Relatório Exercício-Programa 2
MAP - 3121
Métodos Numéricos e Aplicações

Lui Damianci Ferreira - 10770579
Victor A. C. Athanasio - 9784401

1 de Maio de 2020

Resumo

Este relatório, existe como parte integrante da nossa solução para o Exercício Programa 2 proposto pela matéria MAP-3121, Métodos Numéricos e Aplicações.

Nele está documentado não somente o nosso entendimento do problema proposto como também nossa proposta para solução do mesmo e analise sobre os resultados provenientes da aplicação de métodos numéricos.

Tanto o script em **Python** quanto todas as imagens geradas encontram-se em anexo no arquivo **.zip** no qual está contido este relatório.

Conteúdo

1 Introdução

Neste execício-programa (EP), o principal intuito é, lidar com problemas inversos. Após o extensivo estudo sobre problemas diretos, durante a execução do EP1, este EP2 apresenta-se como uma clara continuação. O problema que pretendemos resolver é, dada as medições de temperatura em uma barra no instante T, tendo delimitada as possíveis fontes pontuais iniciais, qual a intensidade [1] de cada uma delas que minimiza um erro quadrático? Posto desta forma, o problema pode ser descrito como um problema de mínimos quadrados.

2 Heranças EP1

Como pedido no enunciado do EP2, algumas funções e funcionalidades deveriam ser trazidas do EP1, sobretudo o método de resolução do problema direto de Crank-Nicolson. Devido a esta necessidade, uma versão *light* do EP1 foi criada. Está versão possui apena o mínimo essencial para o funcionamento do EP2. Os outros métodos de resolução foram retirados e pequenas adaptações foram feitas. Um exemplo destas adaptações é a reorientação do vetor que representa o estado final da barra, para que o mesmo fosse condizente com o requisitado pelo EP2.

A interface entre os EPs foi feita através de uma classe batizada de *crank_nicolson*, que recebe como parâmetros de entrada N, o número de discretizações no domínio do espaço, e P, a posição da fonte pontual que desejamos testas. Está classe possui um método, *execute*, que executa em sua totalidade o algoritmo de Crank-Nicolson, e retorna o vetor que contém o estado final da barra no instante T.

3 Estrutura do código

A estrutura do código foi dividida em 4 arquivos, *Ep1.py*, *Ep2_functions.py*, *testes.py* e *Ep2.py*. O primeiro arquivo contém as heranças do EP1. O segundo arquivo, por sua vez, contém as implementações das tarefas 'A', 'B' e 'C'. O terceiro arquivo contém os modelos de testes propostos e as análises. E, finalmente, o último arquivo possui a interface final com o usuário, e ele é o que deve ser executado.

O *Ep1.py* possui uma estrutura interna baseada em POO (programação orientada a objetos) para maior facilidade dos diversos dados gerados e utilizados durante os procedimentos de Crank-Nicolson.

Os demais arquivos, devido a menor complexidade, e clara segmentação das instruções do EP2, foram construídos baseados em funções, onde cada uma implementa algo que foi pedido no enunciado.

4 Tarefa A

O principal objetivo desta tarefa era a implementação de um código que gerasse os $u_k(T, x_i)$, i = 1, ..., N - 1. Tal objetivo foi cumprido pela implementação da função "*create_us*", que pode ser vista abaixo, e é o ponto de interação entre o EP2 e o EP1.

```
def create_us(plist, N):
    '''Recebe uma lista de pontos e calcula os vetores U's correspondentes
    Recebe tambem o parametro N
    Devolve os vetores U dentro de um vetor maior, cada vetor U é um vetor
↪   coluna.
```

[1] Por intensidade, leia-se os coeficientes a_k

```python
    '''
    ulist = []
    for p in plist:
        u = crank_nicolson(N, p).execute()[1:-1] #slicing para retirada das
        ↪ bordas
        ulist.append(u)
    return np.array(ulist)
```

5 Tarefa B

O principal objetivo desta tarefa era a montagem da matriz e do sistema normal do problema de mínimos quadrados proposto. Tal objetivo foi atingido pela implementação de uma função capaz de calcular o produto interno entre 2 vetores, de uma função para a criação da matriz e uma para a criação do lado direito do sistema. O código[2] pode ser visto abaixo.

```python
@vectorize()
def _prod_interno(x, y):
    '''Executa multiplicacao element_wise dentre 2 vetores'''
    return x * y


@njit(parallel=True)
def prod_interno(x, y):
    '''Executa o produto interno entre 2 vetores'''
    vec = _prod_interno(x, y)
    vec = np.sum(vec)
    return vec


@njit(parallel=True)
def create_matrix_MMQ(uarray):
    '''Funcao que cria a matriz do MMQ
    Ela basicamente executa os produtos internos corretos, na metade inferior
↪   da matrix
    soma a matrix com sua transposta, tirando vantagem da natureza simetrica
↪   do problema,
    para executar menos calculos.'''
    shape = uarray.shape[0]
    matrix = np.zeros((shape, shape))
    for i in range(shape):
        for j in range(shape):
            if j > i:
                matrix[i][j] = prod_interno(uarray[i], uarray[j])
    matrixt = matrix.transpose()
    matrix += matrixt
    for i in range(shape):
```

[2]As *flags* que antecedem as funções, são sinalizadores de uma biblioteca chamada Numba. Esta biblioteca é responsável pelo pela compilação do código de python, em tempo de execução, para linguagem de máquina. Este processo adiciona *overhead* no tempo de execução, porém para múltiplos testes, diminui consideravelmente o tempo total de execução. Mais detalhes podem ser vistos no relatório referente ao EP1. O tempo de execução total foi de 579 segundos, frente aos 851 sem tal otimização

```
            matrix[i][i] = prod_interno(uarray[i], uarray[i])
    return matrix


@njit(parallel=True)
def create_right_side_MMQ(uarray, ut):
    ''''funcao que cria o lado direito do sistema MMQ, funciona de forma
    ↪   analoga
    a funcao que cria a matrix do sistema MMQ'''
    shape = uarray.shape[0]
    matrix = np.zeros(shape)
    for i in range(shape):
        matrix[i] = prod_interno(uarray[i], ut)
    return matrix
```

6 Tarefa C

O principal objetivo desta tarefa era a resolução do sistema do MMQ através de uma decomposição LDL^t. Para a resolução desta tarefa, foi necessário descobrir-se a relação entre uma decomposição LDL^t e uma matriz A genérica. Para tal, executou-se a multiplicação descrita em 1, que possui como resultado a matriz 2[3].

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ L_{10} & 1 & 0 & 0 & 0 \\ L_{20} & L_{21} & 1 & 0 & 0 \\ L_{30} & L_{31} & L_{32} & 1 & 0 \\ L_{40} & L_{41} & L_{42} & L_{43} & 1 \end{bmatrix} * \begin{bmatrix} D_0 & 0 & 0 & 0 & 0 \\ 0 & D_1 & 0 & 0 & 0 \\ 0 & 0 & D_2 & 0 & 0 \\ 0 & 0 & 0 & D_3 & 0 \\ 0 & 0 & 0 & 0 & D_4 \end{bmatrix} * \begin{bmatrix} 1 & L_{10} & L_{20} & L_{30} & L_{40} \\ 0 & 1 & L_{21} & L_{31} & L_{41} \\ 0 & 0 & 1 & L_{32} & L_{42} \\ 0 & 0 & 0 & 1 & L_{43} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} D_0 & D_0L_{10} & D_0L_{20} & \dots \\ D_0L_{10} & D_0L_{10}^2 + D_1 & D_0L_{10}L_{20} + D_1L_{21} & \dots \\ D_0L_{20} & D_0L_{10}L_{20} + D_1L_{21} & D_0L_{20}^2 + D_1L_{21}^2 + D_2 & \dots \\ D_0L_{30} & D_0L_{10}L_{30} + D_1L_{31} & D_0L_{20}L_{30} + D_1L_{21}L_{31} + D_2L_{32} & \dots \\ D_0L_{40} & D_0L_{10}L_{40} + D_1L_{41} & D_0L_{20}L_{40} + D_1L_{21}L_{41} + D_2L_{42} & \dots \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} A_{00} & A_{10} & A_{20} & A_{30} & A_{40} \\ A_{10} & A_{11} & A_{21} & A_{31} & A_{41} \\ A_{20} & A_{21} & A_{22} & A_{32} & A_{42} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{43} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \tag{3}$$

Comparando a matriz 2 com a matriz 3 pudemos extrair as relações 4 e 5. Analisando ambas relações com cuidado, é possível notar que o algoritmo para definir todas as variáveis é, primeiro encontrar D_i e depois encontrar L_{ki} com K: i+1 $\rightarrow$ n e i: 0 $\rightarrow$ n. Com n sendo a dimensão da matriz A_{nxn}

$$L_{xy} = \frac{(A_{xy} - \sum_{k=0}^{x-1} D_k * L_{xk} * L_{yk})}{D_y} \tag{4}$$

[3] A matriz 2 deixou bastante evidente que decomposições deste tipo são eficazes somente para matrizes simétricas

$$D_z = A_{zz} - \sum_{k=0}^{z-1} D_k * L_{zk}^2 \tag{5}$$

A conversão deste código para *python* pode ser vista abaixo.

```
@njit()
def achaLxy(matrix, L, D, x, y):
    '''Encontra o termo Lxy mais detalhes sobre este termo podem ser vistas no relatório'''
    Lxy = matrix[x, y]
    for k in range(x):
        Lxy -= D[k] * L[x, k] * L[y, k]
    return Lxy / D[y]


@njit()
def achaDz(matrix, L, D, z):
    '''Encontra o termo Dz mais detalhes sobre este termo podem ser vistas no relatório'''
    Dz = matrix[z, z]
    for k in range(z):
        Dz -= D[k] * L[z, k] ** 2
    return Dz

@njit(parallel=True)
def LDLt(matrix):
    '''Encontra as matrizes L e D, vale notar que D foi armazena em um vetor. Isso foi feito para eficiencia
    computacional, uma vez que a matriz D é esparsa '''
    size = matrix.shape[0]
    L = np.zeros((size, size))
    D = np.zeros(size)
    for i in range(size):
        D[i] = achaDz(matrix, L, D, i)
        L[i, i] = 1
        for k in range(i + 1, size):
            L[k, i] = achaLxy(matrix, L, D, k, i)
    return L, D
```

A partir da decomposição LDL^t, a solução passa a ter 3 etapas e ser trivial. Tomando como sistema original o sistema 6, podemos fazer as substituições 7, 8 e 9. Podemos resolver os sistemas para Y, Z e por fim para X. A tradução deste algoritimo para *python* pode ser vista abaixo

$$A * X = b \tag{6}$$
$$A = L * D * L^t \tag{7}$$
$$D * L^t * X = Y \tag{8}$$
$$L^t * X = Z \tag{9}$$

```
@njit(parallel=True)
def resolve_U(matrix, b):
```

```
    '''Resolve sistemas superiores'''
    A = matrix
    x = np.zeros(b.shape[0])
    for j in range(A.shape[0] - 1, -1, -1):
        A_row = A[j, :]
        SUM = prod_interno(A_row, x)
        x[j] = b[j] - SUM
        x[j] = x[j] / A_row[j]
    return x


@njit(parallel=True)
def resolve_L(matrix, b):
    '''Resolve sistemas inferiores'''
    A = matrix
    x = np.zeros(b.shape[0])
    for j in range(A.shape[0]):
        A_row = A[j, :]
        SUM = prod_interno(A_row, x)
        x[j] = b[j] - SUM
        x[j] = x[j] / A_row[j]
    return x


@njit(parallel=True)
def resolve_D(D, b):
    '''Resolve sistemas diagonains'''
    return b / D


@njit(parallel=True)
def resolve_LDLt(L, D, b):
    '''Aplica as resolucoes sucessivamente, para resolver um sistema LDLt'''
    b1 = resolve_L(L, b)
    b2 = resolve_D(D, b1)
    b3 = resolve_U(L.transpose(), b2)
    return b3
```

