+prob[k]);
                if (udi->id == dom->id || UNIFORM < cprob)
                  udi = ( tolower(inc->flag) == 'd' ?
                            dom : NOBODY );;
              }
            else
              {
                udi = NOBODY;
              };
          };
```

◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

Finally we set the birth proposals according to the interleaved sandwiching of `L` and `U` as described above. There is a special feature to notice here, concerned with what happens if the alternate sub-population is empty. If `L[k]` is empty, then (by sandwiching) so must all previous `LL[k]` have been empty. Therefore the `inc->Ubirth` branch of the trigraph for `ubi` must always have been invoked in previous visits by the *CFTP* algorithm to this incident. Consequently `inc->Ubirth` must have remained unchanged since its production by the initial simulation by the dominating reversible $M/M/1$ queue, and so its probability of being non-`NULL` is exactly $\texttt{prob}[k]/(1+\texttt{prob}[k])$ as desired.

On the other hand suppose that `U[k]` is empty, so that the `inc->Ubirth` branch of the trigraph for `lbi` is invoked. Then again (once more by sandwiching!) all previous `LL[k]` have been empty. So the reasoning of the previous paragraph applies: the probability of `inc->Ubirth` being non-`NULL` is once again exactly $\texttt{prob}[k]/(1+\texttt{prob}[k])$ as desired.

These are just the candidate probabilities: we also have to apply the interaction as provided by `interaction_fn`. We follow this by a number of assertions designed to root out any problems with non-monotonicity.

"CFTP.c" 28b ≡

```
        ubi = (ldi ? NOBODY : inc->Ubirth);
```

```
            lbi = (udi ? NOBODY : inc->Ubirth);
            bcand = (* interaction_fn) (ubi, lbi, U, L, prob);
            ubi = bcand.ubi;
            lbi = bcand.lbi;
/* DEBUG */ assert(!inc->Lbirth || lbi);
/* DEBUG */ assert(!isupper(inc->flag) || !lbi || ubi);
/* DEBUG */ assert(!islower(inc->flag) || !lbi || ubi);
/* DEBUG */ assert(!lbi || ubi);
/* DEBUG */ assert(!ubi || inc->Ubirth);
```

◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

We interpose higher-level debugging code to trace the detailed behaviour of the populations:

"CFTP.c" 29a ≡

```
if(debug>1)
 {
   printf("LL: %c%c ",
     (inc->Lbirth ? 'b' : ' '), (inc->Ldeath ? 'd' : ' '));
   print_diagnostic (inc, LL[k]);
   printf("L:  %c%c ",
     (lbi ? 'b' : ' '), (ldi ? 'd' : ' '));
   print_diagnostic (inc, L[k]);
   printf("U:  %c%c ",
     (ubi ? 'b' : ' '), (udi ? 'd' : ' '));
   print_diagnostic (inc, U[k]);
   printf("UU: %c%c ",
     (inc->Ubirth ? 'b' : ' '), (inc->Udeath ? 'd' : ' '));
   print_diagnostic (inc, UU[k]);
 };
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

More debugging code now traces birth and death in tabular form:

"CFTP.c" 29b ≡

```
if(debug>1)
 {
  if (inc->Lbirth || inc->Ubirth || lbi || ubi ||
      inc->Ldeath || inc->Udeath || ldi || udi)
    puts(    "\tLL\tL \tU \tUU\tdom");
  if (inc->Lbirth || inc->Ubirth || lbi || ubi)
   {
    printf("b:\t");
    if(inc->Lbirth)
      printf("%d",inc->Lbirth->id); putchar('\t');
    if(lbi)
      printf("%d",lbi->id);         putchar('\t');
    if(ubi)
      printf("%d",ubi->id);         putchar('\t');
    if(inc->Ubirth)
```

```
    printf("%d",inc->Ubirth->id); putchar('\t');
  if(tolower(inc->flag)=='b')
    putchar(inc->flag); putchar('\n');
 };
if (inc->Ldeath || inc->Udeath || ldi || udi)
 {
  printf("d:\t");
  if(inc->Ldeath)
    printf("%d",inc->Ldeath->id); putchar('\t');
  if(ldi)
    printf("%d",ldi->id);         putchar('\t');
  if(udi)
    printf("%d",udi->id);         putchar('\t');
  if(inc->Udeath)
    printf("%d",inc->Udeath->id); putchar('\t');
  if(tolower(inc->flag)=='d')
    putchar(inc->flag); putchar('\n');
 };
puts("");
};
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

We check various assertions and then update the original sandwiching populations LL and UU. Notice that we have not yet altered the birth and death candidates such as inc->Ldeath, so the original sandwiching populations will follow their previous trajectories.

"CFTP.c" 30a ≡

```
assert(!(ldi && lbi));

assert(!LL[k] || L[k]);
assert(!belong(ldi,LL) || inc->Ldeath);
assert(!inc->Lbirth || lbi);

if (inc->Ldeath)
  take_from_populations (LL,inc->Ldeath);
if (inc->Lbirth)
  add_to_populations (LL,inc->Lbirth);
if (inc->Ubirth)
  add_to_populations (UU,inc->Ubirth);
if (inc->Udeath)
  take_from_populations (UU,inc->Udeath);
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

If diagfile is set then we log the transitions for future analysis.

"CFTP.c" 30b ≡

```
if (diagfile)
 {
  fprintf(diagfile,"%d ",(ldi?1:0));
  fprintf(diagfile,"%d ",(lbi?1:0));
```

```
                fprintf(diagfile,"%d ",(udi?1:0));
                fprintf(diagfile,"%d ",(ubi?1:0));

                fprintf(diagfile,"%d %d %d\n",
                               n(L[k]),n(U[k]),k);
               };
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

Now we update the birth and death candidates as given in the incident `inc`, using the constructions we have been carrying out. These are now used to update the new sandwiching populations `L` and `U`.

"CFTP.c" 31a ≡

```
              inc->Ldeath = ldi;
              inc->Lbirth = lbi;
              inc->Ubirth = ubi;
              inc->Udeath = udi;

              if (inc->Ldeath)
                take_from_populations (L,inc->Ldeath);
              if (inc->Lbirth)
                add_to_populations (L,inc->Lbirth);
              if (inc->Ubirth)
                add_to_populations (U,inc->Ubirth);
              if (inc->Udeath)
                take_from_populations (U,inc->Udeath);
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

This completes the processing of this incident, so now we move on to the next. It is inefficient to lower the case of the flag character repeatedly after the first pass, but it makes for a clearer program!

"CFTP.c" 31b ≡

```
              inc->flag = tolower(inc->flag);
              inc = inc->next;
            };
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

At the end of a time-step we print out the current lower- and upper-sandwich populations then increment the negative value of `t`:

"CFTP.c" 31c ≡

```
          if (debug)
            {
              printf("Lower bound on population:");
              print_populations (L);
              SEPARATOR2;
              printf("Upper bound on population:");
              print_populations (U);
```

```
            SEPARATOR2;
            END_VERBATIM;
          };
        t++;
      };
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

The function returns a logical flag indicating whether or not coalescence has been attained, as indicated by a separate function `equal_populations`. If coalescence has occurred then we output the final populations according to the setting of the output switches `concise` and `debug`.

Note: we need to compute the termination test *before* freeing the memory used by `lopop` and `hipop`.

"CFTP.c" 32a ≡

```
    if (debug) BEGIN_VERBATIM;
    test = equal_populations(L,U);
    if (debug) END_VERBATIM;

    if(test)
      {
        if(debug) BEGIN_VERBATIM;
        if(debug) SEPARATOR1;
        if(!concise) puts("Coalesced perfect sample:");
        if(debug) SEPARATOR2;
        if(concise)
          {
            if (postscript) {
              init_postscript ();
              print_populations_postscript (U);
             }
            else {
              print_populations_concisely (U);
             };
          }
        else
          {
            print_populations(U);
          };
        if(debug) SEPARATOR2;
        if(debug) END_VERBATIM;
      };
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

If diagnostic logging is set using `diagfile` then we add a note if we have not yet achieved coalescence.

"CFTP.c" 32b ≡

```
            if (diagfile && !test)
              fprintf(diagfile,"NO COALESCENCE YET!\n");
```
◇

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

The final step is to free the memory used by the populations LL, L, U, and UU.

"CFTP.c" 33a ≡

```
    free_populations (LL);
    free_populations (L);
    free_populations (U);
    free_populations (UU);

    return test;
  }
  ◇
```

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

We add the functions which open and close the logging file:

"CFTP.c" 33b ≡

```
void
open_logfile (char *fname)
{
 diagfile = fopen(fname,"w");
 if (!diagfile)
  {
   printf("Cannot open log file %s\n",fname);
   exit(14);
  };
}
◇
```

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

"CFTP.c" 33c ≡

```
void
close_logfile(void)
{
 if(diagfile) fclose(diagfile);
}
◇
```

File defined by scraps 22bc, 23, 24, 26ab, 27, 28ab, 29ab, 30ab, 31abc, 32ab, 33abc.

The equal_populations function needs to discover whether there are any entries in hipop which do not exist in lopop. We *assume* that lopop is contained in hipop, so that it suffices to check that each of the population types has the same cardinality in each case. This avoids the complexity of having to compare two lists not necessarily of the same order. We place this in pop.c, as it is to do with the Pop structure.

"pop.c" 34 ≡

```
int
equal_populations(Pop ** lopop, Pop ** hipop)
{
  int i = 0,j;
  Pop * p;

  if(debug) {
      puts("Starting termination test");
    };
  while(i<ntypes){
    if(debug) printf("Testing population %d: ",i);
    p = hipop[i];
    j=0;
    while(p){
      j++;
      p = p->next;
    };
    if(debug) printf(" hipop = %d, ",j);
    p = lopop[i];
    while(p){
      j--;
      p = p->next;
    };
   if(debug) printf(" lopop deficit %d\n",j);
    if(j) {
        return FALSE;
      };
    i++;
  };
  return TRUE;
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

The function `CFTP` uses the function `copy_populations`, which returns a copy of its argument, associated in a natural way to `copy_population`:

"pop.c" 35a ≡

```
Pop *
copy_population(Pop *p)
{
  Pop *q;

  if(!p) return (Pop *) NULL;

  q = (Pop *) malloc (sizeof(Pop));
  q->n = n(p);
  q->subj = p->subj;
  q->next = copy_population(p->next);

  return q;
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Here is the associate function:

"pop.c" 35b ≡

```
Pop **
copy_populations(Pop **pop)
{
  Pop **p;
  int k;

  p = (Pop **) malloc (ntypes * sizeof(Pop *));

  for (k=0; k<ntypes; k++)
    p[k] = copy_population(pop[k]);

  return p;
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

We also need the function `free_populations`, which frees the memory taken up by its argument, associated in a natural way to `free_population`:

"pop.c" 35c ≡

```
void
free_population(Pop *p)
{
  Pop *q;

  while(p)
    {
      q = p;
      p = p->next;
      free (q);
    };
```

```
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Here is the associate function:

"pop.c" 36a ≡

```
void
free_populations(Pop **pop)
{
  int k;

  for (k=0; k<ntypes; k++)
    free_population(pop[k]);

  free (pop);
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

The function `belong` returns `TRUE` if its first argument does NOT belong to the appropriate sub-population of the second argument. It is associated to the structure `Pop`, so we place it in `pop.c`.

