```c
/******************************************************************************
 * Nome   : Thiago Pinheiro de Macedo
 * N USP : 5124272
 ******************************************************************************/
/******************************************************************************
 * MAE0699 - Tópicos de Probabilidade e Estatística
 * Prof.: José Carlos Simon de Miranda
 * Exercicio de Implementação #01 (Sem Nome)
 * Desenvolvido utilizando Visual C++ 2005 SP1
 ******************************************************************************/

/* #define PROFILE_WINDOWS */

/* Utiliza "SIMD oriented Fast Mersenne Twister(SFMT)" para rng uniformes; */
#define USE_SFMT

#ifdef PROFILE_WINDOWS
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#ifdef USE_SFMT
#include "SFMT/SFMT.h"
#endif

#ifndef M_PI
const double M_PI  = 3.14159265358979323846264338328L;      /* pi */
#endif

#ifndef M_2PI
const double M_2PI = 6.28318530717958647692528676656L;      /* 2*pi */
#endif

#ifndef M_SQRT2
const double M_SQRT2 = 1.41421356237309504880168872421097L;   /* sqrt(2.0) */
#endif

/* Constantes para geracao de variaveis com distribuicao Gaussiana (Leva) */
const double GAUSSIAN_S    =  0.449871L;
const double GAUSSIAN_T    = -0.386595L;
const double GAUSSIAN_A    =  0.19600L;
const double GAUSSIAN_B    =  0.25472L;
const double GAUSSIAN_R1   =  0.27597L;
const double GAUSSIAN_R2   =  0.27846L;

const double PARABOLOIDE_R = (2.0 / 3.0);
const double PARABOLOIDE_C = 10.1803398874989484820458683436563656L; /* (5.0 * sqrt(5.0) - 1.0);

/* limites para geração dos ensaios */
unsigned int NumFunc = 1;
unsigned int NumPts  = 10000;
unsigned int NumCos  = 30;
unsigned int NumSen  = 30;

static double runif(void)
{
#ifdef USE_SFMT
  return genrand_res53();
#else
  return rand() / (double)RAND_MAX;
#endif
}

void inicRNG(void)
{
#ifdef USE_SFMT
  init_gen_rand(time(NULL));
#else
  srand((unsigned int)time(NULL));
#endif
}

static double geraVarA(const double X, const double Y)
```

```c
{
  return atan(Y * (2*X - 1));
}

static double geraBernoulli(const double p)
{
  double u = runif();
  return (u < p) ? 1 : 0;
}

static double geraCauchy(void)
{
  double u = runif();
  return tan(M_PI * (u - 0.5));
}

static double geraExp(void)
{
  double u = runif();
  return (-2.0 * log(u));
}

static double geraGeo(const double p)
{
  double u = runif();

  if (p == 1)
    return 1;
  return 1 + ceil(log(u) / (double)log(1 - p));
}

/* Ratio method (Kinderman-Monahan); see Knuth v2, 3rd ed, p130.
 * K+M, ACM Trans Math Software 3 (1977) 257-260.
 *
 * [Added by Charles Karney] This is an implementation of Leva's
 * modifications to the original K+M method; see:
 * J. L. Leva, ACM Trans Math Software 18 (1992) 449-453 and 454-455. */
static double geraGaussian(void)
{
  double u, v, x, y, Q;

  /* This loop is executed 1.369 times on average  */
  do {
      /* Generate a point P = (u, v) uniform in a rectangle enclosing
         the K+M region v^2 <= - 4 u^2 log(u). */
      /* u in (0, 1] to avoid singularity at u = 0 */
      u = 1.0 - runif();
      /* v is in the asymmetric interval [-0.5, 0.5).  However v = -0.5
         is rejected in the last part of the while clause.  The
         resulting normal deviate is strictly symmetric about 0