Uma função extra foi criada, para integrar a solução completa do MMQ em um único ambiente. Esta função pode ser vista abaixo.

```
def resolveMMQ(plist, N, uT):
    '''Agrupa os metodos anteriores sob uma unica funcao'''
    uarray = create_us(plist, N)
    matrix = create_matrix_MMQ(uarray)
    b = create_right_side_MMQ(uarray, uT)
    L, D = LDLt(matrix)
    resp = resolve_LDLt(L, D, b)
    return resp, uarray
```

7 Testes

Todos os testes foram estruturados e encontram-se em *Testes.py*. Para os testes C e D, foi necessária uma função que lesse o arquivo *teste.txt* disponibilizado no moodle. Tal função encontra-se abaixo.

```
def read_text(N):
    '''Le o arquivo de teste disponibilizado no moodle, e retorna o vetor U e
    ↪ a posição das fontes'''
    mod = 2048 // N

    f = open("teste.txt", "r")

    positions = f.readline().split()
    for i in range(len(positions)):
        positions[i] = float(positions[i])

    uT = f.read().splitlines()
    uT_serialized = []
    for i in range(len(uT)):
        if i % mod == 0:
            uT_serialized.append(float(uT[i]))

    f.close()

    return positions, np.array(uT_serialized)[1:-1].reshape((N - 1, 1))
```

7.1 Análises realizadas

Em todos os testes, diversas análises foram efetuadas. Para começar, a análise do Erro quadrático. Ela foi efetuada da seguinte forma:

1. Calculo da solução pelo MMQ
2. Subtração do uT fornecido, pela solução calculada
3. Elevação dos termos ao quadrado
4. Soma dos termos
5. Multiplicação por ΔX
6. Retirada da raiz quadrada

Desta forma, o erro quadrático pode ser interpretado como a norma do vetor erro ponto a ponto multiplicada por $\sqrt{\Delta X}$. O código do Erro quadrático pode ser visto abaixo.

```
def our_sol(resp, uarray):
    '''Calcula o vetor solucao baseado nas intensidades do MMQ'''
    sol = np.zeros((uarray.shape[1], 1))
    for i in range(resp.shape[0]):
        sol += resp[i] * uarray[i]
    return sol

```

```python
def Erro_quadratico(N, sol, uT):
    ''''Calcula erro quadrático'''
    global string
    global counter
    DeltaX = 1 / N
    Erro_ponto_a_ponto = uT - sol
    Erro = prod_interno(Erro_ponto_a_ponto, Erro_ponto_a_ponto)
    Erro = np.sqrt(DeltaX*Erro)
    print('Erro quadrático: {}'.format(Erro))
    string += 'Erro quadrático: {}'.format(Erro) #referente a criacao do apendice contendo resultados
    print()
    string += '\n \n' #referente a criacao do apendice contendo resultados
    if counter % 2 == 1: #referente a criacao do apendice contendo resultados
        string = string[:-2] #referente a criacao do apendice contendo resultados
        string += r'\end{multicols}' + '\n \n' #referente a criacao do apendice contendo resultados
    counter += 1 #referente a criacao do apendice contendo resultados
    return Erro
```

Além da análise do erro quadrático, 3 plots foram criados. O primeiro contém a solução fornecida (uT) e a solução calculada, ambas sobrepostas. O segundo plot contém a diferença entre elas [4], representando o erro ponto a ponto. O terceiro plot é uma forma mais intuitiva de visualizar a intensidades das fontes.

Este terceiro gráfico pode ser visualizado na figura 1. Nele pode-se ver 2 barras sobre a posição de cada fonte, suas alturas representam a intensidade da fonte pontual. A da esquerda representa a intensidade calculada pelo método do MMQ, e a da direita a solução exata[5]. Os códigos que executam tais análises encontram-se no apêndice B.

Além das análises, como forma de *feedback* para o usuário, um método para o *print* dos resultados foi criado. O código está representado abaixo e um exemplo pode ser visto na figura 2.

```python
def print_resp(Name, resp):
    ''''Imprime respostas do teste de forma bonita e organizada'''
    global string
    global counter
    print('------------------------------------------------------------'
    '----------------------')
    print(Name, ':')
    if counter % 2 == 0:
        string += r'\begin{multicols}{2}' + '\n' #referente a criacao do apendice contendo resultados
    string += r'\noindent\rule{\linewidth}{0.4pt}' + '\n' #referente a criacao do apendice contendo resultados
```

[4] Vale notar, que se pegarmos esse gráfico, elevá-lo ao quadrado, calcularmos a sua área e tirarmos a raiz dela, teremos o valor do erro quadrático. Isso é notável, pois torna esse gráfico, uma forma visual de enxergarmos o erro quadrático. Essa intuição é valida, pois cada ponto neste gráfico representa um retângulo de base ΔX e altura diferença entre o ponto exato e o calculado

[5] Para os testes C e D, a solução exata do MMQ é a solução proveniente do Teste C com $N = 2048$. Ela pode ser considerada como a exata pois o Erro quadrático dela é da mesma ordem de grandeza (E-13) dos erros quadráticos de sistemas com respostas exatas. Sistema que possuem respostas exatas são os sistemas que podem ser perfeitamente representados pelas fontes pontuais escolhidas.

```python

    string += Name + ':' + '\n \n' #referente a criacao do apendice contendo resultados
    df = pd.DataFrame(columns=['', 'Ak'])
    for i in range(resp.shape[0]):
        df.loc[i, ''] = 'a{} = '.format(i + 1)
        df.loc[i, 'Ak'] = resp[i]
        print('a{} = {}'.format(i + 1, resp[i]))
        string += 'a{} = {}'.format(i + 1, resp[i]) + '\n \n' #referente a criacao do apendice contendo resultados
    string = string[:-2] #referente a criacao do apendice contendo resultados
    string += r'\\'  #referente a criacao do apendice contendo resultados
    df = df.set_index('').dropna()
    print()
    string += '\n'#referente a criacao do apendice contendo resultados
```

Todos estes métodos e análises foram englobados em uma função de finalização, que segue abaixo.

```python
def finalize(Name, resp, uT, uarray, N, exata, plist):
    '''Imprime os resultados e executa as analises de cada teste'''
    print_resp(Name, resp)
    sol = our_sol(resp, uarray)
    Erro = Erro_quadratico(N, sol, uT)
    plot_exataXsol(Name, uT, sol, Erro)
    plot_barra(Name, resp, exata, plist)
    return Erro
```

Figura 1: Exemplo do terceiro tipo de análise.

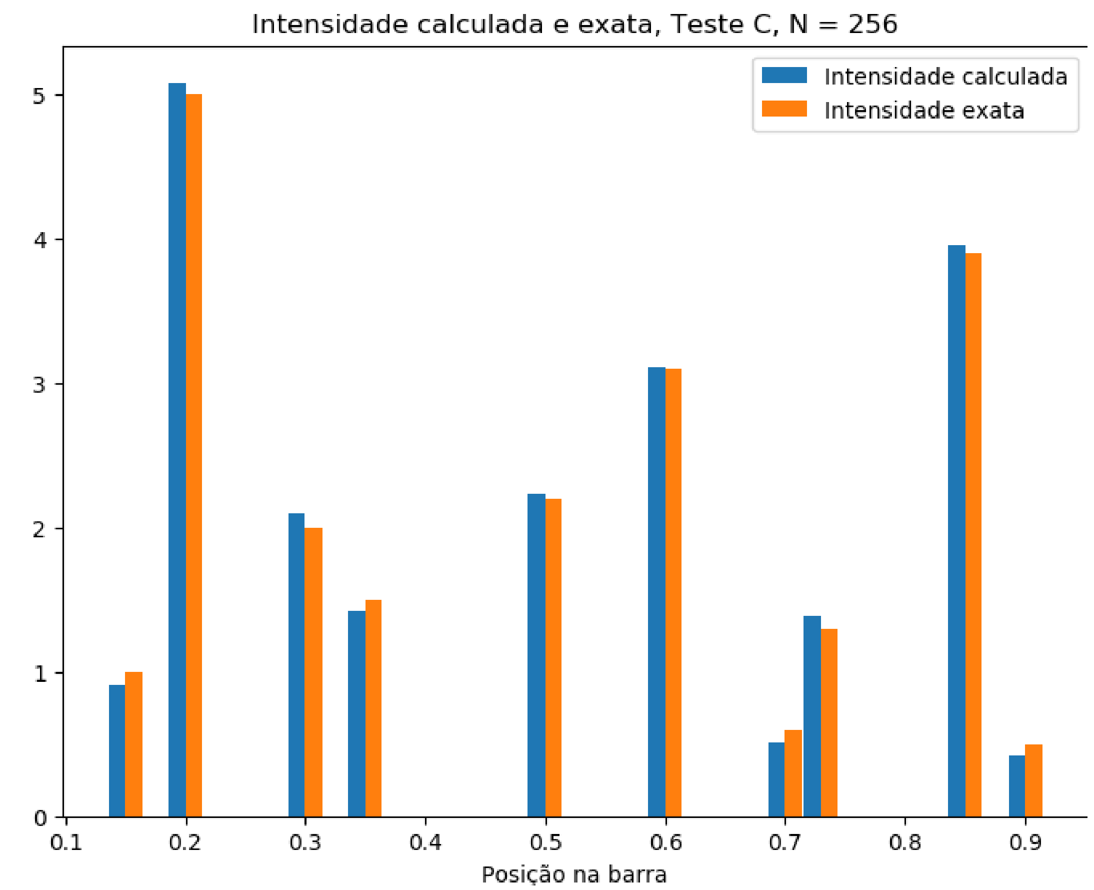


Figura 2: Exemplo de print.

```
------------------------------------------------
Teste D, N = 2048 :
a1 = 0.9493797330503675
a2 = 5.075986918342835
a3 = 1.95911348819509
a4 = 1.510330101759365
a5 = 2.2030506661305864
a6 = 3.105773241551538
a7 = 0.608126852771397
a8 = 1.2939735946842799
a9 = 3.871937852770244
a10 = 0.5320745041480931

Erro quadratico: 0.10477882015184259
```

7.2 Teste A

O teste A consiste de uma validação do método, gerando um vetor uT como uma resposta conhecida.

7.2.1 Código

```python
def TesteA():
    '''Executa teste A'''
    Name = 'Teste A'
    N = 128
    plist = [0.35]
    uarray = create_us(plist, N)
    uT = 7 * uarray[0]
    resp, uarray = resolveMMQ(plist, N, uT)
    exata = np.array([7])
    Erro = finalize(Name, resp, uT, uarray, N, exata, plist)
    return resp, Erro, exata, plist
```

7.2.2 Resultados

Como podemos ver nas figuras 3, 4 e 5, como também no resultado. O modelo MMQ funciona como esperado, determinando com exatidão a solução. O erro quadrático ficou na ordem de E-15, indicando que ele é proveniente da limitação de ponto flutuante dos *Float64* do *numpy*. Isso fica evidente quando olhamos o gráfico do erro ponto a ponto, que fica extremamente próximo ao zero.

