Thiago Pinheiro de Macedo - Nº USP: 5124272
MAE0699 - Tópicos de Probabilidade e Estatística
Prof.: José Carlos Simon de Miranda

Exercício de Implementação #01

Resumo

Para uma função y = F(x) desconhecida, deveria-se estimar um valor H, tal que, para valores de x real no intervalo [0, 1] da reta, com 90% de probabilidade, todos os valores de y = F(x) estariam abaixo de y = H, ou ainda, o valor máximo de F(x) no intervalo [0, 1] da reta real deve ser menor ou igual a tal H, com 90% de probabilidade.

Como F(x) não é conhecida, realizamos diversos ensaios para calcular aproximações de F e tentar realizar alguma inferência quanto aos máximos. Métodos como o estudo da derivada foram sugeridos, mas não foram implementados por imperícia do implementador, o que nos forçou a realizar muito mais ensaios a fim de inferir heuristicamente algo conclusivo acerca do problema.

Problema

Seja :

$$x \in [0,1]$$
$$F : \mathbb{R} \to \mathbb{R}$$
$$x \to y = F(x)$$
$$F(x) = a_0 + \sum_{k=1}^{\infty} \frac{a_k}{2^k} \cos(2k\pi(x+u_k)) + \sum_{k=1}^{\infty} \frac{b_k}{2^{k+|b_k|}} \sin(2k\pi(x^2+v_k))$$

Onde tem-se que:

$$\{u_k, v_w : k \geq 0, w \geq 1\} \quad iid$$
$$u_k \sim Unif(0,1) \sim v_k$$
$$\{a_i, b_j : i \geq 0, j \geq 1\} \quad iid$$
$$a_i \sim A \sim b_j$$
$$A = \arctan(Y(2X-1))$$
$$X \sim Ber(\frac{1}{2})$$
$$Y = y(Z)Z \sim Cauchy(1)$$

Com ***y(t)*** solução da E.D.O:

$$y'' + L_1 y' + L_2 y = E_1 t^2 + E_2 t + E_3 + \gamma(\cos(t), \sin(t), \exp(t)) \quad y(0) = 1, y'(0) = 1$$

Onde :

$$L_1 \sim \exp(\frac{1}{2})$$
$$(L_2 | L1 = l_1) \sim 2 + Geo(\frac{l_1}{1+l_1})$$
$$E_1 \sim N(0,1)$$
$$(E_2 | E_1 = e_1) \sim N(0, {e_1}^2)$$
$$(E_3 | E_1 = e_1, E_2 = e_2) \sim N({e_1}^2 + {e_2}^2, |e_1|^5 + 3{e_1}^2{e_2}^4 + |e_2|^{\sqrt{2}})$$
$$\gamma = (\gamma_1, \gamma_2, \gamma_3) \in \mathbb{R}$$

Sendo GAMMA um vetor distribuído uniformemente sob a superfície:

$$0 \leq \gamma_3 = {\gamma_1}^2 + {\gamma_2}^2 \leq 1$$

Deve-se determinar o menor valor possível para $H \in \mathbb{R}$ tal que

$$P[max(F(x)) \leq H] \geq 0.9$$

Materiais e Métodos

Como a função F não era conhecida, foram necessárias simulações computacionais para calcularmos diversas aproximações da mesma. No processo, utilizamos as seguintes ferramentas para realizar os espectivas taferas:

- Microsoft Visual C++ 2005 SP1
  - Desenvolvimento do programa de simulação ***mae0699_ep01.c***
  - Simulação das variáveis aleatórias.
  - Exportação dos resultados das simulações para arquivos em disco.
- The R Project for Statistical Computing
  - Importamos os resultados gerados pelo programa *mae0699_ep01.c*, e utilizamos as ferramentas estatísticas do *software* R para analisarmos tais resultados.

Para a execução do programa de simulação e das ferramentas de desenvolvimento e análise estatística, forma utilizados computadores pessoais do tipo IBM-PC dotados de processadores Intel Pentium 4 HT 3.0 Ghz, Intel Pentium D 3.2 Ghz e AMD Althlon 64 2800+ 1.8Ghz, e sistema Operacional Microsoft Windows XP Professional. Como não havia disponível a versão 64 bits do sistema operacional, otimizações e compilações específicas para tal caso não foram utilizadas, de modo que em todos os casos os sistemas foram executados em ambiente 32 bits.

Tivemos também a oportunidade de executar o programa de simulação *mae0699_ep01.c* e a ferramenta R de computação estatística em ambiente Ubuntu Linux 7.04, afim de compatibilizar sua execução em ambiente Linux.

