Chapter 6

Implementation

6.1 Implementation of RWS

6.1.1 Implementation of Example 2.9

The computation of Example 2.9 was done by the following C program.

```
/*============================================================*/
/*  file name: rws_example.c                                  */
/*============================================================*/
#include <stdio.h>

#define SAMPLE_NUM 1000000
#define M          100
#define M_PLUS_J   119

/* seed */
char xch[M_PLUS_J] =
   "1110110101" "1011101101" "0100000011" "0110101001" "0101000100"
   "0101111101" "1010000000" "1010100011" "0100011001" "1101111101"
   "1101010011" "111100100";
char ach[M_PLUS_J] =
   "1100000111" "0111000100" "0001101011" "1001000001" "0010001000"
   "1010101101" "1110101110" "0010010011" "1000000011" "0101000110"
   "0101110010" "010111111";

int  x[M_PLUS_J], a[M_PLUS_J];

void longadd(void)   /* x = x + a ( long digit addition ) */
{
 int i, s, carry = 0;
 for ( i = M_PLUS_J-1; i >= 0; i-- ){
   s = x[i] + a[i] + carry;
   if ( s >= 2 ) {carry = 1; s = s - 2; } else carry = 0;
   x[i] =  s;
```

```
 }
}

int maxLength(void)  /* count the longest run of 1's */
{
 int len = 0, count = 0, i;
 for ( i = 0; i <= M-1; i++ ){
  if ( x[i] == 0 ){ if ( len < count ) len = count; count = 0;}
  else count++ ;  /*  if x[i]==1 */
 }
 if ( len < count ) len = count;
 return len;
}

int main()
{
 int n, s = 0;
 for( n = 0; n <= M_PLUS_J-1; n++ ){
  if( xch[n] == '1' ) x[n] = 1; else x[n] = 0;
  if( ach[n] == '1' ) a[n] = 1; else a[n] = 0;
 }
 for ( n = 1 ; n <= SAMPLE_NUM ; ++n ){
  longadd();
  if ( maxLength() >= 6 ) s++;
 }
 printf ( "s=%6d, p=%7.6f\n", s, (double)s/(double)SAMPLE_NUM);
 return 0;
}
/*=============== End of rws_example.c ==================*/
```

6.1.2 Implementation of Example 5.9

The computation of Example 5.9 and Figure 5.2 was done by the following C program.[†1] It uses two functions `m90setseeds` and `m90randombit` of the C language library *random_sampler* which will be introduced in § 6.2.

```
/*=======================================================*/
/*  file name: rws_S500.c                                */
/*=======================================================*/
#include <stdio.h>
#include <time.h>
#include "random_sampler.h"

#define SAMPLE_NUM 10000000
#define M          500
#define M_PLUS_J   523
```

[†1] The output of this program is very long, so redirect it to a text file.

```
int  x[M_PLUS_J], a[M_PLUS_J], hist[M+1];

void longadd(void)   /* x = x + a ( long digit addition ) */
{
 int i, s, carry = 0;
 for ( i = M_PLUS_J-1; i >= 0; i-- ){
   s = x[i] + a[i] + carry;
   if ( s >= 2 ) {carry = 1; s = s - 2; } else carry = 0;
   x[i] =  s;
 }
}

int s500(void)
{
 int s = 0, i;
 for ( i = 0; i <= M-1; i++ ) s += x[i];
 return s;
}

int main()
{
 int n,i;

 m90setseeds(664426,5161592,7773372,84171419,1545);
 for( i = 0; i <= M ; i++ ) hist[i] = 0;
 for( i = 0; i <= M_PLUS_J-1; i++ ) x[i] = m90randombit();
 for( i = 0; i <= M_PLUS_J-1; i++ ) a[i] = m90randombit();

 for ( n = 1 ; n <= SAMPLE_NUM ; n++ ){
  longadd();
  hist[s500()]++;
  if ( n == 1000 || n == 10000 || n == 100000 || n == 10000000 ){
   printf ("%d samples:\n",n);
   for( i = 0; i <= M ; i++ )
    printf ( "%3d : %8.7f\n", i,(double)hist[i]/(double)n);
   printf ("\n");
  }
 }
 return 0;
}
/*=============== End of rws_S500.c ===================*/
```

6.2 C language library : *random_sampler*

We introduce a C language library *random_sampler*, which provides the pseudorandom generator by means of Weyl transformation and the dynamic random Weyl sampling (DRWS).