Teste A:
a1 = 7.000000000000001
Erro quadrático: 1.040163109568946e-15

Figura 3

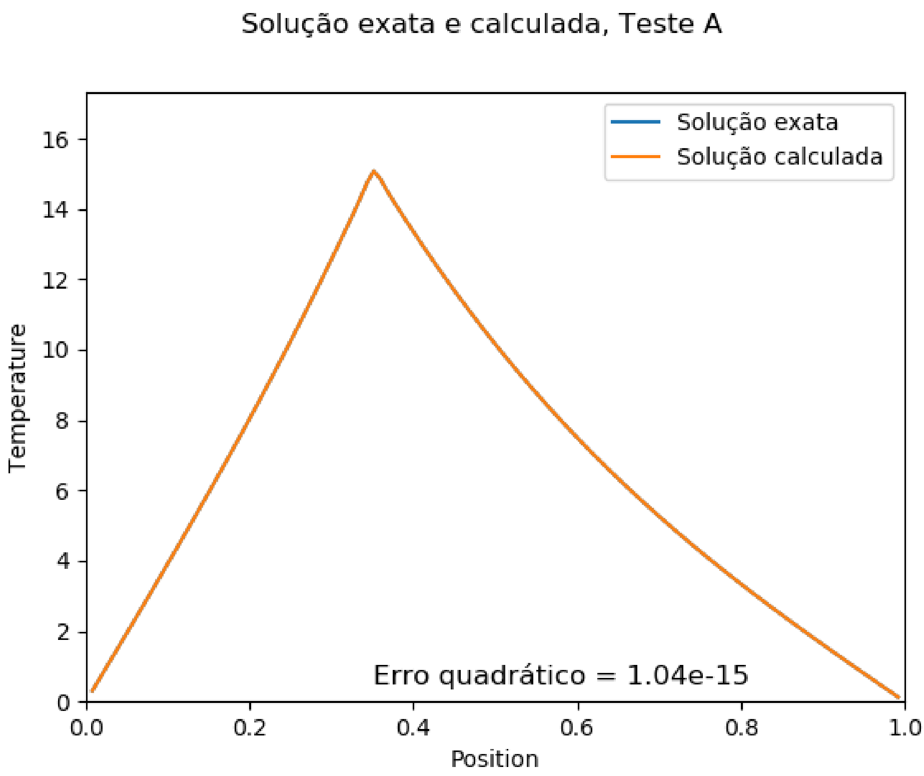


Figura 4

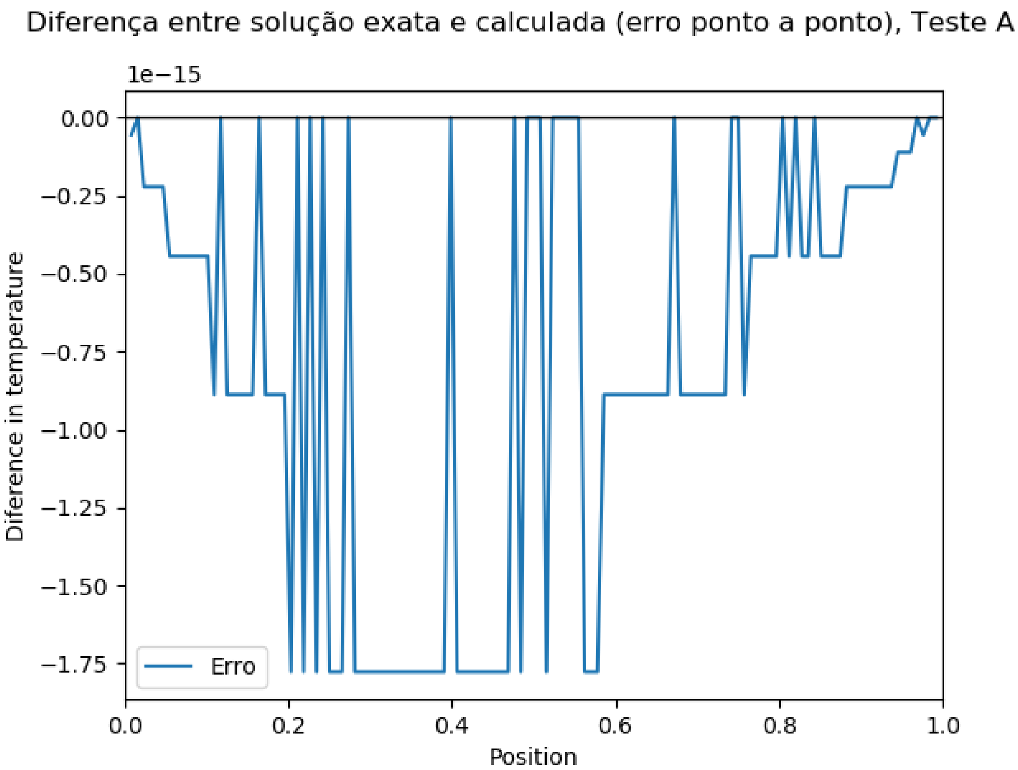
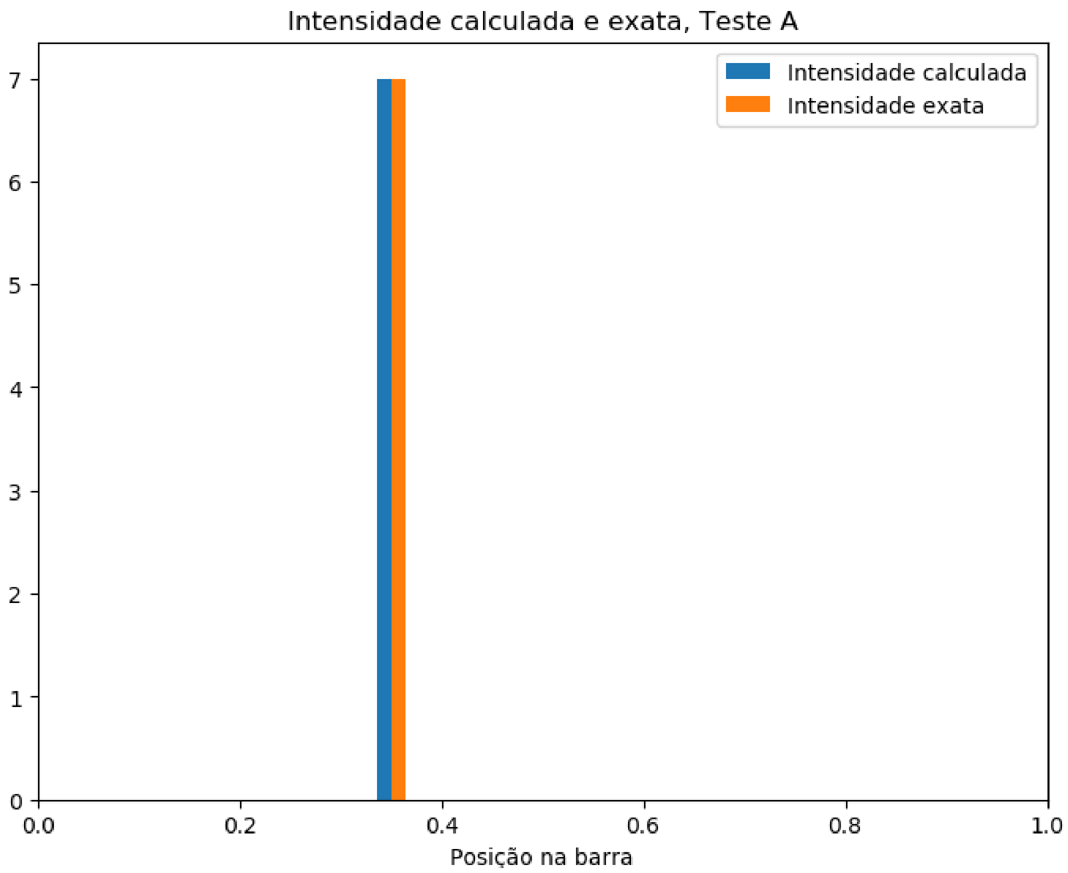


Figura 5



7.3 Teste B

O teste B também consiste de uma validação, porém uma validação mais complexa, envolvendo múltiplas fontes, e, novamente, uT é contruído como combinação linear dos vetores gerados pelas fontes.

7.3.1 Código

```
def TesteB():
    '''Executa teste B'''
    Name = 'Teste B'
    N = 128
    plist = [0.15, 0.3, 0.7, 0.8]
    uarray = create_us(plist, N)
    uT = 2.3 * uarray[0] + 3.7 * uarray[1] + 0.3 * uarray[2] + 4.2 * uarray[3]
    resp, uarray = resolveMMQ(plist, N, uT)
    exata = np.array([2.3, 3.7, 0.3, 4.2])
    Erro = finalize(Name, resp, uT, uarray, N, exata, plist)
    return resp, Erro, exata, plist
```

7.3.2 Resultados

Como podemos ver nas figuras 6, 7 e 8, como também no resultado. O modelo MMQ funciona como esperado, determinando com exatidão a solução. O cenário assemelha-se muito ao do Teste A, no entanto, a ordem do erro passa a ser E-14. Isso nos mostra que os erros de aproximação de *Float64* propagasse, e quanto maior o número de operações executadas, maior será esse erro.

Teste B:
a1 = 2.3000000000000433
a2 = 3.699999999999968
a3 = 0.3000000000000007
a4 = 4.200000000000008
Erro quadrático: 1.1521327611331212e-14