Um fator o qual gostaríamos de enfatizar é que: como todas as variáveis envolvidas nas simulações foram geradas a partir de uma – ou mais -- variável com distribuição uniforme no intervalo real [0, 1], optou-se por não utilizar os métodos disponíveis para tal na *libc* padrão, pois o mesmo é inconsistente em ambiente Windows e não cria elementos com a qualidade desejada para as simulações (tal comentário foge do escopo do problema e deve não será abordado a fundo). A solução então foi utilizar o *SIMD-oriented Fast Mersenne Twister (SFMT)* criado por *Mutsuo Saito* e *Makoto Matsumoto*. Tal algoritmo tem plena aceitação no meio acadêmico científico (tais conceitos e sua qualidade muito superior ao método padrão também fogem do escopo deste documento, e devem ser tratadas separadamente) e foi facilmente incorporado ao programa de simulação

Dada a implementação dos métodos, outro ponto importante era a performance e a qualidade dos valores gerados com distribuição normal a serem calculados. O método padrão para simulação de valores com tal distribuição é conhecido como *Transformação Box-Müller*. Tal transformação, aplicada utilizando coordenadas polares, carece da utilização de funções trigonométricas e logarítmicas, o que o torna computacionalmente deselegante. Optou-se, então por utilizar um algoritmo de aceitação-Rejeição de nome *Algorítimo Ziggurat* o qual pode gerar números pseudo-aleatórios de distribuição normal com performance comparada a geração de números aleatórios com distribuição uniforme.

Descrição Experimental

De posse do programa de simulação compilado, realizamos diversos ensaios para estimar funções que aproximem a função F original. Consideramos tais funções da forma :

$$x \in \mathbb{Q}\,,\ 0 \le x \le 1$$

$$F_n : \mathbb{Q} \to \mathbb{Q}$$

$$F_n(x) \;=\; a_0 + \sum_{k=1}^{NC} \frac{a_k}{2^k} \cos\left(2\mathrm{k}\,\pi\,(x + u_k)\right) + \sum_{k=1}^{NS} \frac{b_k}{2^{k+|b_k|}} \sin\left(2\mathrm{k}\,\pi\,(x^2 + v_k)\right)$$

Considerando :

NP : número de pontos em [0,1];

- NF : número de funções F estimadas;
- NC : número de co-senos em F;
- NS : número de senos em F;

Como o meio computacional não é passível de representação de números reais exatos, consideramos que estamos trabalhando no conjunto dos números racionais. Heuristicamente, assumimos NS e NC iguais a 30, experimentos paralelos mostraram que não muito distante de tais valores, o cálculo dos valores utilizados como parâmetros nas funções trigonométricas envolvidas não contribuía com a convergência da aproximação.

De fato, como os equipamentos utilizados foram compatíveis com a dimensão e complexidade dos cálculos envolvidos, não houveram surpresas quanto aos tempos de execução dos ensaios. Pudemos utilizar diversas combinações de ensaios, dos quais separamos três casos, de onde extraímos e anexamos abaixo alguns gráficos gerados pelo R com base nos resultados do programa de simulação para cada um dos respectivos casos.

Caso 01: Uma função de aproximação com 10.000 pontos calculados;

