MAURÍCIO ARAÚJO BARROS

Cálculo da Função Densidade de Probabilidade em Dados Contínuos em Paralelo

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos, da Universidade de São Paulo

Curso de Engenharia de Computação

ORIENTADOR: Prof. Dr. Carlos Dias Maciel

São Carlos

2012

Barros, Maurício Araújo
B277c Cálcula da função densidade de probabilidade em dados contínuos em paralelo / Maurício Araújo Barros; orientador Carlos Dias Maciel. São Carlos, 2012.

Monografia (Graduação em Engenharia de Computação) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2012.

1. histograma. 2. Python. 3. Função densidade de Probabilidade. 4. IPython. 5. paralelo. 6. numpy. I. Título.

FOLHA DE APROVAÇÃO

Nome: Maurício Araújo Barros

Título: "Cálculo da Função Densidade de Probabilidade em Dados Contínuos em Paralelo"

Trabalho de Conclusão de Curso defendido em 23 / 11 / 2012.

Comissão Julgadora:	**Resultado:**
M.Sc. Jen John Lee SEL/EESC/USP	Aprovado
M.Sc. Wagner Endo SEL/EESC/USP	Aprovado

Orientador:
Prof. Associado Carlos Dias Maciel - SEL/EESC/USP

Coordenador pela EESC/USP do Curso de Engenharia de Computação:

Prof. Associado Evandro Luís Linhari Rodrigues

RESUMO

Dado uma grande amostra de dados, construir um histograma representando a função densidade de probabilidade (FDP) que permita extrair o máximo de informação possível não é uma tarefa trivial. Fazê-lo de forma empírica pode ser muito trabalhoso, principalmente para mais de um experimento. É interessante, então, utilizar um algoritmo já pronto que calcula o histograma com uma quantidade de intervalos ótima.

Para o caso de dados que obedecem a distribuição de Poisson, o algoritmo “A method for bin size selection” (SHIMAZAKI; SHINOMOTO, 2007) já permite fazer esse cálculo automático. O resultado desse algoritmo é a seleção de um intervalo de classe ótimo, mas a implementação deste de maneira eficiente é crucial para processamento cientifico que utiliza enormes quantidades de dados.

A linguagem Python, utilizada em conjunto com as bibliotecas Numpy e Scipy, possui um bom desempenho. A utilização da ferramenta para paralelização IPython melhorou ainda mais a performance do processamento do algoritmo. Foram feitos testes em serial e em paralelo. A execução em paralelo apresentou um ganho em torno de 50% sobre a execução serial.

Concluído os testes de funcionalidade e desempenho, o programa foi utilizado para estimar a FDP de dados experimentais e como resultado foi obtido uma FDP próxima de uma gaussiana, como esperado.

A disponibilização deste programa mostra-se uma promissora forma de facilitar a construção de histogramas nos mais diversos campos de pesquisa devido à linguagem Python ser uma ferramenta padrão dos sistemas derivados Unix e a ferramenta IPython estar disponível, gratuitamente, para instalação nestas.

Palavras-chave: histograma, função densidade de probabilidade, Python, Numpy, IPython, paralelo.

Abstract

Given an amount of data, generate a histogram representing the probability density function (PDF) that allows extract as much information as possible is not an easy task. Doing it empirically may be very painful, especially if there is more than one experimentation. So, it's interesting to use an already made algorithm that calculates the histogram with an optimal number of bins.

For the case of data that obeys the Poisson distribution, the algorithm "A method for bin size selection" (SHIMAZAKI; SHINOMOTO, 2007) already allows this automatic calculation. The result of this algorithm is the selection of a range of great class, but implementing this efficiently is crucial for scientific processing which uses huge amounts of data.

The Python language, used in conjunction with libraries Numpy and Scipy, has a good performance. The use of the IPython tool for parallelization further improved the performance of the processing algorithm. Tests were made in serial and in parallel. Running in parallel showed a gain around 50% over the serial execution.

Completed the tests of functionality and performance, the program was used to estimate the PDF of experimental data and was obtained a PDF next to a Gaussian distribution as expected.

The deployment of this software shows a promising way to ease the construction of histograms in various fields of research due to the fact that the Python language is a standard for Unix-derived systems and the IPython tool is available, free of charge, to install.

Keywords: histogram, probability density function, Python, Numpy, IPython, parallel.

Sumário

RESUMO 5

Sumário 7

1 Introdução e Objetivos 9

1.1 Introdução 9

1.2 Objetivos 10

2 Teoria 11

2.1 Histogramas e a Função Densidade de Probabilidade 11

2.2 Calculo do intervalo de classe do histograma 13

2.3 Algoritmo para cálculo do intervalo ótimo 17

2.4 Python e computação científica 18

2.5 Computação paralela com IPython 19

3 Configuração e Implementação 20

3.1 Configurando o ambiente computacional 20

3.2 Implementação 23

3.2.1 Implementação serial 25

3.2.2 Implementação paralelo 28

3.3 Geração de Dados de testes 30

4 Metodologia 31

4.1 Execução serial $\boldsymbol{e(t) = (t + 1)^8}$: 33

4.2 Execução serial $\boldsymbol{g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)}$ 36

4.3 Execução paralela $\boldsymbol{e(t) = (t + 1)^8}$: 39

4.4 Execução paralela $\boldsymbol{g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)}$: 42

4.5 Estimativa da FDP em dados obtidos experimentalmente 45

4.6 Execução para amostra A 47

4.7 Execução para amostra B 49
4.8 Execução para amostra C 51
5 Análise de Resultados 55
5.1 Validação teórica do Algoritmo 55
5.2 Eficiência do Algoritmo 58
5.3 Resultados com dados empíricos 59
6 Conclusão 60
7 Referências Bibliográficas 61

1 Introdução e Objetivos

1.1 Introdução

Nos diversos campos de pesquisa a análise de quantidades cada vez maiores de dados é, muitas vezes, algo crucial para chegar a uma conclusão assertiva de um experimento. Agrupar dados e construir histogramas que os representem corretamente pode ser algo bem complexo em experimentos que geram milhões de dados, tornando-se uma tarefa que consume um precioso tempo da pesquisa.

Essa construção do histograma precisa ser cuidadosa, pois um intervalo de classe muito grande não é capaz de representar a taxa básica do sinal corretamente e um intervalo de classe muito pequeno faz o histograma flutuar muito, tornando muito difícil discernir a taxa básica do sinal.

Shimazaki e Shinomoto (2007) desenvolveram um algoritmo capaz de calcular computacionalmente o valor ótimo do intervalo de classe de um histograma gerado a partir de um conjunto de dados cuja distribuição seja próxima à distribuição de Poisson. Isso foi possível devido à definição de uma função Custo que minimiza o erro quadrático integrado médio de um sinal.

A implementação deste algoritmo precisa de desempenho, pois as quantidades de dados a serem analisados podem ser muito grandes. Linguagens compiladas como C/C++, são as melhores neste quesito, mas o desenvolvimento e os testes dos códigos feitos nessas linguagens é mais trabalhoso. Contudo, testes realizados e apresentados em: http://www.scipy.org/PerformancePython mostram que linguagem interpretada Python, com as bibliotecas Numpy e Scipy, tem desempenho muito bom comparado a diversas outras linguagens. Sendo gratuita e possuindo portabilidade nas plataformas Unix, unido à sua facilidade de paralelismo através da ferramenta IPython, esta foi a linguagem escolhida para realizar este trabalho.

1.2 Objetivos

Com este trabalho, busca-se agilizar pesquisas que envolvem geração de histogramas a partir de grandes volumes de dados com a entrega da implementação do algoritmo de Shimazaki e Shinomoto (2007) em Python em paralelo. Ir-se-á através deste documento comprovar a eficiência da implementação e demonstrar ao leitor como utilizar o programa. Será explicado, ao longo do trabalho, os seguintes pontos:

- A teoria da qual é derivado o algoritmo computacional
- A implementação detalhada do algoritmo: uma breve explicação sobre o sistemas utilizado, a linguagem Python, a ferramenta IPython e a explicação da codificação.
- Testes realizados para garantir a funcionalidade da implementação
- Aplicação para dados de sinais neurais de gafanhotos
- Discussão dos resultados e conclusão

2 Teoria

2.1 Histogramas e a Função Densidade de Probabilidade

Uma das formas mais antigas de se estimar a função densidade de probabilidade é utilizar histogramas, apesar de ser vastamente conhecido, é importante realizar uma breve revisão do conceito de histograma e fazer algumas definições.

Um histograma é uma representação gráfica de um conjunto de dados que pode ser utilizado para estimar a FDP de funções continuas (SILVERMAN, 1986). Em sua forma mais tradicional, um histograma é um gráfico em duas dimensões (x, y) composto por retângulos justapostos, cujas bases (de tamanhos fixos) definem o intervalo de classe e cuja área é proporcional à freqüência dos dados que se encontram num intervalo de classe do retângulo. A Figura 1 mostra um histograma tradicional.

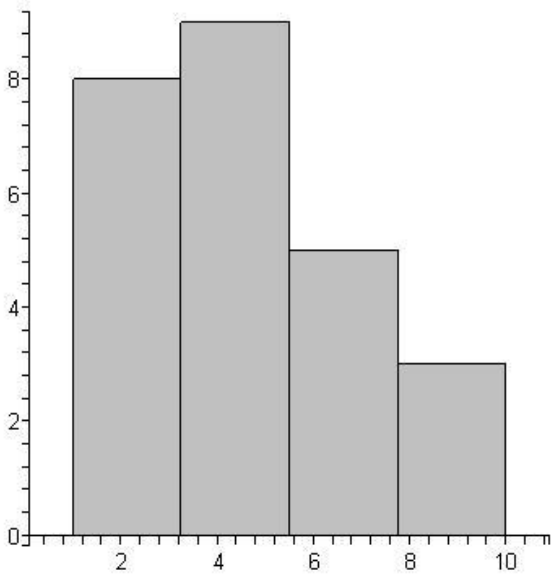

Figura 1 – Histograma tradicional

Os intervalos de classe são definidos como:

$$[x_0 + mh, x_0 + (m+1)h] \qquad (2.1)$$

Onde m é um inteiro positivo ou negativo, x_0 é a origem e h é a largura do intervalo de classe.

A função que descreve um histograma pode ser agora trivialmente definida como:

$$f(x) = X_i \tag{2.2}$$

Onde X_i é a quantidade de dados que se encontram no mesmo intervalo de classe que x.

Contudo, para representar a função densidade de probabilidade é necessário que:

$$\int_{-\infty}^{+\infty} f(x)\, dx = 1 \tag{2.3}$$

Isso é satisfeito dividindo a (Eq 2.2) por nh, sendo n a quantidade total de dados do histograma. Assim um a função que define um histograma tradicional que descreve uma função densidade de probabilidade é dada por:

$$\hat{f}(x) = \frac{X_i}{nh} \tag{2.4}$$

Essa será o histograma que representa a FDP que este trabalho irá calcular.

2.2 Calculo do intervalo de classe do histograma

A parte mais difícil da construção de um histograma que represente corretamente a FDP de um conjunto de dados é achar a melhor largura do intervalo de classe. A escolha errada deste pode levar a funções que não condizem com a realidade. A Figura 2 mostra a comparação entre histogramas construídos a partir de uma mesma base de dados, porém, com diferentes larguras de intervalos.

A estimativa empírica além de, muitas vezes, não levar a uma largura de intervalo ótimo, pode ser extremamente exaustivo para quantidades muito grandes de dados.

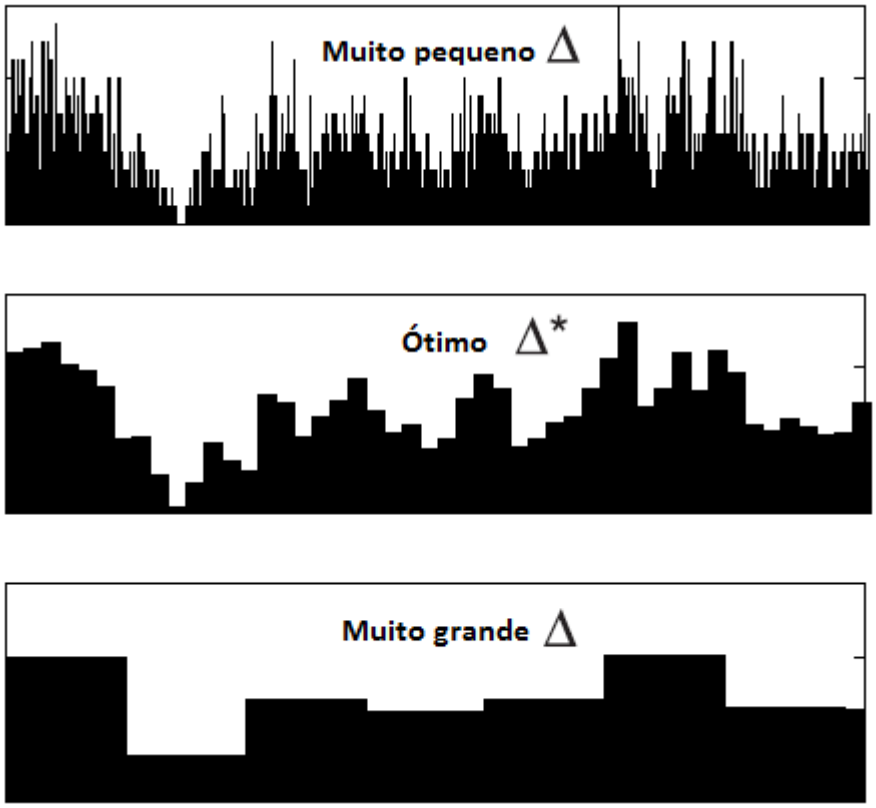

Figura 2 - Diferentes escolhas do intervalo de classe (SHIMAZAKI;SHINOMOTO, 2007)

O primeiro histograma da Figura 2 mostra como a escolha de um intervalo de classe (Δ) muito pequeno pode gerar flutuações muito grandes e o terceiro histograma mostra que a escolha de um intervalo de classe muito grande não permite identificar corretamente as variações mais sutis da taxa básica do sinal. Utilizar o segundo gráfico da Figura 2 permite extrair informações

bem melhores dos dados sendo, portanto uma estimativa melhor da função densidade de probabilidade.

Dado a importância da escolha do Δ e de um possível exaustivo trabalho para seu cálculo manual, foi desenvolvido um algoritmo para cálculo de Δ (SHIMAZAKI; SHINOMOTO, 2007). Segue abaixo sua teoria:

Quanto menor o erro quadrático integral médio (EQIM), dado pela equação Eq 2.5, em período de tempo T, melhor a estimativa.

$$ER = \frac{1}{T}\int_0^T E(\hat{\lambda}_t - \lambda_t)^2 \ dt \qquad (2.5)$$

Sendo:

$\hat{\lambda}_t$ = Função densidade de probabilidade estimada

λ_t = Função densidade de probabilidade ideal

$E()$ refere-se à expectativa das diferentes realizações dos eventos pontuais.

Começa-se com uma FDP $\hat{\lambda}_t$ com um intervalo de classe pequeno e se explora um método que minimize ER. O problema é que não se conhece λ_t.

Definindo-se:

Δ = Intervalo de classe do histograma

n = Quantidade de sequências de sinais

k_i = Número de sinais durante o i-ésimo intervalo de classe

$\hat{\theta}_i = \frac{k_i}{n\Delta}$ = Altura estimada de i-ésimo intervalo de classe

Dado um intervalo de largura Δ, a altura esperada da barra, para $t \in [0, \Delta]$ é dado pela média temporal da equação 2.6:

$$\theta = \frac{1}{\Delta}\int_0^\Delta \lambda_t \ dt \qquad (2.6)$$

E o número total de pontos k que entram neste intervalo obedecem à distribuição de Poisson:

$$p(k|n\Delta\theta) = \frac{(n\Delta\theta)^k}{k!} e^{-n\Delta\theta}, \tag{2.7}$$

onde a expectativa é $\Delta\theta$. Uma estimativa razoável para θ é $\hat{\theta} = {}^{k}/_{n\Delta}$, que é a altura empírica da barra do gráfico para $t \in [0, \Delta]$.

Pode-se reescrever a Eq 2.5, segmentando o período T em N intervalos de tamanho Δ:

$$ER = \frac{1}{\Delta}\int_0^{\Delta} \frac{1}{N}\sum_{i=1}^{N} \{E\left(\hat{\theta}_i - \lambda_{t+(i-1)\Delta}\right)^2\} \; dt \tag{2.8}$$

Redefinindo $\lambda_{t+(i-1)\Delta}$ para λ_t definido para o intervalo $t \in [0, \Delta]$ fica:

$$ER = \frac{1}{\Delta}\int_0^{\Delta} \langle E(\hat{\theta}_i - \lambda_t)^2 \rangle \; dt \tag{2.9}$$

onde a expectativa $E()$ refere-se à média sobre número de pontos, ou $\hat{\theta} = {}^{k}/_{\Delta\theta}$ dado a taxa básica λ_t, ou seu valor médio θ.