- m90random
  The pseudorandom generator by means of Weyl transformation (§ 4.2.1) with $\alpha = (\sqrt{5}-1)/2$, $m = 90$ and $j = 60$ by the discretization (4.6), (4.7) and (4.8).[†2] This is a multi-purpose pseudorandom generator.

- DRWS
  DRWS (§ 5.4.1) with $K = j = 31$ by (5.12), (5.13) and (5.14). This is a pseudorandom generator exclusively for the Monte Carlo integration. Its random source is m90random.

6.2.1 Source code

The source code consists of two files; `random_sampler.c` (body of the library) and `random_sampler.h` (header).

• `random_sampler.c`

```
/*==========================================================*/
/*  file name: random_sampler.c                             */
/*==========================================================*/
#include <stdlib.h>

#define  LIMIT_30  0x3fffffff
#define  LIMIT_31  0x7fffffff
#define  CARRY_31  0x40000000
#define  CARRY_32  0x80000000

static unsigned long  omega[5]; /* for m90random */

struct data_pair_s {            /* for DRWS */
   unsigned long x1;
   unsigned long x2;
   unsigned long a1;
   unsigned long a2;
   struct data_pair_s *next;
};
typedef struct data_pair_s  data_pair_t;

static long location;
static long locmax;
static long locmaxmax=-1;
static data_pair_t  random_list;
static data_pair_t  *current_ptr;

/*==========================================================*/
/*   Functions for pseudo-random generation "m90random"      */
```