Gráfico 1.1: Pontos da função de aproximação

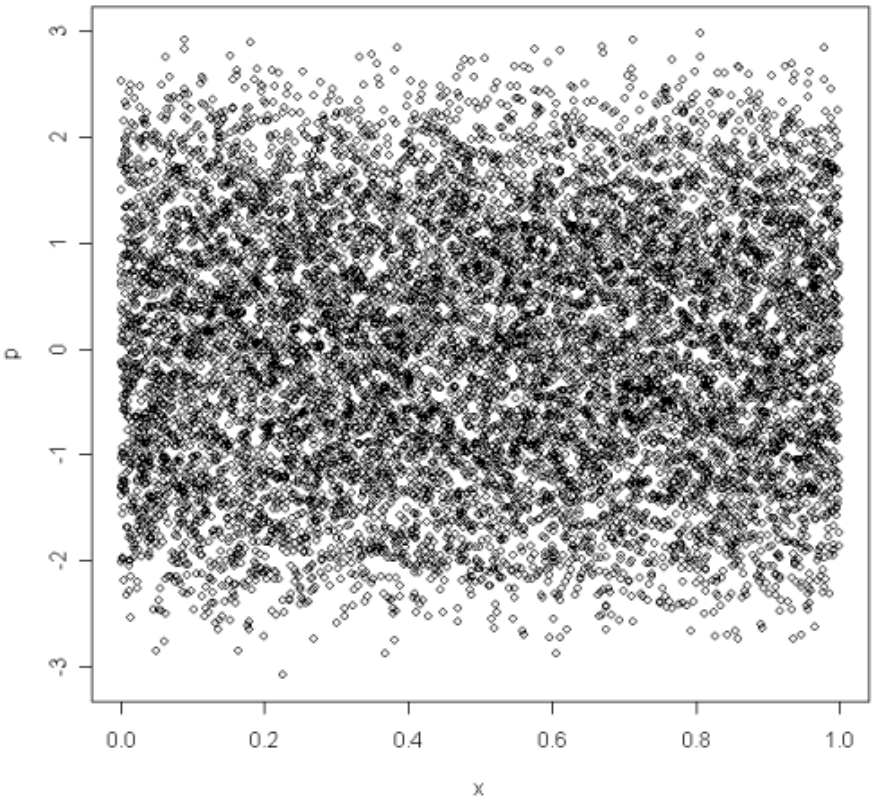
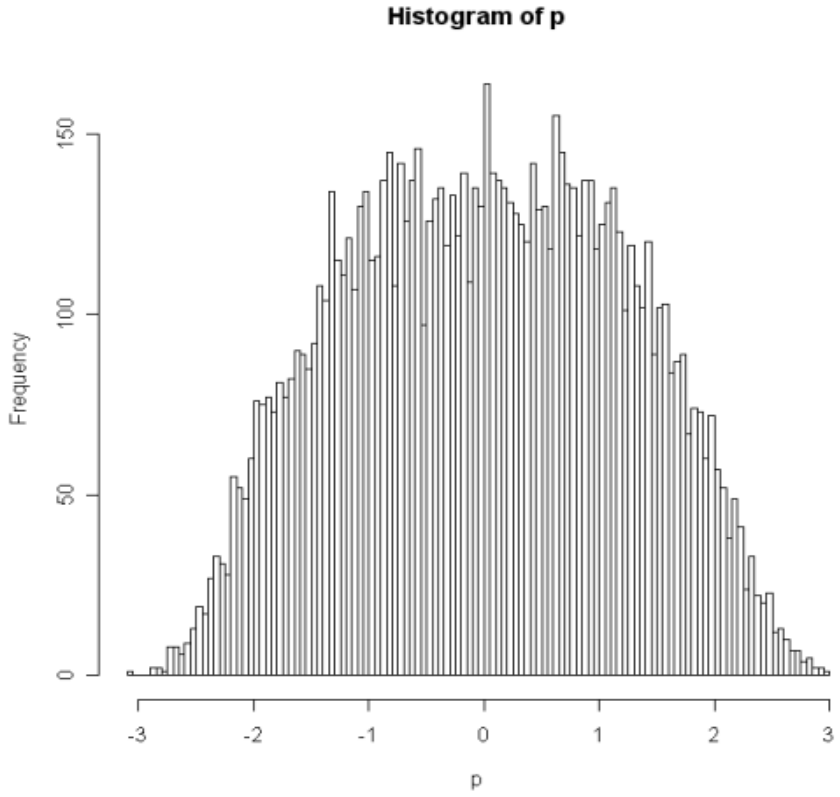


Gráfico 1.2: Histograma dos pontos da função de aproximação



Caso 02: Uma função de aproximação com 100.000 pontos calculados;