Pode-se então decompor a equação do erro em duas partes:

$$ER = \langle E(\hat{\theta} - \theta)^2 \rangle + \frac{1}{\Delta}\int_0^{\Delta} \langle E(\lambda_t - \theta)^2 \rangle \; dt \tag{2.10}$$

O primeiro termo da Eq. 2.10 refere-se à variação estocástica do estimador $\hat{\theta}$ sobre a média esperada θ. E o segundo termo refere-se à flutuação temporal de λ_t sobre sua média θ no intervalo Δ.

O segundo termo da equação ainda pode ser decomposto em mais dois outros termos:

$$\frac{1}{\Delta}\int_0^{\Delta} \langle E(\lambda_t - \langle\theta\rangle + \langle\theta\rangle - \theta)^2 \rangle \; dt = \frac{1}{\Delta}\int_0^{\Delta} \langle(\lambda_t - \langle\theta\rangle)^2\rangle dt \quad + \quad \langle(\theta - \langle\theta\rangle)^2\rangle \tag{2.11}$$

Como o primeiro termo da Eq. 2.11 não depende da largura do intervalo Δ, pois

$$\frac{1}{\Delta}\int_0^{\Delta} \langle(\lambda_t - \langle\theta\rangle)^2\rangle dt = \frac{1}{T}\int_0^{T} (\lambda_t - \langle\theta\rangle)^2 \; dt. \tag{2.12}$$

Define-se a função custo sem este termo na equação, tal como na Eq 2.13

$$C_n(\Delta) = \langle E(\hat{\theta} - \theta)^2 \rangle \quad + \quad \langle (\theta - \langle \theta \rangle)^2 \rangle \tag{2.13}$$

A média esperada θ, contudo, não é uma variável observável, portanto, deve-se retirá-la da equação;

Considerando $E\hat{\theta} = \theta$ e utilizando a regra da decomposição, pode-se escrever

$$\langle E\left(\hat{\theta} - \langle E\hat{\theta} \rangle\right)^2 \rangle = \quad \langle E\left(\hat{\theta} - \theta\right)^2 \rangle \quad + \quad \langle (\theta - \langle \theta \rangle)^2 \rangle \tag{2.14}$$

Então a função custo fica:

$$C_n(\Delta) = 2\langle E(\hat{\theta} - \theta)^2 \rangle \quad - \quad \langle E(\hat{\theta} - \langle E\hat{\theta} \rangle)^2 \rangle \tag{2.15}$$

Assumindo que o processo em estudo obedece à distribuição de Poisson, a variância do numero de pontos k em cada intervalo Δ é igual à média. Assim podemos expressar:

$$E\left(\hat{\theta} - \theta\right)^2 = \frac{1}{n\Delta} E\hat{\theta} \tag{2.16}$$

Finalmente, chega-se a função custo sem θ, definida na Eq 2.17

$$C_n(\Delta) = \frac{2}{n\Delta} \langle E\hat{\theta} \rangle \; - \; \langle E(\hat{\theta} - \langle E\hat{\theta} \rangle)^2 \rangle \tag{2.17}$$

O tamanho ótimo do intervalo de classe pode ser então obtido, achando o que minimiza a função custo, tal como a Eq. 2.14

$$\Delta^* \stackrel{\text{def}}{=} \arg\min C_n(\Delta) \tag{2.18}$$

Trocando a função de expectativa $E()$ pela contagem dos pontos, pode-se chegar ao algoritmo computacional para seleção do tamanho ótimo do intervalo de classe de um histograma.

2.3 Algoritmo para cálculo do intervalo ótimo

1. Ordenar os dados
2. Dividir o período T em N intervalos de largura Δ
3. Calcular a quantidade de dados k_i dentro do $i-$ésimo intervalo.
4. Construir a média e a variância dos dados k_i como

$$\bar{k} \overset{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N} k_i, \text{ e } v \overset{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N}(k_i - \bar{k})^2$$

5. Calcular a função custo

$$C_n(\Delta) = \frac{2\bar{k} - v}{(n\Delta)^2}$$

6. Repetir passos um a quatro, modificando a quantidade de intervalos para achar Δ^* que minimizem a função Custo.

2.4 Python e computação científica

Python é uma linguagem de computação concisa e utilizada para os mais diversos domínios de aplicação. Entre suas vantagens está sua sintaxe compacta, ser orientada a objetos e sua manipulação de alto nível de tipos de dados. Isso tudo, aliado ao fato de ser uma linguagem interpretada, permite a construção e testes de programas para realização de tarefas não triviais rapidamente. Sendo a velocidade de desenvolvimento um fator importante, esta linguagem vem sendo muito bem aceita atualmente.

Para desenvolvimento de aplicações científicas com foco no processamento de dados, o desempenho é um quesito que não pode ser deixado de lado. Por isso a linguagem C/C++ é muito utilizada nessa área. Python, sozinho, não atenderia aos pré-requisitos deste domínio de aplicação, pois não possui desempenho tão bom como esses concorrentes. Contudo, a utilização das bibliotecas Scipy e Numpy, permite a utilização de Python para computação científica.

Numpy é uma biblioteca de Python fundamental para computação cientifica. Ela possui ferramentas para armazenamento e processamento de matrizes. É possível estendê-la com código C e integrar código Fortran existente. Scipy utiliza Numpy como base para prover diversos módulos de alto nível utilizados pela área cientifica em um único pacote.

Um teste de desempenho é apresentado no web site oficial destas bibliotecas, http://www.scipy.org. Nele é apresentado o desempenho de Python utilizando essas bibliotecas contra diversas outras implementações. A Tabela 1 apresenta alguns resultados deste teste.

Tabela 1 - Comparação de desempenho de linguagens e ferramentas computacionais

Tipo de Solução	Tempo (s)
Python + Numpy Expression	29.3
Matlab (estimate)	29.0
Python (estimate)	1500.0
Fast Inline	2.3
Python/Fortran	2.9
Pure C++	2.16

2.5 Computação paralela com IPython

Para processamento de grandes quantidades de dados, hoje é habitual utilizar clusters de computadores para processamento em paralelo por razões de desempenho. Para a linguagem Python, a ferramenta IPython (disponível em: http://ipython.org) foi desenvolvido pela IPython Development Team, para facilitar desenvolvimento de programas em paralelo.

IPython é uma ferramenta que provê dois principais componentes: um ambiente interativo melhorado para Python, e uma arquitetura para computação paralela. Com ele é possível executar códigos escritos em Python paralelamente com pouca modificação do código fonte.

A Figura 3 mostra como um mesmo trecho de código pode ser executado facilmente em serial ou paralelo com IPython.

```
In [61]: dview.block = True

In [62]: serial_result = map(lambda x:x**10, range(32))

In [63]: parallel_result = dview.map(lambda x: x**10, range(32))

In [67]: serial_result==parallel_result
Out[67]: True
```

Figura 3 – Paralelização rápida utilizando IPython (Retirado de: http://ipython.org)

Na linha de código 62 da Figura 3, é executado o método *map()* da linguagem Python em serial que, neste caso, calcula o valor da variável *x* elevado à dez, para *x* indo de 0 a 32, e retorna o vetor o resultado para a variável *serial_result*. Este mesmo cálculo é feito em paralelo apenas na linha de código 63, bastando utilizar a instancia *dview* da interface *Direct* do IPython. A linha 67 compara os resultados em serial e paralelo para legitimá-los.

Por essa facilidade de paralelismo, esta ferramenta é utilizada neste trabalho para o processamento dos dados em paralelo.

3 Configuração e Implementação

3.1 Configurando o ambiente computacional

Para utilizar o programa implementado neste trabalho ou refazê-lo é necessário instalar e configurar as ferramentas Python, IPython e as bibliotecas. Este trabalho foi realizado no sistema operacional (SO) Ubuntu Linux 12.04, portando as descrições que aqui seguem se aplicam a esse ambiente. Para outras plataformas pode ser necessário consultar referências.

O interpretador Python já vem instalado com o Ubuntu, mas é importante confirmar e se necessário obter a versão mais atual. Para isso, basta utilizar o comando que segue abaixo no terminal do Linux:

$sudo dpkg -p python

A versão utilizada neste trabalho foi a 2.7.3. Caso este pacote não esteja instalado, ou esteja numa versão anterior à 2.7, é necessário instalá-lo/atualizá-lo. Para instalá-lo digite o comando:

$sudo apt-get install python2.7.3

É necessário também ter o pacote de desenvolvimento do Python que, pelo Ubuntu, pode ser obtida pelo seguinte comando:

$sudo apt-get install python-dev.

Para instalar as bibliotecas Numpy e Scipy:

$sudo apt-get install python-scipy

Com o pacote Python e as bibliotecas para computação cientifica corretamente instaladas deve-se instalar o IPython. Este pacote, no entanto, depende da biblioteca *zeromq* disponível em: http://www.zeromq.org/. Esta é responsável pelo transporte de mensagens na camada de aplicação para ambientes clusterizados.

Para instalar a *zeromq*, é necessário ter pacotes *libtool, autoconf* e *automake* instalados. Para instalá-los, basta executar o comando:

$sudo apt-get install libtool autoconf automake

Acesse o site http://www.zeromq.org/ e obtenha o arquivo para posix (ou para o outro sistema operacional que estiver utilizando).

Descompacte o arquivo baixado e, dentro da pasta descompactada, execute os seguintes comandos no terminal:

$ ./configure.sh

$make

ou

$sudo make install

para instalar para todos os usuários.

E para criar os *links* e *cache* necessários, execute o comando abaixo no terminal:

$sudo ldconfig

Com o *python* e *zeromq* instalados, resta instalar o IPython. Isto é mais simples de ser feito utilizando a ferramenta *distribute*:

Acesse o site http://pypi.python.org/, procure e obtenha o pacote *distribute*.

Para instalá-lo, na pasta onde o arquivo se encontra utilize o comando:

$python distribute_setup.py

Por último, execute o comando *easy_install* do pacote *distribute* para instalar o IPython e suas principais bibliotecas de uma vez só:

$sudo easy_install ipython[zmq, test, qtconsole, notebook]

Com o ambiente configurado, é possível fazer um teste simples para confirmar seu funcionamento. Iniciam-se quatro threads com o comando:

$ipcluster start -n 4 &

Depois se inicia o *shell* interativo do IPython com comando

$ipython

E, dentro deste *shell*, executa-se os comandos mostrados em *In[1], In [2] e In[3]* abaixo:

```
In[1]: from IPython.parallel import Client

In[2]: rc = Client()

In[3]: rc = Client()
```

A saída *out[3]* , a seguir, mostra que as quatro threads estão funcionando com os respectivos *ids:* 0,1,2 e 3:

```
Out[3]: [0,1,2,3]
```

3.2 Implementação

A implementação deste trabalho foi feita em vários módulos para permitir reutilização. Ao todo foram três módulos desenvolvidos: bin_methods_sc.py, bin_serial_sc.py e bin_parallelx2_sc.py. Este modelo de programação em módulos foi utilizado porque facilita o entendimento e a re-utilização do código.

O módulo bin_methods.py é o módulo que contém os diversos métodos necessários para a execução dos outros módulos. Este é escrito em Python, utilizando as bibliotecas de computação cientifica (Numpy e Scipy) para melhorar a eficiência do algoritmo.

Os módulos bin_serial_sc.py, e bin_parallelx2_sc.py são dois programa escrito em Python prontos para serem executados. Estes utilizam os métodos do módulo bin_methods_sc.py. O bin_serial_sc.py é uma implementação serial e bin_parallelx2_sc.py é uma implementação em paralelo para executar em duas threads.

Para correto entendimento do código, segue abaixo a descrição dos métodos do módulo bin_methods_sc.py, os quais serão utilizados pelos outros módulos. A descrição de cada método apresenta o que é passado como parâmetro: Entrada, o que o método faz: Função (caso necessário), e o que ele retorna: Saída.

op(filestring)

Entrada: Este método aceita como entrada um string com o nome do arquivo que contém os dados. Estes devem estar separados entre si por barra de espaço(" "). O programa não aceitará um arquivo com dados organizados de outra forma.

Função: Processa os dados que estão salvos no arquivo e os transforma em um vetor de floats.

Saída: Retorna o vetor de floats.

bubble(theList)

Estrada: O parâmetro deste método, theList, é o vetor de floats obtidos do método **op()**.

Função: Ordena o vetor passado como entrada.

Saída: Retorna o vetor de floats ordenado.

mean_k(K,N)

Entrada: Este método tem como entrada um vetor de dados K e seu tamanho N.

Saída: Este método retorna a média aritmética (float) dos valores do vetor K.

variance_k(K, N, km)

Entrada: Este método tem como entrada um vetor de dados K, seu tamanho N e a média de seus valores km.

Saída: Este método retorna a variância (float) dos valores do vetor K.

cost(D,K)

Entrada: O tamanho do intervalo de classe D, do histograma dado pelo vetor K.

Saída: a função Custo definida na seção 2.2

delta(L,N)

Entrada: Vetor L com dados a partir do qual o histograma vai ser calculado. Numero de intervalos de classe N em que o histograma será dividido

Saída: Tamanho do intervalo de classe.

dist(N, L, D)

Entrada: Numero de intervalos de classe N em que o histograma será dividido, vetor com dados L e o tamanho do intervalo de classe D.

Saída: Vetor com valores correspondente aos valores de cada intervalo do histograma

min_cost(L)

Entrada: Matriz L[C, K] onde K é o vetor correspondente aos valores de cada intervalo do histograma e C é a função custo deste histograma K.

Saída: Retorna vetor dentro da matriz L que possui a menor função custo

map_bins(List, N)

Entrada: Vetor de dados ordenados List e numero de intervalos de classe do histograma a ser calculado N

Saída: Matriz L[C,K] onde K é o vetor correspondente aos valores de cada intervalo do histograma e C é a função custo deste.

3.2.1 Implementação para execução serial

O programa *bin_serial_sc.py* , que implementa o algoritmo da secção 2.3 em serial, utiliza os métodos definidos pelo arquivo *bin_methods_sc.py*, explicados anteriormente. Para isso, é necessário que ambos os arquivos estejam na mesma pasta.

Para executar o programa é necessário ter preparado o ambiente, como descrito na seção 3.1 até, pelo menos a instalação da biblioteca *numpy*. Feito isso e, com os arquivos *bin_methods_sc.py* e *bin_serial_sc.py* no mesmo diretório, pode-se executar IPython no terminal de sua preferência o comando:

$ipython

E, em seguida executar o comando abaixo para executar o programa:

$run bin_serial_sc.py data_example

Onde *data_example* é o caminho para arquivo que contém os dados a serem processados. É necessário enfatizar que, para correto funcionamento do programa, os dados devem estar organizados separados apenas por barra de espaço entre si e números decimais devem utilizar ponto, não vírgula. Abaixo segue o exemplo de um arquivo com essa formatação:

```
4.37 3.87 4.00 4.03 3.50 4.08 2.25 4.70 1.73 4.93 1.73 4.62 3.43
4.25 1.68 3.92 3.68 3.10 4.03 1.77 4.08 1.75 3.20 1.85 4.62 1.97
4.50 3.92 4.35 2.33 3.83 1.88 4.60 1.80 4.73 1.77 4.57 1.85 3.52
4.00 3.70 3.72 4.25 3.58 3.80 3.77 3.75 2.50 4.50 4.10 3.70 3.80
3.43 4.00 2.27 4.40 4.05 4.25 3.33 2.00
```

Com essa condição satisfeita o programa deve rodar corretamente e retornará um vetor *F* que representa o histograma. Para verificá-lo, basta digitar *F*, no shell IPython. Também é possível checar seu gráfico através do comando abaixo:

$*plot()*

Este comando apresenta o histograma normalizado obtido através da biblioteca *matplotlib.*

A trecho de código abaixo mostra o cabeçalho de *bin_serial.sc.py*:

```
1 import matplotlib.pyplot as plt
2 from bin_methods import *
3 import sys
4 import numpy as np
```

Na linha 2 é onde *bin_serial.sc.py* importa os métodos de *bin_methods.py*. Também é necessário importar da biblioteca numpy e a biblioteca padrão *sys* A primeira aumenta o desempenho da implementação e a segunda será utilizada para leitura dos argumentos passados pelo *shell*. A biblioteca *matplotlib.pyplot* é importada, pois foi desenvolvido um método básico para visualização dos dados. A próxima parte do código, abaixo, carrega os dados e calcula os histogramas:

```
8 L = op(sys.argv[1]);
9
10
11 #Calculate histograms and costs
12 l = len(L)
13 M = map(lambda n: map_hists(L, n), range(1, l));
14 C = map(lambda n: map_costs(L,n, M[n-1]), range(1,l));
```

Na linha 8, no código apresentado acima, é onde é chamado o método *op()* de *bin_methods.py*. Esta linha passa como parâmetro o argumento que usuário escreve invocando *bin_serial_sc.py* pelo console. Este argumento deve ser o caminho para o arquivo que contém os dados dos quais será obtido o histograma.