Figura 6

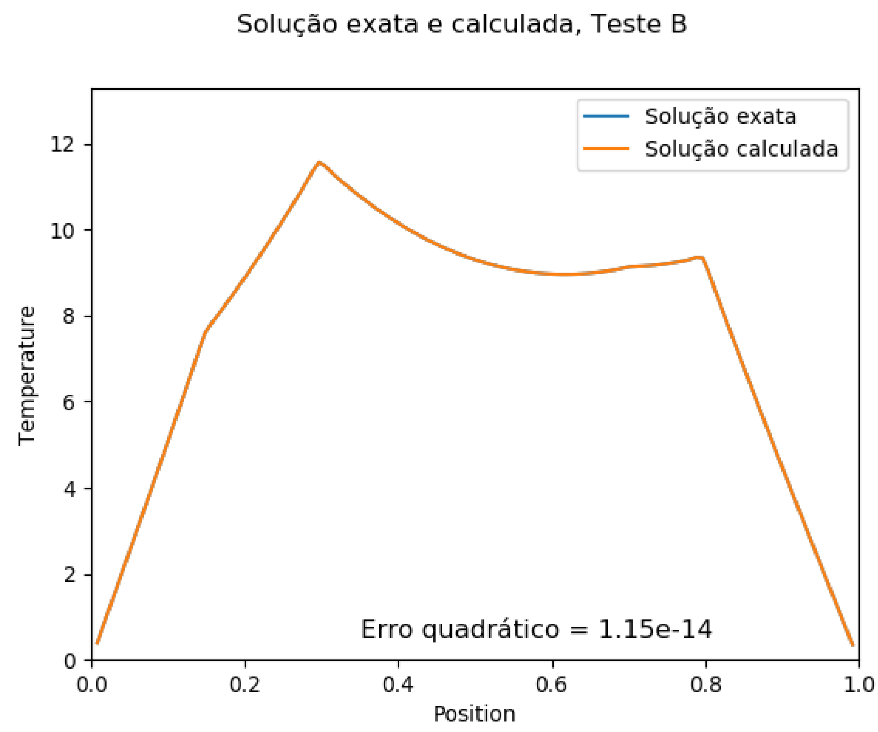


Figura 7

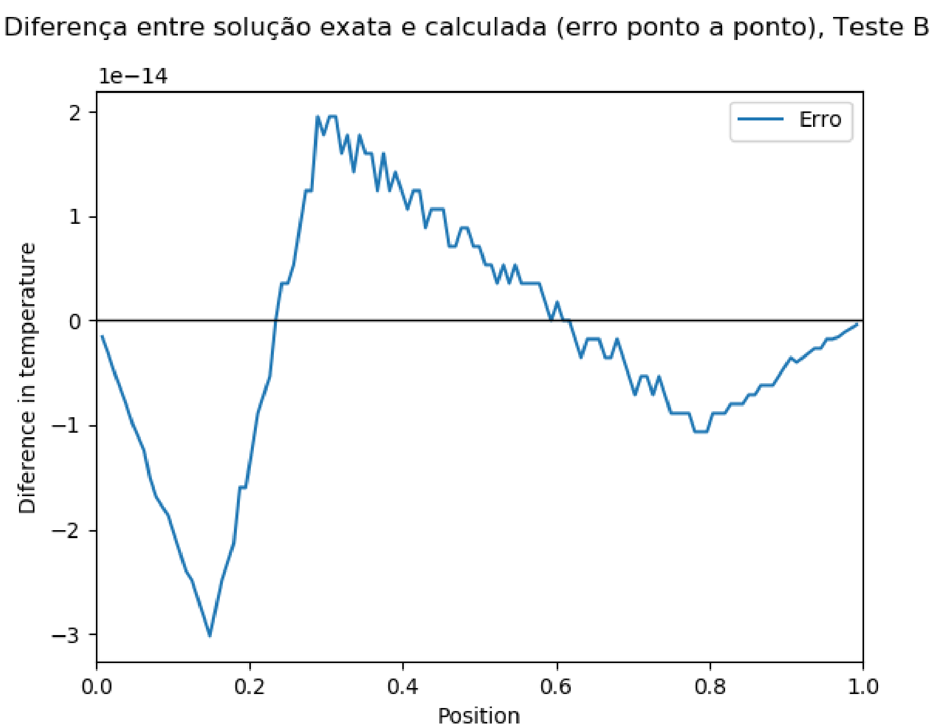
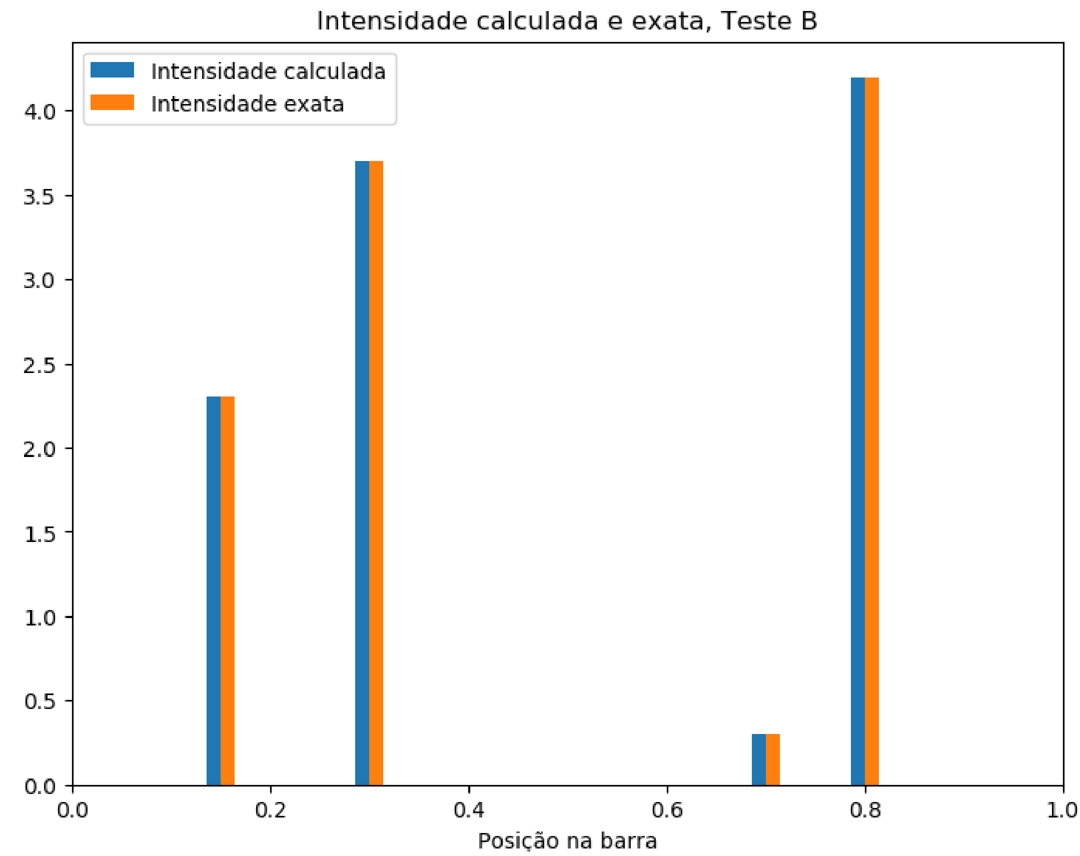


Figura 8



7.4 Teste C

O teste C consiste de um vetor uT fornecido no moodle. Este vetor possui discretização espacial de 2048, e o nosso objetivo é executar o algoritmo sobre ele e *subsamples* (N = 128, 256, 512, 1024 e 2048) dele. A posição das fontes também foi fornecida.

7.4.1 Código

```python
def TesteC(N):
    '''Executa teste C'''
    Name = 'Teste C, N = {}'.format(N)
    plist, uT = read_text(N)
    resp, uarray = resolveMMQ(plist, N, uT)
    exata = np.array([1, 5, 2, 1.5, 2.2, 3.1, 0.6, 1.3, 3.9, 0.5])
    Erro = finalize(Name, resp, uT, uarray, N, exata, plist)
    return resp, Erro, exata, plist
```

7.4.2 Resultados

Os resultados numéricos desta simulação estão disponíveis no apêndice A.

Pode-se ver na figura 9 que o método é bem capaz de aproximar a solução exata, mesmo em cenários de *subsampling*. O erro quadrático fica na ordem de E-2 e E-3 para os testes em amostragem, e salta para E-12 para N = 2048. Isso, novamente, nos indica que tal vetor uT foi construído como combinação linear dos vetores das fontes, e que a solução para este caso é a exata.

Na figura 10 fica claro que o erro não possui viés (negativo ou positivo), mas vale notar que seus picos coincidem com as posições das fontes e dos pontos médios entre elas. E por fim, na figura 11, é possível visualizar o quão distantes as intensidades calculadas ficaram da realidade.

Figura 9: Podemos ver como as solução calculada consegue aproximar-se da solução exata.

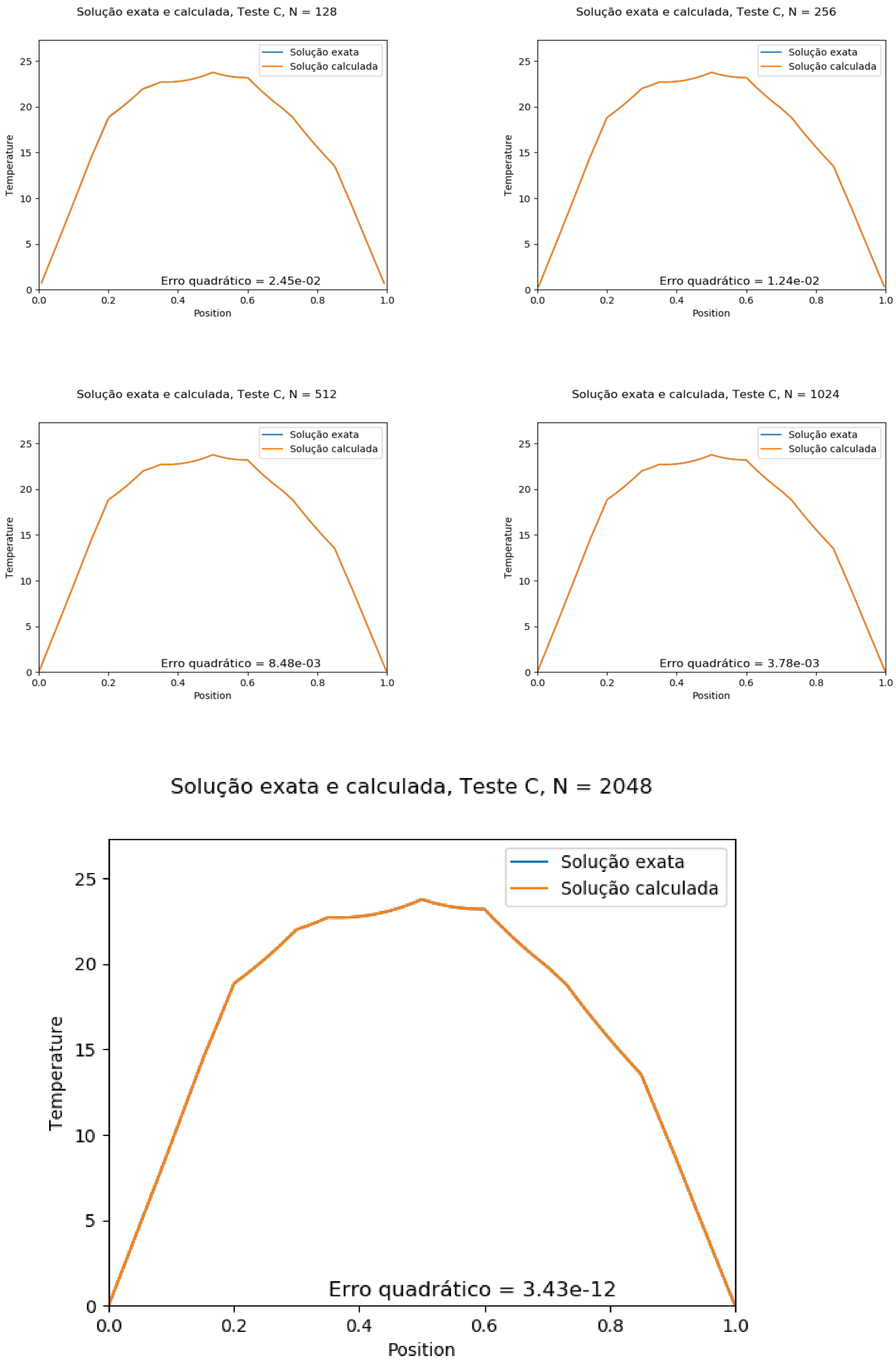


Figura 10: Podemos ver como o erro se distribui ao longo da barra.