Gráfico 2.1: Pontos da função de aproximação

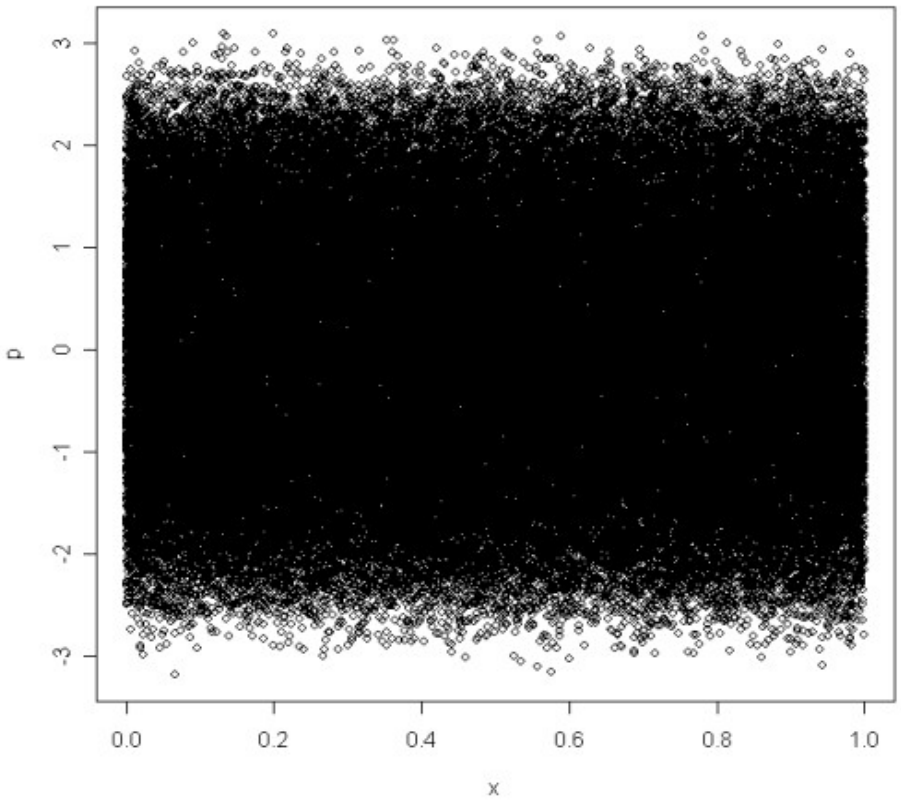
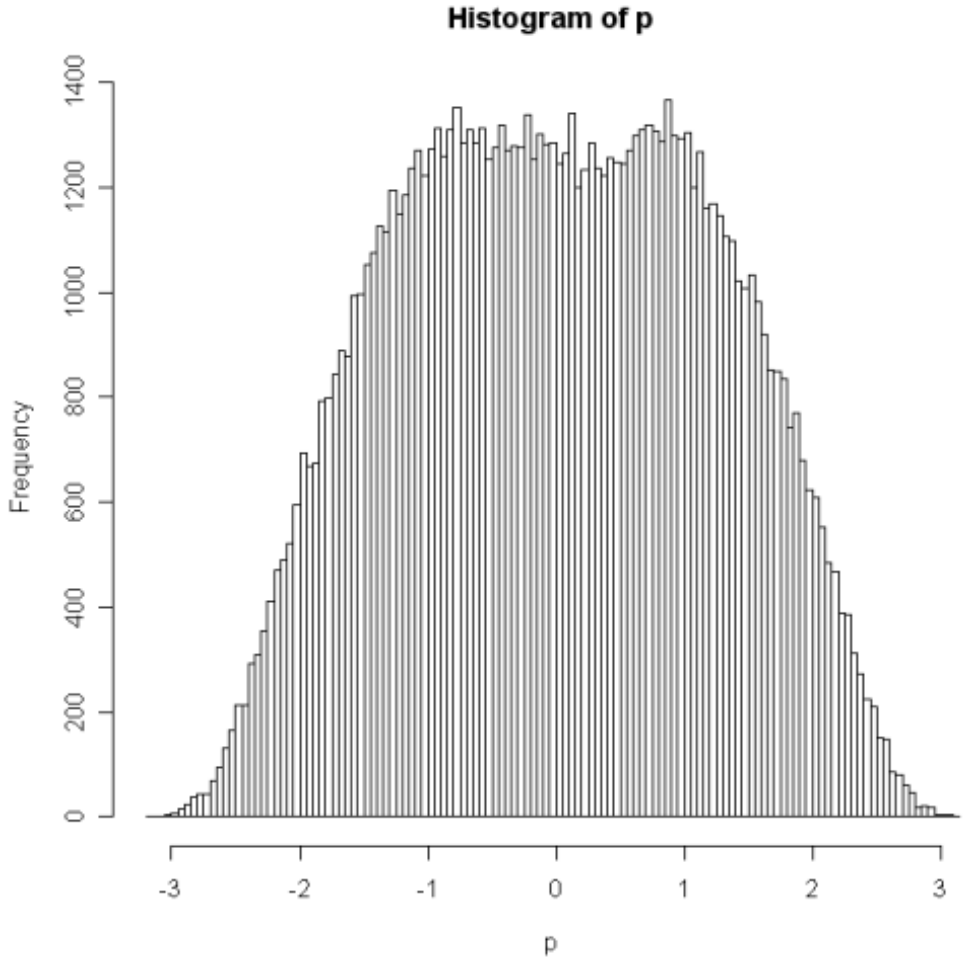


Gráfico 2.2: Histograma dos pontos da função de aproximação



Caso 03: Uma função de aproximação com 1.000.000 pontos calculados;

