```
   Help
/* ---------------------------------- */
/* PSEUDO RANDOM NUMBERS GENERATORS. */
/* ---------------------------------- */

#include "mathtools.h"
#include "optype.h"
#include "random.h"
#include "var.h"

#include <limits.h>
#include <float.h> /*TOCHECK*/
#include <math.h>
#include <stdio.h>

#define INV_M (1.0/M)
#define INV_M1 (1.0/M1)

static long counter= 0;

double ArrayOfRandomNumbers[MONTECARLOMAX];

/*********************************/
/******MONTE CARLO STANDARD******/
/*********************************/

/* ------------------------------------------------
    ---------------------- */
/* Random numbers generator of Knuth :
   It is  based on MRG and it uses a substractiv
    e method  */
/* ------------------------------------------------
    ---------------------- */
void KNUTH(int dimension,double *sample)
{
  static long M= 1000000000;
  static long SEED= 161803398;

  /* Initialize the sequence with a positive seed
     */
  static alea= 1;
```

```
static int inc1, inc2;
static long t_alea[56];
long X_n, y_k;
int i, ii, l;


/* First call to the sequence */
if(counter == 1)
  {
    X_n= SEED- alea;
    X_n%= M;
    t_alea[55]= X_n;
    y_k= 1;
    /* Initialization of the table */
    for(i= 1; i<= 54; i++)
{
  ii= (21*i)%55; /* 21 was chosen to alleviat
  e initial
        nonrandomness problems */
  t_alea[ii]= y_k;
  y_k= X_n - y_k;
  if(y_k < 0)
    y_k+= M;
  X_n= t_alea[ii];
}

    /* Randomization of the elements of the ta
  ble */
    for(l=  1; l<= 4; l++)
{
  for(i= 1; i<= 55; i++)
    {
      t_alea[i]-= t_alea[1+(i+30)%55];
      if(t_alea[i] < 0)
  t_alea[i]+= M;
    }
}
    inc1= 0;
    inc2= 31;  /* 31 is a special value of Knut
  h : 31= 55-24 */
    alea= 1;
```

```
    }

  counter+= 1;

  /* For each call to the sequence, computation
    of a new point */
  if(++inc1 == 56)
    inc1= 1;
  if(++inc2 == 56)
    inc2= 1;
  /* Substractive method*/
  X_n= t_alea[inc1] - t_alea[inc2];

  if(X_n < 0)
    X_n+= M;
  t_alea[inc1]= X_n;
  /* Normalized value */
  *sample=X_n*INV_M;
  return;
}



/* ------------------------------------------
    ------------------------- */
/* Combination of two multiplicative recursive g
    enerators of order 3 (k=3) */
/* ------------------------------------------
    ------------------------- */
void MRGK3(int dimension, double *sample)
{
  static double M1= 4294967087.0;
  static double M2= 4294944443.0;
  static double A12= 1403580.0;
  static double A13N= 810728.0;
  static double A21= 527612.0;
  static double A23N= 1370589.0;
  static double NORM= 2.328306549295728e-10;

  static double x10, x11, x12, x20, x21, x22;
  long k;
```

```
double p1, p2;

/* First call to the sequence */
if (counter == 1 )
  {
    /* Initialization */
    x10=231458761.;
    x11=34125679.;
    x12=45678213.;
    x20=57964412.;
    x21=12365487.;
    x22=77221456.;
    counter+=1;
  }

/* For each call to the sequence, computation
  of a new point */
/* First generator */
p1= A12*x11 - A13N*x10;
k= (long)floor(p1/M1); /*TOCHECK*/
p1-= k*M1;

if(p1 < 0.0)
  p1+= M1;

x10= x11;
x11= x12;
x12= p1;

/* Second generator */
p2= A21*x22 - A23N*x20;
k= (long)floor(p2/M2);/*TOCHECK*/
p2-= k*M2;

if(p2 < 0.0)
  p2+= M2;

x20= x21;
x21= x22;
x22= p2;
```

```
  /* Combination of the two generators */
  if (p1< p2)
    *sample= (p1- p2+ M1)*NORM;
  else
    *sample=(p1- p2)*NORM;
  return;
}



/* ------------------------------------------
    ------------------------- */
/* Combination of two multiplicative recursive g
    enerators of order 5 (k=5) */
/* ------------------------------------------
    ------------------------- */
void MRGK5(int dimension,double *sample)
{
  static double M1= 4294949027.0;
  static double M2=    4294934327.0;
  static double A12=   1154721.0;
  static double A14=   1739991.0;
  static double A15N=  1108499.0;
  static double A21=   1776413.0;
  static double A23=   865203.0;
  static double A25N= 1641052.0;
  static double NORM=  2.3283163396834613e-10;

  static double x10, x11, x12, x13, x14, x20, x21
    , x22, x23, x24;
  long k;
  double p1, p2;

  /* First call to the sequence */
  if (counter == 1)
    {
      /*Initialization*/
      x10= 231458761.;
      x11= 34125679.;
      x12= 45678213.;
```

```
    x13= 7438902.;
    x14= 957345.;

    x20= 57964412.;
    x21= 12365487.;
    x22= 77221456.;
    x23= 816403.;
    x24= 8488912.;
    counter+= 1;
  }

/* For each call to the sequence, computation
  of a new point */
/* First generator with Schrage method */
p1= A12*x13 - A15N*x10;

if(p1> 0.0)
  p1-= A14*M1;

p1+= A14*x11;
k= (long)floor(p1/M1);/*TOCHECK*/
p1-= k*M1;

if(p1< 0.0)
  p1+= M1;

x10= x11;
x11= x12;
x12= x13;
x13= x14;
x14= p1;

/* Second generator with Schrage method */
p2= A21*x24 - A25N*x20;

if(p2> 0.0)
  p2-= A23*M2;

p2+= A23*x22;
k= (long)floor(p2/M2);/*TOCHECK*/
p2-= k*M2;
```

```
  if(p2< 0.0)
    p2+= M2;

  x20= x21;
  x21= x22;
  x22= x23;
  x23= x24;
  x24= p2;


  /*Combination of the two generators */
  if (p1<= p2)
    *sample= (p1- p2+ M1)*NORM;
  else
    *sample=(p1- p2)*NORM;

  return;
}
/* ------------------------------------------