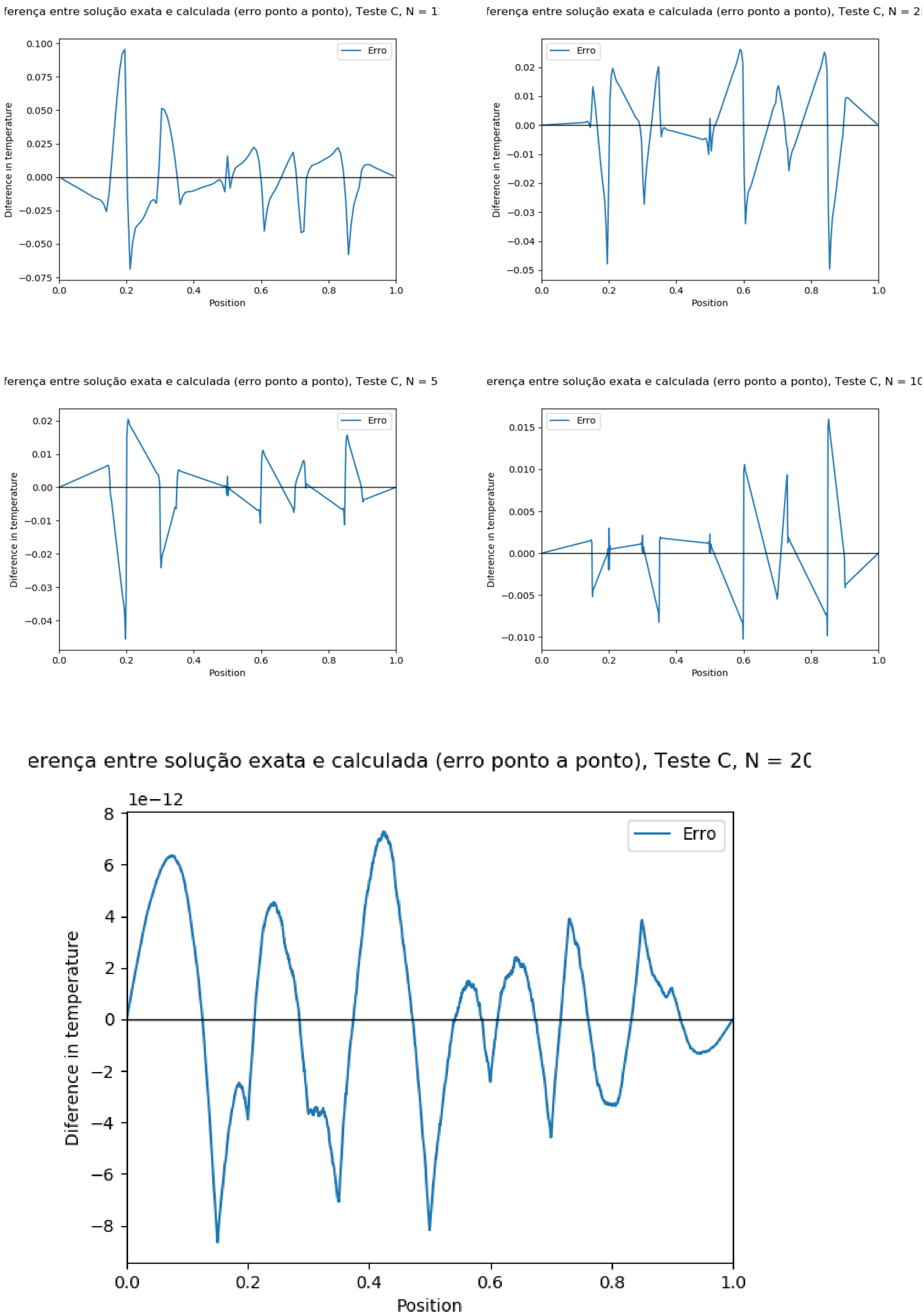
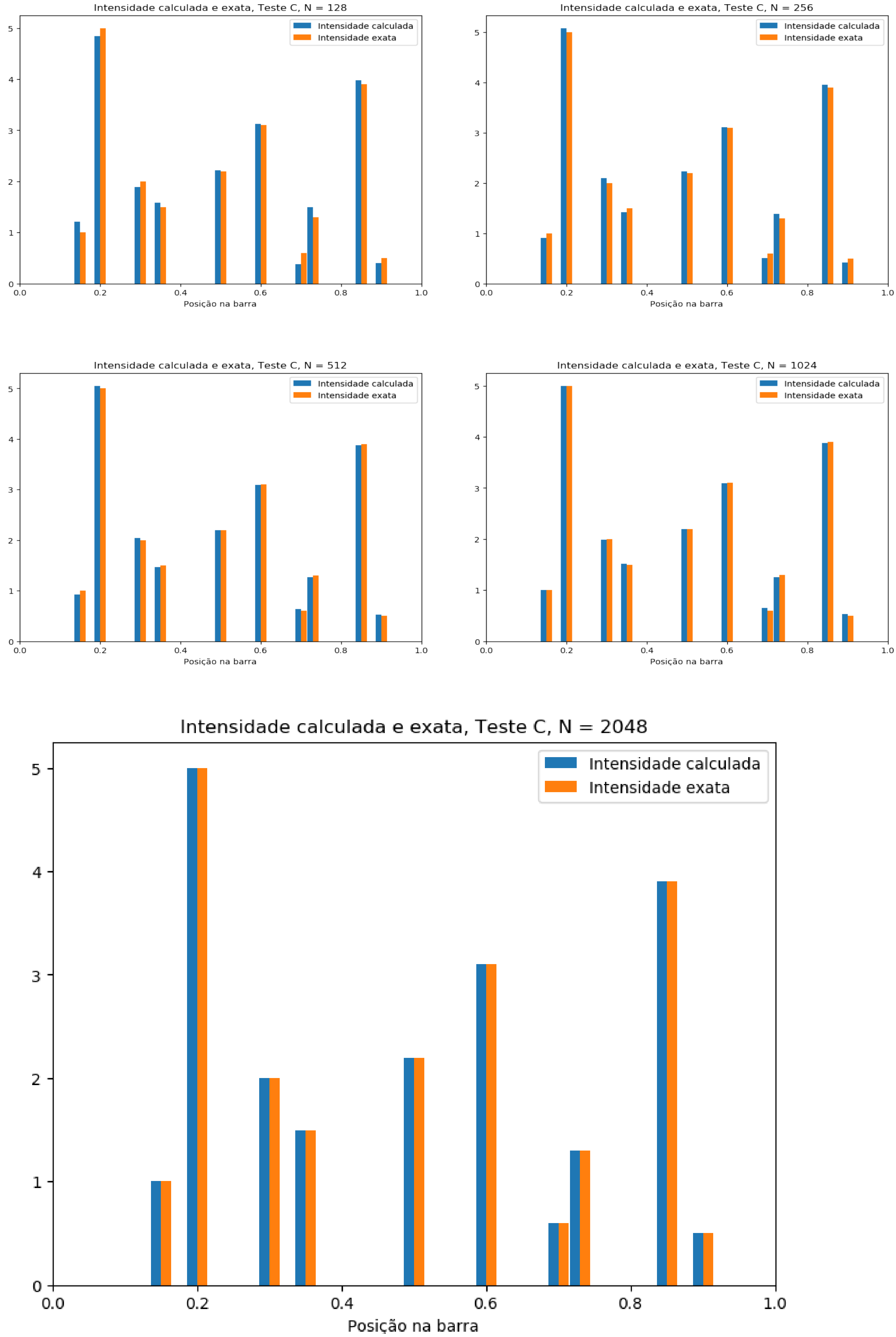


Figura 11: Podemos ver como as intensidades se distribuíram ao longo da barra.



7.5 Teste D

O teste D é uma continuação do teste C, porém com adição de ruído na solução exata. Este ruído foi adicionado de forma aleatória.

7.5.1 Código

```
def TesteD(N):
    '''Executa teste D'''
    Name = 'Teste D, N = {}'.format(N)
    plist, uT = read_text(N)
    multipliers = np.random.random(N - 1)
    multipliers -= 0.5
    multipliers *= 2
    multipliers *= 0.01
    multipliers += 1
    multipliers = multipliers.reshape(N - 1, 1)
    uT = uT * multipliers
    resp, uarray = resolveMMQ(plist, N, uT)
    exata = np.array([1, 5, 2, 1.5, 2.2, 3.1, 0.6, 1.3, 3.9, 0.5])
    Erro = finalize(Name, resp, uT, uarray, N, exata, plist)
    return resp, Erro, exata, plist
```

7.5.2 Resultados

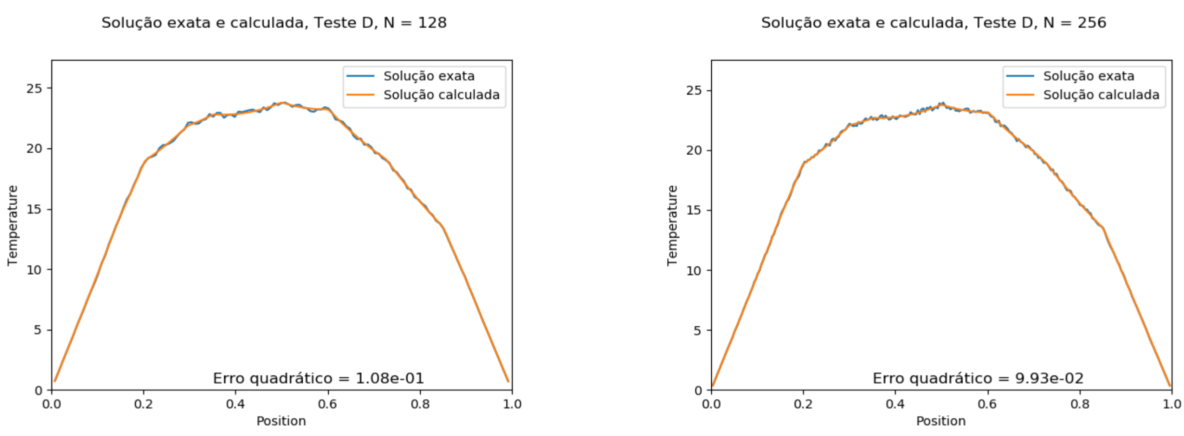
Os resultados numéricos desta simulação estão disponíveis no apêndice A.

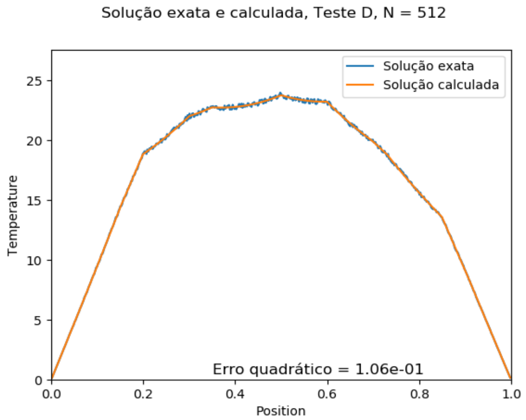
Pode-se ver na figura 12 que o método é bem capaz de aproximar a solução exata, mesmo em cenários de *subsampling* e com ruído. É notável que por natureza, o ruído oscila em torno da solução exata, e o MMQ foi capaz de manter-se centrado apesar disto.

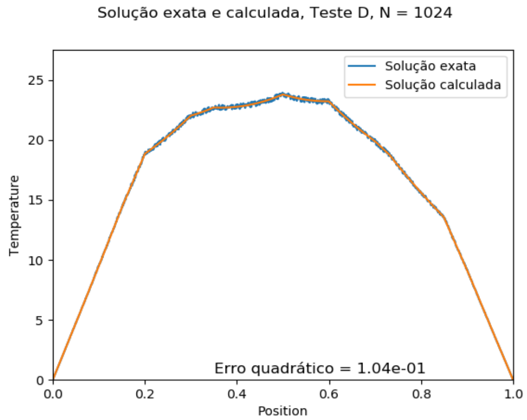
Na figura 13 fica claro que o erro não possui viés (negativo ou positivo), da forma como era esperado. O erro é maior nos pontos em que a temperatura é maior, o que reflete o fato de ele ter sido escolhido como uma variação percentual.