Gráfico 3.1: Pontos da função de aproximação

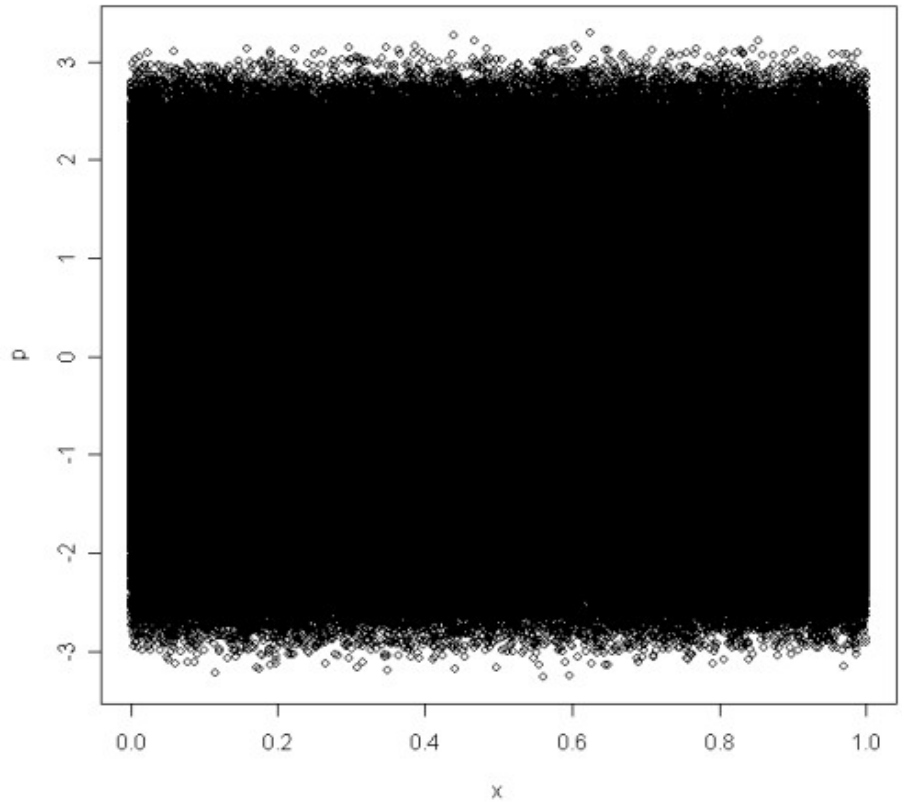
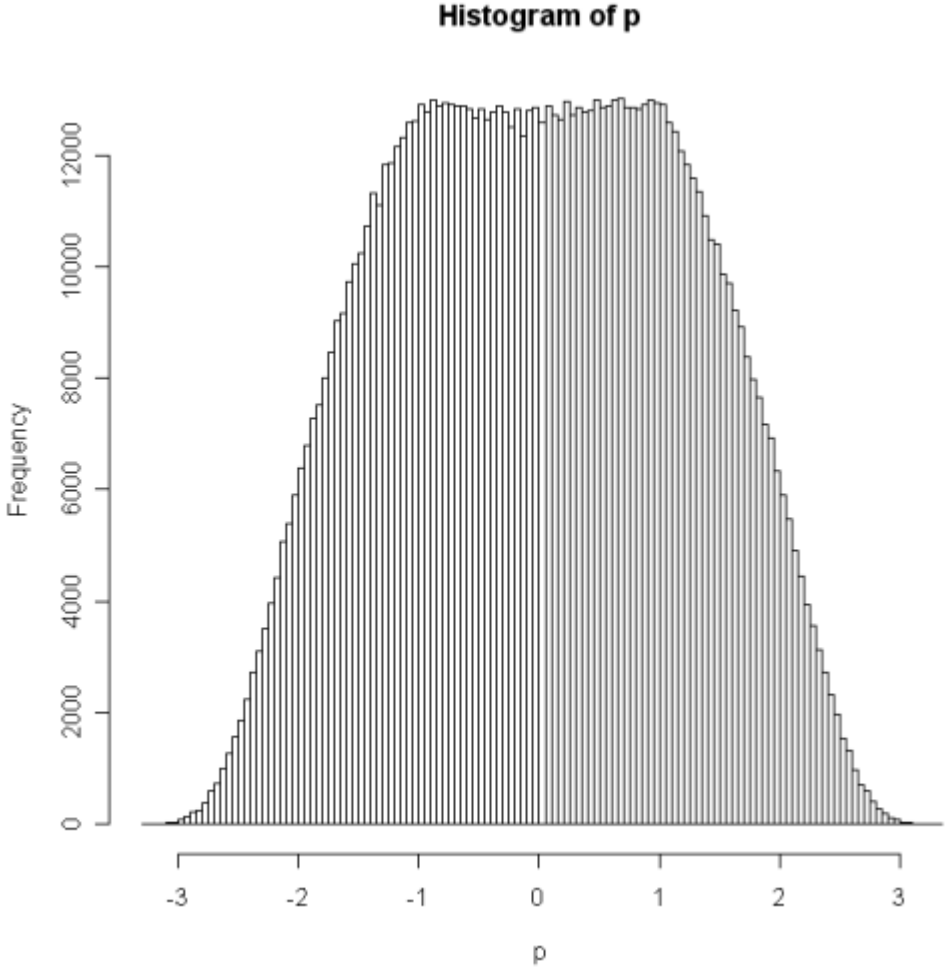