"pop.c" 36b ≡

```
int
belong (Indiv *person, Pop **pop)
  {
    if(!person) return FALSE;
    return belong2(person, pop[person->type]);
  }
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

The recursive function `belong2` carries out the main work here:

"pop.c" 36c ≡

```
int
belong2 (Indiv *person, Pop *p)
  {
    if(!p) return FALSE;
    if((person->id)==(p->subj->id)) return TRUE;
    return belong2(person, p->next);
  }
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

2.3 Birth candidates

In this section we compute the birth candidates for the point process. It is here and only here that the specific model for the point process affects the code! The function for computing the candidates is chosen by `interaction_choice`. This also allows for choice of parameters (the `static` array `arg` holds these).

"cand.c" 37a ≡

```
BP update_births_null(Indiv *ubi, Indiv *lbi,
                      Pop **U, Pop **L, double *prob);

BP
update_births_Poisson (Indiv *ubi, Indiv *lbi,
                       Pop **U, Pop **L, double *prob);

BP
update_births_Strauss (Indiv *ubi, Indiv *lbi,
                       Pop **U, Pop **L, double *prob);

static double *interactionarg;

BP
(* interaction_choice (char *interaction, int postscript))
              (Indiv *ubi, Indiv *lbi,
               Pop **U, Pop **L, double *prob)
{
 int i;
 char *par;
◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

We begin by extracting any parameters, using `strtok`.

"cand.c" 37b ≡

```
 interactionarg = (double *) malloc(2*sizeof(float));
 for(i=0;i<=MAXARGNO;i++) interactionarg[i]=0.0;
 i=0;

 interaction = strtok(interaction,"(");
 par = strtok(NULL,",)");

 while(par && i<=MAXARGNO)
  {
   interactionarg[i] = atof(par);
   par = strtok(NULL,",)");
   i++;
  };
◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

Now we find out which of the interactions is being used, and print out corresponding information if in `postscript` mode. Note that we begin by outputting a "magic number" and font information for the benefit of `unix` systems.

"cand.c" 37c ≡

```
 if(postscript)
   {
     puts ("%!");
     puts ("/Helvetica-Oblique findfont");
```

```
    puts ("/unitft {12 mul} bind def");
    puts ("1 unitft scalefont");
    puts ("setfont");
  };

if (!strcmp(interaction,DEFAULT_INTERACTION))
  {
    if(postscript)
      {
       puts("/cr {");
       puts(" grestore 0 unitft -1 unitft rmoveto gsave");
       puts(" } def");

       puts("/distribution {");
       puts(" gsave");
       puts(" (Independent cells) show");
       puts(" cr");
       puts(" (P[0 points in a cell] = ) show");
       puts(" ((1+p)/(2p exp(p)+1) = ) show");
       puts(" prob0 show");
       puts(" cr");
       puts(" (P[x points in a cell] = ) show");
       puts(" (((x+p)p^x)/((2p exp(p)+1)x!) for x>0) show");
       puts(" grestore");
       puts(" } def");
      };
    return update_births_null;
  };
if (!strcmp(interaction,POISSON_INTERACTION))
  {
    if (postscript)
      {
        puts("/distribution {");
        puts(" (Poisson process of intensity ) show ");
        puts(" intensity show");
        puts(" } def");
      };
    return update_births_Poisson;
  };
if (!strcmp(interaction,STRAUSS_INTERACTION))
 {
   if (!(interactionarg[0]>0.0))
     {
       printf("Radius %f should be positive\n",interactionarg[0]);
       exit(30);
     };
   if (!(interactionarg[1]>0.0 && interactionarg[1]<1.0))
     {
       printf("Gamma parameter %f should lie in (0,1)\n",interactionarg[1]);
       exit(31);
     };
   if(postscript)
     {
       puts("/cr {");
```

```
          puts(" grestore 0 unit -1 unitft rmoveto gsave");
          puts(" } def");

          puts("/radius {");
          printf(" (%f)\n",interactionarg[0]);
          puts(" } def");

          puts("/gamma {");
          printf(" (%f)\n",interactionarg[1]);
          puts(" } def");

          puts("/distribution {");
          puts(" gsave");
          puts(" (Strauss process density: ) show ");
          puts(" gamma show");
          puts(" (^(number of close ordered pairs)) show");
          puts(" cr");
          puts(" (Reference distribution is ) show ");
          puts(" (a Poisson distribution of intensity ) show ");
          puts(" intensity show");
          puts(" cr");
          puts(" (Interaction radius: ) show");
          puts(" radius show");
          puts("  } def");
        };
      return update_births_Strauss;
    };
   exit(99);
   return NULL;
  }
```

◇

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

Here (for use in statistical testing of the code) is `update_births_null` which does nothing! (This does not give rise to a stationary point process.) Here is a typical invocation:

```
mh-cftp -i "null" -n 64 -m 8 -z 0.5
```

"cand.c" 39 ≡

```
BP
update_births_null (Indiv *ubi, Indiv *lbi,
                    Pop **U, Pop **L, double *p)
{
  BP bcand;

  bcand.ubi = ubi;
  bcand.lbi = lbi;

  return bcand;
}
```

◇

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

The function `update_births_Poisson` updates the birth candidates as required by the point process model using the censoring required to deliver a Poisson point process of rate `prob` per cell area $\texttt{scale}^2$. Here is a typical invocation:

```
mh-cftp -i "poisson" -n 64 -m 8 -z 0.5
```

It begins in the same way as for the `null` interaction.

"cand.c" 40 ≡

```
BP
update_births_Poisson (Indiv *ubi, Indiv *lbi,
                       Pop **U, Pop **L, double *prob)
{
  BP bcand;
  int k;
  double p, m;

  bcand.ubi = ubi;
  bcand.lbi = lbi;
```

◇

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

Then we must censor births to reduce birth rate (in a monotonic way) so that at coalescence the process evolves as an immigration-death process with Poisson equilibrium of mean `prob[k]`. Candidate birth rates are as follows (dropping reference to types!):

$$\begin{aligned} U \to U+1 \quad & \text{with probability} & \frac{\texttt{prob}}{\texttt{n(L)}+\texttt{prob}} \quad & \text{if } \texttt{n(L)} > 0 \\ & \text{otherwise} & \frac{\texttt{prob}}{1+\texttt{prob}}\,; & \\ L \to L+1 \quad & \text{with probability} & \frac{\texttt{prob}}{\texttt{n(U)}+\texttt{prob}} \quad & \text{if } \texttt{n(U)} > 0 \\ & \text{otherwise} & \frac{\texttt{prob}}{1+\texttt{prob}}\,; & \end{aligned} \tag{2.13}$$

and we must censor to achieve sandwiching *and* so that at coalescence (so $\texttt{n(U)}=\texttt{n(L)}$)

$$\begin{aligned} U \to U+1 \quad & \text{with probability} \quad \frac{\texttt{prob}}{\texttt{n(L)}+1+\texttt{prob}}\,, \\ L \to L+1 \quad & \text{with probability} \quad \frac{\texttt{prob}}{\texttt{n(U)}+1+\texttt{prob}}\,. \end{aligned}$$

To this end we do not censor at all when `L`, respectively `U`, is empty. Otherwise (for the sake of coalescence) we censor with retention probabilities

$$\begin{aligned} & \frac{\texttt{n(U)}+\texttt{prob}}{\texttt{n(U)}+1+\texttt{prob}} \quad \text{for } \texttt{U}\,, \\ & \frac{\texttt{n(L)}+\texttt{prob}}{\texttt{n(L)}+1+\texttt{prob}} \quad \text{for } \texttt{L}\,. \end{aligned} \tag{2.14}$$

These retention probabilities are calculated as the maximum (for `U`) and minimum (for `L`) appropriate for the target process over the range of sandwiching.

"cand.c" 41a ≡

```
    if (ubi)
     {
      k = ubi->type;
      m = ubi->mark;
      p = prob[k];

      if (L[k])
       {
        if (1-m <= 1/(n(U[k])+1+p))
         bcand.ubi = NOBODY;
       };

      if (U[k])
       {
        if (1-m <= 1/(n(L[k])+1+p))
         bcand.lbi = NOBODY;
       };

     };

    return bcand;
  }
  ◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

The Strauss interaction is based on the Poisson interaction, but adds a penalty term based on the number of points close to the proposed birth candidate. Here is a typical invocation:

```
mh-cftp -i "strauss(1,0.5)" -n 64 -m 8 -z 0.5
```

The first term in parentheses is the interaction radius, the second term is the parameter γ in the weighting

$$\gamma^{\text{number of close ordered pairs}}$$

Notice of course the Strauss censoring has to be done in a sandwiched way, since the interaction is repulsive!

"cand.c" 41b ≡

```
  BP
  update_births_Strauss (Indiv *ubi, Indiv *lbi,
                         Pop **U, Pop **L, double *prob)
  {
    BP bcand;
    double r, ubithresh, lbithresh;
  ◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

First compute the birth candidates as if the interaction is Poisson: if there are no candidates (and by sandwiching it is enough to check `bcand.ubi` then we can return right away.