    -------------------------- */
/* Random numbers generator of  Park & Miller wit
    h Bayes & Durham shuffling
   procedure : the next random number is not obta
    ined from the previous one
   but we use an intermediate table which conta
    ins the 32 precedent random
   numbers and we choose one of them randomly.  *
    /
/* ------------------------------------------
    -------------------------- */
void SHUFL(int dimension,double *sample)
{
  static long A= 16807;        /* multiplier */
  static long M= 2147483647L;    /* 2**31 - 1  */
  static long Q= 127773L;        /*   M div A  */

  static long R= 2836;           /*   M mod A  */
  long N1;

  int j;
```

```
long hi;                  /* high order bit */
static long y= 0;
static long t[32];        /* 32 refers to the si
  ze of a computer word */
/* Initialisation */
static long x;


N1=(M/32);

/* First call to the sequence */
if (counter == 1)
  {
    x= 1043618065;

    /* After 8 "warm-ups", initialisation of th
  e shuffle table */
    for (j= 39; j>= 0; j--)
{
  hi= x/Q;
  /*Schrage's method to avoid overflows */
  x= A*(x- hi*Q)- R*hi;
  if (x < 0)
    x+= M;
  if (j< 32)
    t[j]= x;
}
    y= t[0];
  }
counter+= 1;


/* For each call to the sequence, computation
  of a new point */
hi= x/Q;
x= A*(x-hi*Q)- R*hi;
if (x < 0)
  x+= M;

/* Shuffling procedure of Bayes & Durham */
/* Index j dependent on the last point */
```

```
  j= y/N1;
  /* Next point dependent on j */
  y= t[j];

  t[j]= x;
  *sample= INV_M*y;

  return;
}



/* ------------------------------------------
    -------------------------- */
/* Random numbers generator of L'Ecuyer with Bay
    es & Durham shuffling
   procedure :
   Combination of two short periods LCG to obta
    in a longer period generator.
   The period is the least common multiple of th
    e 2 others.  */
/* ------------------------------------------
    -------------------------- */

void ECUYER(int dimension,double *sample)
{
  static long A1= 40014;        /* multiplier of
    the 1st generator */
  static long A2= 40692;        /* multiplier of
    the 2nd generator */
  static long M1= 2147483647;   /* 2**31 - 1   */

  static long M2= 2147483399;   /* 2**31 - 249 */
  static long Q1= 53668;        /* m1 div a1   */

  static long Q2= 52774;        /* m2 div a2   */

  static long R1= 12221;        /* m1 mod a1   */
  static long R2= 3791;         /* m2 mod a2   */
  long N1;
```

```
static long x;
static long y= 978543162;
int j;
long hi;                /* high order bit */
static long z= 0;
static long t[32];      /* 32 is the size of a
  computer word */

N1= (M1/32);

/* First call to the sequence */
if (counter == 1)
  {
    x= 437651926;
    y= x;

    /* After 8 "warm-ups", initialisation of th
  e shuffle table */
    for (j= 39; j>= 0; j--)
{
  /* Park & Miller's generator */
  hi= x/Q1;
  x= A1*(x-hi*Q1) - R1*hi;
  if (x < 0)
    x+= M1;
  if (j < 32)
    t[j]= x;
}
    z= t[0];
  }
counter+= 1;

/* For each call to the sequence, computation
  of a new point */
/* First generator */
hi= x/Q1;
x= A1*(x-hi*Q1) - R1*hi;
if (x < 0)
  x+= M1;

/* Second generator */
```

```
  hi= y/Q2;
  y= A2*(y-hi*Q2) - R2*hi;
  if (y < 0)
    y+= M2;

  /* Shuffling procedure of Bqyes & Durham */
  /* Index j dependent on the last point */
  j= z/N1;
  /* Next point dependent on j */
  z= t[j]- y;
  t[j]= x;


  /* To avoid 0 value */
  if (z < 1)
    z+= M1-1;

  *sample= INV_M1*z;

  return;
}


/* ------------------------- */
/* ------------------------- */
/* Tausworthe Algorithm   */
/* ------------------------- */

/* maximal number of combined generators */
#define TAUS_MAX 10


/* ---------------------------------------------
    ------ */
/* Generation of a random bit
   Algorithm based on a prime polynomial :
   here we choose x^18 + x^5 + x^2 + x + 1 .
   It is described in 'Numerical Recipes in C'
    page 296. */
/* ---------------------------------------------
    ------ */
```

```
int bit_random()
{
  static int compt = 1;
  int degre = 18;
  static unsigned long a;
  unsigned long new_bit;

  /* Initialisation for the n first values */
  /* random number over [1, 2^18] ; 2^18= 262144
    */
  if(compt == 1)
    {
      a= 176355;
    }
  compt++;

  /* Next bit calculation by the recurrence rela
    tion  */
  new_bit= (a & (1<<17)) >> 17
    ^ (a & (1<<4)) >> 4
    ^ (a & (1<<1)) >> 1
    ^ (a & 1);
  a <<= 1;
  /* The new bit is shift on the right */
  a ^= new_bit;
  /* The most left bit is put to 0 */
  a ^= (1 << degre);

  return((int)new_bit);
}



/* ------------------------------------------
    ----------------- */
/* Generation of a word of k random bits. */
/* ------------------------------------------
    ----------------- */
unsigned long random_word(int k)
{
  int i, bit;
```

```
  unsigned long mot;

  mot= 0;
  for(i= 0; i< k; i++)
    {
      bit= bit_random();
      mot= (mot<<1) ^ bit;
    }
  return(mot);
}

/* ---------------------------------------------
    ----------------- */
/*  Tausworthe  Algorithm
   Combination  of J Tausworthe generators
             u(n)[j]= u(n-r)[j] ^ u(n-k)[j],
   with parameters k, q, r, s and t.