         (provided that v is symmetric once v = -0.5 is excluded). */
      v = runif() - 0.5;
      /* Constant 1.7156 > sqrt(8/e) (for accuracy); but not by too
         much (for efficiency). */
      v *= 1.7156;
      /* Compute Leva's quadratic form Q */
      x = u - GAUSSIAN_S;
      y = fabs(v) - GAUSSIAN_T;
      Q = x * x + y * (GAUSSIAN_A * y - GAUSSIAN_B * x);
      /* Accept P if Q < r1 (Leva) */
      /* Reject P if Q > r2 (Leva) */
      /* Accept if v^2 <= -4 u^2 log(u) (K+M) */
      /* This final test is executed 0.012 times on average. */
  } while (Q >= GAUSSIAN_R1 && (Q > GAUSSIAN_R2 || v * v > -4 * u * u * log (u)));
  /* Return slope */
  return (v / u);
}

static double geraRaioUnifParaboloide(void)
{
  double u = runif();
  double r = pow((1.0 + (u * PARABOLOIDE_C)), PARABOLOIDE_R);
  return(sqrt(r - 1.0) / 2.0);
}

static double geraVarY_Ex(double t)
```

```c
{
  /*
  Resolver EDO para obter Y
  y'' + L1 * y' + L2 * y = (E1 * t ^ 2) + (E2 * t) + E3 + Gamma(cos(t), sin(t), exp(t));
  y'' + L1 * y' + L2 * y = (E1 * t ^ 2) + (E2 * t) + E3 + GammaX * cos(t) + GammaY * sin(t)
  */

  double L1 = geraExp();
  double L2 = 2 + geraGeo(L1 / (1 + L1));
  double E1 = geraGaussian();
  double E2 = E1 * geraGaussian();
  double E3_mean = (E1*E1) + (E2*E2);
  double E3_sigma = sqrt(pow(fabs(E1), 5.0) + (3.0 * (E1*E1) * pow(E2, 4.0)) + pow(fabs(E2),
  double E3 = E3_mean + E3_sigma * geraGaussian();
  double theta = M_2PI * runif();
  double radius = geraRaioUnifParaboloide();
  double GammaX = radius * cos(theta);
  double GammaY = radius * sin(theta);
  double GammaZ = radius * radius;
  double delta = (L1*L1) - (4.0 * L2);
  double Yp = 0.0; /* Solucao Particular; */
  double Yc = 0.0; /* Solucao Complementar (caso homogeneo) */
  double r1, r2, c1, c2;
  double YpA, YpB, YpC;
  double A, B;
  double sqrtDelta;

  /*
  Condicoes Iniciais:
  y (0) = 1;
  y'(0) = 1;
  */

  if(delta > 0)
  {
    /*
    Yc  =       c1 * exp(r1 * t) +      c2 * exp(r2 * t);
    Yc' = r1 * c1 * exp(r1 * t) + r2 * c2 * exp(r2 * t);
    Yc (0) =      c1 +      c2 = 1 => c1 = (1 - c2);
    Yc'(0) = r1 * c1 + r2 * c2 = 1 => r1 * (1 - c2) + r2 * c2 = 1 => c2 = (1 - r1) / (r2 - r
    */
    sqrtDelta = sqrt(delta);
    r1 = (-L1 + sqrtDelta) / 2.0;
    r2 = (-L1 - sqrtDelta) / 2.0;
    c2 = (1 - r1) / (r2 - r1);
    c1 = (1 - c2);
    Yc = c1 * exp(r1 * t) + c2 * exp(r2 * t);
  }
  else if(delta < 0)
  {
    /*
    Yc  = exp(A * t) * (c1 * cos(B * t) + c2 * sin(B * t));
    Y'c = exp(A * t) * ((A * c1 + B * c2) * cos(B * t) + (A * c2 - B * c1) * sin(B * t));
    Yc (0) = c1          = 1;
    Y'c(0) = A * c1 + B * c2 = 1 => c2 = (1 - A) / B;
    Y'c    = A * c1 + B * c2 = 1;
    A = -b / 2 * a = -b / 2;
    B = sqrt(4 * L2 - L1^2) / 2 * a;
    */
    A = -(L1 / 2.0);
    B = sqrt(fabs(delta)) / 2.0;
    c2 = (1 - A) / B;
    Yc = exp(A * t) * (cos(B * t) + c2 * sin(B * t));
  }
  else
  {
    /*
    Yc  =      c1 * exp(r * t) +     c2 * t * exp(r * t);
    Yc' =  r * c1 * exp(r * t) + r * c2 * t * exp(r * t);
    Yc (0) =     c1 + c2  = 1 => c2 = (1 - c1);
    Yc'(0) = r * c1       = 1 => c1 = (1 / r);
    */
    r1 = (-L1 / 2.0);
    c1 = (1 / r1);
    c2 = (1 - c1);
    Yc = (c1 + c2 * t) * exp(r1 * t);