[†2]The prefix "m90" indicates $m = 90$.

```
/* Initialization */
void  m90setseeds( unsigned long s0, unsigned long s1,
                   unsigned long s2, unsigned long s3,
                   unsigned long s4 )
{
   omega[0] = s0 & LIMIT_30; omega[1] = s1 & LIMIT_30;
   omega[2] = s2 & LIMIT_30; omega[3] = s3 & LIMIT_30;
   omega[4] = s4 & LIMIT_30;
}

/* Returns the current seeds */
void  m90getseeds( unsigned long *sp0, unsigned long *sp1,
                   unsigned long *sp2, unsigned long *sp3,
                   unsigned long *sp4 )
{
   *sp0 = omega[0]; *sp1 = omega[1]; *sp2 = omega[2];
   *sp3 = omega[3]; *sp4 = omega[4];
}

/* Generates a random bit */
char  m90randombit(void)
{
   static unsigned long alpha[5] = { /* Data of (sqrt(5)-1)/2 */
      0x278dde6e, 0x17f4a7c1, 0x17ce7301, 0x205cedc8, 0x0d042089
   };
   char data_byte;
   union bitarray {
      unsigned long of_32bits;
      char of_8bits[4];
   } data_bitarray;
   int j;

   for ( j=4; j>=1; ){
      omega[j] += alpha[j];
      if ( omega[j] & CARRY_31 ){ omega[j] &= LIMIT_30; omega[--j]++; }
      else --j;
   }
   omega[0] += alpha[0]; omega[0] &= LIMIT_30;
   data_bitarray.of_32bits = omega[0] ^ omega[1] ^ omega[2];
   data_byte =  data_bitarray.of_8bits[0] ^ data_bitarray.of_8bits[1]
              ^ data_bitarray.of_8bits[2] ^ data_bitarray.of_8bits[3];
   data_byte ^= ( data_byte >> 4 );
   data_byte ^= ( data_byte >> 2 );
   data_byte ^= ( data_byte >> 1 );
   return( 1 & data_byte );
}

/* Generates a 31 bit random integer */
```

```
unsigned long m90random31(void)
{
   int j;
   unsigned long b=0;
   for ( j=0; j<30; j++ ) { b |= m90randombit(); b <<= 1; }
   b |= m90randombit();
   return b;
}

/* Generates a 31 bit random real in [0,1) */
double m90randomu(void)
{
   return (double)m90random31()/CARRY_32;
}

/*==========================================================*/
/*   Functions for dynamic random Weyl sampling "DRWS"      */

/* Initialization */
void init_drws(void)
{
   locmax = -1; location = -1; random_list.next = 0;
}

/* Finalization */
void end_drws(void)
{
   data_pair_t *previous_ptr;
   if (random_list.next != 0){
      current_ptr = random_list.next;
      previous_ptr = &random_list;
      while (current_ptr -> next !=0){
         previous_ptr = current_ptr;
         current_ptr = current_ptr -> next;
      }
      free(current_ptr);
      previous_ptr -> next = 0;
      end_drws();
   }
}

/* Returns the locmax */
long get_locmax(void)
{
   return locmax;
}

/* Sets the locmaxmax */
void set_locmaxmax(long n)
```

```
{
   locmaxmax = n;
}

/* Sets the first location */
void set_first_location(void)
{
   location = -1; current_ptr = &random_list;
}

/* Generates a dynamic random Weyl sample (31 bit integer) */
unsigned long drws31(void)
{
   data_pair_t *p;

   location++;
   if ((locmaxmax > 0)&&(location > locmaxmax )) return m90random31();

   if ( location > locmax ){
      p = (data_pair_t *) malloc(sizeof(data_pair_t));
      if ( p == 0 ) return CARRY_32;

      current_ptr -> next = p;
      p -> x1 = m90random31(); p -> x2 = m90random31();
      p -> a1 = m90random31(); p -> a2 = m90random31();
      p -> next = 0;
      locmax++;
   }
   current_ptr = current_ptr -> next;
   current_ptr -> x2 += current_ptr -> a2;
   current_ptr -> x1 += current_ptr -> a1;
   if ( current_ptr -> x2 & CARRY_32 ) {
      current_ptr -> x2 &= LIMIT_31;
      current_ptr -> x1 ++;
   }
   return ( current_ptr -> x1 &= LIMIT_31 );
}

/* Generates a dynamic random Weyl sample in [0,1) */
double drwsu(void)
{
   unsigned long drws31copy = drws31();
   if ( drws31() == CARRY_32 ) return -1.0;
   else return (double)drws31copy/(double)CARRY_32;
}
/*=============== End of random_sampler.c ===================*/
```

• `random_sampler.h`

```
/*==========================================================*/
/*  file name: random_sampler.h                             */
/*  (header for random_sampler.c)                           */
/*==========================================================*/

/* Constant */

#define RANDMAX   0x80000000

/* Functions for pseudo-random generation "m90random"  */

extern  void           m90setseeds(unsigned long, unsigned long,
                                   unsigned long, unsigned long,
                                   unsigned long);
extern  void           m90getseeds(unsigned long *, unsigned long *,
                                   unsigned long *, unsigned long *,
                                   unsigned long *);
extern  char           m90randombit(void);
extern  unsigned long  m90random31(void);
extern  double         m90randomu(void);

/* Functions for dynamic random Weyl sampling "DRWS"  */

extern  void           init_drws(void);
extern  void           end_drws(void);
extern  long           get_locmax(void);
extern  void           set_locmaxmax(long);
extern  void           set_first_location(void);
extern  unsigned long  drws31(void);
extern  double         drwsu(void);

/*=============== End of random_sampler.h ===================*/
```

6.2.2 Specification of constant and function

The constant and the functions included in *random_sampler* are;

- Constant
  - `RANDMAX`
    Its value is `0x80000000` $= 2^{31} = 2,147,483,648$.
- m90random
  - `void m90setseeds(unsigned long, unsigned long, ...);`
    assigns 5 `unsigned long` integers as a *seed* to initialize m90random. This seed corresponds to $\tilde{x}$ of (4.6)(4.7) in § 4.2.1. In any Monte Carlo methods,

all results are function of the seed, and hence, this initialization must always be done.