   Generator :
             v= =(u[0] ^ u[1] ... ^ u{J-1])/2^32.


   L= 32 length of a word.    */
/* ---------------------------------------------
    ----------------- */
void TAUS(int dimension,double *sample)
{
  int L= 32;
  int J= 3;

  int i;
  static unsigned long u[TAUS_MAX];
  static unsigned long c[TAUS_MAX];
  unsigned long b;
  static int k[TAUS_MAX], q[TAUS_MAX];
  static int s[TAUS_MAX], r[TAUS_MAX], t[TAUS_MAX
    ];
  unsigned long v= 0;

  /* First call to the sequence. Initialisation *
    /
  if (counter == 1)
```

```
  {
    /* Choice of the parameters to have ME-CF g
  enerators (cf L'Ecuyer) */
    k[0]= 31; q[0]= 13; s[0]= 12;
    r[0]= k[0]- q[0]; t[0]= k[0]- s[0];

    k[1]= 29; q[1]= 2; s[1]= 4;
    r[1]= k[1]- q[1]; t[1]= k[1]- s[1];

    k[2]= 28; q[2]= 3; s[2]= 17;
    r[2]= k[2]- q[2]; t[2]= k[2]-s[2];

    /* constant c : k bits to one and (L-k) bi
  ts to zero */
    /* c[j]= 2^32 - 2^(L-k[j]) */
    c[0]= 4294967294ul;
    c[1]= 4294967288ul;
    c[2]= 4294967280ul;

    /* initialisation of each generator */
    u[0]= 0; u[1]= 0; u[2]= 0;
    for(i= 0; i< J; i++)
{
  /* The first k bits are chosen randomly */
  u[i]= random_word(k[i]);
  /* The next L-k bits are initially fixed to
  zero */
  u[i] <<= (L- k[i]);
  /* Now they are computed with the recurrence
   on u */
  b= u[i] << q[i];
  b ^= u[i];
  b >>= k[i];
  u[i] ^= b;
}

    counter+= 1;
  }

/* For each call to the sequence, computation
  of a new point */
```

```
  for(i= 0; i< J; i++)
    {
      /* Calculus of the next point for the J gen
    erators */
      /* 6 steps explained by L'Ecuyer */
      b=(u[i]<<q[i]) ^ u[i];          /* Steps 1
    and 2 */
      b >>= t[i];                    /* Step 3 */
      u[i]= (u[i] & c[i]) << s[i];   /* Steps 4
    et 5 */
      u[i] ^= b;                     /* Step 6 */
      /* Combination : XOR between the J genera
    tors */
      v^= u[i];
    }

  /* Normalization by 1/2^32 */
  *sample=v * 2.3283064365e-10;

  return;
}
```

```
/* ****************************** */
/* ****************************** */
/* *****QUASI MONTE CARLO   ***** */
/* ****************************** */
/* ------------------------------------------
    ------------------*/
/* QUASI RANDOM NUMBERS ; LOW DISCEPANCY SEQUENCE
    S.
   SQRT, Van der Corput - Halton, Faure, Sobol, g
    eneralized Faure,
   Niederreiter */
/* ------------------------------------------
    ------------------*/
/* ------------------------------------------
    ------------------*/
```

```
/* Inverse cumulative distribution of the standar
    d gaussian variable.
/* ----------------------------------------------
    ------------------*/



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

#define DIM_MAX_QMC 300

#define MAXI 33
long Comb[MAXI][MAXI];/*Binomial Coefficients*/

/* ----------------------------------------*/
/* Table for the n first prime numbers */
/* ----------------------------------------*/
void prime_number(int n, int prime[])
{
  int i, bool;
  int tested_val= 3, nbre= 1;

  prime[0]= 2;

  while(nbre < n)
    {
      bool = 0;
      for(i = 0; i < nbre; i++)
  {
    /* Test if the value is divisible by one of
    the first prime numbers */
    if((tested_val % prime[i]) == 0)
      {
        bool = 1;
        break;
      }
  }
      /* If no divisor was found, the number is
```

```
    prime */
      if(bool == 0)
  {
    prime[nbre]= tested_val;
    nbre += 1;
  }
      /* Test for odd integers only */
      tested_val+= 2;
    }
}



/* ------------------------------------------
    ----------------------- */
/* Search the smallest element of a table great
    er than a given threshold.
   Used for the prime numbers */
/* ------------------------------------------
    ----------------------- */
int search_value(int seuil, int tab[], int dim)
{
  int min, max, indice;

  min= 0;
  max= dim- 1;
  indice= (int)dim/2;

  while((max- min) >0)
    {
      if(tab[indice]< seuil)
  min= indice+1;
      else
  max= indice;
      indice= (min + max)/2;
    }
  return(tab[indice]);
}
```

```
/* ----------------------------------------*/
/* Computation of the binomial coefficients.  */
/* ----------------------------------------*/
void binomial(int Max)
   /*Max should be less than 33 otherwise C[n][
    p] could exceed LONG_MAX*/
{
  int n, p, i;


  Comb[0][0]= 1;
  for (n= 1; n< Max; n++)
    {
      Comb[n][0]= 1;
      Comb[n][n]= 1;
      i= n-1;
      for (p= 1; p<= i; p++)
    Comb[n][p]= Comb[i][p]+ Comb[i][p-1];
    }
  return;
}

/* ------------------------------------------
    ------------------*/
/* SQRT Sequence */
/* Computation of the next element for the dim-
    dimensional sequence. */
/* ------------------------------------------
    ------------------*/

void SQRT(int dim, double X_m[])
{
  static int dimension=0;
  int i;
  static int prime[DIM_MAX_QMC];
  static double alpha[DIM_MAX_QMC];

  /* Verification of the dimension. It must not
    change without reinitializing */
  if(dimension != dim)
    counter= 1;
```

```
  /* First call : initialisation */
  if(counter == 1)
    {
      dimension= dim;
      prime_number(dim, prime);
      for(i=0; i<dim; i++)
  {
    alpha[i]= sqrt(prime[i]);
  }
    }

  /* For each call to the sequence, computation
    of a new point */
  for(i= 0; i< dim; i++)
    X_m[i]= ((counter*alpha[i])-floor(counter*al
    pha[i]));

  counter+= 1;

  return;
}




/* ---------------------------------------------
    -----------*/
/* Van der Corput sequence in base p.