As linhas 12 à 14, ainda sobre código anterior, fazem o cálculo da função **Custo** definida na secção 2.2 para larguras de intervalos de classe iterativamente decrementas. A linha 13 calcula todos os histogramas com um intervalo de classe à *I* intervalos de classe, onde *I* é o quantidades de dados sendo analisados. Por último, a linha 14 calcula a função custo para esses histogramas.

Com o vetor *C* contendo o custo de todos os histogramas e com a matriz *M* contendo todos os histogramas calculados, o histograma de menor custo é trivialmente calculado encontrando o índice do vetor *C* com o menor custo, e calculando a o histograma associado a este índice na matriz *M*. Isto é feito pelo trecho de código apresentado abaixo:

```
16 #Looking for minimum Cost
17 i = np.asarray(C).argmin();
18 F = M[i];
```

Neste, a variável *i* recebe o argumento do menor valor do vetor *C* e depois *F* recebe o histograma correspondente na matriz *M*.

Por último, o próximo código apresenta o método definido para apresentar o histograma não normalizado do resultado final. Este apenas faz uma chamada ao metódo *graph()* de *bin_methods_sc.py* passando os parâmetros já calculados.

```
20 def plot():
21     graph(F,L);
22     return;
```

3.2.2 Implementação para execução em paralelo

A implementação em paralelo, como já mencionado na seção 2.4, difere pouco da implementação serial em termos de codificação. Contudo, para executar o programa em paralelo é necessário inicializar as *engines* do IPython e mapear as maquinas que farão parte do cluster.

Para execução em uma máquina em paralelo, basta digitar o comando abaixo no *shell* do Linux;

$ipcluster start --n 4

Este comando inicia quatro *engines* locais para executar o código em threads diferentes. O parâmetro *start* inicia as *engines* e o parâmetro *–n* especifica quantas serão iniciadas. Neste caso são iniciadas quatro *engines.* Para utilizar mais ou menos threads, basta trocar o número 4 pela quantidade desejada de threads.

Com as *engines* corretamente inicializadas, executar o programa *bin_parallel_sc.py* é semelhante a executar *bin_serial_sc.py*. Com *bin_methods_sc.py* e *bin_parallel_sc.py* no mesmo diretório deve-se inicializar o IPython como descrito na seção anterior e, em seguida executar o comando abaixo:

$run bin_serial_sc.py data_example

Onde *data_example*, novamente, é o caminho para arquivo que contém os dados a serem processados. E, novamente, os dados devem estar organizados separados apenas por barra de espaço entre si e números decimais devem utilizar ponto, não vírgula.

A codificação possui algumas mudanças em relação ao código serial, mas continua bem parecido. O trecho abaixo apresenta as bibliotecas e os métodos que a programa importa:

```
1 from Ipython.parallel import Client
2 import matplotlib.pyplot as plt
3 from bin_methods import *
4 import sys
5 import numpy as np
```

Neste caso, além das bibliotecas importadas no programa serial, é necessário importar também *Client* de *IPython.parallel*. Esta contém as interfaces para falar com as engines em execução.

Com a biblioteca importada, é necessário instanciar uma interface DirectView do IPython que permite passar comandos para serem executados nas engines inicializadas. Isso é feito no código abaixo:

```
8 c = Client();
9 view = c[:];
10 view.block=True;
11
12 view.run('bin_methods.py');
```

Neste caso, na linha 8 do trecho anterior, uma instancia dos Clients() é criada na variável *c* e na linha 9, uma instancia da interface DirectView, *view*, é criada. Configura-se então a instancia *view* para esperar o resultado na linha 10 e executa-se *bin_methods.py* em todas a engines para que estas conheçam os métodos definidos nele.

O arquivo é aberto e armazenado como um vetor *numpy* em *L* da mesma forma que em *bin_serial_sc.py*. A diferença é que o vetor *L* deve ser armazenado nas *engines*, para posterior processamento interno destas. Isso é feito na linha 18 do trecho de código abaixo:

```
14 #open file and return array of data
15 L = op(sys.argv[1]);
16
17 #Send variable L to the engines
18 view['L'] = L;
19
20 #Calculate costs and histograms in parallel
21 l = len(L);
22 M = view.map(lambda n: map_hists(L,n),range(1,l));
23 view['M'] = M;
24 C = view.map(lambda n: map_costs(L,n,M[n-1]),range(1,l));
```

O método *map* da instancia *view* funciona como um *map* tradicional, mas dividindo o a execução das iterações entre as diversas instâncias visualizadas por *view*. Portanto, as linhas 22 e 24 do código acima fazem o cálculo dos histogramas de das funções Custo destes em paralelo.

O resto do código de *bin_parallel_sc.py* é idêntico a *bin_serial_sc.py*, pois não foi utilizado paralelismo para achar o custo mínimo na matriz *C* e nem para desenhar o gráfico.

3.3 Geração de Dados de testes

Para realizar os testes foi necessário gerar dados pontuais que representassem um modelo de Poisson. Para isso foi utilizado o processo de geração de dados não homogêneos (HEEGER, 2000).

Para um espaço de tempo $t \in [0, \mathrm{T}]$, para qual queremos amostrar uma função $r(t)$, divide-se o tempo em vários intervalos δt, tal que $r(t)\delta t \ll 1$. Então se gera uma seqüência de números randômicos $x[i]$, uniformemente distribuídos entre zero e um, para cada δt. Depois, deve-se amostrar $r(t)$, também para cada intervalo δt, obtendo-se $r[i]$. Para cada intervalo de tempo δt, caso $x[i] \leq r[i]/(\max(\mathrm{r[i]}))$, um ponto é gerado naquele instante.

Através deste método, foram gerados duas amostras de dados, uma para a função $e(t) = (t+1)^8$ e para $g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)$ para o cálculo de suas respectivas funções densidades de probabilidade. Estas foram escolhidas, pois suas FDPs são bem conhecidas.

Depois de realizado os testes com estas funções conhecidas, foram utilizados dados do trabalho feito por Maciel et al. (2012) para confirmar a utilidade deste método em dados empíricos. Foi escolhida uma amostra de dados dentro do conjunto de dados recebidos e, só então, foi calculado a FDP dos dados.

4 Metodologia

A fim de verificar a assertividade da implementação, foram utilizadas as duas diferentes distribuições geradas, de acordo com a seção 3.3, para construir os gráficos das funções densidade de probabilidade. A primeira, foi a função $e(t) = (t+1)^8$, para qual o gráfico é apresentado na Figura 4.

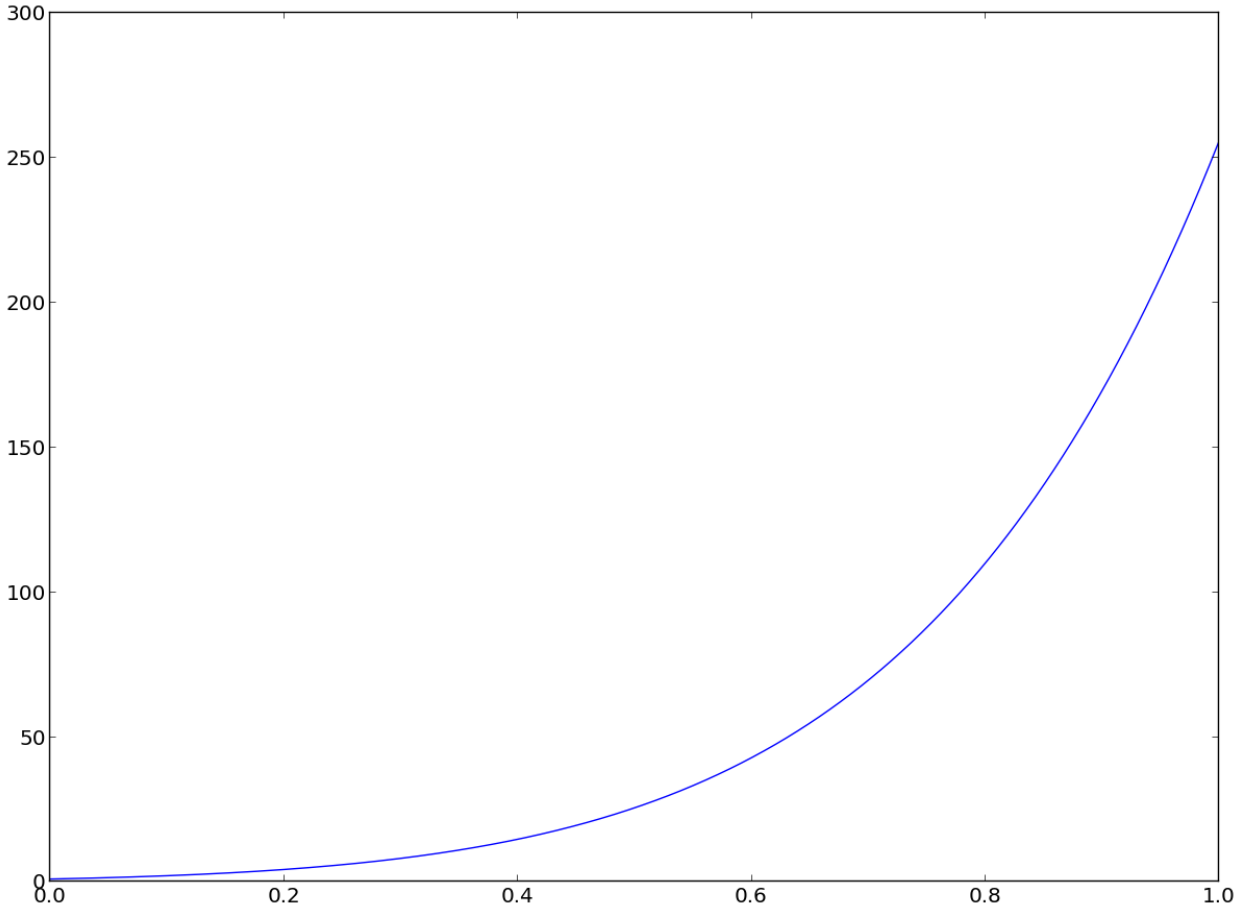

Figura 4– Gráfico de $e(t) = (t+1)^8$ por t em segundos

E o gráfico da segunda função $\boldsymbol{g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)}$ é apresentado na Figura 5.

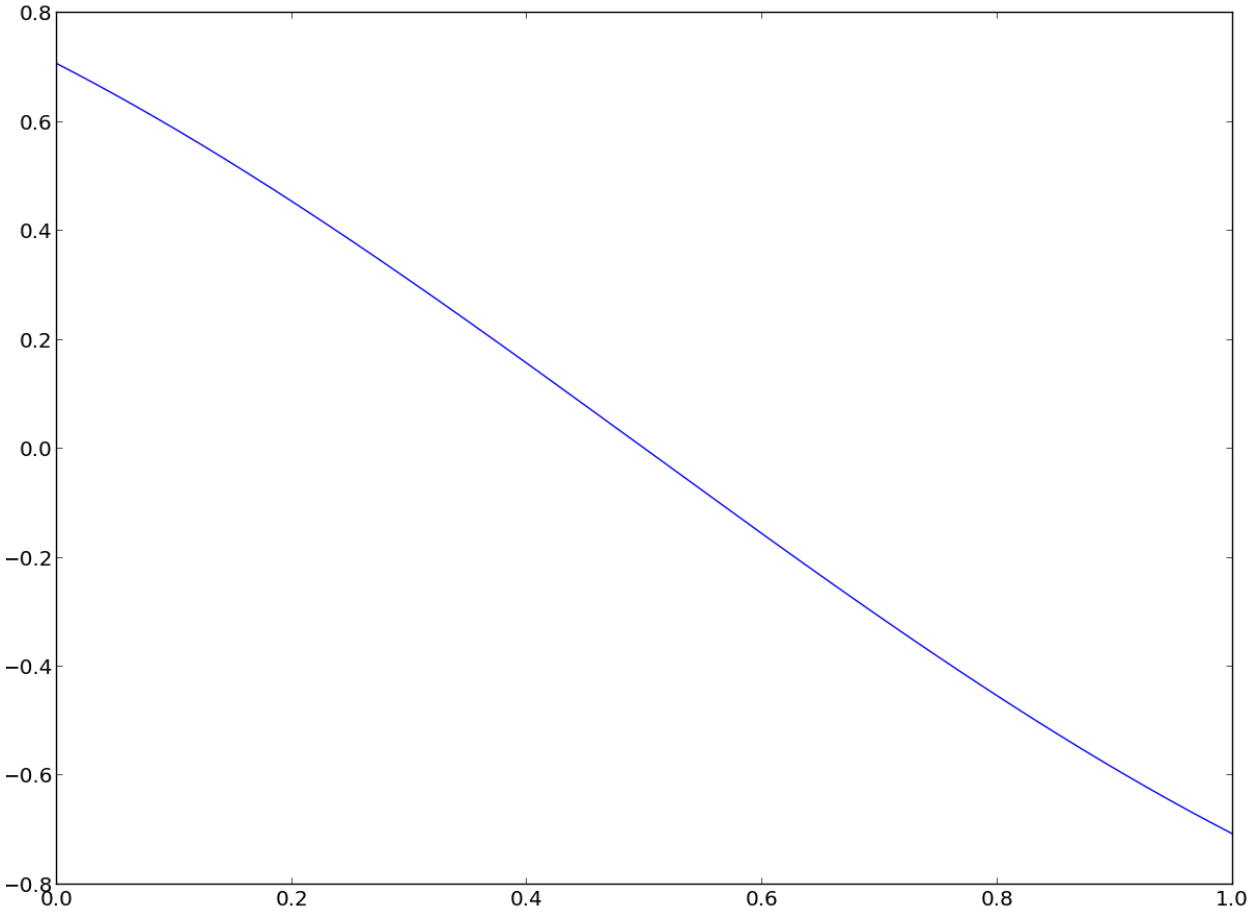

Figura 5 – Gráfico de $\boldsymbol{g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)}$ por t em segundos

A partir dessas funções foram geradas cem mil amostras pontuais a partir do método descrito na seção 3.3 para $t \in [0,1]$, para calcular da função densidade de probabilidade com o algoritmo descrito na seção 2.3.

Para ambas as funções foram, então, realizadas execuções em serial e em paralelo. Para cada uma dessas quatro execuções foram tomados dez tempos de execução e obtido a média para posterior discussão sobre a eficiência em paralelo.