"cand.c" 42a ≡

```
  bcand = update_births_Poisson (ubi, lbi, U, L, prob);
  if (!bcand.ubi)
    return bcand;
◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

Otherwise we must collect parameters and set integer thresholds:

"cand.c" 42b ≡

```
  r = interactionarg[0];
  ubithresh =
    (log ((bcand.ubi)->mark) / log (interactionarg[1]));
  lbithresh = ubithresh;
  assert (ubithresh > 0);
◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

We now use a subsidiary function `update_births_Strauss2` (defined in `loc.c`) to decrement the thresholds once for each relevant neighbour. The relevant birth candidate is killed if a threshold becomes negative:

"cand.c" 42c ≡

```
  return update_births_Strauss2 (bcand, U, L, ubithresh, lbithresh, r);
}
◇
```

File defined by scraps 13b, 37abc, 39, 40, 41ab, 42abc.

2.4 The schedule

This can be varied to taste, so long as we visit each cell sufficiently often. Here we vary systematically over the types `0. 1`, . . . , `ntypes-1`, and then back again through `ntypes-1`, . . . , `1. 0`.

"sched.h" 42d ≡

```
/*
 * File sched.h ⟨revision information 1c⟩
 */
int * make_schedule (int);
◇
```

"sched.c" 42e ≡

```
/*
 * File sched.c ⟨revision information 1c⟩
 */

#include <stdlib.h>

#include "mh-cftp.h"
#include "sched.h"

int *
```

```
make_schedule (int nostype)
{
  int i, listsize, *list;

  listsize = 2 * nostype;
  list = (int *) malloc (2 * nostype * sizeof (int));

  for (i = 0; i < listsize; i++)
    list[i] = (i < nostype) ? i : listsize - 1 - i;
  return list;
}
```
◇

Chapter 3

Creation of incidents

3.1 Making an incident

We emphasize the peculiarity of time-flow in the extension phases of the simulation. Since we actually construct the dominating process in reverse time, deaths appear to be births and *vice versa*! (consider the perspective of the White Queen in *Alice through the Looking Glass*!).

We first construct the function in `inc.h` which makes a null inc. This sets up a new incident to point on to its second argument, with `type` and `time` set correctly, but with pointers all pointing to `NULL` structures.

"inc.c" 44 ≡

```
Inc *
make_null_incident (int k, Inc *c, double t)
{
 Inc *n;

 n = (Inc *) malloc (sizeof(Inc));
 n->type = k;
 n->time = t;
 n->flag = ' ';
 n->next = c;

 n->Lbirth = NOBODY;
 n->Ldeath = NOBODY;

 n->Ubirth = NOBODY;
 n->Udeath = NOBODY;

 return n;
}
◇
```

File defined by scraps 7b, 44.

We now construct the function `make_incident`. In the code for `make_incident` a choice has to be made between

1. a *death* in the upper-sandwich population, which is where most of the work is done since we then have to create a new individual (remember we are running backwards through time when extending the CFTP simulation!);
2. a *birth*, which must point to a randomly chosen individual from the current population;
3. a *"non-incident"*, where nothing happens, because a possible birth is rendered ineffective since there is no available death in the future.

Within a fixed population, in an extension phase the overall birth rate is fixed and so is the overall death rate (assuming there are any individuals left around to die!). This means that individual populations behave as $M/M/1$ queues, as noted above.

"sim.c" 45a ≡

```
Sim *
make_incident (int k, double birthprob, Sim *s, double t)
{
 Inc *n;

 n = make_null_incident (k, s->inc, t);

 n->flag = (UNIFORM < birthprob)?'B':'D';
 if (n->flag == 'B' && !(s->U)[k])
   n->flag = 'N';
◇
```

File defined by scraps 6b, 45ab, 48a.

We must now set the birth or death, alter the upper-sandwich population in accordance with what has been set, and return the resulting `Sim`.

"sim.c" 45b ≡

```
 switch(n->flag)
 {
 case 'B': n->Ubirth = make_birth((s->U)[k]);
           break;
 case 'D': n->Udeath = make_death(k);
           break;
 case 'N':
           break;
 default:  exit(2);
           break;
 };

 s->inc = n;
 if (n->Ubirth) take_from_populations (s->U, n->Ubirth);
 if (n->Udeath) add_to_populations (s->U, n->Udeath);

 return s;
}
◇
```

File defined by scraps 6b, 45ab, 48a.

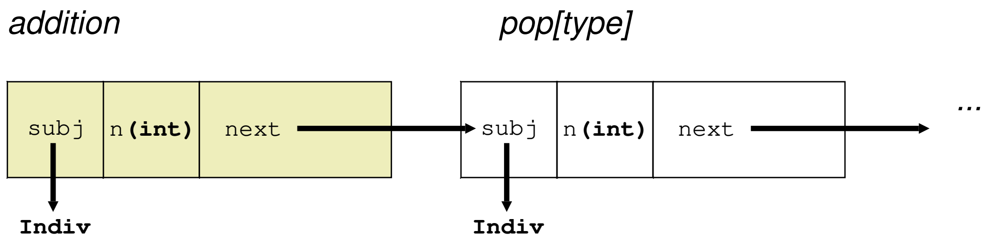

Figure 3.1: When add_to_population is applied, a replacement sub-population is created by putting the new individual at the head of the previous sub-population list.

We can visualize the change which results from calling add_to_populations as shown in Figure 3.1. The required change after calling take_from_populations is that we delete the individual in question from the list (s->U)[k].

The functions add_to_population, take_from_population, and their associates are used to make these changes. The changes concern the Pop structure and so are declared in pop.h (see §1.4) and implemented in pop.c.

We begin with add_to_population.

"pop.c" 46a ≡

```
Pop *
add_to_population(Pop * p, Indiv * s)
{
  Pop *q;

  assert(!p || s->type == p->subj->type);

  q=p;
  p = (Pop *) malloc(sizeof(Pop));

  p->n = n(q)+1;
  p->subj = s;
  p->next = q;

  return p;
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Its associate add_to_populations selects the sub-population in accordance with the type of the subject. Notice that the associate does its work by a side-effect!

"pop.c" 46b ≡

```
void
add_to_populations(Pop ** p, Indiv * s)
{
  p[s->type] = add_to_population (p[s->type], s);
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Now consider `take_from_population`. There are two cases. The easiest is when the individual concerned is at the head of the list.

"pop.c" 47a ≡

```
Pop *
take_from_population(Pop * p, Indiv * s)
{
  Pop *q, *r;

  assert(p);
  assert(s->type == p->subj->type);

  if (s->id == p->subj->id)
    {
      q=p;
      p=p->next;
      free (q);
    }
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Otherwise we have to work a little harder:

"pop.c" 47b ≡

```
  else
    {
      r = p;
      do
        {
          assert (r->next);
          q=r;
          (q->n)--;
          r=r->next;
        }
      while (s->id != r->subj->id);
      q->next = r->next;
      free (r);
    };

  return p;
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

Once again, the associate `take_from_populations` does its work by a side-effect!

"pop.c" 47c ≡

```
void
take_from_populations(Pop ** p, Indiv * s)
{
  p[s->type] = take_from_population (p[s->type], s);
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

We now construct the function `make_initial_incident`, which is a simpler version in which the upper-sandwich process always experiences a death.

"sim.c" 48a ≡

```
Sim *
make_initial_incident (int k, Sim *s, double t)
{
 return make_incident(k, 0, s, t);
}
◇
```

File defined by scraps 6b, 45ab, 48a.

3.2 Implementing death in reverse-time

The major subsidiary function `make_death` arranges for the death of an individual. The first step is to create the individual who dies (remember we are constructing the simulation backwards in time!) and to assign to it a location. To construct the location we must call on a separate procedure `location`, which takes account of the `type` of the individual.

"death.h" 48b ≡

```
/*
 * File death.h ⟨revision information 1c⟩
 */
extern Indiv *make_death (int);
◇
```

We include in the definition of `make_death` a static variable `unique_id`, which tracks the unique identity number we assign to each `Indiv`.

"death.c" 48c ≡

```
/*
 * File death.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "inc.h"
#include "loc.h"
/*#include "check.h"*/
#include "death.h"
#include "rand.h"

Indiv *
make_death(int type)
{
 static unique_id = 1;
 Indiv *subject;

 subject = (Indiv *) malloc (sizeof(Indiv));
 subject->loc  = location (type);
```

```
 subject->type = type;
 subject->mark = UNIFORM;
 subject->id   = unique_id++;

 return subject;
}
◇
```

3.3 Implementing birth

The `make_birth` function is similar to the `make_death` function, but does not involve creation of an individual. We must return a random member of the relevant sub-pop.

"birth.h" 49a ≡

```
/*
 * File birth.h ⟨revision information 1c⟩
 */
extern Indiv *make_birth (Pop *) ;
◇
```

"birth.c" 49b ≡

```
/*
 * File birth.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
/*#include "check.h"*/
#include "pop.h"
#include "birth.h"

Indiv *
make_birth(Pop* pop)
{
  return pick_random(pop);
}
◇
```

3.4 Utilities for populations

The function `make_birth` uses a utility function `pick_random` to select a random individual from a sub-population. This is made available as a function associated with the `Pop` structure (See §1.4). The function `pick_random` returns a random individual from a sub-population. Note that in a commented-out line we adjust the output if the random number generator achieves its maximum value – a problem with the `Windows 95` version of the code using `rand()`.

"pop.c" 49c ≡

```
Indiv *
pick_random (Pop *p)
{
 int counter;

 assert(n(p));

 if(!p) return NOBODY;

 counter = (int) floor(UNIFORM*(n(p)));
 /*if(counter==n) counter--;*/
 assert(counter >= 0);
 assert(counter < n(p));

 while(counter)
  {
   p = p->next;
   counter --;
  };

 return p->subj;
}
◇
```

File defined by scraps 9b, 10a, 34, 35abc, 36abc, 46ab, 47abc, 49c.

3.5 Generating locations

The `location` function returns the location of a new individual. Eventually we could modify this even to allow for multi-type point processes!

`"loc.h"` 50a ≡