   Computation of the next element of the sequen
    ce.  */
/* ---------------------------------------------
    -----------*/
void VANDERCORPUT(int p, double X_m[])
{
  int coeff;
  double y= 0;
  int x= 0;
  int puissance= 1;

  /* Digit expansion of n in base p */
```

```
  while ((counter-x)>0)
    {
      coeff= ((counter-x)/puissance)%p;
      x += coeff*puissance;
      puissance *= p;
      y += coeff*1.0/puissance;
    }
  X_m[0]= y;

  counter+= 1;
}




/* ---------------------------------------------
    --------------- */
/* Halton sequence. Bases p={p1, p2, ..., pd} of
    prime numbers ;
   Computation of the next element of the sequen
    ce X_n
   (index 0 to d-1) */
/* ---------------------------------------------
    --------------- */
void HALTON(int dim, double X_n[])
{
  static int dimension=0;
  int i;
  static int prime[DIM_MAX_QMC];
  int coeff;
  double y;
  int x, puissance;

  /* Verification of the dimension. It must not
    change without reinitializing */
  if(dimension != dim)
    counter= 1;

  /* First call : initialization */
  if( counter == 1)
    {
```

```
      dimension= dim;
      prime_number(dim, prime);
    }

  /* For each call to the sequence, computation
    of a new point */
  for(i = 0; i< dim; i++)
    {
      puissance= 1;
      x= 0; y= 0;
      /* radical inverse function */
      while ((counter-x)>0)
  {
    coeff= ((counter-x)/puissance)%prime[i];
    x += coeff*puissance;
    puissance *= prime[i];
    y += coeff*1.0/puissance;
  }
      X_n[i]= y;
    }
  counter+= 1;

  return;
}

/* ------------------------------------------
    ------------------*/
/* FAURE sequence */
/* r smallest odd prime number greater than d, th
    e dimension of the sequence
   Computation of the next element --> U_n[] (ind
    ex 0 to d-1) */
/* ------------------------------------------
    ------------------*/
/* maximal dimension  for the FAURE sequence */
#define DIM_MAX_FAURE 300
/* maximal samples for the FAURE sequence */
#define MAX_SAMPLE_FAURE 10000000

void FAURE(int d, double U_n[])
{
```

```
int coeff[MAXI];
int b[MAXI];
static int dimension=0, r;
int prime[DIM_MAX_QMC];
int x= 0, puissance1= 1, puissance2;
int indice, i, j, k, somm;


/* Verification of the dimension. It must not
  change without reinitializing */
if(dimension != d)
  counter= 1;

/*First call to the sequence */
if(counter == 1)
  {
    dimension=d;
    if((d == 2)||(d == 1))
  r= 3;
    else
{
  prime_number(d, prime);
  r=search_value(d,prime,d);
}
  }

/* Initialization */
for (i=0; i< d; i++)
  {
    U_n[i]= 0.;
  }

indice= 0;
/* For each call to the sequence, computation
  of a new point */

/* r-digit expansion of n --> first term of th
  e sequence */
while ((counter-x) > 0)

  {
```

```
      coeff[indice]= ((counter-x)/puissance1)%r;
      x += coeff[indice]*puissance1;
      puissance1 *= r;
      U_n[0] +=(double)coeff[indice]/(double)pui
    ssance1;
      indice +=1;
    }

  /* Other terms of the sequence */
  /* Successive transformations of the r-digit
    expansion.*/
  for (k=1; k< d; k++)
    {
      puissance2= r;
      for (j=0; j< indice; j++)
  {
    somm= 0;
    for (i=j; i< indice; i++)
      somm += Comb[i][j]*coeff[i];
    b[j]= somm%r;
  }
      for (j=0; j< indice; j++)
  {
    coeff[j]= b[j];
    U_n[k] += (double)coeff[j]/(double)puissance
    2;
    puissance2 *= r;
  }
    }
  counter+= 1;

  return;
}


/* ------------------------------------------
    ----------------- */
/* SOBOL SEQUENCE in dimension d (d<=39)
   X_n[] contains the terms of the n-th element
   The algorithm is based on a relation between X
    (n) and X(n-1)    */
```

```
/* Cf Numerical recipes in C, pages 312-313 */
/* ------------------------------------------
    ----------------- */

/* maximal dimension for the SOBOL sequence */
#define DIM_MAX_SOBOL 39
/* maximal length of bits for the  SOBOL sequence
     */
#define BIT_MAX_SOBOL 30

void SOBOL(int d, double X_n[])
{
  int i, j, k, P_j, deg_j;
  unsigned long aux, dim;
  static double facteur;
  static unsigned long initialX_n[DIM_MAX_SOBOL+1
    ];

  /* Degree of the DIM_MAX primitive polynomials
    */
  static int deg[DIM_MAX_SOBOL+1]={0, 1, 2, 3, 3,
     4, 4, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 7, 7,
     7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
     8, 8, 8};

  /* Index of each primitive polynomial. It deter
    mines the coefficients bi */
  static int P[DIM_MAX_SOBOL+1]={0, 0, 1, 1, 2, 1
    , 4, 2, 13, 7, 14, 11, 4, 1, 16, 13, 22, 19, 25,
     1, 32, 4, 8, 7, 56, 14, 28, 19, 50, 21, 42, 31,