Gráfico 3.2: Histograma dos pontos da função de aproximação



Realizamos então diversos ensaios, simulando muitas funções com o mesmo número de pontos. Para cada função calculada, armazenávamos o maior valor que a mesma assumia, dentre os calculados. Com base nestes máximos, criamos uma lista de máximos estimados. De tais tipos de ensaios, extraímos um, do qual seguem abaixo alguns gráficos:

Caso 04: Estudo dos máximos de 10.000 funções de aproximação, com 10.000 pontos calculados;

Gráfico 4.1: Pontos de máximo (sem ordenação)

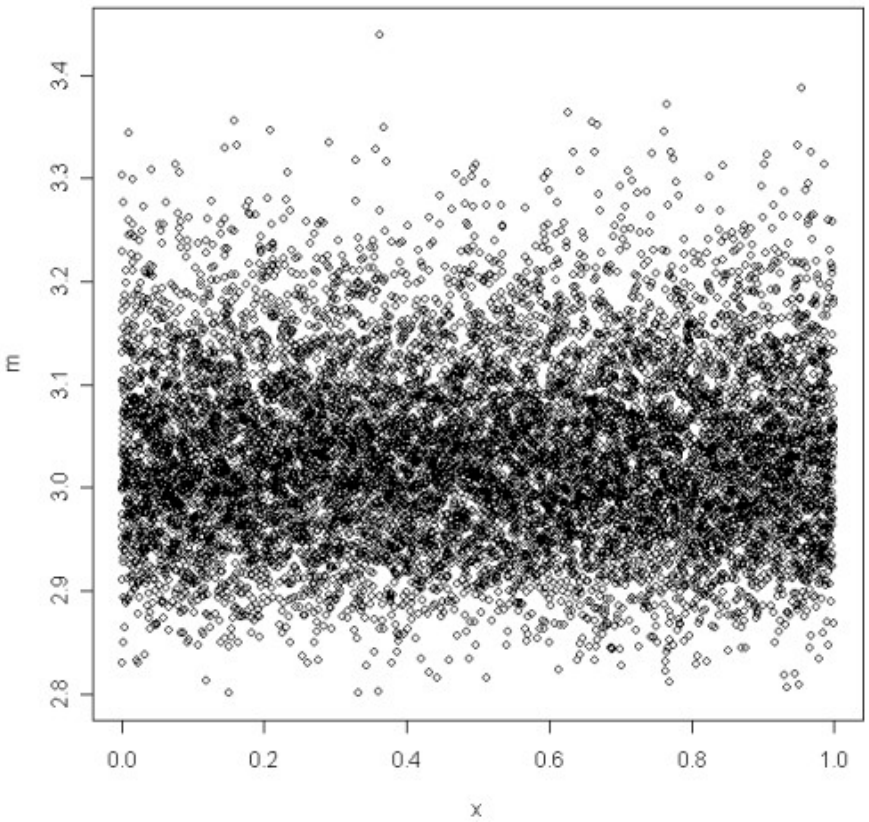


Gráfico 4.2: Pontos de máximo ordenados

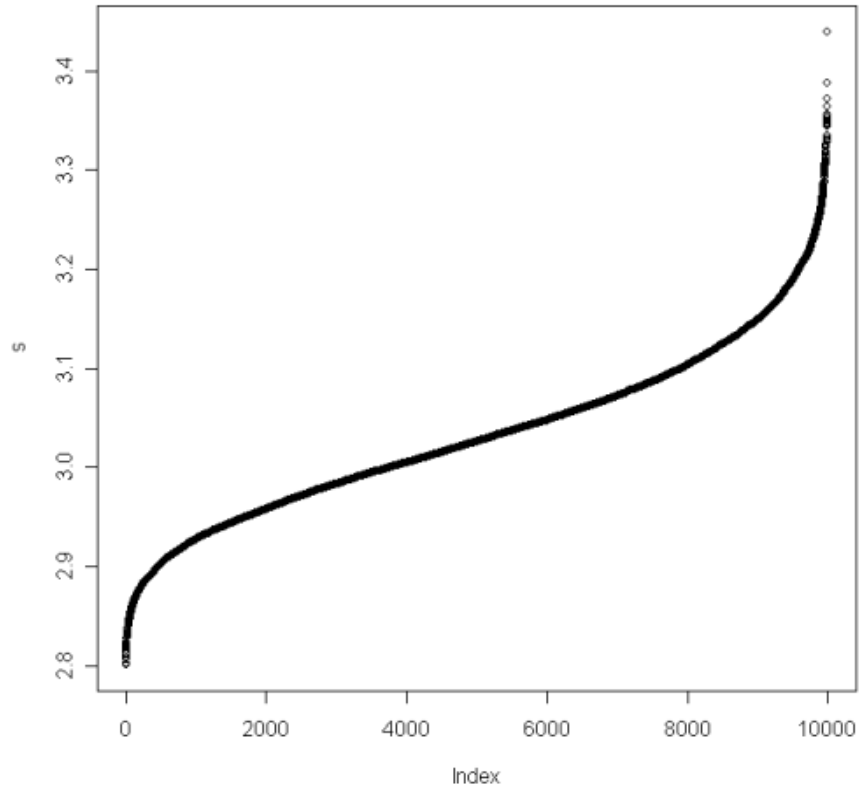


Gráfico 4.3: Histograma dos pontos de máximo

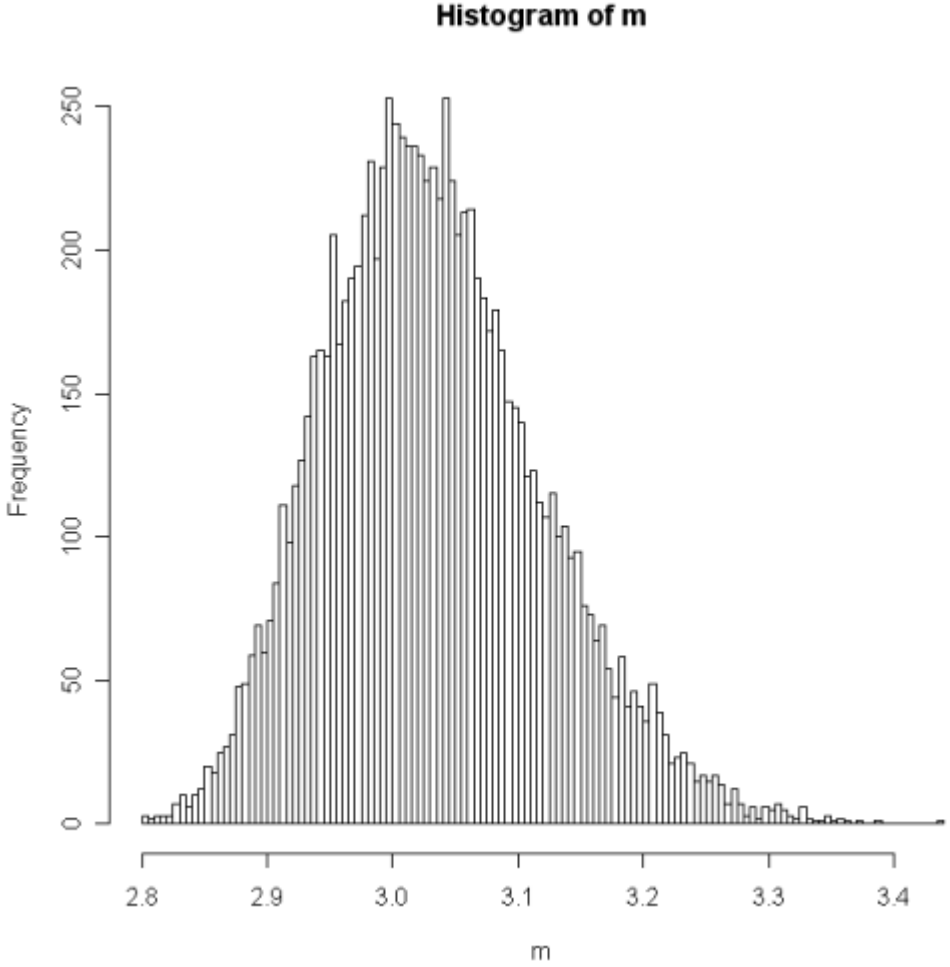
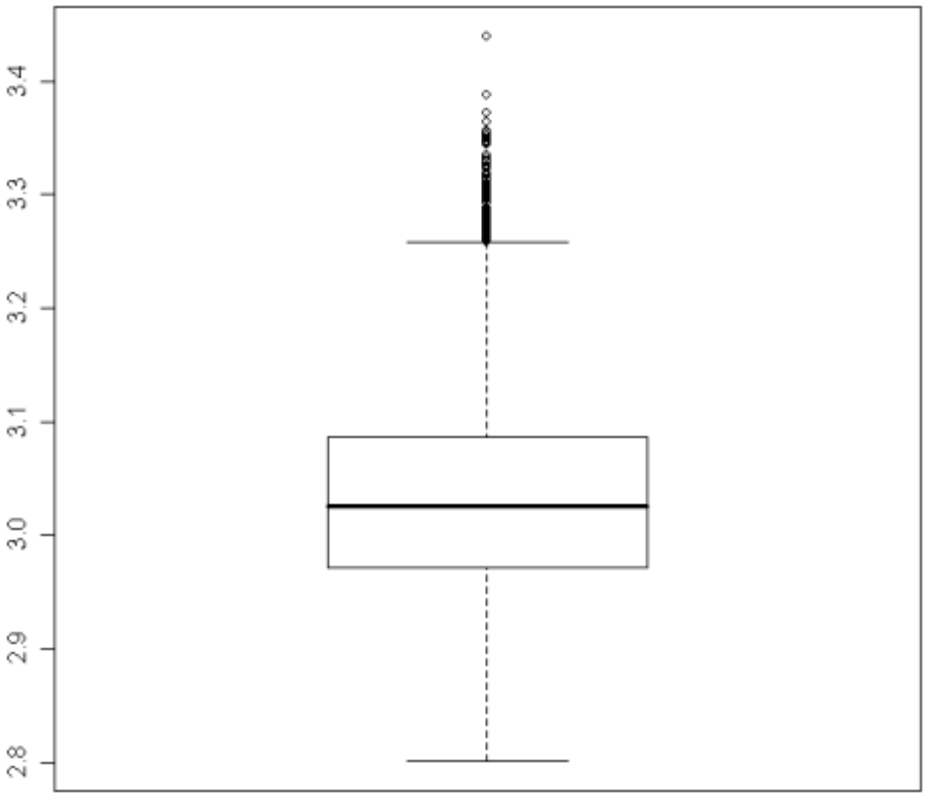