```
/*
 * File loc.h ⟨revision information 1c⟩
 */

extern Pt location (int);
◇
```

File defined by scraps 50ab, 51a.

There are various functions relating to the geometry of the window:

`"loc.h"` 50b ≡

```
extern void set_xgrid (int);
extern void set_scale (double);
extern int get_xgrid (void);
extern int get_ygrid (void);
extern double get_scale (void);
extern void state_geometry(int);
◇
```

File defined by scraps 50ab, 51a.

We also export update_births_Strauss2, which is required for the Strauss interaction. It is convenient to define it here as it makes use of static variables holding the window geometry.

"loc.h" 51a ≡

```
extern BP update_births_Strauss2 (BP, Pop **, Pop **, double, double, double);
◇
```

File defined by scraps 50ab, 51a.

We define static variables xgridvar, ygridvar, scalevar to define the geometry of the rectangular window with which this program deals. Defaults are determined by XGRID, SCALE; values are altered by command-line switches *via* the functions set_xgrid, set_scale.

"loc.c" 51b ≡

```
/*
 * File loc.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "inc.h"
#include "loc.h"
#include "rand.h"
#include "display.h"

static int xgridvar = XGRID;
static int ygridvar = 1;
static double scalevar = SCALE;

◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

We declare a private function neighbours to count (recursively) the number of individuals in a given population which are within a threshold distance of a specified point:

"loc.c" 51c ≡

```
int neighbours (double, double, Pop *, double, int);
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

The function set_xgrid determines the overall geometry of the rectangular region: its argument is the length in cells of the x-axis of the rectangle. It sets the global variables xgridvar and ygridvar, having checked first for compatibility with the global variable ntypes (the number of cells).

"loc.c" 51d ≡

```
void set_xgrid(int x)
{
 div_t result;

 if (x>ntypes) {
  puts("Illegal geometry: xgrid value exceeds number of cells.");
  exit (17);
 };

 result = div(ntypes,x);
 if (result.rem != 0) {
  puts("Illegal geometry: xgrid value doesn't divide number of cells.");
  exit (18);
 };
 ygridvar = result.quot;
 xgridvar = x;
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

The function `set_scale` determines the scale of the rectangular region: its argument is the length in units of distance of a square cell side.

"loc.c" 52a ≡

```
void set_scale(double s)
{
 scalevar = s;
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

The functions `get_xgrid`, `get_ygrid`, `get_scale` return details of the cell configuration:

"loc.c" 52b ≡

```
int get_xgrid (void)
{
 return xgridvar;
}
int get_ygrid (void)
{
 return ygridvar;
}
double get_scale (void)
{
 return scalevar;
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

The function `state_geometry` prints out a summary of the geometry, after performing a check to see the dimensions are sensible.

"loc.c" 53a ≡

```
void state_geometry(int concise)
{
 if (!concise)
   printf("Region geometry is (%d x %d) cells of side-length %f\n",
   xgridvar,ygridvar,scalevar);
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

The `location` function creates a random point chosen uniformly in the square cell determined by the type given as its argument.

"loc.c" 53b ≡

```
Pt location(int type)
{
 Pt pt;
 div_t result;

 result = div(type,xgridvar);

 pt.x = scalevar*(result.rem  + UNIFORM);
 pt.y = scalevar*(result.quot + UNIFORM);
 return pt;
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

Here is the function used by the Strauss interaction:

"loc.c" 53c ≡

```
BP
update_births_Strauss2 ( BP bcand, Pop ** U, Pop ** L,
       double ubithresh, double lbithresh, double r)
{
  double x, y, r2;
  int i, j, k, i0, i1, j0, j1;
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

We begin by establishing the coordinates of the candidate point and computing the squared radius

"loc.c" 53d ≡

```
  x = ((bcand.ubi)->loc).x;
  y = ((bcand.ubi)->loc).y;

  r2 = r * r;
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

We now compute a rectangular region containing the disk to be scanned, expressed as a rectangle in cell coordinates:

"loc.c" 54a ≡

```
        i0 = (int) floor ((x - r) / scalevar);
        i0 = (i0 > 0 ? i0 : 0);
        j0 = (int) floor ((y - r) / scalevar);
        j0 = (j0 > 0 ? j0 : 0);

        i1 = (int) ceil ((x + r) / scalevar);
        i1 = (i1 < xgridvar ? i1 : xgridvar);
        j1 = (int) ceil ((y + r) / scalevar);
        j1 = (j1 < ygridvar ? j1 : ygridvar);

        if (debug) {
          printf("Cand (%5.1f,%5.1f) (r=%5.1f) ",x,y,r);
          printf("in cell %d, ",(bcand.ubi)->type);
          printf("rect (%d,%d,%d,%d), ",i0,j0,i1,j1);
          printf("wind (%d,%d,%d,%d)\n",0,0,xgridvar,ygridvar);
        };

        assert (0 <= i0 && i0 <= i1 && i1 <= xgridvar);
        assert (0 <= j0 && j0 <= j1 && j1 <= ygridvar);
      ◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

Notice the assertions concerning the ordering of the rectangle coordinates!

We also check that the point in question lies within the rectangle:

"loc.c" 54b ≡

```
        assert (i0 * scalevar <= x && x <= i1 * scalevar);
        assert (j0 * scalevar <= y && y <= j1 * scalevar);
      ◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

We now scan the whole rectangular region (the work involved in scanning just the disk seems too complicated to justify the saving in effort; we tried this in an earlier version of the program but introduced programming errors in so doing).

"loc.c" 54c ≡

```
        for(j=j0; j<j1; j++){
          k = j*xgridvar;
          for(i=i0; i<i1; i++){
            ubithresh -= neighbours (x, y, L[k+i], r2, 0);
            lbithresh -= neighbours (x, y, U[k+i], r2, 0);
            /* Note the crossover here! */
          };
        };
      ◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

Finally we use the resulting threshold computations to determine whether or not to allow the birth candidates to proceed. Note we check for monotonicity before returning the required value!

"loc.c" 54d ≡

```
  if (ubithresh <= 0)
    bcand.ubi = NOBODY;
  if (lbithresh <= 0)
    bcand.lbi = NOBODY;

  assert (bcand.ubi || !bcand.lbi);

  return bcand;
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

Here is the private function for neighbour counting: it works recursively down the population checking for the neighbourhood relation. Notice that the radius argument `r2` is the *squared* radius!

"loc.c" 55 ≡

```
int
neighbours (double x, double y, Pop * p, double r2, int n)
{
  if (!p)
    return n;
  return neighbours (x, y, p->next, r2,
                     (pow (x - (p->subj->loc).x, 2)
                      + pow (y - (p->subj->loc).y, 2)
                      < r2 ? n + 1 : n)
    );
}
◇
```

File defined by scraps 51bcd, 52ab, 53abcd, 54abcd, 55.

Chapter 4

Checking consistency

The functions in this chapter perform expensive consistency checks. They should *never* be called except *via* `assert`.

"check.h" 56a ≡

```
/*
 * File check.h ⟨revision information 1c⟩
 */
extern int check_population (Pop *);
extern int check_populations (Pop **);
extern int check_inclusion (Pop ** lopop, Pop ** hipop);
extern int check_equality (Pop ** lopop, Pop ** hipop);
◇
```

"check.c" 56b ≡

```
/*
 * File check.c ⟨revision information 1c⟩
 */

#include "mh-cftp.h"
#include "pop.h"
#include "check.h"
◇
```

File defined by scraps 56bc, 57ab, 58.

The function `check_population` checks consistency of populations. We simply chase down the sequence of pointers of the `population` argument `pop`, decrementing the counter `pop_number` as we go. If the two arguments are consistent then the counter should be zero at the end of the list, and so `check_population` will return a non-zero integer (signalling truth!).

"check.c" 56c ≡

```
int
check_population (Pop *pop)
{
 int pop_number;

 if(!pop) return TRUE;
```

```
 pop_number = n(pop);
 while(pop)
 {
  pop = pop->next;
  pop_number--;
 };

 return !pop_number;
}
◇
```

File defined by scraps 56bc, 57ab, 58.

We also add the obvious associated function `check_populations` which applies the above to each sub-population of a `Pop **` object!

`"check.c"` 57a ≡

```
int
check_populations (Pop **pop)
{
  int k, check = TRUE;

  k = ntypes;

  do
    {
      k--;
      check = check && check_population (pop[k]);
    }
    while (k>0);

  return check;
}
◇
```

File defined by scraps 56bc, 57ab, 58.

The sandwiching construction depends on the `Pop` arrays `lopop`, `hipop` one being included in the other. The function `check_inclusion` performs this check, assuming that the constituent lists of `lopop` are sub-lists of the corresponding constituent lists of `hipop`. The method is to walk down each `Pop` list in `lopop`, and for each individual in turn to move down the corresponding list in `hipop` until a match is made or the end of the `hipop` list is reached. If the end of the `hipop` list is reached then there is no match (assuming the sub-list property mentioned at the head of this paragraph!), and the value `FALSE` is returned. If successful matches are made in all cases then the value `TRUE` is returned.

`"check.c"` 57b ≡

```
int
check_inclusion (Pop ** lopop, Pop ** hipop)
{
  int i;
  Pop *p, *q;
```

```
  for (i=0; i<ntypes; i++){
    p = lopop[i];
    q = hipop[i];
    while(p){
      while(q && q->subj->id != p->subj->id){
        q = q->next;
      };
      if(!q)
        return FALSE; /* Match or sublist property fails! */
      p = p->next;
    };
  }

  return TRUE;
}
◇
```

File defined by scraps 56bc, 57ab, 58.

We also add the obvious associated function `check_equality`:

`"check.c"` 58 ≡

```
int
check_equality (Pop ** lopop, Pop ** hipop)
{
  return
    (
     check_inclusion(lopop,hipop)
     &&
     check_inclusion(hipop,lopop)
    );
}
◇
```

File defined by scraps 56bc, 57ab, 58.