     62,37, 41, 55, 59, 14, 56, 21};

  /* Sobol's constants C_i_(j) */
  static unsigned long C[DIM_MAX_SOBOL+1][BIT_MAX
    _SOBOL+1];
  /* Initial values for C */
  /* The second dimension is the maximum degree
    of a primitive polynomial +1 */
  static unsigned long C_init[DIM_MAX_SOBOL+1][9]
    ={
    {0,0, 0, 0, 0,  0,  0,  0,   0},
```

```
  {0,1, 0, 0, 0,  0,  0,  0,   0},
  {0,1, 1, 0, 0,  0,  0,  0,   0},
  {0,1, 3, 7, 0,  0,  0,  0,   0},
  {0,1, 1, 5, 0,  0,  0,  0,   0},
  {0,1, 3, 1, 1,  0,  0,  0,   0},
  {0,1, 1, 3, 7,  0,  0,  0,   0},
  {0,1, 3, 3, 9,  9,  0,  0,   0},
  {0,1, 3, 7, 13, 3 , 0,  0,   0},
  {0,1, 1, 5, 11, 27, 0,  0,   0},
  {0,1, 3, 5, 1,  15, 0,  0,   0},
  {0,1, 1, 7, 3,  29, 0,  0,   0},
  {0,1, 3, 7, 7,  21, 0,  0,   0},
  {0,1, 1, 1, 9,  23, 37, 0,   0},
  {0,1, 3, 3, 5,  19, 33, 0,   0},
  {0,1, 1, 3, 13, 11, 7,  0,   0},
  {0,1, 1, 7, 13, 25, 5,  0,   0},
  {0,1, 3, 5, 11, 7,  11, 0,   0},
  {0,1, 1, 1, 3,  13, 39, 0,   0},
  {0,1, 3, 1, 15, 17, 63, 13,  0},
  {0,1, 1, 5, 5,  1,  27, 33,  0},
  {0,1, 3, 3, 3,  25, 17, 115, 0},
  {0,1, 1, 3, 15, 29, 15, 41,  0},
  {0,1, 3, 1, 7,  3,  23, 79,  0},
  {0,1, 3, 7, 9,  31, 29, 17,  0},
  {0,1, 1, 5, 13, 11, 3,  29,  0},
  {0,1, 3, 1, 9,  5,  21, 119, 0},
  {0,1, 1, 3, 1,  23, 13, 75,  0},
  {0,1, 3, 3, 11, 27, 31, 73,  0},
  {0,1, 1, 7, 7,  19, 25, 105, 0},
  {0,1, 3, 5, 5,  21, 9,  7,   0},
  {0,1, 1, 1, 15, 5,  49, 59,  0},
  {0,1, 1, 1, 1,  1,  33, 65,  0},
  {0,1, 3, 5, 15, 17, 19, 21,  0},
  {0,1, 1, 7, 11, 13, 29, 3,   0},
  {0,1, 3, 7, 5,  7,  11, 113, 0},
  {0,1, 1, 5, 3,  15, 19, 61,  0},
  {0,1, 3, 1, 1,  9,  27, 89,  7},
  {0,1, 1, 3, 7,  31, 15, 45,  23},
  {0,1, 3, 3, 9,  9,  25, 107, 39}
};
```

```
/*First call to the sequence */
if(counter == 1)
  {
    /* Initialization of the full array C[i][j]
   */
    for (j=1; j<=DIM_MAX_SOBOL; j++)
initialX_n[j]= 0;

    facteur=1.0/(1L << BIT_MAX_SOBOL);

    for (j=1; j<=DIM_MAX_SOBOL; j++)
{
  deg_j = deg[j];
  /* Values of C_init in C */
  for (i= 0; i<= deg_j; i++)
    {
      C[j][i]= C_init[j][i];
    }
  /* Recurrence relation to compute the other
  C[i][j] */
  for (i= deg_j+1; i<= BIT_MAX_SOBOL; i++)
    {
      P_j= P[j];
      aux= C[j][i-deg_j];
      aux ^= (C[j][i-deg_j] << deg_j);
      for (k= deg_j-1; k>=1; k--)
  {
    /* Test for the coefficient b(k) */
    if (P_j & 1)
      aux ^= (C[j][i-k] << k);
    P_j >>= 1;
  }
      /* Final value for C[i][j] */
      C[j][i]= aux;
    }
}
  }

/* Calculation of a new quasi-random vector on
  each call */
```

```
  dim= counter;
  /* Research of the rightmost 0 bit */
  for (i=1; i<=BIT_MAX_SOBOL; i++)
    {
      if ((dim & 1) == 0)
  break;
      dim >>= 1;
    }
  /* Computation of the term n from the term (n-1
    ) */
  for (j=1; j<= d; j++)
    {
      initialX_n[j] = (initialX_n[j])^ (C[j][i]<<
     (BIT_MAX_SOBOL-i));
      /* normalization */
      X_n[j-1]= initialX_n[j] *facteur;
    }
  counter+= 1;

  return;
}




/* ------------------------------------------
    -- */
/* NIEDERREITER SEQUENCE in dimension d, in base
    2 */
/* X_n[] contains the terms of the n-th element
   The algorithm is based on a relation between X
    (n) and X(n-1)    */
/* /* ------------------------------------------
    ----- */

/* maximal dimension for the NIEDERREITER sequen
    ce */
#define DIM_MAX_NIED 12
/* maximal length of bits for the NIEDERREITER se
    quence */
#define BIT_MAX_NIED 30
```

```
void NIEDERREITER(int d, double X_n[])
{
  int i, j;
  unsigned long saut, gray, dim;
  static double facteur;
  static unsigned long initial_d, initialX_n[DIM_
    MAX_NIED+1];

  /* Niederreiter's constants */
  /* Une fonction de calcul de ces coefficients
    est donnee dans Owen.c */
  static unsigned long C[BIT_MAX_NIED+1][DIM_MAX_
    NIED+1]={
    {0,1073741824,1073741824,1610612736,187904819
    2,1879048192,2013265920,
     2013265920,2013265920,2080374784,2080374784,
    2080374784,2080374784},
    {0,536870912,1610612736,1207959552,1644167168
    ,1644167168,1887436800,
     1887436800,1887436800,2015363072,2015363072,
    2015363072,2015363072},
    {0,268435456,1342177280,939524096,1174405120,
    1442840576,1635778560,
     1769996288,1769996288,1885339648,1885339648,
    1885339648,1952448512},
    {0,134217728,2013265920,2046820352,503316480,
    771751936,1132462080,
     1400897536,1535115264,1625292800,1692401664,
    1692401664,1759510528},
    {0,67108864,1140850688,1577058304,742391808,1
    312817152,260046848,
     796917760,1065353216,1172307968,1306525696,1
