Arthur P. Da Fonseca (10773096)
Sergio A. G. Fuentes (10770200)

EP 2

Métodos Numéricos e Aplicações
10 de Abril de 2020

Introdução

Este relatório faz referência à resolução do segundo exercício-programa de Métodos Numéricos e Aplicações (MAP3122) para os alunos de Engenharia de Computação da Escola Politécnica da USP.

O problema aqui resolvido procura a causa (*pressão acústica* da onda) a partir da consequência de um fenômeno físico observado (*fonte* da onda).

Foi possível, após a realização dos exercícios propostos, aproximar uma solução para a *equação da onda*, que é uma equação diferencial parcial cuja solução é justamente a *pressão acústica* da onda, u(t, x).

É importante salientar o programa aqui descrito é capaz de modelar problemas extremamente relevantes para a área da geofísica, incluindo desde o mapeamento de subsolos para prospecção petróleo até a detecção da origem de terremotos.

Exercício 1

Neste exercício o objetivo era resolver o problema direto representado pela seguinte equação diferencial parcial:

$$\partial^2_{tt}u(t,x) - c^2\partial^2_{xx}u(t,x) = f(t,x)\ em\ [0,T]\times[0,1]$$
$$u(0,x) = u_0(x)\ em\ [0,1]$$
$$\partial_t u(0,x) = u_1(x)\ em\ [0,1]$$
$$u(t,0) = 0\ em\ [0,T]$$
$$u(t,1) = 0\ em\ [0,T]$$

Considerando que para os problemas apresentados neste exercício-programa as condições iniciais são nulas e o valor de T é 1, temos:

$$\partial^2_{tt}u(t,x) - c^2\partial^2_{xx}u(t,x) = f(t,x)\ em\ [0,1]\times[0,1]$$
$$u(0,x) = 0\ em\ [0,1]$$
$$\partial_t u(0,x) = 0\ em\ [0,1]$$
$$u(t,0) = 0\ em\ [0,1]$$
$$u(t,1) = 0\ em\ [0,1]$$

A *função de fonte* f(t,x) é definida como:

$$f(t,x) = \begin{cases} 1000c^2(1-2\beta^2t^2\pi^2)e^{-\beta^2t^2\pi^2}, & x = x_c \\ 0, & x \neq x_c \end{cases}$$

Buscando aproximar numericamente a solução da equação diferencial observada, foram feitas as seguintes simplificações para as derivadas primeiras:

$$\partial_t u(t,x) \approx \frac{u(t+\Delta t/2,x) - u(t-\Delta t/2,x)}{\Delta t} \quad \text{e} \quad \partial_x u(t,x) \approx \frac{u(t,x+\Delta x/2) - u(t,x-\Delta x/2)}{\Delta x}$$

Analogamente, as derivadas segundas podem ser calculadas assim:

$$\partial^2_{tt}u(t,x) \approx \frac{\partial_t u(t+\Delta t/2,x) - \partial_t u(t-\Delta t/2,x)}{\Delta t} \quad \text{e} \quad \partial^2_{xx}u(t,x) \approx \frac{\partial_x u(t,x+\Delta x/2) - \partial_x u(t,x-\Delta x/2)}{\Delta x}$$

Agora ao se combinar as 4 expressões anteriores, as seguintes expressões são deduzidas:

$$\partial^2_{tt}u(t,x) \approx \frac{\partial_t u(t+\Delta t/2,x) - \partial_t u(t-\Delta t/2,x)}{\Delta t}$$

$$\partial^2_{xx}u(t,x) \approx \frac{\partial_x u(t,x+\Delta x/2) - \partial_x u(t,x-\Delta x/2)}{\Delta x}$$

Para que se possa calcular de forma algorítmica a solução do problema inverso, deve-se discretizar as variáveis t e x, de forma a se convergir para os valores exatos da função u(t, x) quanto menores forem os intervalos de discretização.

Desta forma, temos, na região [0, T]×[0, 1] :

$$\begin{cases} t_{\mathrm{i}} = i\Delta t, \quad com \;\; i = 0, ..., n_{\mathrm{t}} \;\; e \;\; \Delta = T/n_{\mathrm{t}} \\ x_{\mathrm{j}} = j\Delta x, \; com \;\; j = 0, ..., n_{\mathrm{x}} \;\; e \;\; \Delta t = 1/n_{\mathrm{x}} \end{cases}$$

Agora, para que se calcule u(t_{i}, x_{j}), deve-se partir das condições iniciais, a passos constantes, conforme propõe o Método de Euler.

Ao se aplicar sucessivas vezes o método, obter-se-á uma boa aproximação $u_i^j \approx u(t_i, x_j)$ da função u(t,x) calculada em t_i, x_j.

Denotaremos doravante $f(t_i, x_j)$ por f_i^j.