- `void m90getseeds(unsigned long *, unsigned long *, ...);`
  saves the present status of m90random to 5 `unsigned long` variables. Passing them to `m90setseeds`, we can make m90random recover the saved status.
- `char m90randombit();`
  returns a 1 bit integer (0 or 1) at random. This corresponds to $Y_n^{(m)}(\tilde{x}; \lfloor\alpha\rfloor_{m+j})$ defined by (4.6), (4.7) and (4.8). $Y_n^{(m)}(\tilde{x}; \lfloor\alpha\rfloor_{m+j})$ is the parity of the upper m bit of $\tilde{x} + n\lfloor\alpha\rfloor_{m+j}$, which is quickly calculated here.
- `unsigned long m90random31();`
  returns an unsigned 31 bit integer ($0 \sim 2^{31}-1 =$ `RANDMAX` $-1 =$ `0x7fffffff`) at random. This function makes 31 calls of `m90randombit()` to make a 31 bit integer.
- `double m90randomu();`
  returns a $[0, 1]$-valued real number in 31 bit precision at random. More precisely, it returns `m90random31()/RANDMAX`.

• DRWS
Here we assume that the integrand in question satisfies Assumption 1.9 in § 1.3.1, and we use the symbols appeared in § 5.4.3.

- `void init_drws();`
  initializes DRWS. This function must be called at the beginning of DRWS.
- `void end_drws();`
  releases the computer memory that DRWS has used. This function must be called at the end of DRWS.
- `void set_first_location();`
  should be called once, before Z_1 is generated to produce each sample.
- `unsigned long drws31();`
  returns an unsigned 31 bit integer. Call this function when you generate $Z_1, Z_2, \ldots$ to compute each sample of f. If the current number of Z_l's gets bigger than the upper limit specified by `set_locmaxmax`, *random_sampler* switches from DRWS to i.i.d.-sampling, i.e., `drws31()` calls `m90random31()` and returns its value. If the memory is exhausted, `drws31()` returns `RANDMAX` to warn the programmer.
- `double drwsu();`
  returns a $[0, 1]$-valued real number in 31 bit precision. Call this function when you generate $Z_1, Z_2, \ldots$ to compute each sample of f. More precisely, it returns `drws31()/RANDMAX`, unless the memory is exhausted. If it is exhausted, `drwsu()` returns -1.0 to warn the programmer.
- `long get_locmax();`
  returns the maximum number T of $Z_1, \ldots, Z_T$ that have been currently generated to sample f.

– `void set_locmaxmax(long);`
  specifies the upper limit of the number of Z_l's. Assigning -1 means specifying no upper limit. In the default setting, no upper limit is specified.

6.2.3 Sample codes

• m90random

Some functions of the pseudorandom generator m90random have been used in § 6.1.2. They are also used in the following program `drws.c`.

• DRWS

A sample program `drws.c` below computes the mean of f of Example 1.12 in § 1.3.1. Namely, it computes the mean of the first time when the total number of Heads becomes 5 in successive coin tosses.

```
/*==============================================================*/
/*    drws.c : A sample program for DRWS                        */
/*==============================================================*/
#include <stdio.h>
#include "random_sampler.h"

#define SAMPLE_SIZE  1000000

int main()
{
   unsigned long halfmax = RANDMAX >> 1;
   long i;
   int number_of_heads, f;
   double sum_of_f;

   m90setseeds(0,53,0,0,0);
   init_drws();

   sum_of_w=0.0;
   for ( i=1; i <= SAMPLE_SIZE; i++ ){
      number_of_heads=0;
      f=0;
      set_first_location();
      while ( number_of_heads < 5 ){
         f++;
         if ( drws31() >= halfmax ) number_of_heads++;
      }
      sum_of_f += f;
   }
   printf("Mean of hitting time = %f\n", sum_of_f/SAMPLE_SIZE);
```

```
32:   printf("locmax = %d\n", get_locmax());
33:   end_drws();
34:   return 0;
35:}
```

Comments are given with line numbers.