```

```
  }

  /* Solucao particular do caso nao homogeneo; */
  YpA = (E1 / L2);
  YpB = (E2 - 2 * L1 * YpA) / L2;
  YpC = ((GammaX * cos(t) + GammaY * sin(t) + GammaZ * exp(t)) - 2 * YpA - L1 * YpB + E3) /
  Yp = YpA * (t*t) + YpB * t + YpC;
  return(Yp + Yc);
}

static double geraVarY(double t)
{
  double Y;
  do {
    /* previde a utilizaçao de valores nao numericos (NaN); */
    Y = geraVarY_Ex(t);
  } while(Y != Y);
  return Y;
}

static double geraFuncao(double* pValores)
{
  register unsigned int nIdxP, nIdxS, nIdxC;
  const double  stepSeq = (1 / (double)NumPts);
  double* pVal;
  double base2;
  double X, Y, a, u, x, fMax;

  pVal = pValores;
  nIdxP = 0;
  x = 0.0;
  fMax = 0.0;
  while(nIdxP < NumPts)
  {
    Y = geraVarY(geraCauchy());
    X = geraBernoulli(0.5);
    *pVal = geraVarA(X, Y);
    base2 = 2;
    for(nIdxC = 0; (nIdxC < NumCos); nIdxC++)
    {
      Y = geraVarY(geraCauchy());
      X = geraBernoulli(0.5);
      a = geraVarA(X, Y);
      u = runif();
      *pVal += (a * cos(M_2PI * nIdxC * (x + u)) / base2);
      base2 *= 2;
    }
    for(nIdxS = 0; (nIdxS < NumSen); nIdxS++)
    {
      Y = geraVarY(geraCauchy());
      X = geraBernoulli(0.5);
      a = geraVarA(X, Y);
      u = runif();
      *pVal += (a * sin(M_2PI * nIdxS * (x*x + u)) / pow(2.0, (nIdxS + fabs(a))) );
    }
    if(fMax < *pVal) fMax = *pVal;
    nIdxP++;
    pVal++;
    x += stepSeq;
  }
  return fMax;
}

void DumpArray2File(char* szArquivo, double* pValores, unsigned int nSize)
{
  FILE* fpArq = NULL;
  double* pVal;
  unsigned int i;

  fpArq = fopen(szArquivo, "w+b");
  if(fpArq != NULL)
  {
    i = 0;
    pVal = pValores;
    while(i++ < nSize)
    {
```

```
      fprintf(fpArq, "%.16f\n", *pVal);
      pVal++;
    }
    fclose(fpArq);
  }
}

int main(int argc, char* argv[])
{
  unsigned int i = 0;
  double* pValores;
  double* pValMax;
  double* pMaxAtu;
  char    szArquivo[255];
  time_t  timeAtu;
#ifdef PROFILE_WINDOWS
  DWORD   dwTime;
#endif


  for(i = 1; i < argc; i++)
  {
    if(*argv[i] == '-')
    {
      switch(toupper(*(argv[i]+1)))
      {
        case 'F': NumFunc = (unsigned int)atol(argv[i]+2); break;
        case 'P': NumPts  = (unsigned int)atol(argv[i]+2); break;
        case 'C': NumCos  = (unsigned int)atol(argv[i]+2); break;
        case 'S': NumSen  = (unsigned int)atol(argv[i]+2); break;
      }
    }
  }

  printf("NumFunc = %d; NumPts  = %d; NumCos  = %d; NumSen  = %d;\n",
         NumFunc,NumPts, NumCos, NumSen);

  inicRNG();

  pValMax = calloc(NumFunc, sizeof(double));
  pValores = calloc(NumPts, sizeof(double));

  pMaxAtu = pValMax;
  for(i = 0; i < NumFunc; i++)
  {
#ifdef PROFILE_WINDOWS
    dwTime = GetTickCount();
    *pMaxAtu = geraFuncao(pValores);
    dwTime = GetTickCount() - dwTime;
    printf("F(%d);\tTime(%ld);\n", i, dwTime);
#else
    *pMaxAtu = geraFuncao(pValores);
    if(i%10==0) printf("F(%d)\n", i);
#endif
    pMaxAtu++;
  }

  timeAtu = time(NULL);

  sprintf(szArquivo, "F%dP%dPTS_%ld.TXT", NumFunc, NumPts, timeAtu);
  DumpArray2File(szArquivo, pValores, NumPts);

  sprintf(szArquivo, "F%dP%dMAX_%ld.TXT", NumFunc, NumPts, timeAtu);
  DumpArray2File(szArquivo, pValMax, NumFunc);

  free(pValMax);
  free(pValores);

  return 0;
}
```