Chapter 5

Randomness

The random number generator needs careful attention in this program. The simplest possibility, using the (unsatisfactory) default `rand()` is given in Table 5.1. The current version `ran3` is due to Knuth.

In the header we define macros to invoke the random-number generator (done this way to enable us to switch random number generators easily!). Note the `INITIALIZE_RANDOM` argument should be a positive integer, so that the initializing value `idum` is negative!

We also include functions to return a Geometrically distributed random variable (the parameter is the success probability when the random variable counts the number of successes before a failure), an exponentially distributed random variable (the parameter is the rate) and a Poisson distributed random variable (the parameter is the mean).

"rand.h" 59 ≡

```
/*
 * File rand.h ⟨revision information 1c⟩
 */
#define INITIALIZE_RANDOM(seed) idum=(- (long) seed)
#define UNIFORM ran3(&idum)

#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)

extern double ran3(long *);
extern long idum;

extern int Geometric(double);
extern double exponential(double);
extern int Poisson(double);
◇
```

5.1 Uniform variate

Note an issue here: in `Windows 95` the random number generator `rand()` actually *achieves* its (rather low) maximum value `RAND_MAX` = 32767 with embarrassing frequency. This can give rise to programming problems (see Section 3.4 for a case where we had to code round this! however, with the generator actually used now, this is no longer a problem) and also is a measure of the poor quality of the generator. The actual code (see Table 5.1) included some debugging lines to trace these problems. However in the following we use `ran3`, due to [7]. This particular code was taken from [10] by my research assistant Yuzhi Cai.

"rand.c" 60 ≡

```
/*
 * File rand.c ⟨revision information 1c⟩
 */

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

#include "rand.h"

long idum;

double ran3(long *idum)
{
    static int inext, inextp;
    static long ma[56];
    static int iff=0;
    long mj, mk;
    int i, ii, k;

    if(*idum<0 || iff ==0)
    {
        iff = 1;
        mj=MSEED-(*idum<0 ? -*idum: *idum);
        mj %=MBIG;
        ma[55] = mj;
        mk = 1;
        for(i=1; i<=54; i++)
        {
            ii=(21*i) % 55;
            ma[ii] = mk;
            mk=mj-mk;
            if(mk < MZ) mk += MBIG;
            mj=ma[ii];
        }

        for(k=1; k<=4; k++)
            for(i=1; i<=55; i++)
            {
                ma[i] -= ma[1+(i+30)%55];
                if(ma[i]<MZ) ma[i] += MBIG;
            }
        inext=0;
```

```
        inextp =31;
        *idum=1;
    }

    if(++inext == 56) inext=1;
    if(++inextp == 56) inextp =1;
    mj=ma[inext]-ma[inextp];
    if(mj<MZ) mj += MBIG;
    ma[inext]=mj;
    return mj*FAC;
}
◇
```

File defined by scraps 60, 61, 62ab.

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

define INITIALIZE_RANDOM(seed) srand(seed)
#define UNIFORM wskrandom()

extern double wskrandom();

double
wskrandom()
{
  static int i=0;
  int r;
  i++;
  r = rand();
  if(r==RAND_MAX) printf("Call [%d] of rand is max at %d\n",i,r);
  return (((double) r)/RAND_MAX);
}
```

Table 5.1: This function was used to provide access to `rand()` with added debugging features.

5.2 Geometric variate

"rand.c" 61 ≡

```
int
Geometric(double prob)
{
 int n = 0;
 assert(prob>0 && prob<1);
 while(UNIFORM<prob) n++;
 return n;
}
◇
```

File defined by scraps 60, 61, 62ab.

5.3 Exponential variate

The implementation of the exponential random variable is routine (we do not even include a check on whether `UNIFORM` is zero or one!). However we do `assert` that the `rate` is positive.

"rand.c" 62a ≡

```
double
exponential(double rate)
{
 assert(rate>0);
 return -log(UNIFORM)*rate;
}
◇
```

File defined by scraps 60, 61, 62ab.

5.4 Poisson variate

The implementation of the Poisson random variable is similarly unsophisticated. Notice however that it works when the mean parameter `mu` is zero!

"rand.c" 62b ≡

```
int
Poisson(double mu)
{
 int n=0;
 assert(mu>=0);
 while((mu-=exponential(1))>0) n++;
 return n;
}
◇
```

File defined by scraps 60, 61, 62ab.

Chapter 6

Display

6.1 Functions to display output

The first display function `print_incident` prints out a trace of each `Inc` in turn following from a specified `Inc`. The second, `print_populations`, prints a snap-shot of the populations at the time of invocation of the function. Minor variants are as follows:

- `print_populations_concisely` just prints out the numbers:
- `print_populations_postscript` *etc* prints postscript code.

The third, `print_history`, uses both `Inc` and `Pop` information to print a more informative display.

"display.h" 63 ≡

```
/*
 * File display.h ⟨revision information 1c⟩
 */
extern void print_incident (Inc *);
extern void print_populations (Pop **);
extern void print_populations_concisely (Pop **);
extern void print_history (Inc *, Pop **);
extern void print_populations_postscript (Pop **);
extern void init_postscript (void);
extern void close_postscript (int, char **, double);
extern void print_diagnostic (Inc *, Pop *);

extern int debug;

◇
```

The function `print_incident` calls a subsidiary `print_incident2` which works recursively through the list of incidents specified by its argument. The function `print_populations` runs through an array of sub-populations, subjecting each one to `print_population2`. All output (except for a single line produced for each sub-population), is carried out in `print_incident2`, `print_population2`, and a subsidiary function `print_individual`.

"display.c" 64a ≡

```
/*
 * File display.c ⟨revision information 1c⟩
 */

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

#include "mh-cftp.h"
#include "inc.h"
#include "display.h"
#include "loc.h"

/*
   Postscript files and units:
   notice that PSUNITS determines the
   setting of units in preamble.psc below.
 */
#define PSINIT "preamble.psc"
#define PSUNITS 72
#define PSX 1
#define PSY 1

void print_incident2 (Inc *);
void print_population2 (Pop *);
void print_history2 (Inc *, char *);
void print_individual (Indiv *);
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

"display.c" 64b ≡

```

void
print_incident (Inc * inc)
{
  SEPARATOR2;
  puts("Incident stream (print_incident)");
  print_incident2 (inc);
  SEPARATOR1;
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

"display.c" 65a ≡

```
void
print_populations (Pop ** p)
{
  int k;
  for (k = 0; k < ntypes; k++)
    {
      print_population2 (p[k]);
    };
  putchar ('\n');
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

"display.c" 65b ≡

```
void
print_populations_concisely (Pop ** p)
{
  int k;
  for (k = 0; k < ntypes; k++)
    {
      printf (" %d\n", n(p[k]));
    };
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

Here is the code to print out a `postscript` description of the point configuration.

"display.c" 65c ≡

```
void
print_populations_postscript (Pop ** p)
{
  int k;
  Pop * q;

  for (k = 0; k < ntypes; k++)
    {
      q = p[k];
      while (q) {
          printf("%f unit %f unit point\n",
            (q->subj->loc).x, (q->subj->loc).y);
          q = q->next;
        };
    };
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

This needs to be combined with a preamble. Notice that the setting of `units` is synchronized with the setting of `PSUNITS` within `init_postscript` below..

"preamble.psc" 65d ≡

```
% Sub-unit for tickmarks and crosses
/unitsq {unit 10 div} bind def

% Point, marked by cross
% Syntax:   x  y  point
/point {
    gsave
    2 setlinewidth
    0.5 setgray
    newpath
                        moveto
    -1 unitsq  0 unitsq rmoveto
     2 unitsq  0 unitsq rlineto
    -1 unitsq -1 unitsq rmoveto
     0 unitsq  2 unitsq rlineto
     0 unitsq -1 unitsq rmoveto
    closepath
    stroke
    grestore
    } bind def

% Box
% Syntax: dx dy x y box
/box {
    newpath
                           moveto
         dup        0 exch rlineto
    exch dup        0      rlineto exch
         -1 mul     0 exch rlineto
         -1 mul     0      rlineto
    closepath
    stroke
    } bind def

% Cell
% Syntax: dx dy x y cell
/cell {
    gsave
    0.5 setlinewidth
    0.85 setgray
    box
    grestore
    } bind def

% Frame
% Syntax: dx dy x y frame
/frame {
    3 index 2 unitsq add
    3 index 2 unitsq add
    3 index 1 unitsq sub
    3 index 1 unitsq sub
    gsave
```

```
    5 setlinewidth
    0.90 setgray
    box
    grestore
    gsave
    1 setlinewidth
    0.75 setgray
    box
    grestore
    } bind def
◇
```

File defined by scraps 65d, 68.

which is included into the `postscript` output by `init_postscript` (together with a bounding box, a shift of origin, a frame, and an indication of the layout of the cells!).