E por fim, na figura 14, é possível visualizar o quão distantes as intensidades ficaram da realidade. Vale ressaltar que para N = 128, o ruído fez as fontes nas posições 0.7 e 0.73, devido a sua proximidade, serem "confundidas", chegando a indicar uma intensidade negativa para a fonte 0.7 (compensada por uma intensidade muito maior que a esperada na fonte 0.73).

Figura 12: Podemos ver como as solução calculada consegue aproximar-se da solução exata.







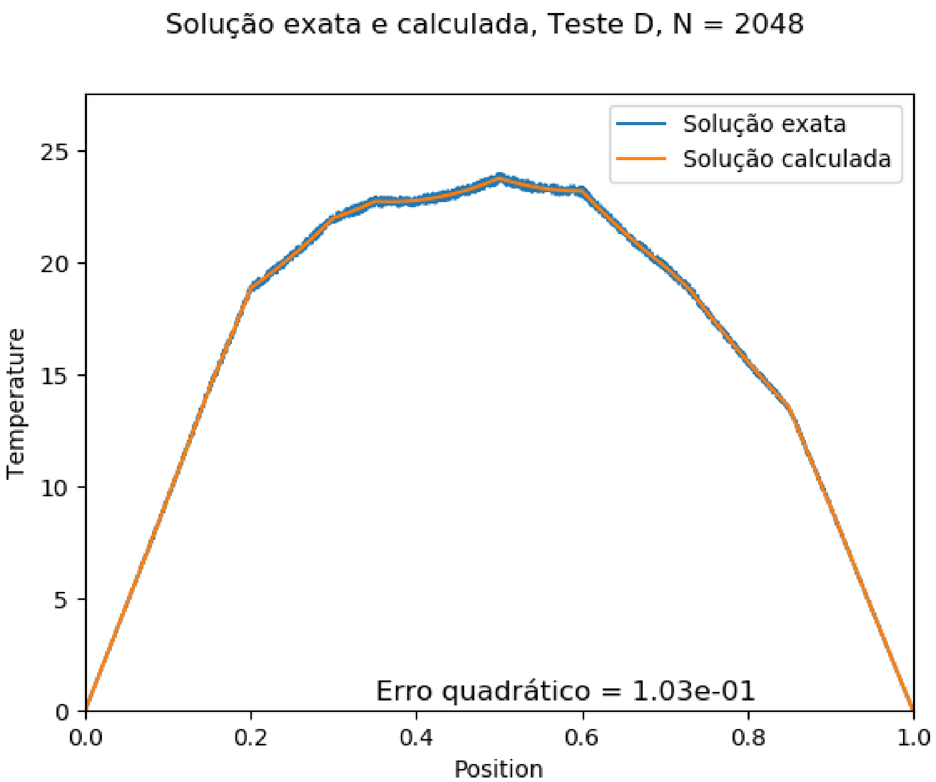
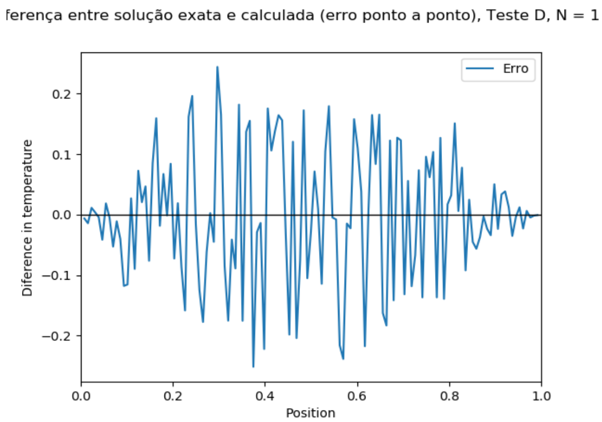
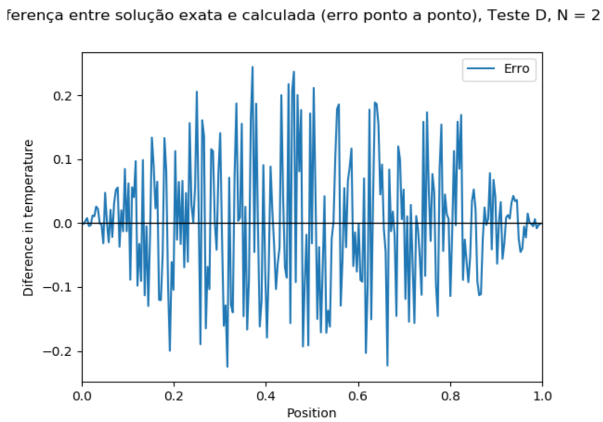
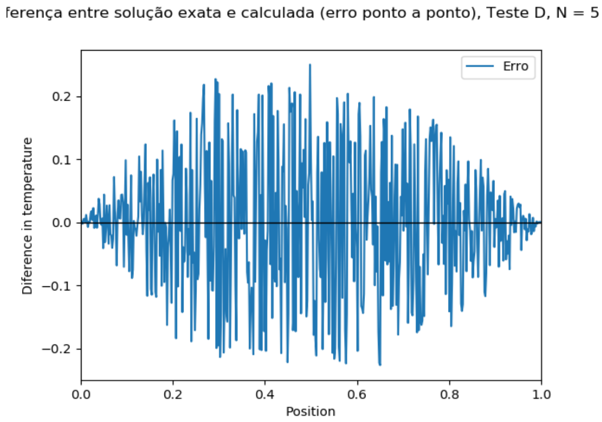
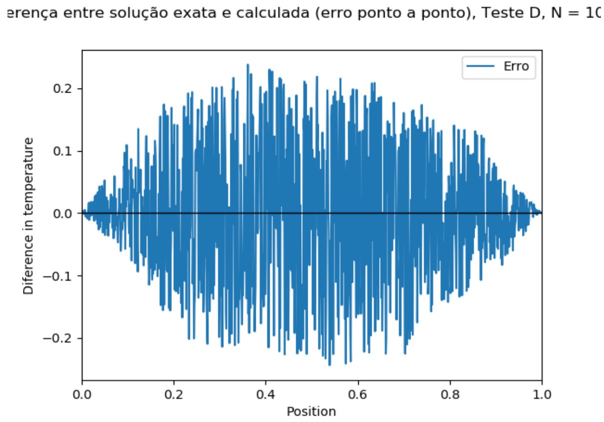


Figura 13: Podemos ver como o erro se distribui ao longo da barra.









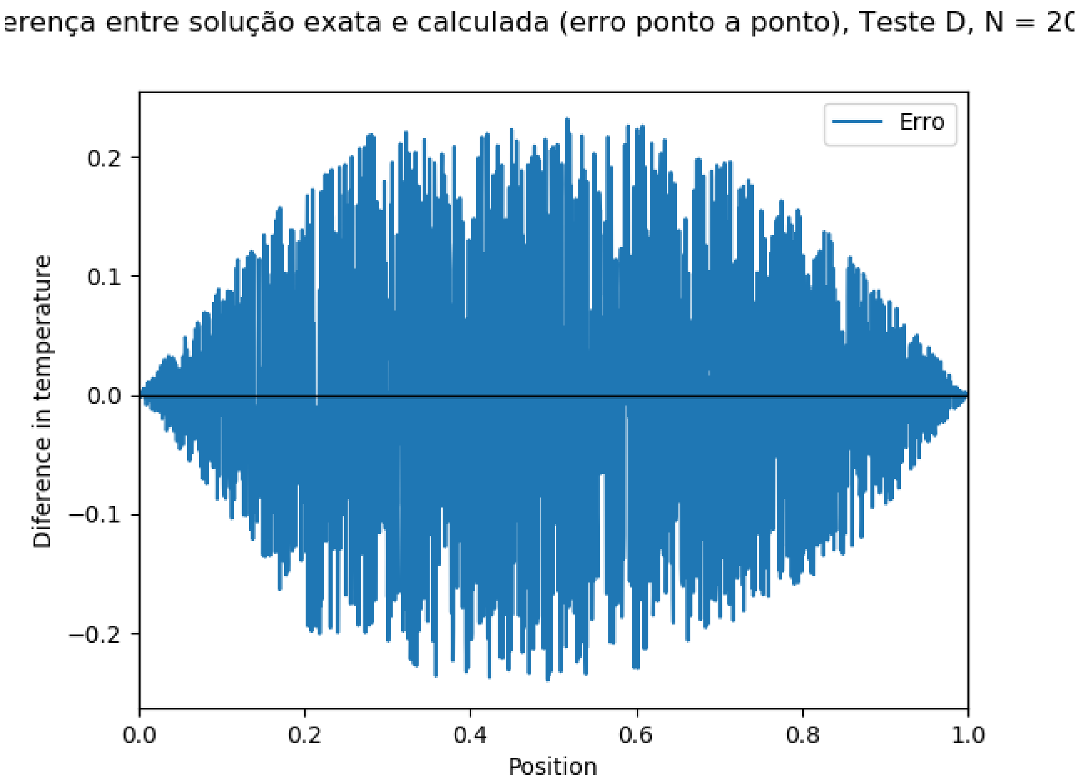
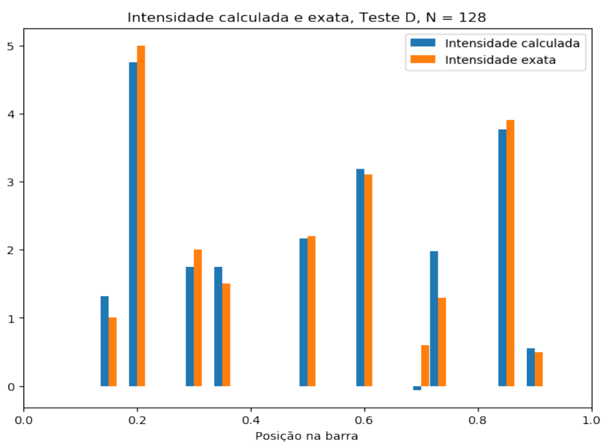
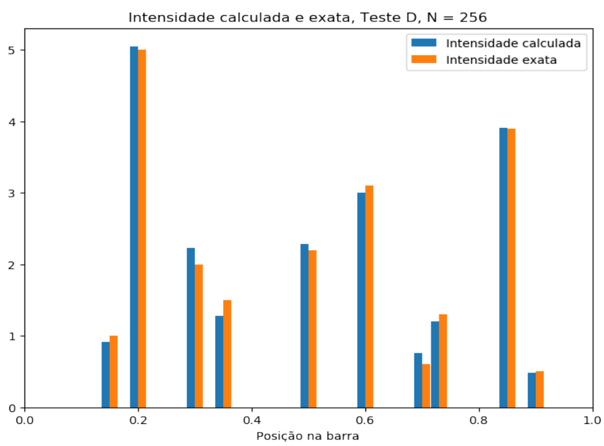
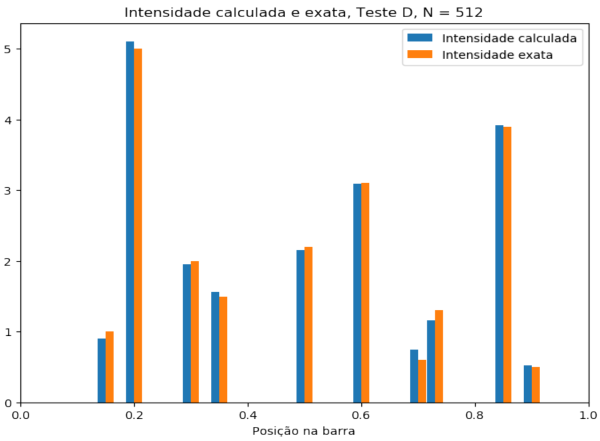
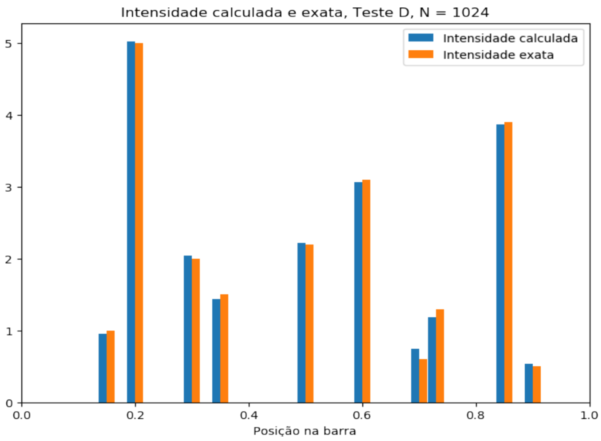