    239416832,1373634560},
    {0,33554432,1711276032,914358272,1522532352,5
    15899392,386400256,
     1602748416,2131230720,266338304,467664896,33
    3447168,601882624},
    {0,16777216,1426063360,1702887424,935329792,1
    069547520,639107072,
     932708352,1981284352,465633280,937492480,669
```

```
057024,1205927936},
{0,8388608,2139095040,1260388352,2106064896,2
106064896,1287127040,
 1740111872,1815609344,933429248,1810038784,1
338179584,197328896},
{0,4194304,1077936128,1046478848,1800929280,1
767374848,435683328,
 1341652992,1484259328,1869021184,1407647744,
461832192,327614464},
{0,2097152,1616904192,2127036416,1152909312,1
387790336,863010816,
 393248768,946896896,1592721408,669974528,858
718208,655294464},
{0,1048576,1347420160,1572339712,456196096,66
1716992,1851883520,
 644448256,2020179968,973012992,1275002880,16
52490240,1243545600},
{0,524288,2021130240,827981824,639827968,1286
275072,1422622720,
 1154711552,1893433344,2011039744,333383680,1
090455552,339675136},
{0,262144,1145307136,1614675968,1548156928,42
5656320,697794560,
 162496512,1774157824,1807554560,597563392,10
0603904,679417856},
{0,131072,1717960704,1216249856,952508416,817
766400,1537673216,
 458721280,1543473152,1465595904,1125922816,2
01209856,1426012160},
{0,655336,1431633920,945651712,1908695040,190
3452160,1069946880,
 1043306496,1073190912,714504192,102332416,40
2487296,771651584},
{0,32768,2147450880,2050039808,1669980160,165
5300096,2139371520,
 2094512128,2137503744,1359804416,137558016,8
04976640,1541273600},
{0,16384,1073758208,1583374336,1192936448,146
1445632,2004908032,
 1907357696,1984428032,574156864,342224960,15
42844480,865859648},
```

```
{0,8192,1610637312,919095296,511552512,780127
232,1736994944,
 1809315968,1812428928,1081139392,686547136,9
38205376,1664612544},
{0,4096,1342197760,1706571776,754088960,13206
55872,1201168768,
 1471148416,1476817280,79806912,1442300352,18
11333568,1114634688},
{0,2048,2013296640,1264220672,1538054272,5225
98528,255345536,
 794291072,939811712,161711040,802130880,1473
022912,83819456},
{0,1024,1140868096,1044274688,924431744,10447
38432,376967040,
 1580193664,2013284224,325519296,1604198336,7
31389888,234684352},
{0,512,1711302144,2130622080,2088397696,20931
48032,611882760,
 878683912,1887440776,651102082,993804162,139
3639298,536412034},
{0,256,1426085120,1574823296,1794429712,17745
12912,1215931928,
 1614267928,1770004376,1235158790,1987673862,
570654470,1005715270},
{0,128,2139127680,823225120,1168671280,140154
9360,410242232,
 1223658552,1535129528,389942798,1825766990,1
143404110,1944254158},
{0,64,1077952576,1610636896,458752112,6561393
76,812620280,
 299341944,1065379832,777854046,1501951134,20
8530654,1743054302},
{0,32,1616928864,1207973576,649592930,1283443
810,1758968688,
 590295160,2131249016,1488664830,856416638,48
4104702,1340654590},
{0,16,1347440720,939550136,1563492422,4146862
94,1369962081,
 1180590193,1981288049,827816380,1645658814,9
66114238,468813820},
{0,8,2021161080,2046839642,941817886,82458846
```

```
  2,726658251,
   205277419,1815616739,1722809210,1143770494,1
  997176636,935528440},
  {0,4,1145324612,1577074238,1879506988,1879405
  006,1578619287,
   410522071,1484272071,1298134710,209263358,18
  46933048,1871120370},
  {0,2,1717986918,914390782,1644568666,16443575
  34,1009722151,
   947430191,946922383,513795374,483606012,1546
  380400,1596917668},
  {0,1,1431655765,1702911453,1174691895,1443162
  927,2019444294,
   2020722270,2020198302,1027523167,1032223673,
  1014481121,1048512329}};


/* First call to the sequence */
if (counter == 1)
  {

    /* Initialization of initX_n[] */
    for (i=1; i<=DIM_MAX_NIED; i++)
initialX_n[i]= 0;

    facteur= 1.0/(double)(1UL << (BIT_MAX_NIED+
  1));
    saut = 1L << 12;
    initial_d = saut;

    /* Gray code of saut */
    gray = saut^(saut >> 1);
    for (i=0; i<=BIT_MAX_NIED; i++)
{
  if (gray == 0)
    break;
  for (j= 1; j<= DIM_MAX_NIED; j++)
    {
      if ((gray & 1) == 0)
  break;
      /* XOR sum */
```

```
        initialX_n[j] ^= C[i][j];

      }
    gray >>= 1;
  }
    }

  /* Calculation of a new quasi-random vector on
    each call */
  dim= initial_d++;

  /* Research of the rightmost 0 bit */
  for (i=0; i<=BIT_MAX_NIED; i++)
    {
      if ((dim & 1) == 0)
  break;
      dim >>= 1;
    }
  /* Computation of the term n from the term n-1
    */
  for (j=1; j<= d; j++)
    {
      X_n[j-1]= (double)initialX_n[j]*facteur;
      initialX_n[j] ^= C[i][j];
    }
  counter+= 1;
  return;
}



/* Simulation of a standard gaussian variable
   with the inverse function.