"display.c" 67 ≡

```
void init_postscript (void)
{
 int ch, i, j, x, y,
     x0, y0, x1, y1,
     xorigin, yorigin,
     xborder, yborder;
 double s;
 FILE *psinit;

 x = get_xgrid();
 y = get_ygrid();
 s = get_scale();

 xorigin = (int) (1.0*PSX*PSUNITS);
 yorigin = (int) (1.5*PSX*PSUNITS);
 xborder = (int) (0.5*PSX*PSUNITS);
 yborder = (int) (1.0*PSY*PSUNITS);
 x0 = xorigin - xborder;
 y0 = yorigin - 2*yborder;
 x1 = xorigin + (int) (x*s*PSUNITS) + xborder;
 y1 = yorigin + (int) (y*s*PSUNITS) + yborder;
 printf("%%%%Bounding Box: %d %d %d %d\n",x0,y0,x1,y1);

 puts("%% Basic unit of linear measurement");
 printf("/unit {%d mul} bind def\n",PSUNITS);

 psinit = fopen(PSINIT,"r");
 if(psinit == NULL){
   printf("mh-cftp cannot find %s\n",PSINIT);
   exit(200);
 };

 while( (ch=getc(psinit)) != EOF) putchar(ch);

 fclose(psinit);
```

```
  printf("%d %d %d %d box\n",x1-x0,y1-y0,x0,y0);

  printf("%d %d translate\n",xorigin,yorigin);
  printf("%f unit %f unit 0 unit 0 unit frame\n", x*s, y*s);

  for (i = 0; i<x; i++)
  for (j = 0; j<y; j++)
    {
      printf("%f unit %f unit %f unit %f unit cell\n",
        s, s, i*s, j*s);
    };

}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

Finally, `close_postscript` needs a postamble. We define this here as an *unbound* `postscript` procedure in the preamble:

`"preamble.psc"` 68 ≡

```
/postamble {
    0 unit -2 unitsq moveto
    0 unit -2 unitft rmoveto
    (Cell geometry: ) show
    xgrid show
    ( by ) show
    ygrid show
    ( cells;) show
    ( cell size: ) show
    scale show
    ( by ) show
    scale show
    ( units) show

    0 unit -2 unitsq moveto
    0 unit -4 unitft rmoveto
    (Parameter p: ) show
    defaultp show

    0 unit -2 unitsq moveto
    0 unit -5 unitft rmoveto
    distribution

    showpage
} def
◇
```

File defined by scraps 65d, 68.

(in `unix` or `Windows 95` we can copy across a file, but in `RISC OS` this produces sporadic `malloc` errors . . .)

"display.c" 69 ≡

```
void close_postscript (int wordnumber, char **word, double defaultp)
{
 int i;

 puts ("0 unit -2 unitsq moveto");
 puts ("0 unit -1 unitft rmoveto");
 printf ("(Output produced by: ");
 i = wordnumber;
 while (i)
   {
     printf (" %s", word[wordnumber - i]);
     i--;
   };
 puts (") show");

 printf("/xgrid {(%d)} bind def\n",get_xgrid());
 printf("/ygrid {(%d)} bind def\n",get_ygrid());
 printf("/scale {(%3.2f)} bind def\n",get_scale());
 printf("/defaultp {(%3.2f)} bind def\n",defaultp);
 printf("/intensity {(%f)} bind def\n",
   defaultp/pow(get_scale(),2));
 printf("/prob0 {(%f)} bind def\n",
   (1+defaultp)/(2*defaultp*exp(defaultp)+1));

 puts("postamble");
/*
 int ch, i;
 FILE *psclose;

 psclose = fopen(PSCLOSE,"r");
 if(psclose == NULL){
   printf("mh-cftp cannot find %s\n",PSCLOSE);
   exit(201);
 };

 while( (ch=getc(psclose)) != EOF) putchar(ch);

 fclose(psclose);
*/
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

to deliver a full `postscript` file which may be previewed (for example with `ghostscript`) or printed. Here is an example of a typical run for `unix`:

```
mh-cftp -n 40 -m 8 -z 0.5 -p 0.5 -P  | ghostview -
```

The function `print_incident2` calls `print_individual` to print out information specific to the individual, then prints out a semi-graphical summary of the incident being reported. Note that null incidents are ignored!

"display.c" 70a ≡

```
void
print_incident2 (Inc * inc)
{
  int i;
  Indiv *subj;

  if (inc)
    {
      subj = (inc->Udeath ? inc->Udeath : inc->Ubirth);
      if (subj)
        {
          print_individual (subj);
          printf (" [%4.1f ", inc->time);
          printf ("%c]", inc->flag);
          for (i = 0; i < subj->id; i++)
            putchar (' ');
          putchar ((inc->Udeath) ? 'v' : '^');
          print_incident2 (inc->next);
        };
    }
  else
    {
      puts ("\nEND");
    };
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

The function `print_population2` recurses through the list which is its first argument, calling `print_individual` to print out information specific to each individual.

"display.c" 70b ≡

```
void
print_population2 (Pop * p)
{
  if (p)
    {
      print_individual (p->subj);
      print_population2 (p->next);
    };
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

The function `print_individual` prints out a basic summary of the information about one individual. This function is called by both `print_incident2` and `print_population2`.

"display.c" 71a ≡

```
void
print_individual (Indiv * one)
{
  Pt pt;
  printf ("\n%d: %4d ", one->type, one->id);
  pt = one->loc;
  printf (" (%3.2f,%3.2f)", pt.x, pt.y);
  printf ("<%3.2f>",one->mark);
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

The function `print_history` is based on `print_incident`, but uses `Pop` information to trace lifelines. We begin by setting up a string `s` which will be printed out at each incident, tracking the lifelines of individuals. The length `n` of the string is determined by the maximum `id` code, computed by running through incidents to find the earliest death (because the `id` values are created in order according to order of death running back in time!). If there are no deaths then there can be no incidents, and the program terminates with a message.

"display.c" 71b ≡

```
void
print_history (Inc * inc, Pop ** pop)
{
  int n, i;
  char *s;
  Pop *p;
  Inc *cur;

  if (!debug)
    return;
  cur = inc;
  while (cur && !cur->Udeath)
    cur = cur->next;
  if (!cur)
    {
      printf ("print_history applied to ");
      puts ("incident sequence with no deaths");
      exit (20);
    };
  n = cur->Udeath->id;

  s = (char *) malloc ((n + 2) * sizeof (char));
  memset (s, ' ', n + 1);
  s[n + 1] = '\0';
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

We now place the character `'|'` at locations where individuals are alive at the start and print this initial string. We then invoke a recursive function `print_history2` (a variant of `print_incident1`) which tracks the lifelines of the various individuals in the simulation.

"display.c" 72a ≡

```
    for (i = 0; i < ntypes; i++)
      {
        p = pop[i];
        while (p)
          {
            s[p->subj->id] = '|';
            p = p->next;
          };
      };
    BEGIN_VERBATIM;
    SEPARATOR2;
    puts ("Current simulation history (print_history)");
    print_history2 (inc, s);
    free (s);
    SEPARATOR1;
    END_VERBATIM;
  }
  ◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

The subsidiary function `print_history2` prints out details of the incident (unless it is null) and then calls itself recursively.

"display.c" 72b ≡

```
  void
  print_history2 (Inc * inc, char *s)
  {
    Indiv *subj;
    char flag;

    subj = NOBODY;
    flag = 'n';

    if (inc)
      {
        if (inc->Udeath)
          {
            subj = inc->Udeath;
            flag = 'd';
          }
        else if (inc->Ubirth)
          {
            subj = inc->Ubirth;
            flag = 'b';
          };
        if (subj)
          {
            print_individual (subj);
            printf (" [%4.1f ", inc->time);
            printf ("%c]", flag);
```

```
            s[subj->id] =
              ( (flag) == 'd' ? 'v' : '^');
            printf (s);
            s[subj->id] =
              ( (flag) == 'd' ? ' ' : '|');
          };
        print_history2 (inc->next, s);
      }
    else
      {
        puts ("\nEND");
      };
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

Finally here is a function to produce some high-level diagnostic output.

"display.c" 73 ≡

```
void
print_diagnostic (Inc * inc, Pop * p)
{
  int r;
  if (p)
    {
      printf("(%4.1f)[%d:%d] {",
        inc->time, inc->type,n(p));
      r=n(p);
      while(p)
        {
          printf("%2d", p->subj->id);
          r--;
          p=p->next;
          if(p) putchar(',');
        };
      if(r) printf(",X");
      printf("}\n");
    }
  else
    {
      printf("(%4.1f)[%d:0] {  }\n",
        inc->time, inc->type);
    };
}
◇
```

File defined by scraps 64ab, 65abc, 67, 69, 70ab, 71ab, 72ab, 73.

6.2 Sample output

Here is output from a test-run of the resulting program using the `null` interaction:

```
$Source: /home/fisher/wsk/ms/Moeller/CPrograms/MH.w,v $ $Revision: 3.28 $
       $Date: 1999/03/10 12:55:50 $
   seed = 1
   t1 = -2
   ntypes = 2
   default probability = 0.500000
   interaction is null
Region geometry is (1 x 2) cells of side-length 1.000000
Debugging output is attached at level 1:
===================

===================
Simulating over [-2,0]

CFTP simulation using null interaction
Initial populations at time -2.000000
Lower bound on population:
-------------------
Upper bound on population:
1:    2  (0.48,1.53)<0.18>
-------------------

-------------------
Current simulation history (print_history)

1:    3  (0.23,1.67)<0.53> [-2.0 b]  |^
1:    3  (0.23,1.67)<0.53> [-2.0 d]  |v
1:    2  (0.48,1.53)<0.18> [-1.0 d]  v
0:    1  (0.87,0.53)<0.63> [-1.0 b] ^
0:    1  (0.87,0.53)<0.63> [ 0.0 d] v
END
===================

Evolution at time -2.000000
Lower bound on population:
-------------------
Upper bound on population:
1:    2  (0.48,1.53)<0.18>
-------------------

Evolution at time -1.000000
Lower bound on population:
0:    1  (0.87,0.53)<0.63>
-------------------
Upper bound on population:
0:    1  (0.87,0.53)<0.63>
-------------------

Starting termination test
Testing population 0:  hipop = 1,  lopop deficit 0
Testing population 1:  hipop = 0,  lopop deficit 0
```

```
===================
Coalesced perfect sample:
-------------------

0:    1  (0.87,0.53)<0.63>
-------------------
```

Chapter 7

Testing the output

To assure ourselves that the program is behaving as it should, we run various statistical tests on the output.

7.1 Testing when the interaction is `null`

When the interaction is `null` we expect output to be a sequence of independent random variables with the distribution described in Chapter 2.2, Eq. (2.10), p. 26.

In fact the result from the invocation

```
mh-cftp -x -i null -p 0.5 -t 11 -n 800 > mh-N.tst
```

will be a file headed by a single line of identifying text (`unix` version given here)

```
%Output from mh-cftp -x -i null -p 0.5 -t 11 -n 800
```

followed by $n = 800$ numbers arranged one per line. If the program is behaving correctly then these should be distributed according to Eq. (2.10) with parameter $p = 0.5$.