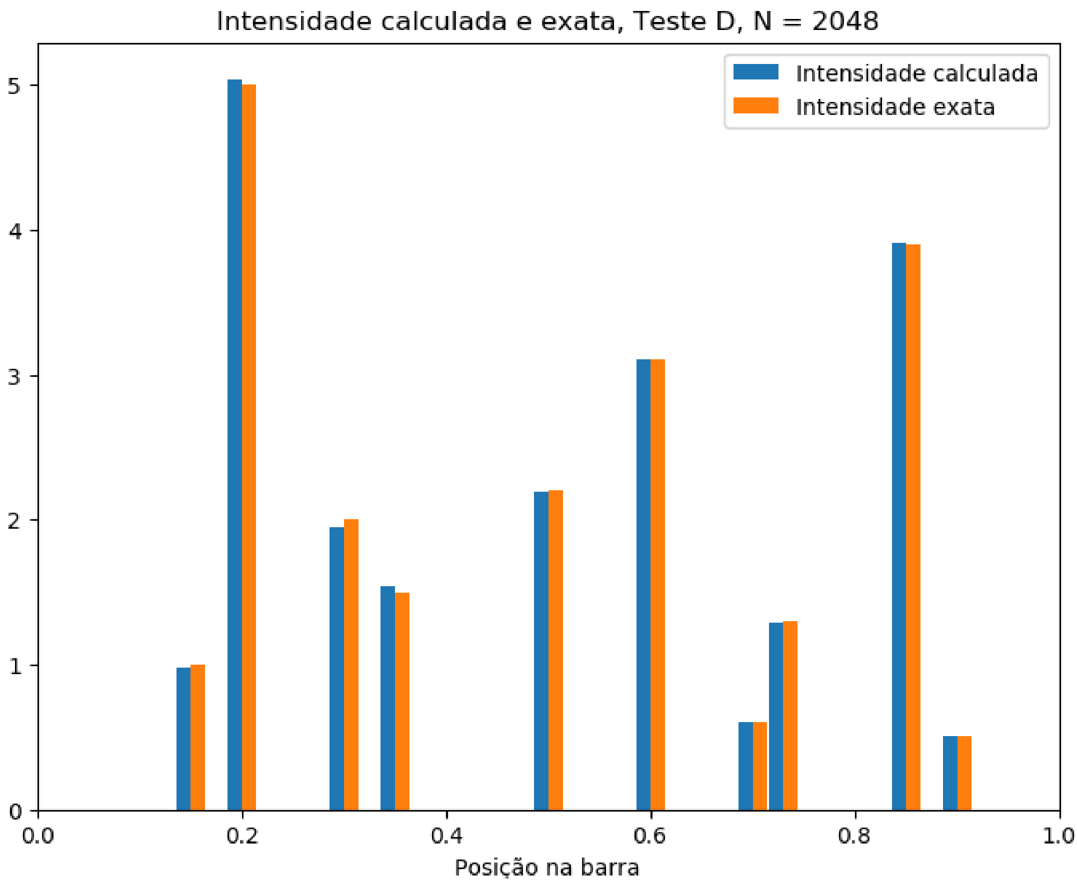
Figura 14: Podemos ver como as intensidades se distribuíram ao longo da barra.









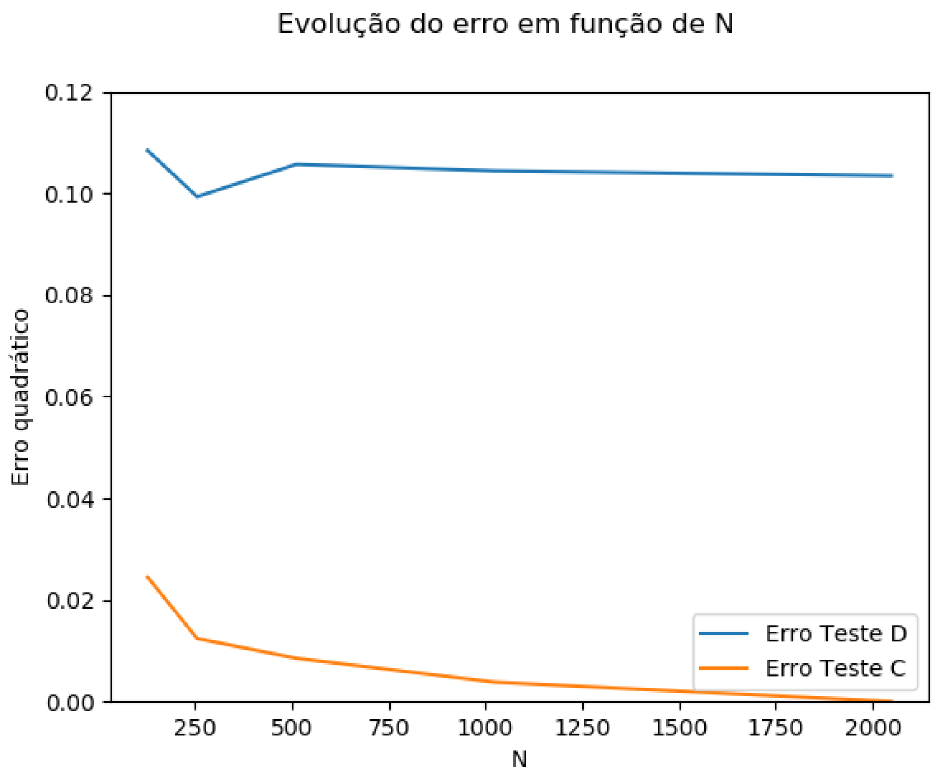


8 Análises Gerais

Além das análises específicas de cada teste (Tipo e N), algumas análises foram criadas para verificarem a influência do refinamento da malha nos testes C e D.

A primeira análise for a evolução do Erro com N, e pode ser vista nas figura 15. No Teste C, sem ruído, o erro aproxima-se de zero com o refinamento da malha, porém com a inserção do ruído, esta tendência se quebra, e o erro fica constantemente próximo a 0.1, sem nenhuma tendência clara. Vale ressaltar também, que, naturalmente, o erro no Teste C é menor que no Teste D, devido a presença de ruído.

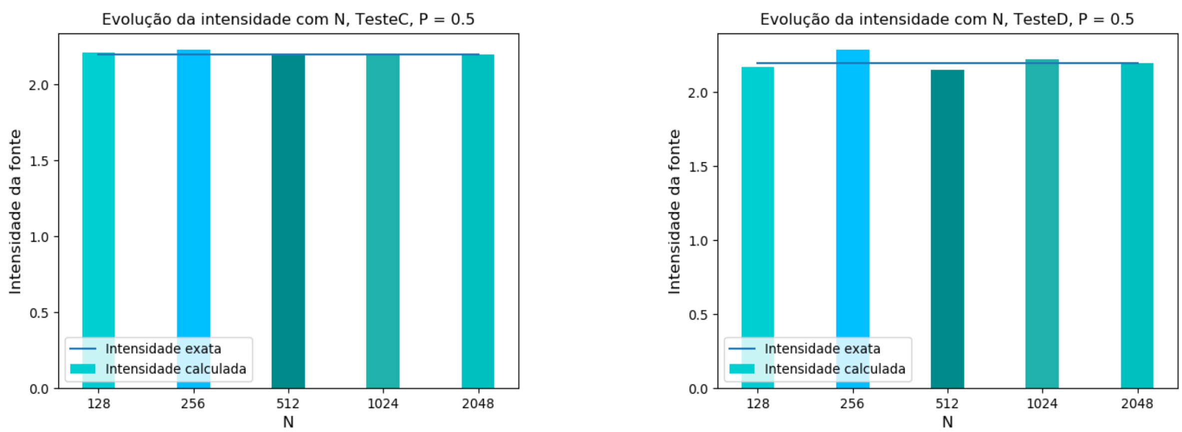
Figura 15



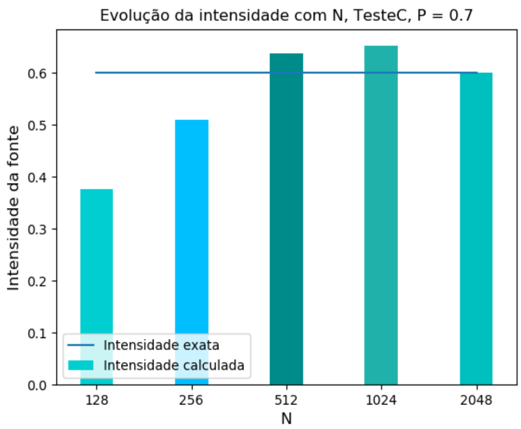
A segunda análise ocorreu sobre cada fonte individualmente, vendo a variação de sua intensidade, segundo N. A linha horizontal no gráfico representa a intensidade exata, enquanto as barras representam a intensidade calculada. Na figura 16, podemos ver como algumas fontes mantiveram-se estáveis, apesar da mudança do refinamento, e como outras mudaram radicalmente, em ambos cenários.

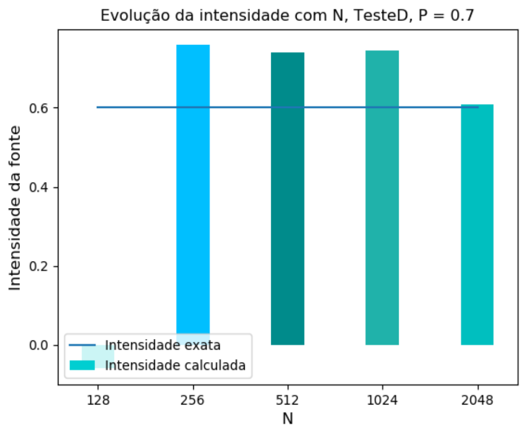
Vale notar também que a diferença entre as intensidades calculadas pelo Teste D e pelo Teste C são mínimas, dado N suficientemente grande. Isso mostra que o método pode ser aplicado em situações práticas, onde sempre haverão erros de leitura instrumental, e que estes erros, no entanto, não afetarão[6] de forma significativa a análise.

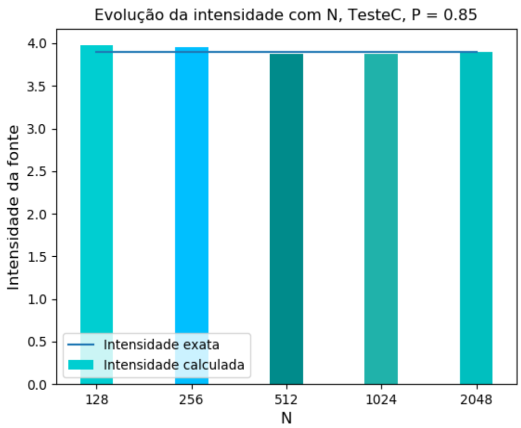
Figura 16: Podemos ver como as intensidades, de cada fonte, variaram com N.