   Used for Quasi Monte Carlo simulation */
double Inverse_erf(double x)
{
  static double c0= 2.515517;
  static double c1= 0.802853;
  static double c2= 0.010328;
  static double d1= 1.432788;
```

```
  static double d2= 0.189269;
  static double d3= 0.001308;

  double t,sign;

  if(x>0.5)
    {
      sign= +1.0;
      x= 1.0-x;
    }
  else
    {
      sign= -1.0;
    }
  t=sqrt(-2.0*log(x));
  return sign*(t -((c2*t+c1)*t+c0)/(1.0+t*(d1+t*(
    d2+d3*t))));
}



double Inverse_erf_Moro(double x)
{
  static double a[4] = {
    2.50662823884,
    -18.61500062529,
    41.39119773534,
    -25.44106049636};
  static double b[4] = {
    -8.47351093090,
    23.08336743743 ,
    -21.06224101826,
    3.13082909833};
  static double c[9] ={
    0.337475482276147,
    0.9761690190917186,
    0.1607979714918209,
    0.0276438810333863,
    0.0038405729373609,
    0.0003951896511919,
    0.0000321767881768,
```

```
    0.000002888167364,
    0.0000003960315187};

  double u,r;

  u=x-0.5;
  if (fabs(u)<0.42)
    {
      r=u*u;
      r=u*(((a[3]*r+a[2])*r+a[1])*r+a[0])/((((b[3
    ]*r+b[2])*r+b[1])*r+b[0])*r+1.0);
      return(r);
    }

  r=x;
  if(u>0.0)
    r=1.0-x;
  r=log(-log(r));
  r=c[0]+r*(c[1]+r*(c[2]+r*(c[3]+r*(c[4]+r*(c[5]+
    r*c[6]+r*(c[7]+r*c[8]))))));
  if (u<0.0)
    r=-r;
  return(r);
}

/* ********************************************
    ************************ */
/* ***************************** */
/* ***** Random Arrays      ***** */
/* ***************************** */

/* Maximum dimension for random sequences */
#define DIM_MAX 10000

/*Random Number Generator Array*/
Generator Random[]=
{
  {"KNUTH",&KNUTH,MC,DIM_MAX},
  {"MRGK3",&MRGK3,MC,DIM_MAX},
  {"MRGK5",&MRGK5,MC,DIM_MAX},
  {"SHUFL",&SHUFL,MC,DIM_MAX},
```

```
  {"L'ECUYER",&ECUYER,MC,DIM_MAX},
  {"TAUSWORTHE",&TAUS,MC,DIM_MAX},

  {"SQRT",&SQRT,QMC,DIM_MAX_QMC},
  {"HALTON",&HALTON,QMC,DIM_MAX_QMC},
  {"FAURE",&FAURE,QMC,DIM_MAX_FAURE},
  {"SOBOL",&SOBOL,QMC,DIM_MAX_SOBOL},
  {"NIEDERREITER",&NIEDERREITER, QMC,DIM_MAX_NIED
    },
  {"NULL{n",NULL,MC,DIM_MAX}
};

/*The number of generators is kept in the flag G
    EN_NUMBER in optype.h*/

int InitGenerator(int type_generator, int simula
    tion_dim,long samples)
{

  /*For the Faure sequences we need to compute bi
    nomial coefficients
    wich are storable with a Long Type only up
    to the number 32.
  The corresponding sample value is MAX_SAMPLE_
    FAURE*/
  if((Random[type_generator].Compute)==&FAURE)
  if (samples>MAX_SAMPLE_FAURE)
    return WRONG;

  if(simulation_dim >= Random[type_generator].Dim
    ension)
    return DIMENSION_QMC_SEQ_EXCEDEED;

  counter=1;

  return OK;
}



int Rand_Or_Quasi(int type_generator)
```

```
{
  return Random[type_generator].RandOrQuasi;
}


/* FOnction test pour le generateur SOBOL*/
/* Utilisee car pour l'instant la suite de SOBOL
    genere un segmentation fault a l'affichage des
    parametres, bien que les cqlculs soient effectues
    correctement */
int test_sobol(int type_generator)
{
  if (Random[type_generator].Compute==&SOBOL)
    return(0);
  else return(1);
}




/* ---------------------------------------- */
/* Simulation of a Gaussian random variable */
/* For a pseudo random generator.