We test the output at version `3.6` in `S-plus` as follows. First we set `p` to match the invocation of `mh-cftp`:

"S-plus.txt" 76 ≡

```
# mh-cftp -x -i null -p 0.5 -t 11 -n 800 > mh-N.tst
  p <- 0.5      # use -p value set in command-line
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

We then input the dataset, compute the number of cases, and calculate theoretical and empirical means:

"S-plus.txt" 77a ≡

```
pop <- scan("mh-N.tst", skip=1)
n <- length(pop)

p*(2*p+1)/(2*p+exp(-p))
mean(pop)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

These means are 0.6224593 and 0.62625 respectively, which is encouraging.

We now consider how to perform a χ^2-test. We first tabulate the data, noting that `tabulate` counts from 1 upwards!

"S-plus.txt" 77b ≡

```
ef <- tabulate(1+pop,1+max(pop))
print(ef)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

The tabled values are

```
 435 256  87  18   3   1
```

Now compute the theoretical frequencies with respect to these cells, using the empirical frequencies to check the required number of cells:

"S-plus.txt" 77c ≡

```
p1 <- (1+p)*p/(2*p*exp(p) + 1)

x <- c(1:max(pop))
xf <- gamma(x+1)
pp <- c(p1/p,p1*((x+p)/(1+p))*p^(x-1)/xf)

pp[length(pp)] <- 1-sum(pp) + pp[length(pp)]

tf <- n*pp
print(tf)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

```
 453.0488026 226.5244013  94.3851672  22.0232057   3.5394438
   0.4789795
```

We have to abbreviate `ef` and `tf`, since the last two theoretical frequencies are too small, summing to less than 5.

"S-plus.txt" 78a ≡

```
    ef <- c(ef[c(1:3)],sum(ef[c(4:6)]))
    tf <- c(tf[c(1:3)],sum(tf[c(4:6)]))
    df <- length(ef)-1
  ◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

Finally we compute the χ^2-statistic and its p-level:

"S-plus.txt" 78b ≡

```
    x2 <- sum((ef-tf)^2/tf)
    pchisq(x2,df)
   ◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

For our example `S-plus` computes the statistic as 5.759543, and χ^2 at 3 degrees of freedom has probability 0.8760866 of being less than this, which is acceptable.

Note that the range of validity of the approximation used in the χ^2 test requires expected counts greater than 5 (unless there are more than about ten categories, in which case we can allow about 20% of the expected counts to be less than 5 so long as non are less than 1: see for example [2, §7.5]), so it is hard to get good information about the tails! However we are using the test here mainly as an indicative tool. Later on we will use generalized linear models to obtain a more sophisticated view of what the simulation is actually doing.

7.2 Testing when the interaction is `poisson`

When the interaction is `poisson` we expect output to be a sequence of independent random variables with Poisson distribution of mean set by the `-p` switch. Again we use a χ^2 test, as follows.

"S-plus.txt" 78c ≡

```
  # mh-cftp -x -i poisson -p 0.5 -t 11 -n 800 > mh-P.tst
    p <- 0.5      # use -p value set in command-line
   ◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

We input the new dataset, compute the number of cases, calculate theoretical and empirical means, and tabulate the data:

"S-plus.txt" 78d ≡

```
    pop <- scan("mh-P.tst", skip=1)
    n <- length(pop)

    p
    mean(pop)
  ◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

The means are now 0.5 and 0.49125, which again is encouraging.

"S-plus.txt" 79a ≡

```
ef <- tabulate(1+pop,1+max(pop))
print(ef)
◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

The tabled values are

```
 480 258  53   7   2
```

We now compute the theoretical frequencies with respect to these cells:

"S-plus.txt" 79b ≡

```
x <- c(0:max(pop))
xf <- gamma(x+1)
pp <- (p^x/xf)*exp(-p)

pp[length(pp)] <- 1-sum(pp) + pp[length(pp)]

tf <- n*pp
print(tf)
◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

```
 485.224528 242.612264  60.653066  10.108844   1.401298
```

Again we need to combine classes to ensure expected frequencies are sufficiently large.

"S-plus.txt" 79c ≡

```
ef <- c(ef[c(1:3)],sum(ef[c(4:5)]))
tf <- c(tf[c(1:3)],sum(tf[c(4:5)]))
df <- length(ef)-1
◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

Finally we compute the χ^2-statistic and its p-level:

"S-plus.txt" 79d ≡

```
x2 <- sum((ef-tf)^2/tf)
pchisq(x2,df)
◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

Here `S-plus` computes the statistic as 2.545285, and χ^2 at 3 degrees of freedom has probability 0.5328373 of being less than this, which is acceptable.

7.3 Tracing the transitions

For a closer examination of what is going on, we can use the `-l` switch to record transitions. This is actually what is carried out in `make test` as above: the extra switch setting for the Poisson interaction case is `-l MH-PT.log`, which logs the transitions in the file `MH-PT.log`. The format is as follows, recording

- whether the transition is `lbi` (lower birth);
- whether it is `ldi` (lower death);
- whether it is `ubi` (upper birth);
- whether it is `udi` (upper death);
- the number in the lower sandwich;
- the number in the upper sandwich;
- and finally the type of the individual concerned.

```
ldi lbi udi ubi nL nU Type
0 0 0 0 0 0 0
0 0 0 1 0 1 1
0 0 0 0 0 0 2
0 0 0 0 0 0 3
0 0 1 0 0 3 4
0 0 1 0 0 3 5
0 0 0 0 0 0 6
0 1 0 1 0 0 7
0 0 0 0 0 0 8
...
```

We now analyze this in `S-plus`. By searching for the phrase "NO COALESCENCE YET" we adjust the `-t` switch, increasing it till the phrase does not occur. This is because otherwise we would overlay several sandwiched transitions, which would reduce the amount of independence present. (Note that in any case the transitions exhibit dependence through their relationship *via* the generating Markov chain. In fact we are using a pseudo-likelihood rather than the true likelihood.)

"S-plus.txt" 80a ≡

```
# mh-cftp -x -i poisson -p 0.5 -t 11 -n 800 -l MH-PT.log > mh-P.tst
  Transitions <- read.table("MH-PT.log", header=T)
  attach(Transitions)
  names(Transitions)
# [1] "ldi"  "lbi"  "udi"  "ubi"  "nL"   "nU"   "Type"
◇
```

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

We can now use `table` to extract useful information:

"S-plus.txt" 80b ≡

```
table(nL,nU,lbi)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

yields the table

```
, , 0
     0    1   2   3   4  5  6 7 8 9 10 11 12 13 14
0 6782  546 470 277 112 60 23 4 3 1  1  2  3  1  1
1    0 3561  78  53  11  8  3 0 0 0  0  0  0  0  0
2    0    0 888  15   7  1  0 0 0 0  0  0  0  0  0
3    0    0   0 132   1  1  0 0 0 0  0  0  0  0  0
4    0    0   0   0  15  0  0 0 0 0  0  0  0  0  0

, , 1
     0   1   2  3 4 5 6 7 8 9 10 11 12 13 14
0 3347  64  43 10 5 3 0 0 0 0  0  0  0  0  0
1    0 895  11  6 1 0 0 0 0 0  0  0  0  0  0
2    0   0 136  1 1 0 0 0 0 0  0  0  0  0  0
3    0   0   0 17 0 0 0 0 0 0  0  0  0  0  0
4    0   0   0  0 0 0 0 0 0 0  0  0  0  0  0
```

where rows are labelled by `nL` (number in `L[k]`) and columns by `nU`. We can now calculate the frequency of birth using the `S-plus` function `tapply`:

"S-plus.txt" 81a ≡

```
 pevent <- function(x) sum(x)/length(x)
 tapply(lbi, nU, pevent)
#         0         1         2         3          4          5
# 0.3304374 0.1893012 0.1168512 0.0665362 0.04575163 0.04109589
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

(omitting levels giving zero frequency). In fact *under coalescence* we expect the probability of birth, under the `poisson` interaction, to be given by

$$\frac{\texttt{prob[k]}}{\texttt{n(U[k])}+1+\texttt{prob[k]}}\,. \tag{7.1}$$

Here is the calculation in `S-plus`:

"S-plus.txt" 81b ≡

```
 p <- 0.5
 u <- c(0:5)
 p/(u+1+p)
# 0.33333333 0.20000000 0.14285714 0.11111111 0.09090909 0.07692308
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

However in general, as can be determined by reference to the transition probability equations (2.13,2.14), the actual (averaged) probabilities should be slightly less for positive values of `u`, which accords with the comparison.

We can carry out a similar calculation for `ubi`:

"S-plus.txt" 82a ≡

```
 tapply(ubi, nL, pevent)
#         0         1         2         3 4
# 0.3325395 0.2014264 0.1315539 0.1125828 0
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

and for `udi`:

"S-plus.txt" 82b ≡

```
 tapply(udi,nU,pevent)
# 0         1         2         3         4        5         6 7 8 9 10
# 0 0.6810107 0.8093481 0.8571429 0.9084967 0.890411 0.9615385 1 1 1  1
# 11 12 13 14
#  1  1  1  1
 u <- c(0:14)
 u/(u+p)
# 0.0000000 0.6666667 0.8000000 0.8571429 0.8888889 0.9090909
# 0.9230769 0.9333333 0.9411765 0.9473684 0.9523810 0.9565217
# 0.9600000 0.9629630 0.9655172
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

and finally `ldi`:

"S-plus.txt" 82c ≡

```
 tapply(ldi,nL,pevent)
# 0         1         2         3 4
# 0 0.6812189 0.8112488 0.8543046 1
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

7.3.1 Analysing Upper Births in S-plus

Bearing in mind the low frequency of the cases of high `nU` and `nL`, these empirical frequencies and conditional probabilities seem plausibly in agreement. We fit a `glm` to the `ubi` response using the theoretical probabilities, and compare it to fits adding

- a factor identifying when `nU` is zero (the most likely deviation!),
- a factor for the different levels of `nL`,
- and finally a factor for the different levels of `nU`.