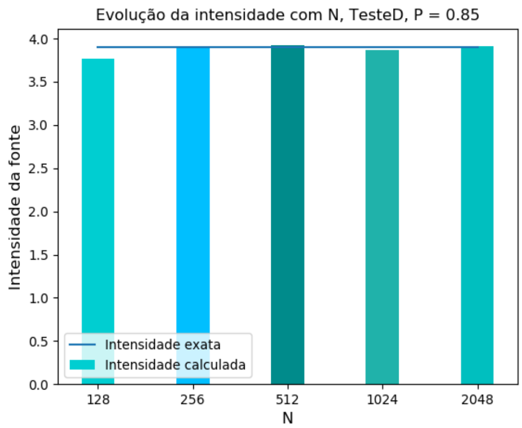


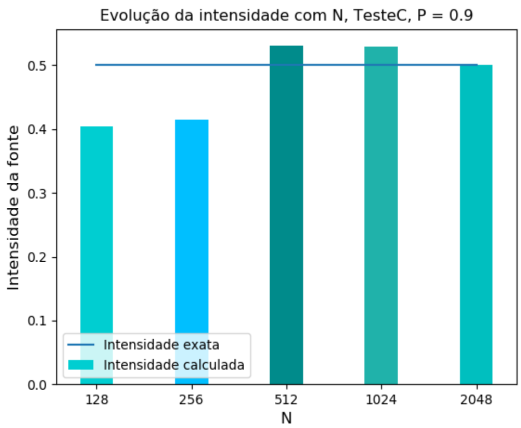
[6]Dado que este erro seja bem menor que a medida propriamente dita.

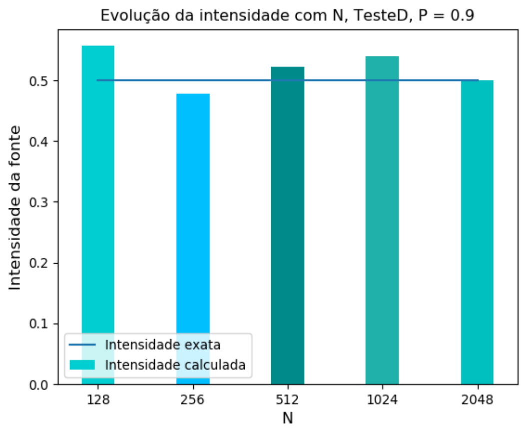












9 Conclusões

Podemos concluir através desta aplicação prática do MMQ, que ele é um método bastante robusto capaz de lidar até mesmo com ruídos. Ele é ótimo para otimizar recursos e minimizar um determinado tipo de erro.

Com essa alta capacidade de minimização, ele passa a ser um algoritmo extremamente versátil para diversos problemas, e como visto neste exercício programa, pode ser implementado de forma pouco custosa, computacionalmente falando (algo que muitos algoritmos de minimização não podem, como por exemplo *gradient descent*). Tornando este método ideal para diversas aplicações.

A Resultados Numéricos

Teste A:
a1 = 7.000000000000001

Erro quadrático: 1.040163109568946e-15

Teste B:
a1 = 2.3000000000000433
a2 = 3.699999999999968
a3 = 0.3000000000000007
a4 = 4.200000000000008
Erro quadrático: 1.1521327611331212e-14

Teste C, N = 128:
a1 = 1.209123179204358
a2 = 4.839258715746446
a3 = 1.887240855757625
a4 = 1.583399931863184
a5 = 2.2145040462885657
a6 = 3.121294778776268
a7 = 0.3773402863725357
a8 = 1.492348288122205
a9 = 3.9751388015990363
a10 = 0.40414515364858933
Erro quadrático: 0.024453403799691034

Teste C, N = 256:
a1 = 0.9045010343195337
a2 = 5.077572635561381
a3 = 2.1008535954798013
a4 = 1.4141556850879553
a5 = 2.2292450130533794
a6 = 3.1046138569914934
a7 = 0.5094525973922925
a8 = 1.3865087904557782
a9 = 3.94987864615182
a10 = 0.4148931283298714
Erro quadrático: 0.012363464048870192

Teste C, N = 512:
a1 = 0.928688378496588
a2 = 5.053707844483995
a3 = 2.043701048906051
a4 = 1.4676706728722735
a5 = 2.1967633320027744
a6 = 3.091131168891181
a7 = 0.637587516383272
a8 = 1.2716872153135235
a9 = 3.8780948673256566
a10 = 0.530556778642094
Erro quadrático: 0.008476628330998064

Teste C, N = 1024:
a1 = 1.0072813220840864
a2 = 4.992443012459354
a3 = 1.9858767276285576
a4 = 1.5132584652247445
a5 = 2.19269283768325
a6 = 3.0951528759315297
a7 = 0.6523266477808756
a8 = 1.253789889048031
a9 = 3.8796670569340175
a10 = 0.5297366252990546
Erro quadrático: 0.0037793104634000784

Teste C, N = 2048:
a1 = 1.000000000058666
a2 = 5.000000000044658
a3 = 2.0000000000251816
a4 = 1.5000000000431903
a5 = 2.200000000054578
a6 = 3.100000000031434
a7 = 0.6000000000471646
a8 = 1.2999999999780663
a9 = 3.8999999999675494
a10 = 0.49999999998891276
Erro quadrático: 3.4283787053057287e-12

Teste D, N = 128:
a1 = 1.318222126517231
a2 = 4.752296000517362
a3 = 1.7474294034170015
a4 = 1.7438302186136774
a5 = 2.167874825356
a6 = 3.1886344205349877
a7 = -0.06046219630666361
a8 = 1.9760473200419018
a9 = 3.763662672476851
a10 = 0.5566260094336855
Erro quadrático: 0.10839481767729643

Teste D, N = 256:
a1 = 0.909392879669273
a2 = 5.049476922744297
a3 = 2.225495365454652
a4 = 1.2804211063888822
a5 = 2.286520197512047
a6 = 3.0042259401348
a7 = 0.7585839667004386
a8 = 1.2017159750350377
a9 = 3.9095679492563367
a10 = 0.4776667326271345
Erro quadrático: 0.09930281685825725

Teste D, N = 512:
a1 = 0.8971668552304166
a2 = 5.109619576740172
a3 = 1.950928539480227
a4 = 1.558476061134911
a5 = 2.15367326871897
a6 = 3.09440473928149
a7 = 0.7404509420074445
a8 = 1.1645720284995296
a9 = 3.9203971168710816
a10 = 0.5226195277491588
Erro quadrático: 0.10568117698540226

Teste D, N = 1024:
a1 = 0.9522493040192224
a2 = 5.027822662803196
a3 = 2.0460306410199465
a4 = 1.4397222519250406
a5 = 2.22028103288598
a6 = 3.0692311647141732
a7 = 0.7454729077428466
a8 = 1.1810951297386119
a9 = 3.872303956270896
a10 = 0.5395011862116126
Erro quadrático: 0.10439190973093909

Teste D, N = 2048:
a1 = 0.9796040541727287
a2 = 5.038121284714059
a3 = 1.9425791229428881
a4 = 1.5452032133138403
a5 = 2.1947723732160878
a6 = 3.1029859808298985
a7 = 0.6076984490105852
a8 = 1.28203405504971
a9 = 3.9115220934007544
a10 = 0.49935626178704695
Erro quadrático: 0.10342468073753705

B Funções de Plot

```
def plot_exataXsol(Name, vector, sol, Erro):
    '''Plota a solucao exata, nossa solucao e a diferenca entre ambas no instante T'''
    N = vector.shape[0] + 1
    xspace = np.linspace(0, 1, N + 1)[1:-1]

    plt.clf()
    plt.plot(xspace, vector)
    plt.plot(xspace, sol)
    plt.legend(['Solução exata', 'Solução calculada'], loc='upper left')
    plt.ylabel('Temperature')
    plt.xlabel('Position')
    plt.suptitle('Solução exata e calculada, ' + Name)
    plt.text(0.35, 0.5, 'Erro quadratico = {}'.format(np.format_float_scientific(Erro, 2)), dict(size=12))
    plt.savefig('{}.png'.format('plots/exataXcalculada' + Name))
    plt.show()

    plt.clf()
    plt.plot(xspace, vector - sol)
    plt.legend(['Erro'], loc='upper left')
    plt.ylabel('Diference in temperature')
    plt.xlabel('Position')
    plt.suptitle('Diferença entre solução exata e calculada, ' + Name)
    plt.savefig('{}.png'.format('plots/erro' + Name))
    plt.show()


def plot_barra(Name, resp, exata, plist):
    '''Plota grafico de barras representando as intensidades calculadas e as exatas'''
    plt.clf()
    fig = plt.figure(frameon=False)
    ax = fig.add_axes([0, 0, 1, 1])
    X = np.array(plist)

    espessura = 0.014
    ax.bar(X - espessura / 2, resp, width=espessura)
    fig.add_axes(ax)
    ax.bar(X + espessura / 2, exata, width=espessura)
    ax.legend(labels=['Intensidade calculada', 'Intensidade exata'])
    ax.set_title('Intensidade calculada e exata, {}'.format(Name))
    ax.set_xlabel('Posição na barra')
    ax.set_ylabel('')

    fig.add_axes(ax)
    fig.savefig('{}.png'.format('plots/barras' + Name), bbox_inches='tight', pad_inches=0)
    fig.show()
```

```python


def plot_serie_barra(Name, resps, exata, plist):
    '''Plota a evolucao da intensidade de cada fonte em funcao do refinamento da malha, plota tambem uma linha que
    contem a resposta exata '''
    width = 0.35
    matrix = resps[0]
    for i in range(1, len(resps)):
        matrix = np.vstack((matrix, resps[i]))
    for i in range(len(plist)):
        plt.clf()
        data = matrix[:, i:i + 1].reshape(5)
        data = pd.DataFrame({
            'Intensidade calculada': data,
            'Intensidade exata': np.ones(len(data)) * exata[i]
        })
        my_colors = list(
            islice(cycle(['darkturquoise', 'deepskyblue', 'darkcyan', 'lightseagreen', 'c']), None, len(data)))

        data['Intensidade calculada'].plot(kind='bar', width=width, color=my_colors,
                                           title='Evolução da intensidade com N, P = {}'.format(plist[i]), legend=True)

        data['Intensidade exata'].plot()
        ax = plt.gca()
        ax.set_xticklabels(('128', '256', '512', '1024', '2048'))
        # ax.set_xticklabels(('128', '256', '512'))
        ax.set_xlabel("N", fontsize=12)
        ax.set_ylabel("Intensidade da fonte", fontsize=12)
        ax.legend(labels=['Intensidade calculada', 'Intensidade exata'])
        # plt.show()
        plt.savefig('{}.png'.format('plots/barras_pp' + Name + 'P=' + str(plist[i])))


def plot_serie_erro(Name, erros):
    '''Plota a evolucao do erro com o refinamento da malha'''
    xspace = ['128', '256', '512', '1024', '2048']
    plt.clf()
    plt.plot(xspace, erros)
    plt.legend(['Erro'], loc='upper left')
    plt.ylabel('Erro quadrático')
    plt.xlabel('N')
    plt.suptitle('Evolução do erro, ' + Name)
    plt.savefig('{}.png'.format('plots/erroXn' + Name))
    plt.show()
```