4.1 Execução serial $e(t) = (t+1)^8$:

Custo mínimo: -646.660

Tamanho do intervalo de classe ótimo: 0,3014

Número de intervalos de classe para custo mínimo: 846

A Figura 6 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

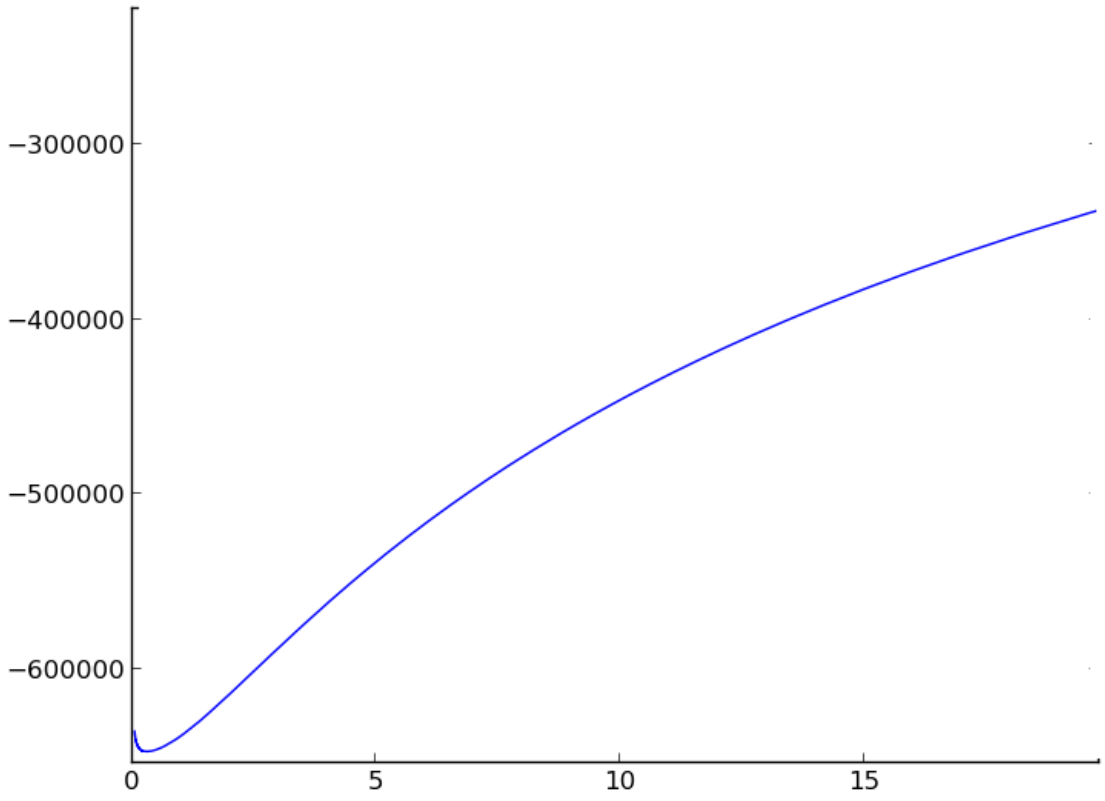

Figura 6 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a função $e(t)$

A Figura 7 mostra o gráfico da função densidade de probabilidade:

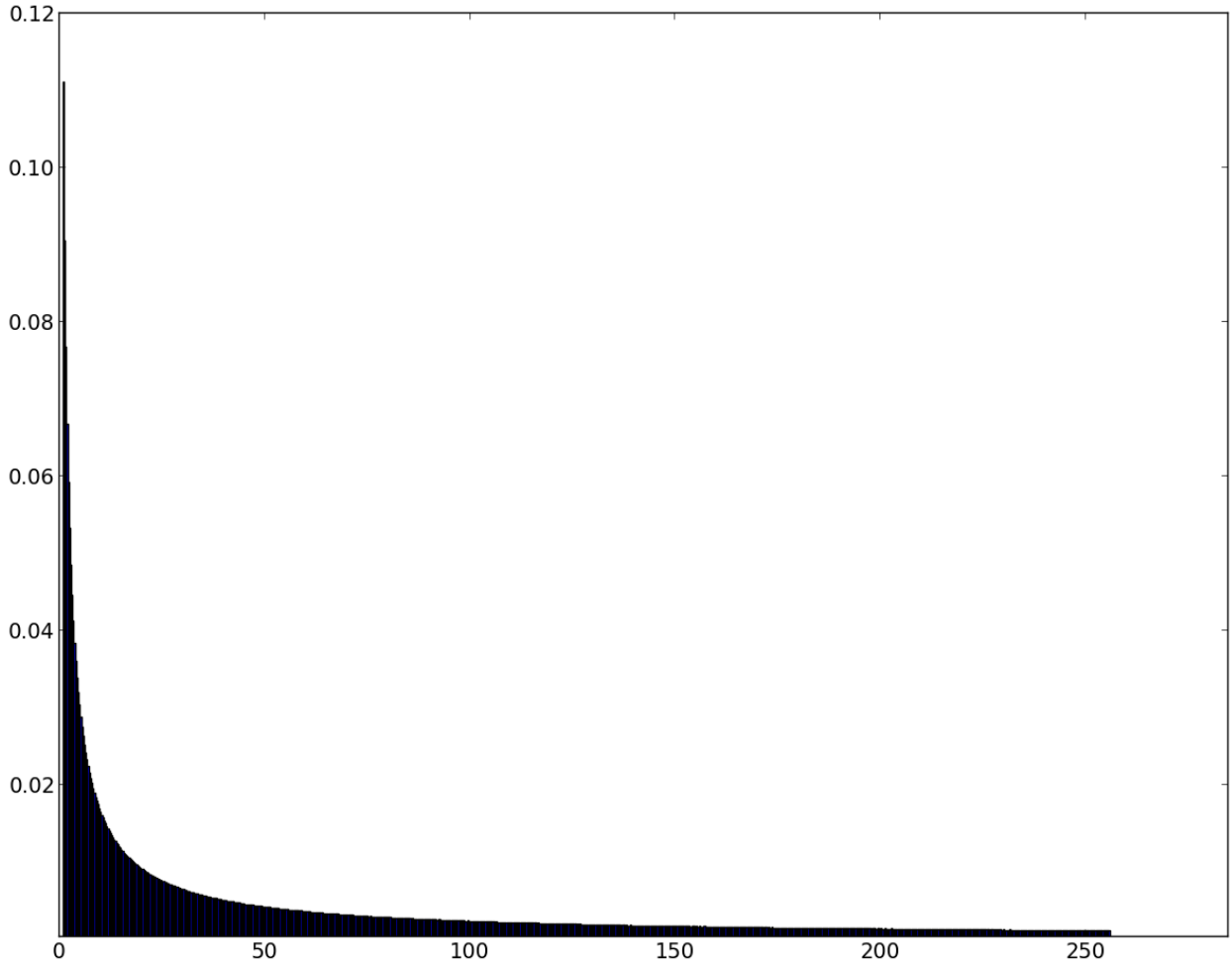

Figura 7 – Gráfico do histograma ótimo que representa a FDP de $e(t)$

A Tabela 2 mostra os tempos de execução tomados.

Tabela 2- Medição do cálculo da densidade de probabilidade de $e(t) = (1+t)^8$ em serial

Nº da Execução	Real Time(s)	User Time(s)	Sys Time (s)
1	20.788	19.061	0.860
2	20.773	19.089	0.812
3	20.748	18.957	0.936
4	20.820	18.981	0.984
5	20.448	19.101	0.760
6	20.898	19.121	0.892
7	20.874	19.105	0.908
8	20.907	19.085	0.928
9	20.480	18.973	0.928
10	20.544	19.081	0.896
Média	**20.728**	**19.055**	**0.890**

A coluna *Sys Time* representa o tempo que o sistema operacional demorou executando as *system calls*. O *User Time* refere-se ao tempo que o código demorou para ser executado no terminal do usuário. O *Real Time* representa o tempo total que o código leva para ser executado. No caso da implementação para execução em serial, ele é equivalente à *Sys Time* + *User Time*.

4.2 Execução serial $g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)$:

Custo mínimo: -48.806.116

Tamanho do intervalo de classe ótimo: 0,0707

Número de intervalos de classe para custo mínimo: 20

A Figura 8 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

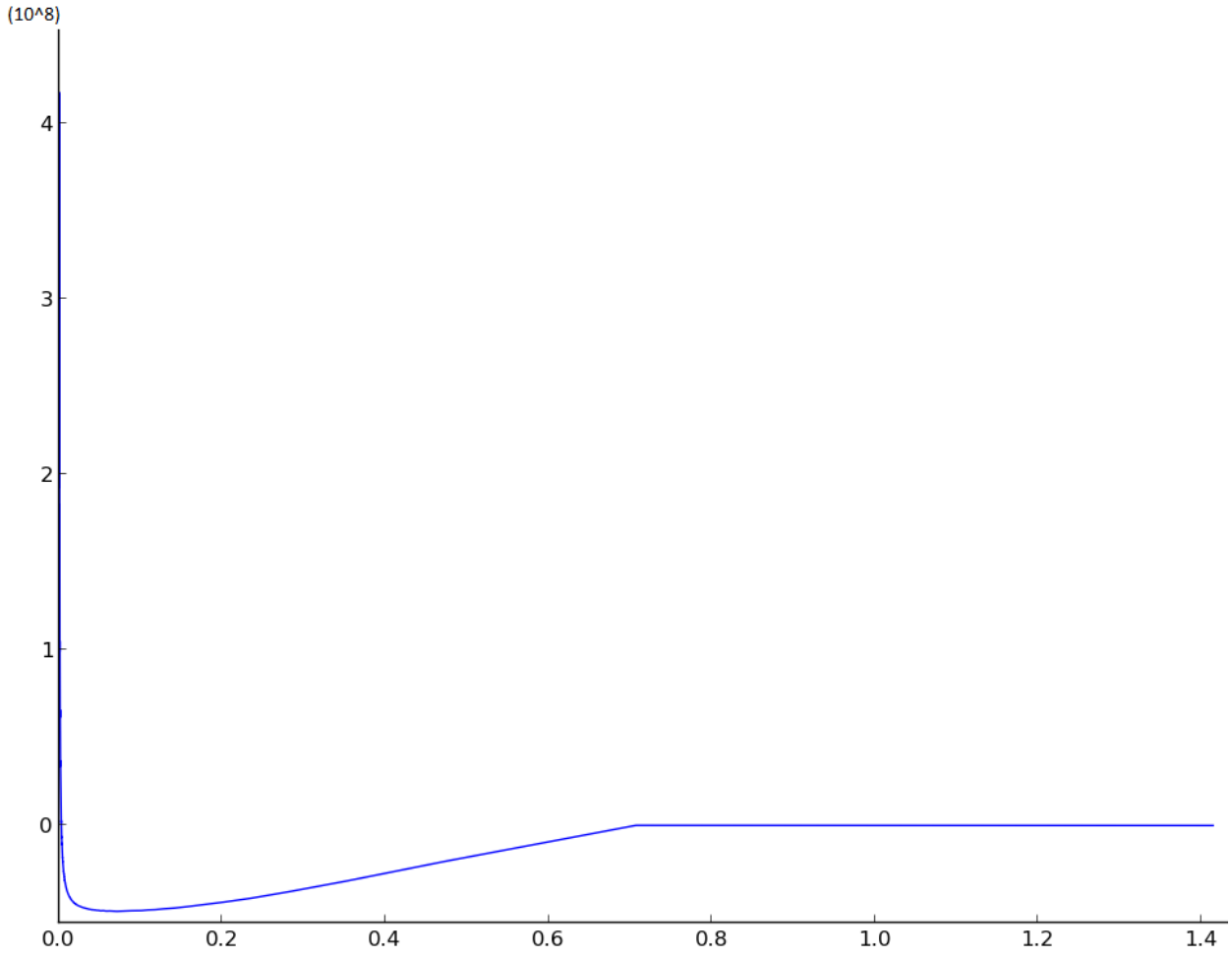

Figura 8 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a função $g(t)$

A Figura 9 mostra o gráfico da função densidade de probabilidade:

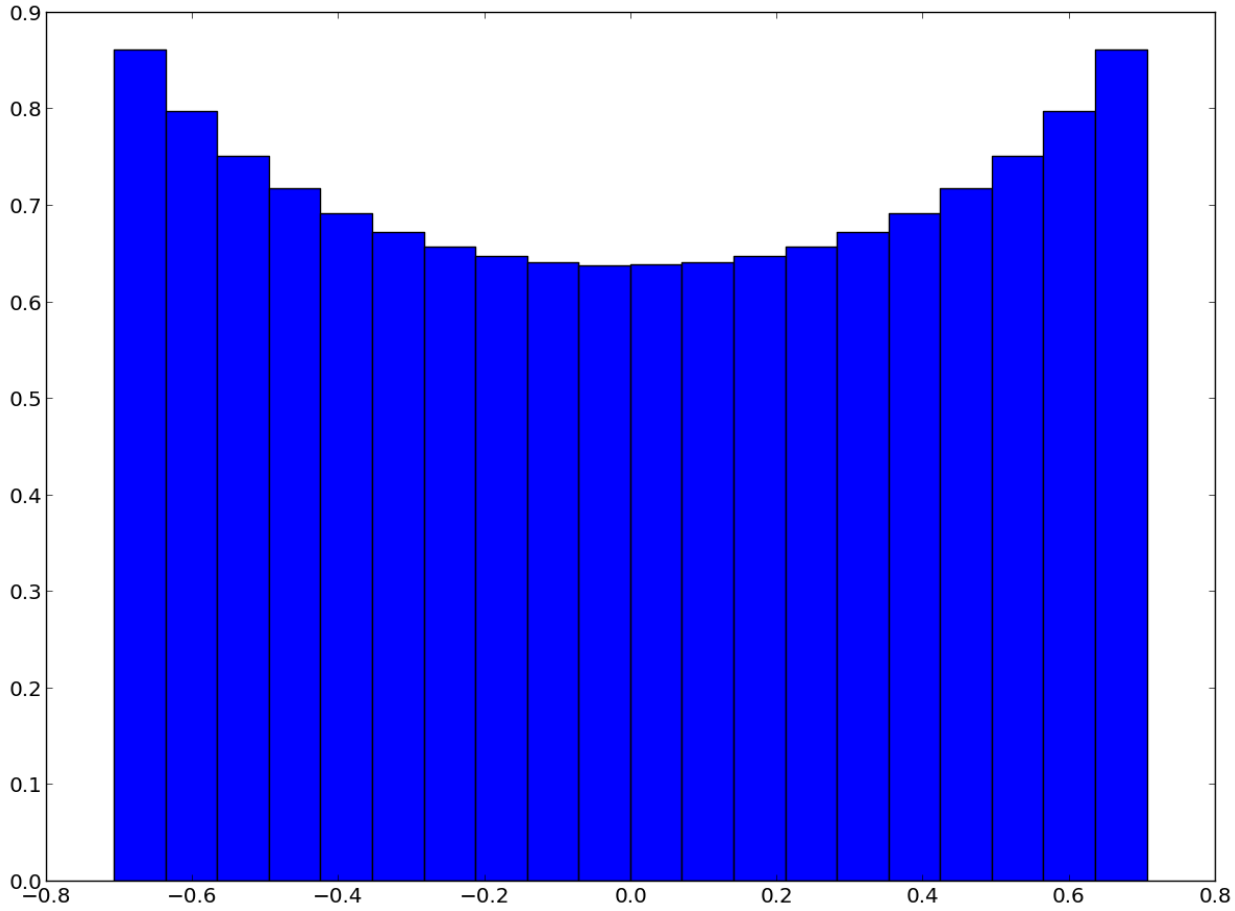

Figura 9 – Gráfico do histograma ótimo que representa a FDP de g(t)

A Tabela 3 mostra os tempos de execução tomados.

Tabela 3 - Medição do cálculo da densidade de probabilidade de $g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)$ em serial

Nº da Execução	Real Time(s)	User Time(s)	Sys Time (s)
1	23.740	20.041	0.948
2	21.371	19.905	0.908
3	21.267	19.765	0.928
4	21.258	19.709	0.992
5	21.288	19.861	0.860
6	21.237	19.777	0.900
7	21.294	19.825	0.900
8	21.215	19.749	8.680
9	21.301	19.793	0.948
10	21.405	19.857	0.94
Média	**21.537**	**19.828**	**1.700**

4.3 Execução paralela $e(t) = (t+1)^8$:

Custo mínimo: -646.660

Tamanho do intervalo de classe ótimo: 0,3014

Número de intervalos de classe para custo mínimo: 846

A Figura 10 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

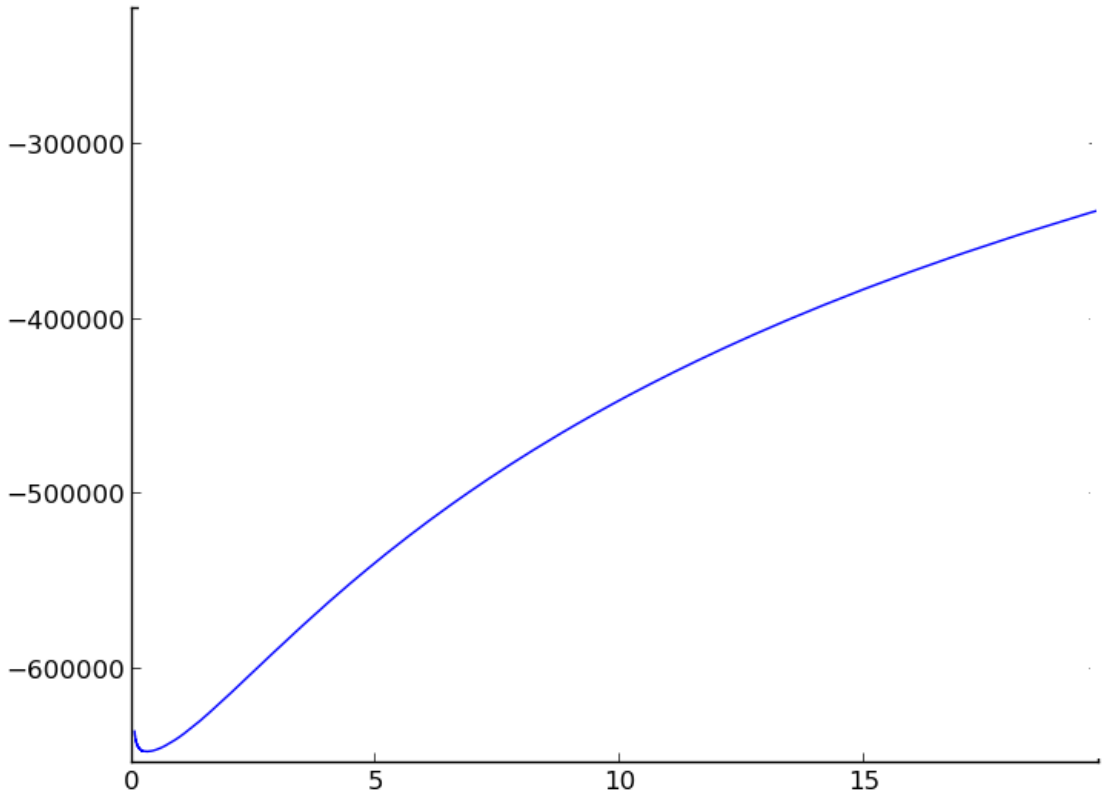

Figura 10 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a função $e(t)$

A Figura 11 mostra o gráfico da função densidade de probabilidade:

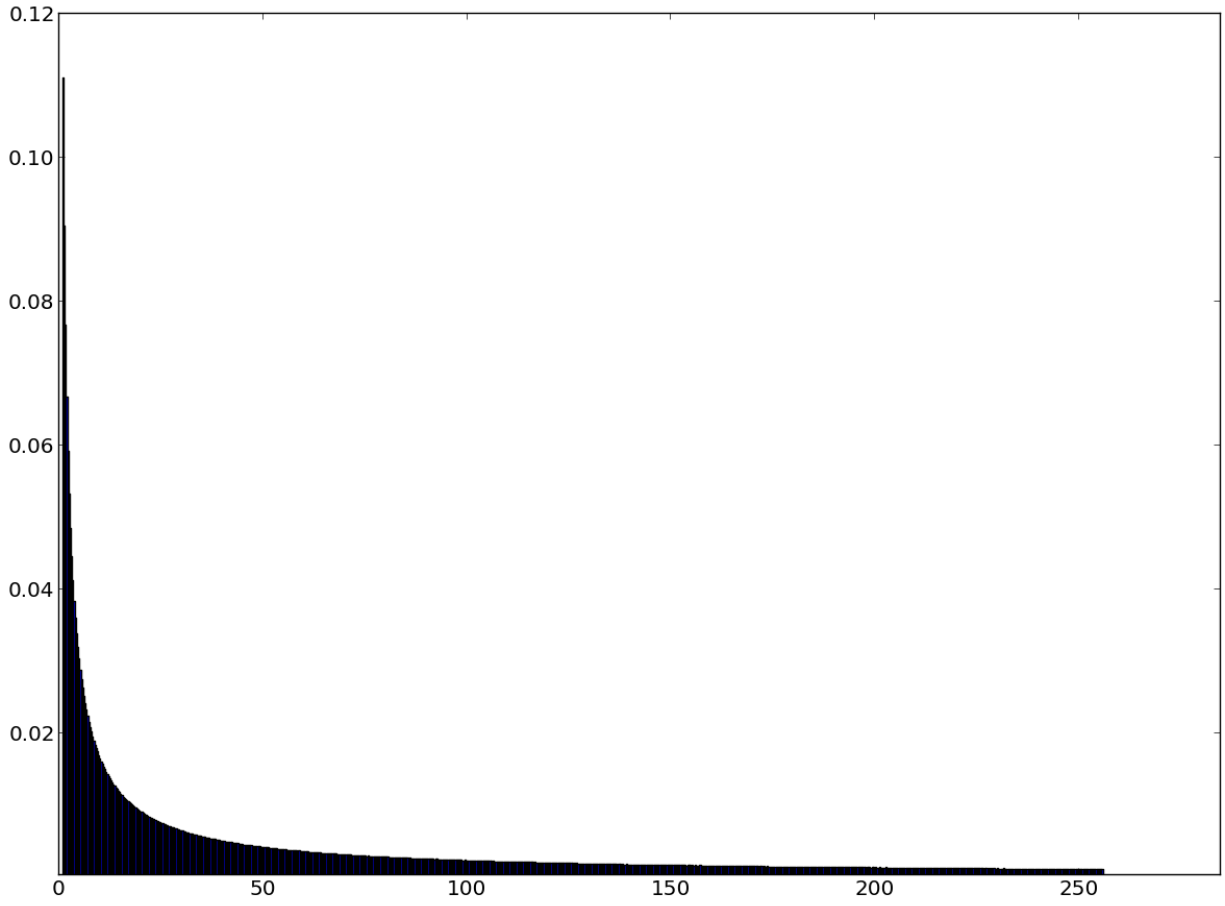

Figura 11 – Gráfico do histograma ótimo que representa a FDP de $e(t)$

A Tabela 4 mostra os tempos de execução.

Tabela 4 - Medição do cálculo da densidade de probabilidade de $\mathbf{e}(t) = (1+t)^8$ em paralelo

Nº da Execução	Real Time(s)	User Time(s)	Sys Time(s)
1	14.382	1.668	0.352
2	14.595	1.744	0.356
3	13.297	1.668	0.304
4	13.664	1.592	0.380
5	14.630	1.812	0.344
6	14.381	1.728	0.364
7	14.084	1.768	0.340
8	13.799	1.636	0.372
9	13.620	1.608	0.372
10	13.295	1.568	0.392
Média	13.974	1.679	0.357

Neste caso o *Real Time* não é igual ao *User Time* + *Sys Time*, pois maior parte do código é executado nas threads iniciadas do IPython.

4.4 Execução paralela $g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)$:

Custo mínimo: -48.806.116

Tamanho do intervalo de classe ótimo: 0,0707

Número de intervalos de classe para custo mínimo: 20

A Figura 12 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

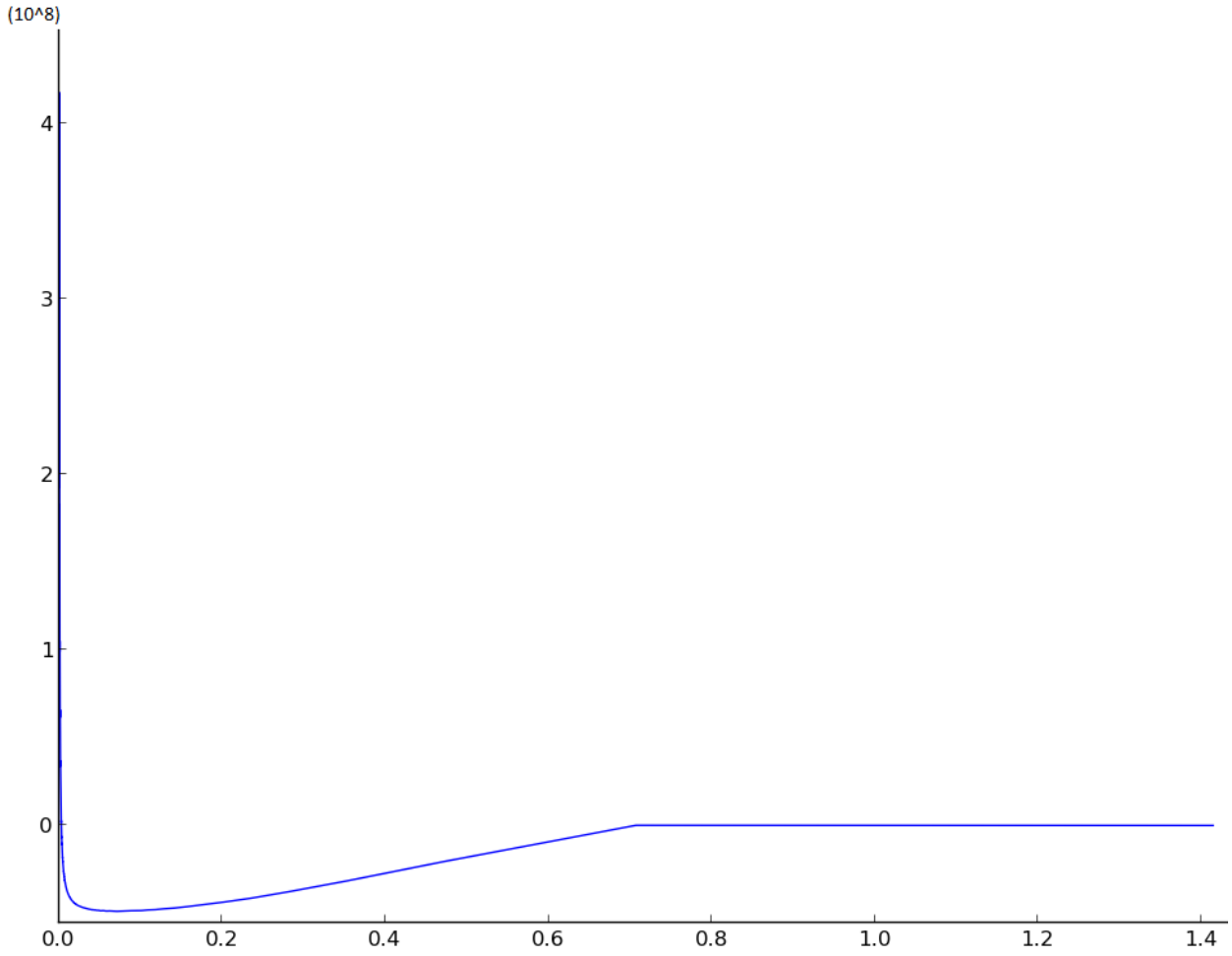

Figura 12 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a função $g(t)$

A Figura 13 mostra o gráfico da função densidade de probabilidade:

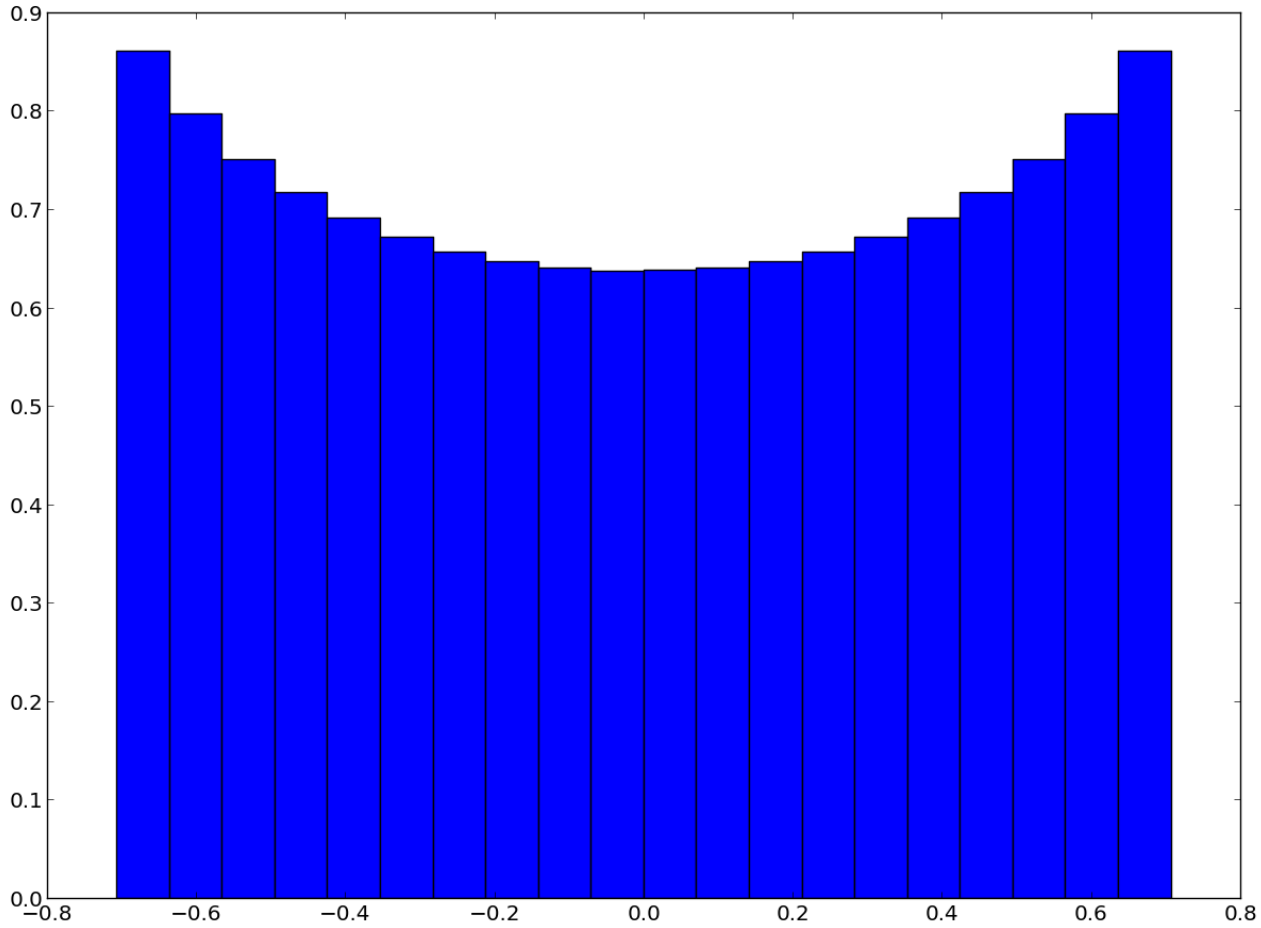

Figura 13 – Gráfico do histograma ótimo que representa FDP de g(t)

A Tabela 5 mostra os tempos de execução tomados.

Tabela 5 - Medição do cálculo da densidade de probabilidade de $g(t) = \cos\left(\frac{\pi}{4} + \frac{\pi}{2}t\right)$ em paralelo

Nº da Execução	Real Time(s)	User Time(s)	Sys Time(s)
1	13.704	1.624	0.376
2	13.511	1.600	0.388
3	13.653	1.676	0.320
4	13.562	1.608	0.400
5	13.706	1.724	0.336
6	13.763	1.652	0.348
7	13.547	1.64-	0.376
8	13.707	1.628	0.364
9	13.579	1.620	0.384
10	13.657	1.644	0.372
Média	13.639	1.642	0.366

4.5 Estimativa da FDP em dados obtidos experimentalmente

Para comprovar a utilidade deste método para dados obtidos na prática, foi utilizado um conjunto de dados obtidos a experimentalmente (MACIEL et al. ,2012). Este conjunto de dados obtidos, apresentados na Figura 14, representa a resposta motora das patas de um gafanhoto a um estimulo elétrico com distribuição gaussiana.

A partir deste conjunto, foram recortados quatro subconjuntos de dados a partir de 50s, com intervalos de, respectivamente, 0.1s, 0.2s, 0.3s e 0.5s. Estes sub-recortes foram feitos para verificar como o programa responde ao acréscimo de dados. A Figura 16 A, B, C, D mostra os recortes feitos no conjunto inicial de dados relativos respectivamente, a 0.1s, 0.2s, 0.3s e 0.5s