"S-plus.txt" 82d ≡

```
# Fitting ubi
 bpU <- (nL==0)*p/(1+p) + (nL!=0)*(p/(nL+p))*((nU+p)/(nU+1+p))
 logit.bpU <- log(bpU/(1-bpU))

 nLzero <- factor(nL==0)
 nUzero <- factor(nU==0)
```

```
nLfactor <- factor(nL)
nUfactor <- factor(nU)

ubifit <- glm(ubi ~ -1+logit.bpU, family=binomial)
summary(ubifit)

ubifit.ZL <- update(ubifit, ~ . + nUzero + nLfactor)
ubifit.ZLU <- update(ubifit.L, ~ . + nLzero + nUfactor)
anova(ubifit.ZLU)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

The summary for (what we hope is) the correct model `ubifit` is

```
Call: glm(formula = ubi ~ -1 + logit.bpU, family = binomial)
Deviance Residuals:
      Min       1Q     Median       3Q      Max
 -0.89792 -0.89792 -0.6638535 1.485462 2.105217

Coefficients:
            Value Std. Error  t value
logit.bpU 1.01013 0.01785739 56.56647

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 24398.78 on 17600 degrees of freedom

Residual Deviance: 20532.7 on 17599 degrees of freedom

Number of Fisher Scoring Iterations: 3
```

which is highly reassuring! Particularly notice the coefficient of `logit.bpU` can plausibly be taken to be 1. Note this is binary data, so the assessment of significance of the residual deviance is not clear.

The analysis of deviance reassures us that the theoretical probabilities give a satisfactory explanation of the transition probabilities:

```
Analysis of Deviance Table

Binomial model

Response: ubi

Terms added sequentially (first to last)
          Df Deviance Resid. Df Resid. Dev
     NULL                 17600   24398.78
logit.bpL  1 3867.508     17599   20531.27
   nUzero  2    2.176     17597   20529.10
 nLfactor  3    2.747     17594   20526.35
   nLzero  0    0.000     17594   20526.35
 nUfactor 13   11.666     17581   20514.68
```

Thus *none* of the other plausible contributions to the fit are significant, which is just as it should be.

7.3.2 Analysing Lower Births in S-plus

The calculations for `lbi` are similar, using

"S-plus.txt" 84a ≡

```
# Fitting lbi
 bpL <- (nU==0)*p/(1+p) + (nU!=0)*(p/(nU+p))*((nL+p)/(nL+1+p))
 logit.bpL <- log(bpL/(1-bpL))

 lbifit <- glm(lbi ~ -1+logit.bpL, family=binomial)
 summary(lbifit)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

```
Call: glm(formula = lbi ~ -1 + logit.bpL, family = binomial)
Deviance Residuals:
        Min         1Q     Median       3Q      Max
 -0.8985673 -0.8985673 -0.6648977 1.484674 2.654079

Coefficients:
             Value Std. Error  t value
logit.bpL 1.007603 0.01652859 60.96121

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 24398.78 on 17600 degrees of freedom

Residual Deviance: 19226.99 on 17599 degrees of freedom

Number of Fisher Scoring Iterations: 4
```

"S-plus.txt" 84b ≡

```
 lbifit.ZU <- update(lbifit, ~ . + nLzero + nUfactor)
 lbifit.ZUL <- update(lbifit.ZU, ~ . + nUzero + nLfactor)
 anova(lbifit.ZUL)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

```
Analysis of Deviance Table

Binomial model

Response: lbi

Terms added sequentially (first to last)
          Df Deviance Resid. Df Resid. Dev
     NULL                 17600   24398.78
```

```
logit.bpL  1 5171.788      17599   19226.99
   nLzero  2    0.186      17597   19226.81
 nUfactor 14    3.677      17583   19223.13
   nUzero  0    0.000      17583   19223.13
 nLfactor  3    4.393      17580   19218.74
```

Again the fit is satisfactory.

7.3.3 Analysing Deaths in S-plus

The death rates can be investigated in the same way:

"S-plus.txt" 85a ≡

```
# Fitting udi
 dpU <- nU/(nU+p)
 logit.dpU <- log(dpU/(1-dpU))

 udifit <- glm(udi ~ -1+logit.dpU, family=binomial,
               subset = (nU!=0))
 summary(udifit)

 udifit.1 <- update(udifit, ~ . + nUfactor + nLfactor)
 anova(udifit.1)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

"S-plus.txt" 85b ≡

```
# Fitting ldi
 dpL <- nL/(nL+p)
 logit.dpL <- log(dpL/(1-dpL))

 ldifit <- glm(ldi ~ -1+logit.dpL, family=binomial,
               subset = (nL!=0))
 summary(ldifit)

 ldifit.1 <- update(ldifit, ~ . + nLfactor + nUfactor)
 anova(ldifit.1)
```
◇

File defined by scraps 76, 77abc, 78abcd, 79abcd, 80ab, 81ab, 82abcd, 84ab, 85ab.

Both these analyses give satisfactory results: we shall not take space by going into details.

7.3.4 Comparison with conventional Markov chain Monte Carlo

We report a comparison with a conventional Metropolis-Hastings simulation of a Strauss point process carried out by Jens Lund and Elke Thönnes. Their simulation had as equilibrium a Strauss process in the unit square based on Poisson intensity parameter 50, interaction parameter 0.5, interaction radius 0.1. Based on 400000 steps they obtained a mean number of points of 31.24 with associated variance 21.13. The corresponding invocation of our program is

```
mh-cftp -i "strauss(0.1,0.5)" -n 100 -m 10 -z 0.1 -p 0.5
```

and 4000 replicates with varying seed yield a mean number of points of 32.23 with associated variance 23.22. This represents reasonable agreement!

Appendix: the Makefile

Here is the Makefile for the project. To make the executable and run a typical test, use `make` alone. To make the LaTeX documentation into a two-up `postscript` file, use `make doc`. Preview the documentation using `make preview`.

"Makefile" 87 ≡

```
# Makefile to weave a coding project, written by W.S. Kendall.
# derived from: $Source: /home/fisher/wsk/ms/Moeller/CPrograms/MH.w,v $
#               $Revision: 3.30 $
#               $Date: 2000/01/17 11:13:19 $

# Actions
CC = gcc
CFLAGS = -Wall -O3
LINK = gcc
LINKFLAGS = -lm

# Variables
PROJECT   = MH
TARGET    = mh-cftp
OBJECTCODE= \
        birth.o         cand.o          CFTP.o          check.o \
        death.o         display.o       inc.o           loc.o   \
        mh-cftp.o       pop.o           rand.o          sched.o \
        sim.o

# Rule pattern
.SUFFIXES: .o .c

.c.o:; $(CC) $(CFLAGS) -o $@ -c $*.c

# Demonstration
DEMOFLAGS = -d -i null -p 0.5 -n 2 -t 2
DEMOOUT   = $(PROJECT).out

# Options for null, Poisson, Strauss interactions
NULLFLAGS = -i null             -p 0.5 -n 36 -m 6
POISFLAGS = -i poisson          -p 0.5 -n 36 -m 6
STRAFLAGS = -i "strauss(1,0.5)" -p 0.5 -n 36 -m 6
```

◇

File defined by scraps 87, 88, 89.

Main targets are:

- `main`, the main target, which invokes `NuWeB` because of the reference to the LaTeX file, but actually tests the resulting `C` code;
- `doc`, which creates a two-up postscript file of documentation;

Note that much of the action is concealed by being prescribed in previous included files!

"Makefile" 88 ≡

```
# Main target
main: $(PROJECT).tex null Poisson Strauss

# Other targets
null: $(TARGET)
	-$(TARGET) $(NULLFLAGS)
	echo

Poisson: $(TARGET)
	-$(TARGET) $(POISFLAGS)
	echo

Strauss: $(TARGET)
	-$(TARGET) $(STRAFLAGS)
	echo

doc: $(PROJECT).tex demo $(PROJECT).ps2
```

◇

File defined by scraps 87, 88, 89.

Various auxiliary targets include, in particular, the special information required to link the object files together to form the executable. Notice that the target `MH.dvi` may have to be made repeatedly in order to sort out labels.

"Makefile" 89 ≡

```
# Auxiliary targets
$(TARGET): $(OBJECTCODE)
        $(LINK) -o $@ $(OBJECTCODE) $(LOADLIBS) $(LINKFLAGS)

demo: $(TARGET)
        -$(TARGET) $(DEMOFLAGS) > $(DEMOOUT)

$(PROJECT).tex: $(PROJECT).w demo
        nuweb $(PROJECT)

$(PROJECT).ps2: $(PROJECT).ps
        twoup $(PROJECT).ps > $(PROJECT).ps2

$(PROJECT).ps: $(PROJECT).dvi
        dvips $(PROJECT)

$(PROJECT).dvi: $(PROJECT).tex
        latex $(PROJECT)
◇
```

File defined by scraps 87, 88, 89.

Bibliography

[1] P. Briggs. `NuWeB` Version 0.87b: A simple literate programming tool. Published on the World-Wide Web by `preston@cs.rice.edu`, 1992.

[2] C. Chatfield. *Statistics for technology.* Chapman and Hall, London, second edition, 1978.

[3] J.A. Fill and M. Machida. Stochastic monotonicity and realizable monotonicity. preprint, Department of Mathematical Sciences, Johns Hopkins University, Baltimore, 1998.

[4] W.S. Kendall. On some weighted Boolean models. In D. Jeulin, editor, *Advances in Theory and Applications of Random Sets*, pages 105–120, Singapore, 1997. World Scientific.

[5] W.S. Kendall. Perfect simulation for the area-interaction point process. In L. Accardi and C.C. Heyde, editors, *Probability Towards 2000*, pages 218–234, New York, 1998. Springer Verlag.

[6] W.S. Kendall and J. Møller. Perfect simulation of point processes using Metropolis-Hastings algorithms. Research report 347, Department of Statistics, University of Warwick, 1999.

[7] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 2 edition, 1981.

[8] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[9] Silvio Levy and Donald E. Knuth. `CWEB` user manual: The CWEB system of structured documentation. Technical Report STAN-CS-83-977, Stanford University, October 1990. Available for anonymous ftp from `labrea.stanford.edu` in directory `pub/cweb`.

[10] W.H. Press, S.A. Teukowsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in* `C`. Cambridge University Press, Cambridge, 2 edition, 1992.

Index: Identifiers used in the code

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

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.