05: loads the header `random_sampler.h`.

12: The constant `RANDMAX` is defined in `random_sampler.h` as `0x80000000`. The maximum value that the functions

`m90random31(), drws31()`

can take is `RANDMAX`−1. In this line, the variable `halfmax` is defined as half of it, i.e., as `0x40000000`.

17: initializes the pseudo-random generator m90random. The arguments should be set by the user, but here, to make the program short, we fixed them.

18: initializes DRWS.

21: The body of `for` loop is repeated `SAMPLE_SIZE` $= 1,000,000$ times.

24: prepares to generate the first Z_1.

27: Each time `drws31()` is called, a 31 bit integer is generated, which corresponds to $Z_1, Z_2, \ldots$. If it is larger than `halfmax`, which occurs with probability $1/2$, the variable `number_of_heads` increases by 1.

28: The end of the loop of `while` in line 25. When `number_of_heads` $= 5$, the thread comes out of the loop. By that time, the number of calls of `drws31()` varies by circumstances.

29: The value of `f` in the right-hand side is the realized value of the random variable f.

31: Getting out of the `for` loop in line 21, the experiment is over. The mean of W is estimated by

`sum_of_w/SAMPLE_SIZE`.

This sample program outputs 10.000073 for it.

32: `get_locmax()` returns the maximum number T of $Z_1, \ldots, Z_T$ that have been generated through the whole process. This sample program outputs 37 for it.

33: Finally, `end_drws()` releases the memory used by DRWS.

The point of using DRWS of *random_sampler* is to call

`set_first_location()`

before generating each sample. Then, call `drws31()` or `drwsu()` to generate $Z_1, Z_2, \ldots$ in oder.

6.2.4 Restrictions of use

• m90random: The upper limit of sample size

The upper limit of sample size that `m90randombit()` can generate is thought to be the critical sample number defined by (4.18), which is $N_c^{(90)}(10000) = 8.7 \times 10^{14}$ bits. The upper limit of sample size that `m90random31()` and `m90randomu()` can generate is thought to be $N_c^{(90)}(10000)/31 = 2.8 \times 10^{13}$.†3

• DRWS: The upper limit of sample size

The upper limit of sample size of DRWS is $2^{32} = 4,294,967,296$. This means, for example, that `SAMPLE_SIZE` in the sample program (§ 6.2.3) must not be larger than $4,294,967,296$.†4

• DRWS: Memory administration

For each Z_l, DRWS spends 160 bits (= 20 bytes) †5 of computer memory. For example, the sample program (§ 6.2.3) outputs `locmax = 37`, which means that it spent $37 \times 20 = 740$ byte memory. Recently, computers have a lot of memory, and so, usually, this is not a big problem. But, in some large scale computations, namely, when the probability that f requires too many Z_l's is not negligible, DRWS may exhaust the memory.

So, it is recommended to call `get_locmax()` to always check how much memory is currently used. The more practical method is the following; set the upper limit by `set_locmaxmax` so as to switch to i.i.d.-sampling from DRWS, if the current number of Z_l's exceeds the upper limit.

†3 It is actually possible to generate more samples than the upper limit of sample size stated here, but in that case, non-zero correlation that cannot be ignored might appear.

†4 It is actually possible to generate more samples than the upper limit of sample size stated here, but in that case, the pairwise independence of samples is not assured.

†5 In the case where `unsigned long` is 32 bit = 4byte.