As expressões anteriores, compiladas e substituídas por essa nova notação ficam da seguinte forma:

$$\frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta t^2} \; - \; c^2 \frac{u_i^{j+1} - 2u_i^j + u_i^{j-1}}{\Delta x^2}, \quad \text{para i} \geq 1$$

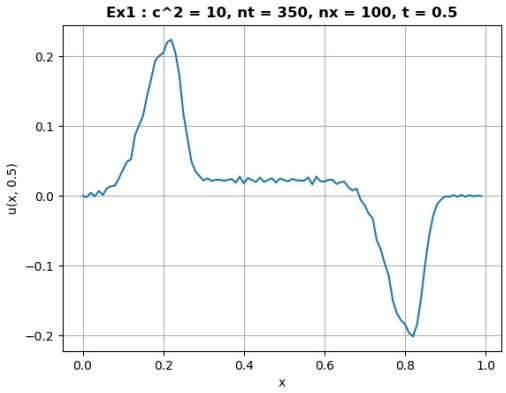
Isolando o termo de interesse u_{i+1}^j, teremos:

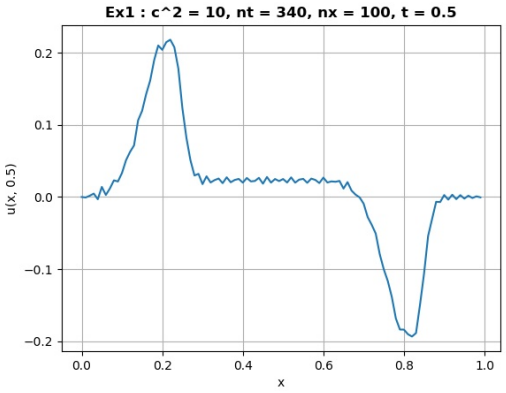
$$u_{i+1}^j = -u_{i-1}^j + 2(1-\alpha^2)u_i^j + \alpha^2(u_i^{j+1} + u_i^{j-1} + \Delta t^2 f_i^j, \quad \text{para i} \geq 1$$

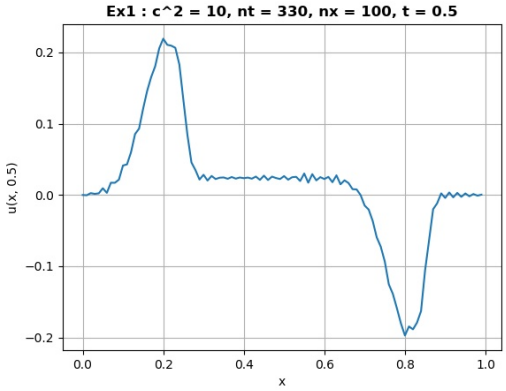
Com $\alpha = c\Delta t/\Delta x$.

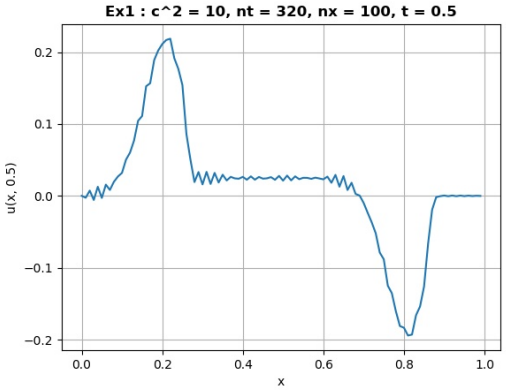
Agora é possível perceber que um certo valor de $u(t_i, x_j)$ depende sempre de 4 valores anteriores a ele próprio, que são $u_i^j, u_{i-1}^j, u_i^{j+1}$ e u_i^{j-1}. Para encontrar as respostas da equação diferencial parcial(EDP) foi necessário criar uma matriz que a representa, ela tem tamanho $n_t + 1$ e $n_x + 1$ para assim poder conter todo o intervalo para a qual está definida a equação, ou seja, o intervalo $[0, 1] \times [0, 1]$. Cada linha representa um tempo e cada coluna uma posição, como mostrado anteriormente na relação entre i e t_i e j e x_j, esses i e j representam o número da linha e da coluna da matriz,respectivamente. Para respeitar as condições iniciais estabelecidas as primeiras duas linhas foram preenchidas por zeros e para respeitar as condições de contorno a primeira e última colunas também são formadas por zeros, com isso, começando a partir da segunda linha pode-se calcular os demais $u(t_i, x_j)$, as respostas da EDP, seguindo a iteração apresentada acima.

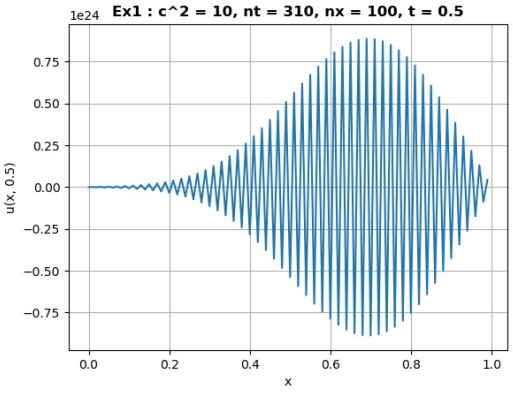
Após elaborar todo o código para resolver a equação diferencial parcial, foram realizados os testes apresentados no enunciado deste exercício-programa, o primeiro deles foi estabelecer $c^2 = 10$ e $n_x = 100$, começar com $n_t = 350$ e ir decrescendo o valor dele de dez em dez unidades, para observar em qual valor ocorreria a perda da estabilidade do método, a qual é representada por uma ampla oscilação nos valores de u(x,t), pelos gráficos abaixo podemos concluir que o problema ocorre na passagem entre $n_t = 310$ e $n_t = 320$.