   Used for Monte Carlo Simulation */
/* ---------------------------------------- */
double Gauss_AbramowitzStegun(int type_generator)
{
  double fac, rsq, v1, v2;
  static int iset= 0;
  static double gset;
  static double random_number;

  if(iset==1)
    {
      iset= 0;
      return gset;
    }

  do
    {
      Random[type_generator].Compute(0,&random_
```

```
    number);
      v1= 2.0*random_number-1.0;

      Random[type_generator].Compute(0,&random_
    number);
      v2= 2.0*random_number-1.0;

      rsq= v1*v1 + v2*v2;
    }
  while(rsq>=1.0 || rsq==0);

  fac= sqrt(-2.0*log(rsq)/rsq);
  gset= v1*fac;
  iset= 1;

  return v2*fac;
}



/* ---------------------------------------- */
/* Simulation of a Gaussian random variable */
/* ---------------------------------------- */
double Gauss_BoxMuller(int type_generator)
{
  double xs,ys;
  static double random_number;

  Random[type_generator].Compute(0,&random_numb
    er);
  xs=random_number;

  Random[type_generator].Compute(0,&random_numb
    er);
  ys=random_number;

  return sqrt(-2.0*log(xs))*cos(2.0*PI*ys);
}


/* Simulation of a Gaussian standard variable fo
```

```
    r Monte Carlo Simulation,
   that is with the Gauss_AbramowitzStegun func
    tion. */
double GaussMC(int dimension, int create_or_retri
    eve, int index, int type_generator)
     /* ****************************************
    ************************ */
     /* Usage:                 */
     /*  This function has the same parameters
    than the GaussQMC  one,
  but simulation is always called for only one
    dimension
  You can't simulate a n-dimensional vector,
  but it isn't necessary because there is no pr
    oblem of independence. */
     /* ****************************************
    ********************** */
{
  double g;

  g=  Gauss_BoxMuller(type_generator);
  /*Random[type_generator].Compute(dimension,Ar
    rayOfRandomNumbers);
    g=  Inverse_erf(ArrayOfRandomNumbers[0]);*/
  return(g);
}



/* Simulation of a Gaussian standard variable fo
    r Quasi Monte Carlo Simulation,
   that is with the inverse-erf function.
   This function can be called for the generatio
    n of a n-dimensional vector of
   independent variables: call to a n-dimensional
     low-discrepancy sequence. */
double GaussQMC(int dimension, int create_or_re
    trieve, int index, int type_generator)
     /* ****************************************
    ************************ */
```

```
     /* Usage:                */
     /*  If create_or_retrieve==CREATE (in optype.h
    )    */
     /*         -put dimension random numbers (g
    enerated with the
          QuasiGenerator type_generator) in the
    array
          ArrayOfRandomNumbers.    */
     /*         -returns ArrayOfRandomNumbers[0]
        */
     /*  else (create_or_retrieve==RETRIEVE) (in optype.h
    )    */
     /*         -returns ArrayOfRandomNumbers[
    index]    */
     /* ***************************************
    ********************* */
{
  if (create_or_retrieve == CREATE)
    {
      Random[type_generator].Compute(dimension,Ar
    rayOfRandomNumbers);/*NoCheck on dimension?*/
      return  Inverse_erf(ArrayOfRandomNumbers[0]
    );
    }
  else {
    return Inverse_erf(ArrayOfRandomNumbers[ind
    ex]);
  }
}



/* According to the generator type, Monte Carlo
    or Quasi Monte Carlo,
   selection of the appropriate gaussian functio
    n, GaussMC or GaussQMC. */
double (*Gaussians[])(int,int,int,int)={GaussMC,G
    aussQMC,NULL};
/* ***********************************************
    ************************ */
/*Usage:            */
```

```
/*  g=Gaussians[Rand_Or_Quasi(type_generator)](
    int dimension,int create_or_retrieve, int index,
    int type_generator)*/
/* ******************************************
    ************************ */




/* ------------------------------------- */
/* Simulation of a Bernoulli random variable */
/* ------------------------------------- */
int Bernoulli(double p, int type_generator)
{
  Random[type_generator].Compute(0,ArrayOfRandomN
    umbers);
  if (ArrayOfRandomNumbers[0]<p)
    return 1;
  else
    return  0;
}

/* ------------------------------------- */
/* Simulation of a Poisson random variable */
/* ------------------------------------- */
long Poisson(double lambda, int type_generator)
{
  double u;
  double a = exp(-lambda);
  long n = 0;
  static double random_number;

  Random[type_generator].Compute(0,&random_numb
    er);
  u = random_number;
  while (u>a){
   Random[type_generator].Compute(0,&random_numb
    er);
    u *= random_number;
    n++;
  }
```

```
  return n;
}


/* ---------------------------------------- */
/* Simulation of an Uniform random variable
   with one of the random numbers generators. */
/* ---------------------------------------- */
double Uniform(int type_generator)
{
  Random[type_generator].Compute(0,ArrayOfRandomN
    umbers);

  return ArrayOfRandomNumbers[0];
}

/* ----------------------------------------------
    -- */
/* Simulation of an Uniform random vector of siz
    e D
   with one of the random numbers generators. */
/* ----------------------------------------------
    -- */
double D_Uniform(int dimension, int create_or_re
    trieve, int index, int type_generator, int mc_or_q
    mc)
{
  int i;
  double random_number;
  /* ******************************************
    ********************** */
  /* Usage:                 */
  /*  If create_or_retrieve==CREATE (in optype.h
    )    */
  /*         -put dimension random numbers in th
    e array
         ArrayOfRandomNumbers.    */
  /*         -returns ArrayOfRandomNumbers[0]
     */
  /*  else (create_or_retrieve==RETRIEVE) (in optype.h
    )    */
```

```
  /*          -returns ArrayOfRandomNumbers[ind
    ex]    */
  /* ****************************************
    ****************** */
  if (create_or_retrieve == CREATE)
    {
      if(mc_or_qmc == MC)
  {
    for(i=0;i<dimension;i++)
      {
        Random[type_generator].Compute(0,&
    random_number);
        ArrayOfRandomNumbers[i]= random_number;
      }
  }
      else
  Random[type_generator].Compute(dimension,Array
    OfRandomNumbers);
      return  ArrayOfRandomNumbers[0];
    }
  else
    {
      return ArrayOfRandomNumbers[index];
    }
}


void Display_Generators(void)
{
  int i;
  char string[MAX_CHAR_X3];

  i=0;
  while (Random[i].Compute!=NULL)
  {
    strcpy(string,"%d:{t");
    strcat( string,Random[i].Name);
    strcat( string,"{n");
    Fprintf(TOSCREEN,string,i);
    i++;
  }
```

```
  Fprintf(TOSCREEN,"{n");
  return;
}

int InitMc(void)
{
  binomial(MAXI);

  return OK;

}
```

References