Gráfico 4.4: Box-Plot dos pontos de máximo



Resultados

Com base nos dados obtidos em simulações, utilizamos a função *summary*() do R para obter as estatísticas dos máximos do ***Caso 04,*** e montamos a tabela abaixo:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2.800	2.971	3.026	3.033	3.086	3.440

Vê-se que a média e mediana da amostra estão muito próximas a 3.0

Utilizamos também a função *quantile(m, 0.9),* a qual nos retorna o valor que delimita a amostra ordenada em 90%, e obtivemos o valor de **3.148692,** valor este mais próximo à media do que do máximo (tal situação também é visível no *Box-Plot* do ***Caso 04***) o que nos leva a crer que, se executarmos ainda mais repetições do experimento, com o mesmo número de pontos (10.000), as medidas devem se aproximar ainda mais, pois pontos significativos estarão, com maior probabilidade, próximos à media e a mediana da amostra.

Conclusão

Com isso – apesar de utilizamos métodos de força bruta heurística, e não métodos mais sofisticados como a análise das variações dos pontos calculados – podemos estimar **H** como próximo a **3.0**, ainda que formalmente não possamos garantir o resultado com 90% de probabilidade. Usamos, sim, que a convergência dos máximos para as medidas de posição central deve ocorrer quando o número de amostras de máximos (isto é, as funções calculadas e seus respectivos pontos).

Apêndice A

Links para maiores informações:

- Microsoft Visual C++ 2005 SP1
  http://msdn.microsoft.com/visualc/
- The R Project for Statistical Computing
  http://www.r-project.org/
- SIMD-oriented Fast Mersenne Twister (SFMT)
  http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/
- Transformação Box-Müller
  http://en.wikipedia.org/wiki/Box_muller
- Algorítimo Ziggurat
  http://en.wikipedia.org/wiki/Ziggurat_algorithm

Apêndice B

Código fonte

```
/*****************************************************************************
 * 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 = 1.41421356237309504880168872420 97L;   /* 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 = 10.180339887498948482045868343656L; /* (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)
{
  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)
{
  /*
  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) +
GammaZ * exp(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),
M_SQRT2));
  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 - r1);
    */
    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) / L2;
  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;
}
```