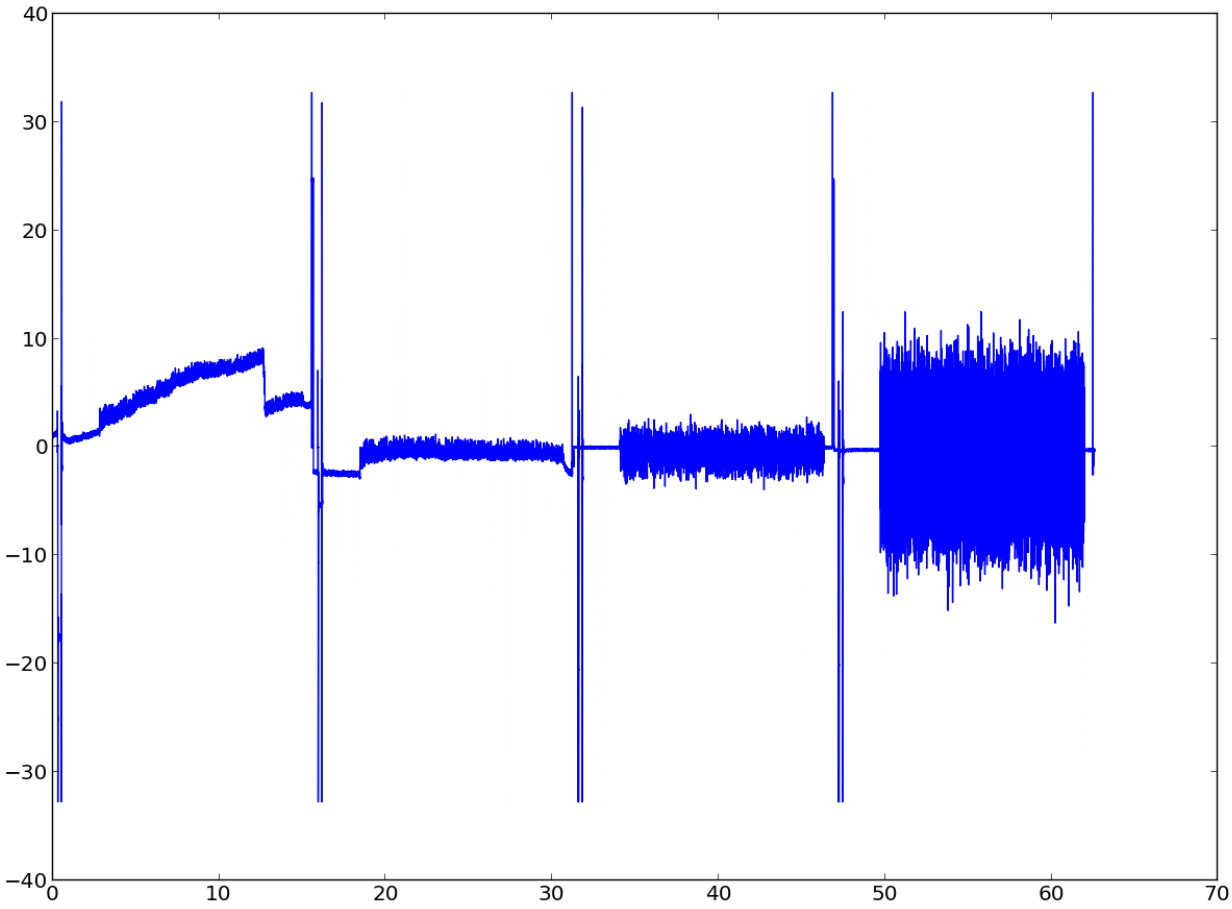

Figura 14 - Dados experimentais iniciais

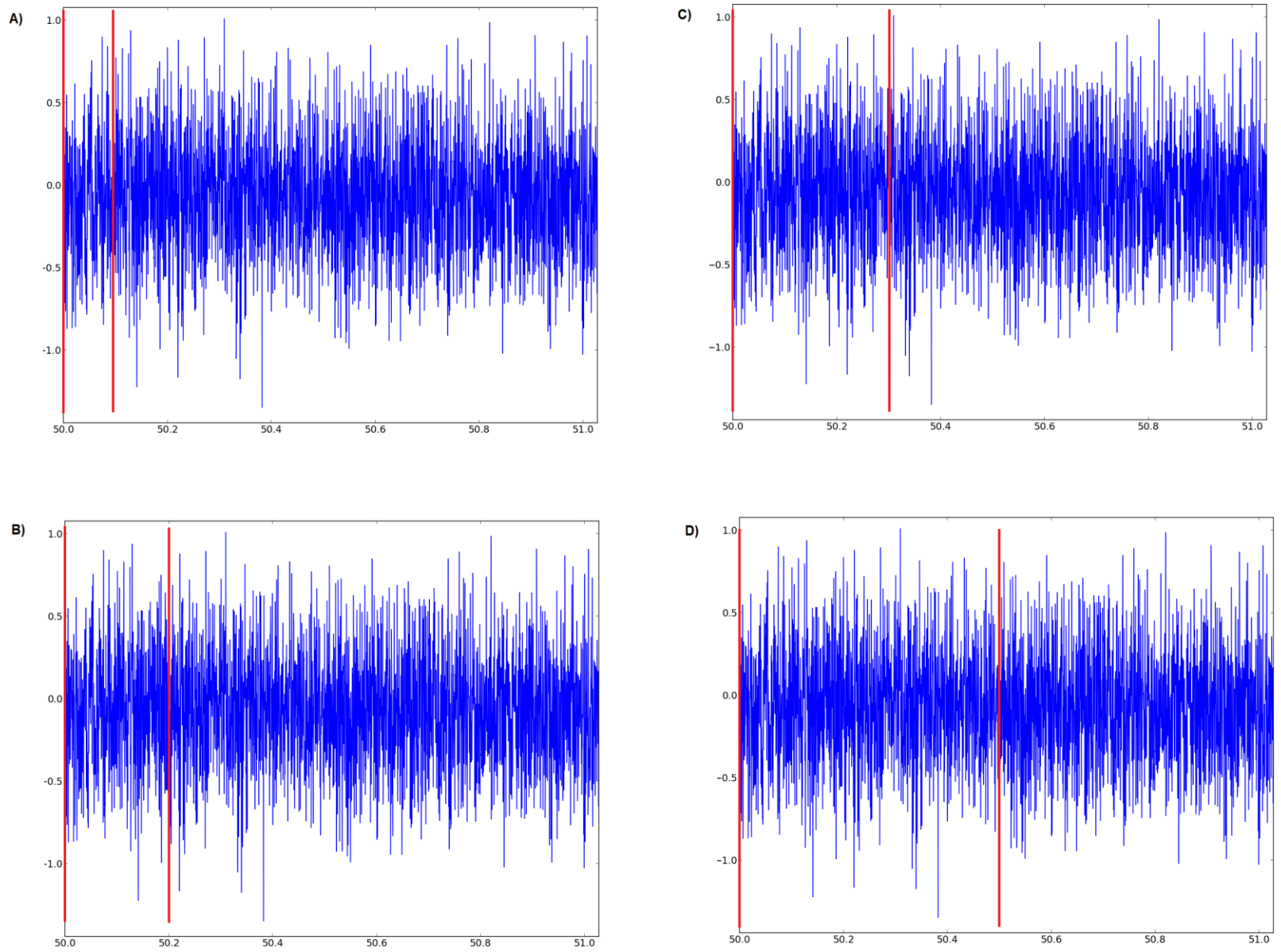

Figura 15 - Amostras A, B, C e D (entre delimitação em vermelho)

4.6 Execução para amostra A

Custo mínimo: -18.653.658

Tamanho do intervalo de classe ótimo: 0,0421

Número de intervalos de classe para custo mínimo: 42

A Figura 16 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

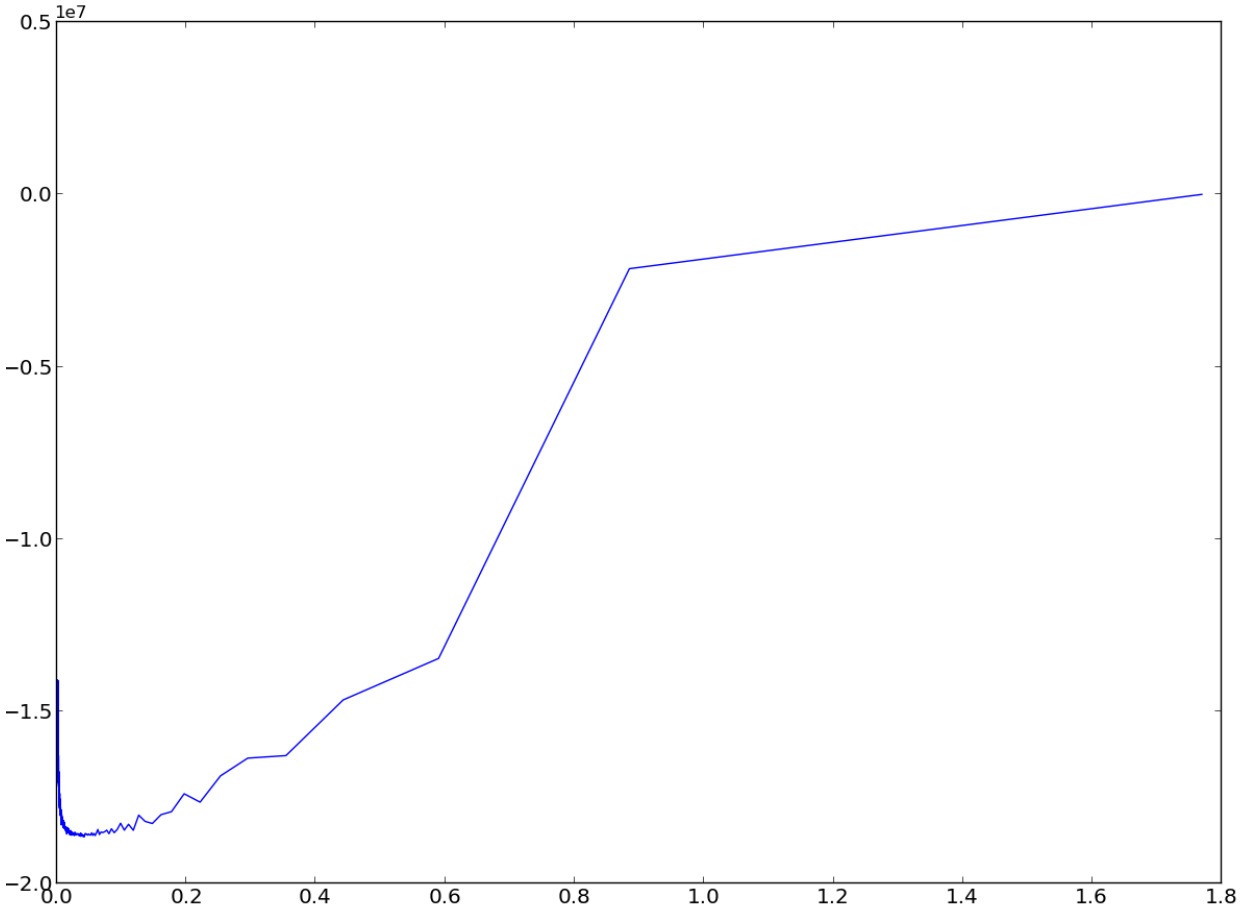

Figura 16- Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a amostra A

A Figura 17 mostra o gráfico da função densidade de probabilidade:

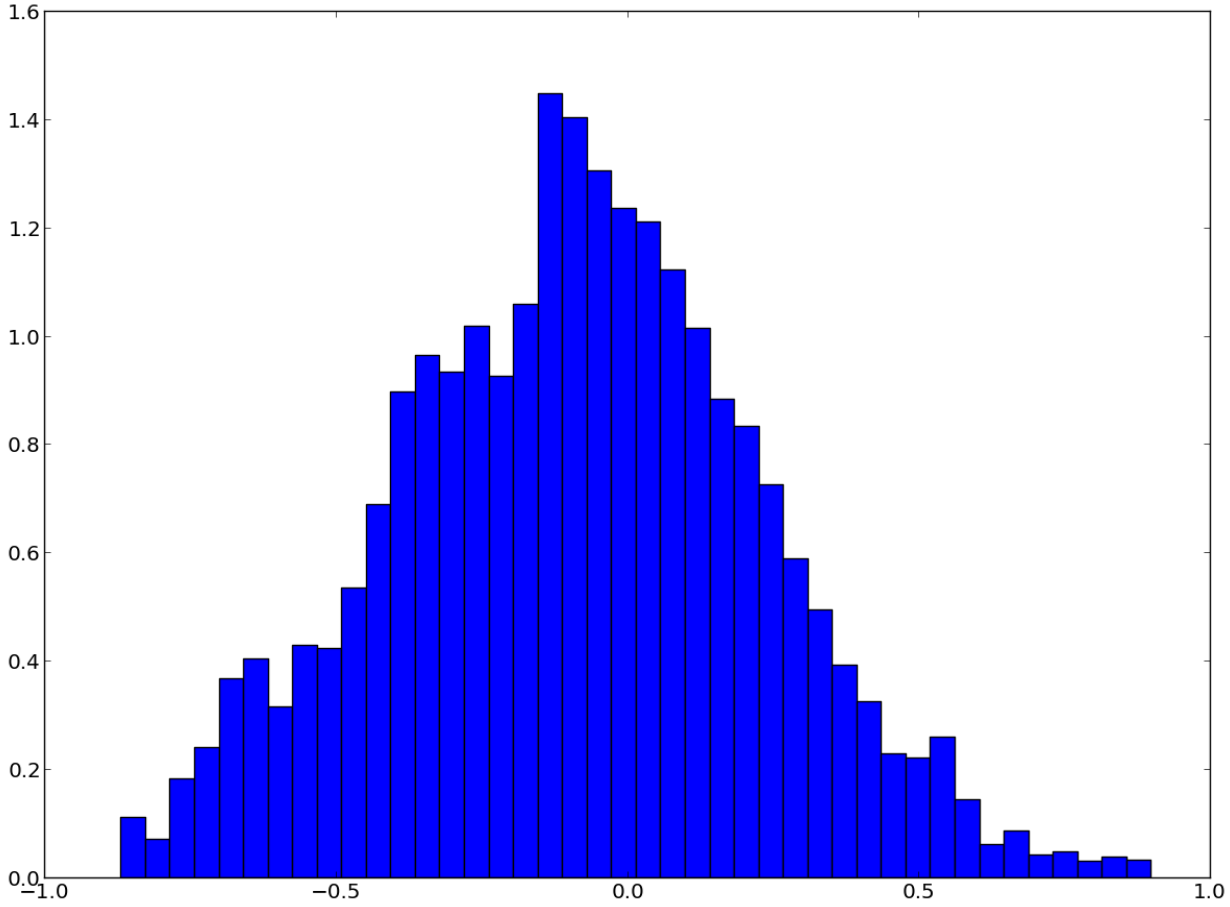

Figura 17 – Gráfico do histograma ótimo que representa a função FDP da amostra A

4.7 Execução para amostra B

Custo mínimo: -75.838.573

Tamanho do intervalo de classe ótimo: 0,0227

Número de intervalos de classe para custo mínimo: 95

A Figura 18 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

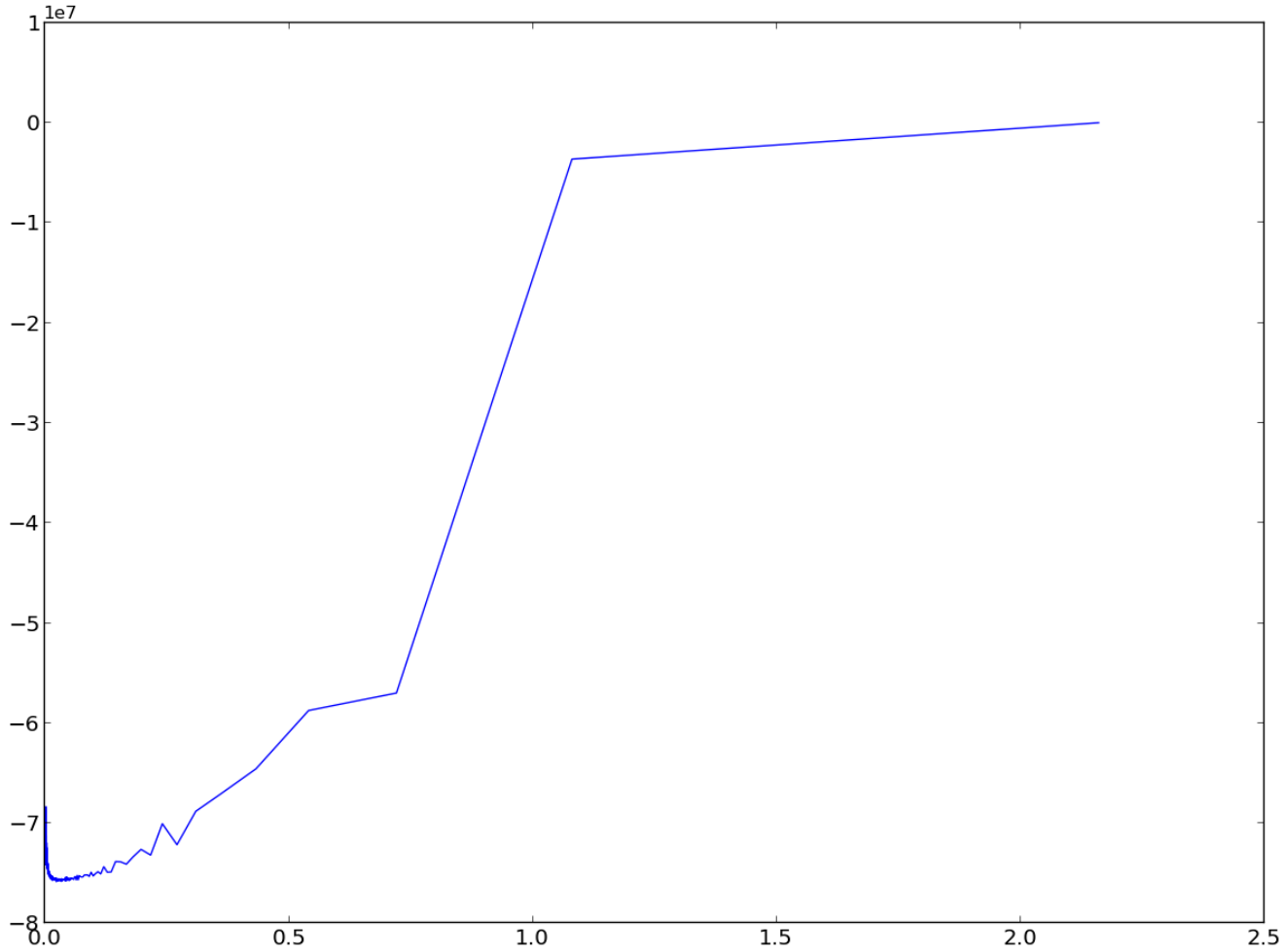

Figura 18 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a amostra B

A Figura 19 mostra o gráfico da função densidade de probabilidade:

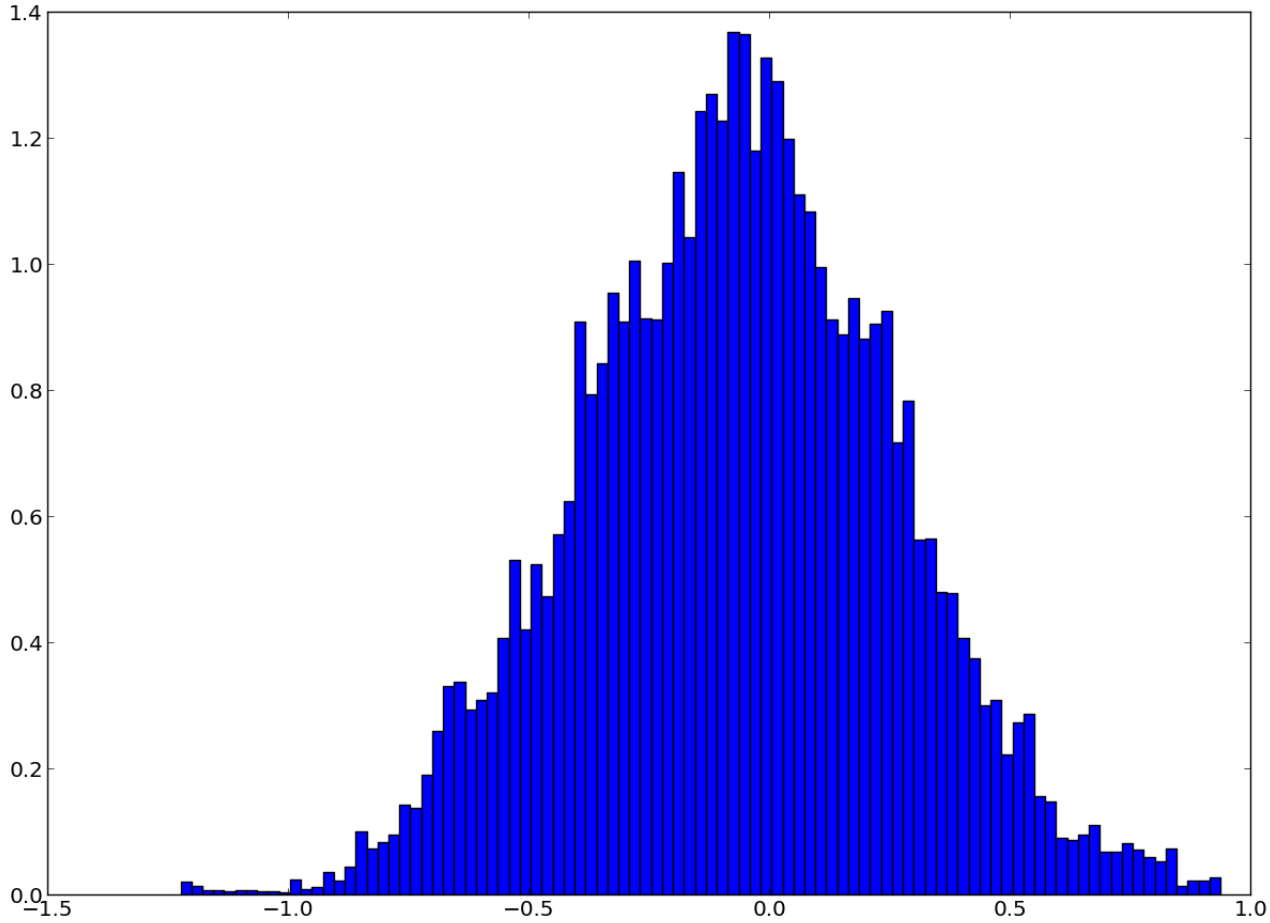

Figura 19 – Gráfico do histograma ótimo que representa a função FDP da amostra B

4.8 Execução para amostra C

Custo mínimo: -175.317.349

Tamanho do intervalo de classe ótimo: 0,0013

Número de intervalos de classe para custo mínimo: 1611

A Figura 20 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

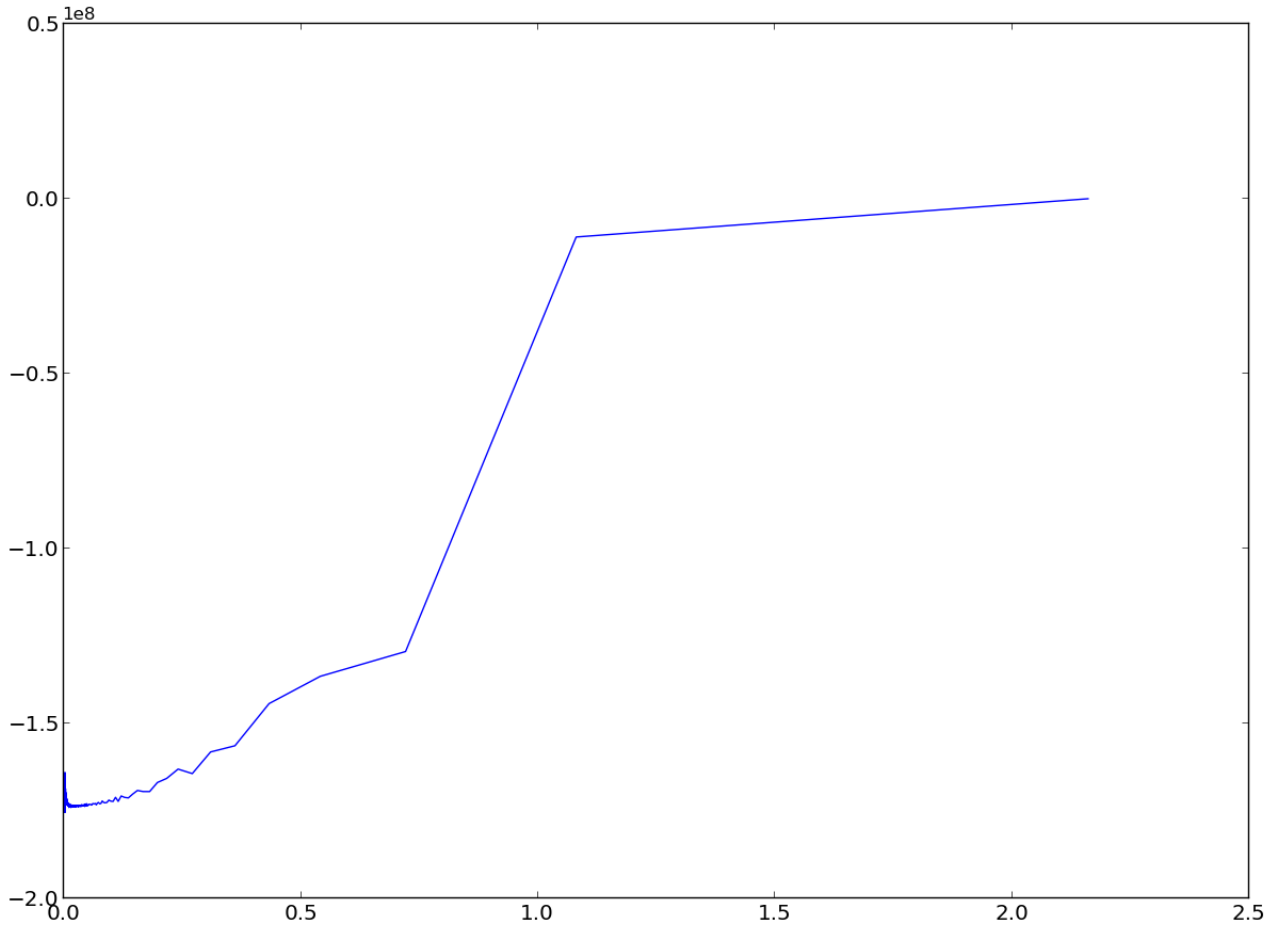

Figura 20 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a amostra C

A Figura 21 mostra o gráfico da função densidade de probabilidade:

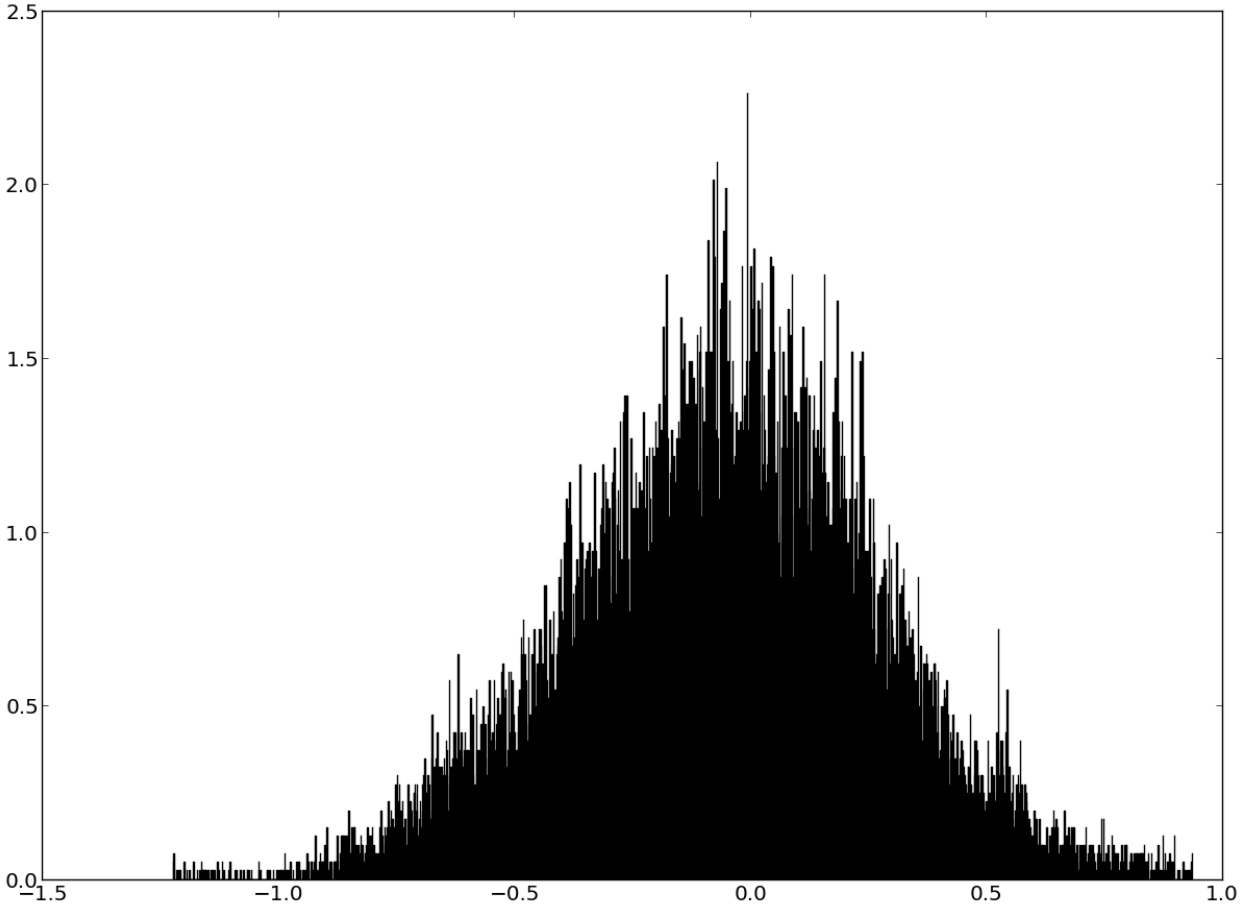

Figura 21 – Gráfico do histograma ótimo que representa a função FDP da amostra C

4.9 Execução para amostra D

Custo mínimo: -536.809.220

Tamanho do intervalo de classe ótimo: 0,0012

Número de intervalos de classe para custo mínimo: 1965

A Figura 22 mostra o gráfico da função custo em função to tamanho do intervalo de classe:

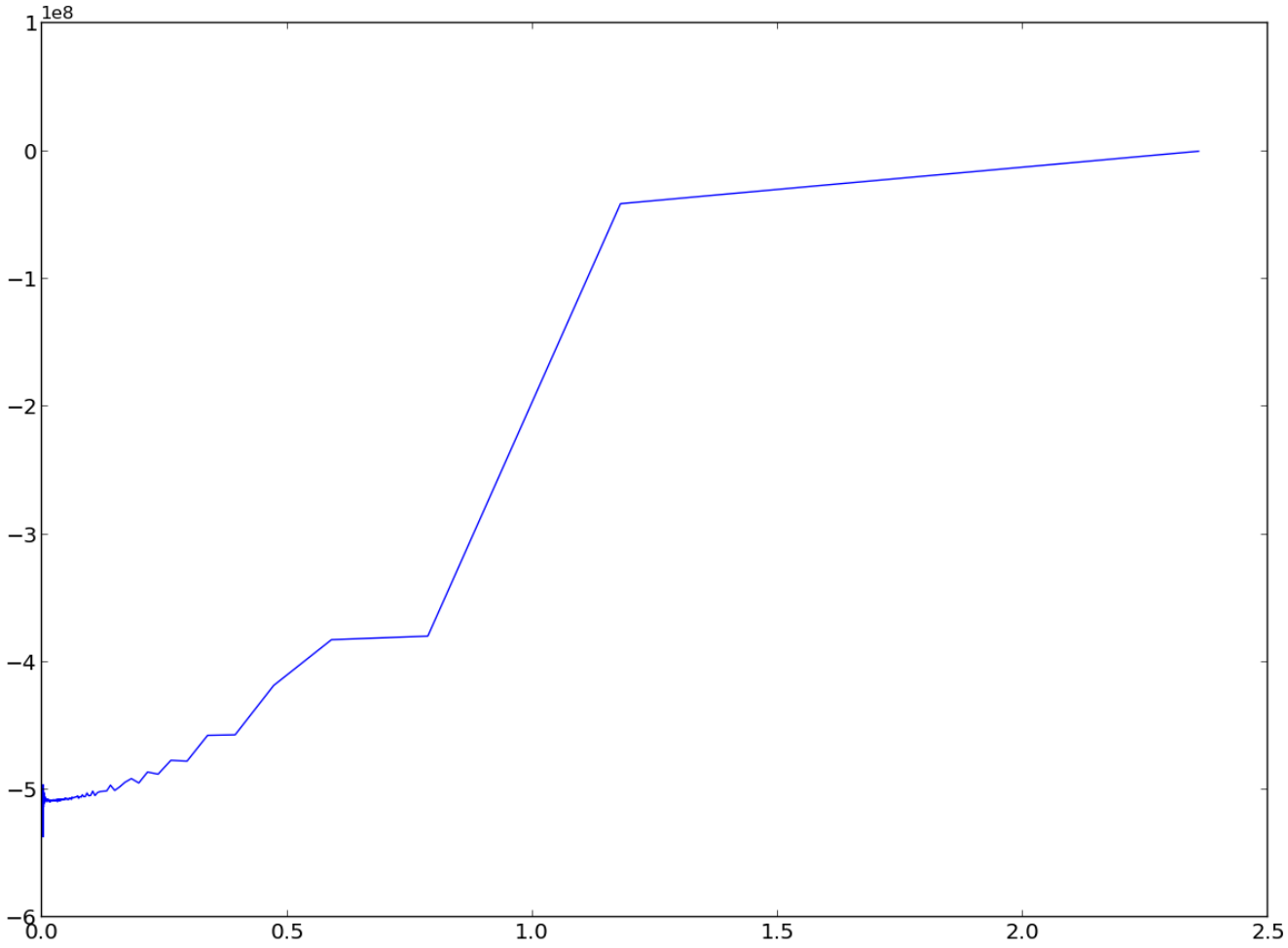

Figura 22 - Gráfico da função Custo, $C(\Delta)$, pelo intervalo de classe, Δ, para a amostra D

A Figura 23 mostra o gráfico da função densidade de probabilidade:

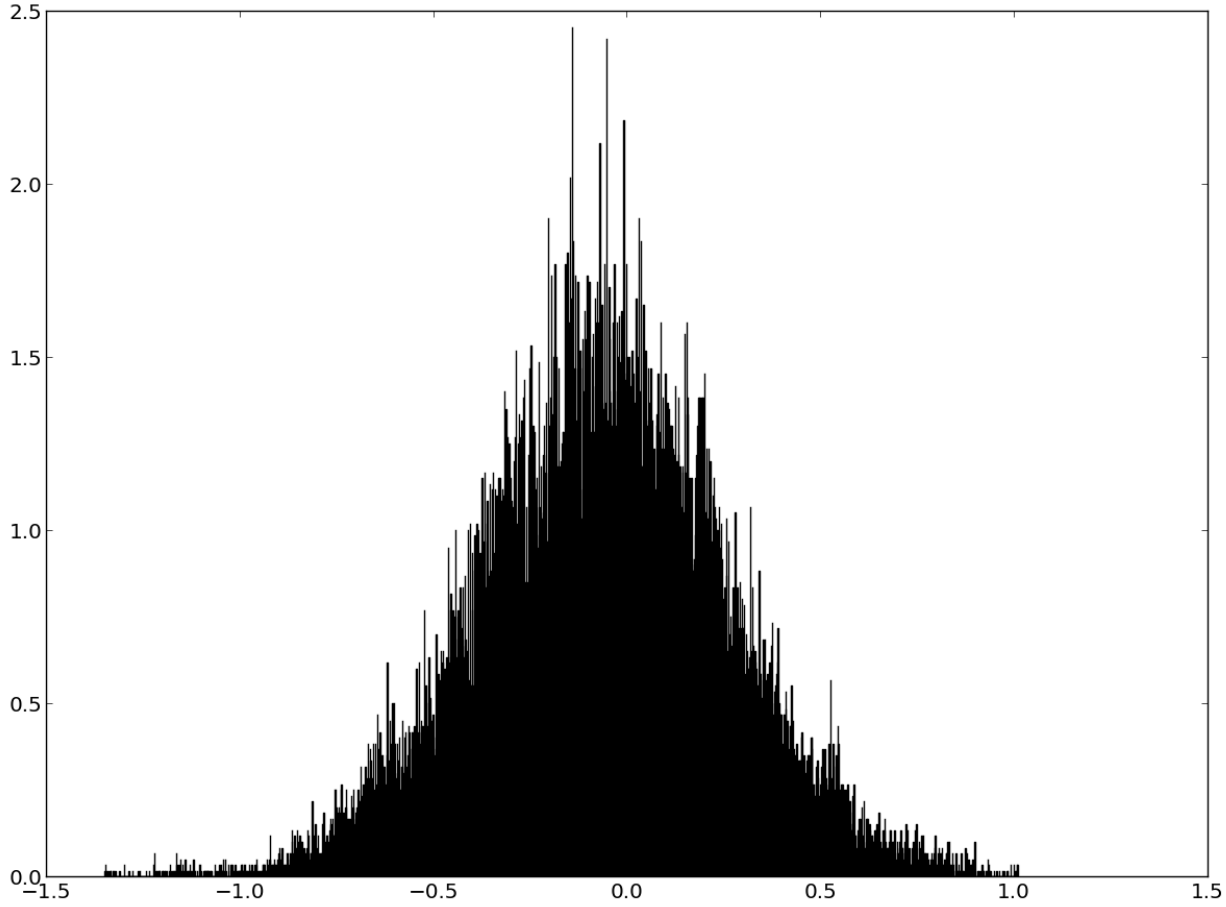

Figura 23 – Gráfico do histograma ótimo que representa a função FDP da amostra D

5 Análise de Resultados

5.1 Validação teórica do Algoritmo

O cálculo da FDP das funções $e(t)$ e $g(t)$ foi muito próximo ao FDP teórico destas funções. As Figuras 24 e Figura 25 mostram uma superposição dos gráficos das FDPs analíticas e empíricas de $e(t)$ e $g(t)$, respectivamente. A linha em vermelho é a teórica dada pelas fórmulas abaixo:

$$FDP\big(y = e(t)\big) = \frac{y^{\frac{-7}{8}}}{8} \qquad Eq\ 5.1$$

$$FDP\big(y = g(t)\big) = \frac{2}{\pi\sqrt{1 - y^2}} \qquad Eq\ 5.2$$

Na implementação em paralelo, os resultados obtidos e apresentados nas seções 4.3 e 4.4, com exceção do tempo de execução, foram exatamente iguais aos obtidos em serial, confirmando a assertividade do paralelismo. Os gráficos da função custo e o histograma final também foram idênticos.

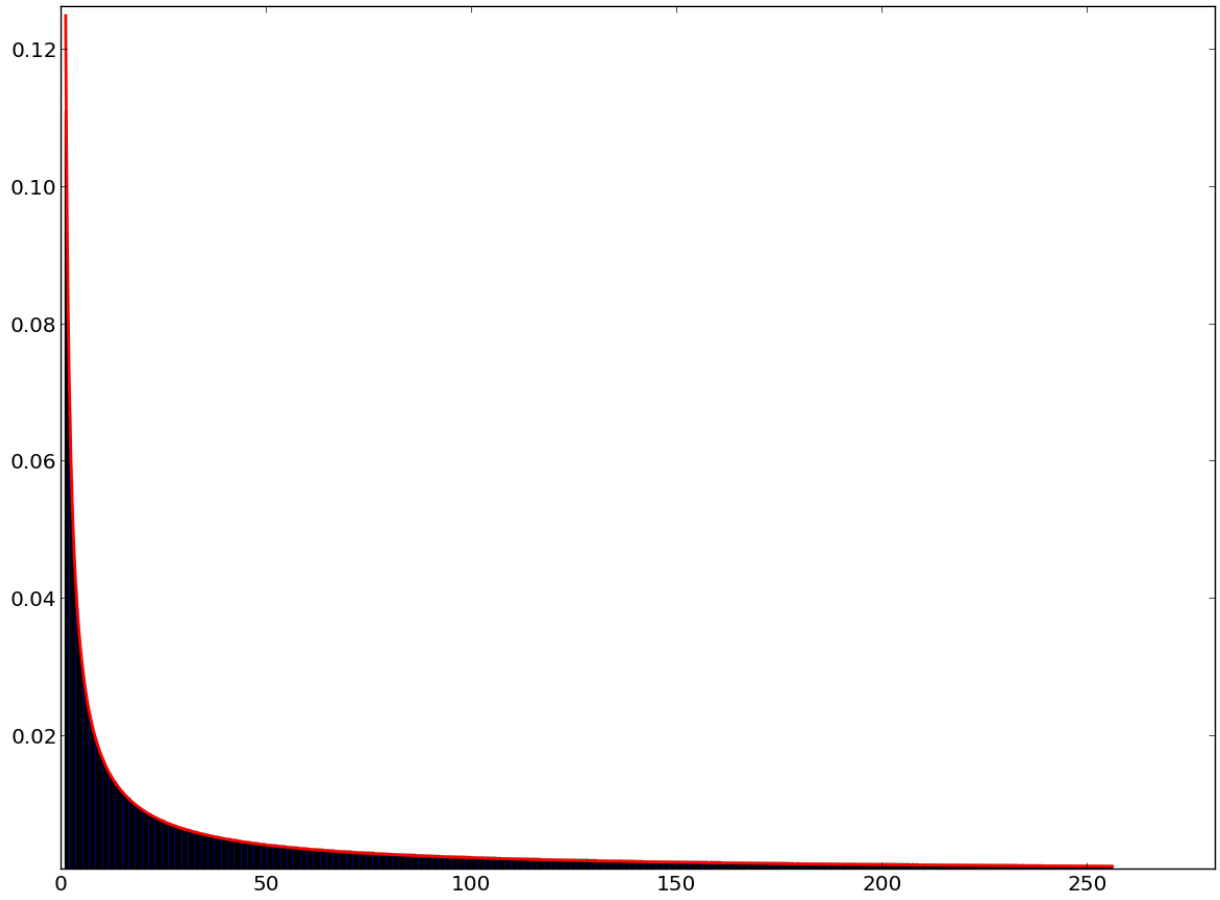

Figura 24 - Gráfico da FDP teórica de $e(t)$ (vermelho) pelo histograma obtido pelo programa (preto)

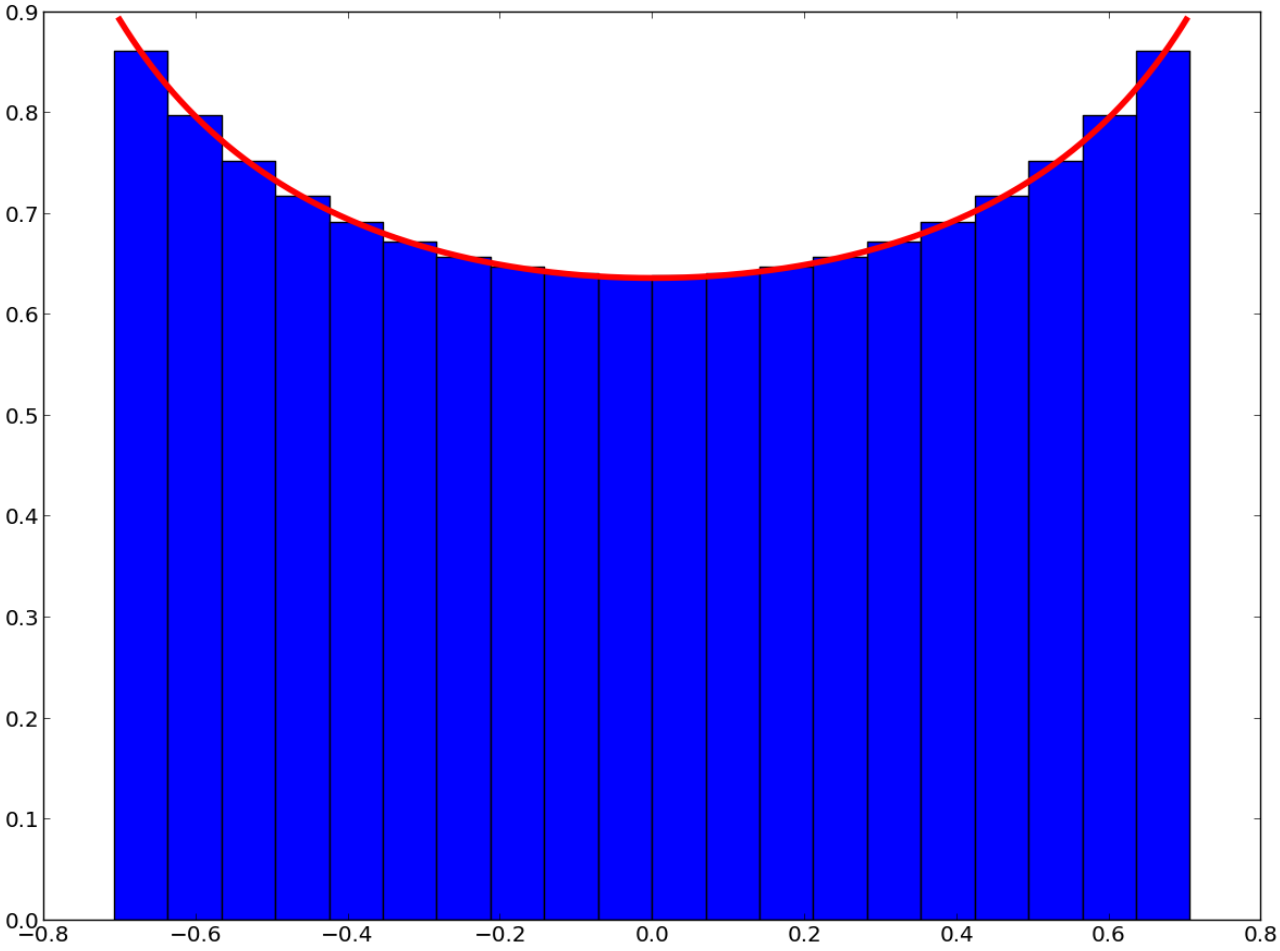

Figura 25 - Gráfico da FDP teórica de g(t) (vermelho) pelo histograma obtido pelo programa (azul)

5.2 Eficiência do Algoritmo

As Tabelas 6 e 7 mostram um comparativo dos resultados serial e paralelo.

Tabela 6 - Tempo de execução médio de e(t) e g(t)

Tempo de execução médio:	Serial(s)	Paralelo(s)
e(t)	20.728	13.974
g(t)	21,537	13.638

Tabela 7 - Ganho de desempenho com utilização de paralelismo

	Ganho com Paralelismo
e(t)	**1.48**
g(t)	**1.57**
Ganho Médio	**1.53**

A utilização do paralelismo permitiu, portanto, um ganho em torno de 50%. Apesar de razoável, não chega próximo à duas vezes. A causa desta limitação pode ser atribuída a dois fatores principais: a execução de uma parte do código em serial e à comunicação interna da ferramenta IPython. Este trabalho utilizou dados que levaram um tempo curto para serem processados, porém caso seja necessário mais tempo de processamento o ganho será maior.

5.3 Resultados com dados empíricos

Percebe-se, para os dados experimentais, que o algoritmo funcionou bem, construindo um histograma razoavelmente próximo de uma distribuição gaussiana e sem grandes flutuações até a amostra contendo 20.000 dados (amostra B). Com 30.000 dados em diante, nota-se que o algoritmo não encontrou um custo mínimo. A função Custo deveria se comportar-se como na Figura 26. Neste caso ideal a função custo vai diminuindo de acordo com a diminuição com a largura do intervalo de classe até chegar a um mínimo e depois começa a aumentar novamente. Já nos histogramas das Figuras 21 e 23 (amostras com 30.000 e 50.000 dados) a função Custo vai diminuindo, depois oscila uma pouco e começa a descer abruptamente. Mesmo aumentando o número de iterações do algoritmo a função custo continuou descendo mais. Portanto, não deve existir custo mínimo neste caso e isso explica as grandes flutuações nos histogramas encontrados.

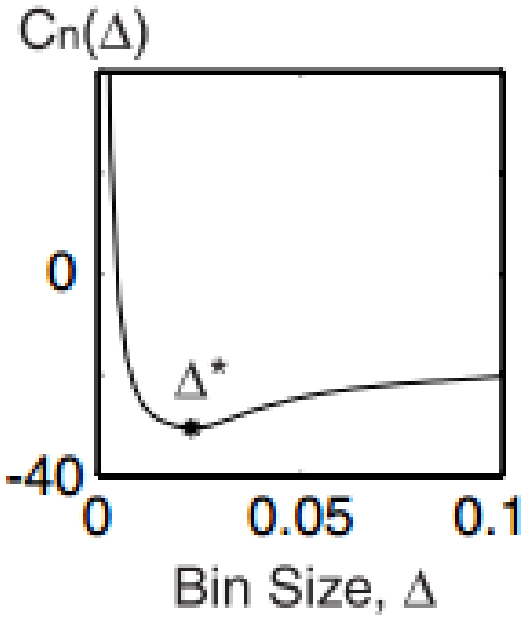

Figura 26 – Comportamento ideal da função Custo em relação à largura do intervalo (Δ) (SHIMAZAKI;SHINOMOTO, 2007)

6 Conclusão

Este método de cálculo de um histograma ótimo provou-se viável de ser utilizado, apesar de possuir suas restrições. Ele não pode ser utilizado para todas as situações e quaisquer amostras de dados. Saber escolher a quantidade certa de dados, amostrados de forma suficientemente rápida (gerando boas quantidades de dados não repetidos) é crucial para conseguir a resposta adequada.

Ademais, as ferramentas utilizadas neste trabalho (IPython e bibliotecas cientificas), mostraram-se simples e eficientes, apesar de ser necessário uma configuração inicial maior para rodar em duas maquinas diferentes, os ganhos de uma computação paralela com código enxuto justificam sua utilização.

Vale ressaltar, também, que é importante saber se a FDP em questão tende ao infinito em algum ponto e se possui variações muito bruscas, pois neste caso a utilização deste algoritmo necessitará de dados amostrados ainda mais rapidamente e, mesmo assim, pode não apresentar o resultado esperado.

7 Referências Bibliográficas

C. D. MACIEL, D. M. SIMPSON E P.L NEWLAND. Inference about multiple pathways in motor control limb in locust. Biosignals SciTePress (2012) , p. 69-75.

ERIC JONES, TRAVIS OLIPHANT, PEARU PETERSON e outros. Scipy: Open Source Scientific Tools for Python. 2001. URL: http://www.scipy.org

FERNANDO PEREZ, BRIAN E. GRANGER, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, Junho 2007, doi:10.1109/MCSE.2007.53. URL: http://ipython.org

HEEGER, David. Poisson Model of Spike Generation. 5 de Setembro de 2000. Disponível em: http://www.cns.nyu.edu/~david/handouts/poisson.pdf. Acessado em 01 de Novembro de 2012.

NUMPY DEVELOPERS. Scientific Computing Tools for Python – Numpy. Disponível em: http://numpy.scipy.org . Acesso em Outubro de 2012

PERFORMANCEPYTHON. Disponível em: http://www.scipy.org/PerformancePython. Acessado em Outubro de 2012

PYTHON SOFTWARE FOUNDATION. Python Programming Language – Official Website. Disponível em: http://www.python.org. Acesso em Outubro de 2012

SHIMAZAKI H. e SHINOMOTO S., A method for selecting the bin size of a time histogram. Neural Computation (2007) Vol. 19(6), 1503-1527

SILVERMAN, B.W.. *Density Estimation for Statistics and Data Analysis*. Monographs on Statistics and Applied Probability, London: Chapman and Hall, 1986. Disponível em: http://ned.ipac.caltech.edu/level5/March02/Silverman/paper.pdf.

ZEROMQ: The Intelligent Transport Layer. Disponível em: http://www.zeromq.org/. Acessado em Outubro de 2012