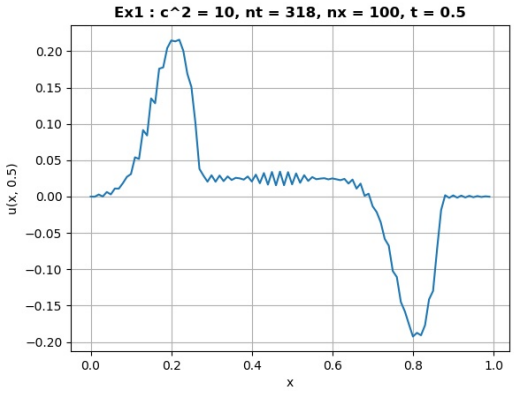


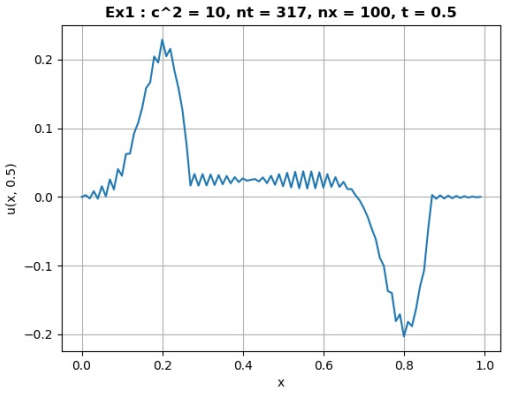


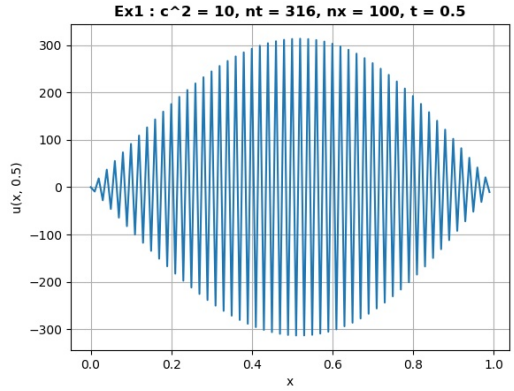


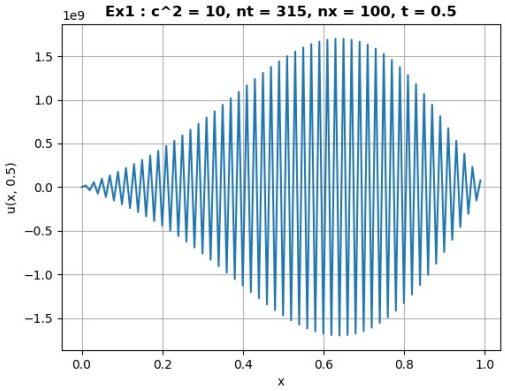


Para obter com maior precisão o caso em que começam a ocorrer problemas de estabilidade, testamos mais alguns casos, chegando à conclusão de que o método deixa de convergir para a resposta correta para n_t menor ou igual a 316.

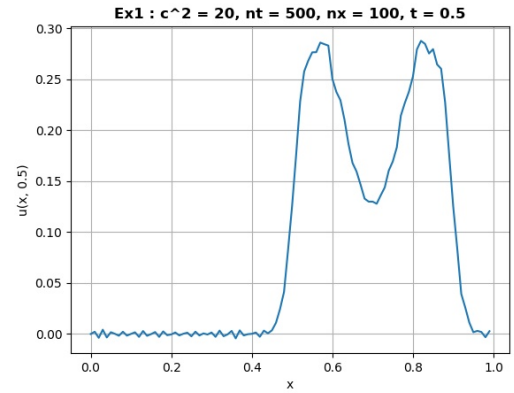


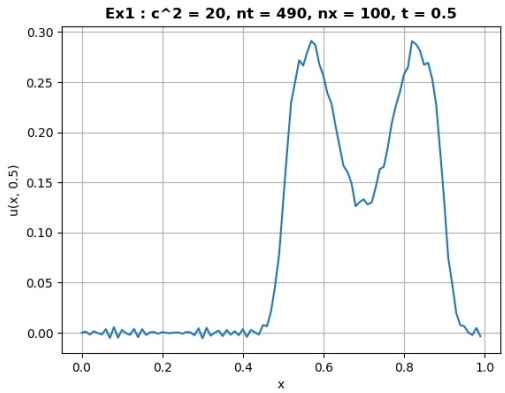


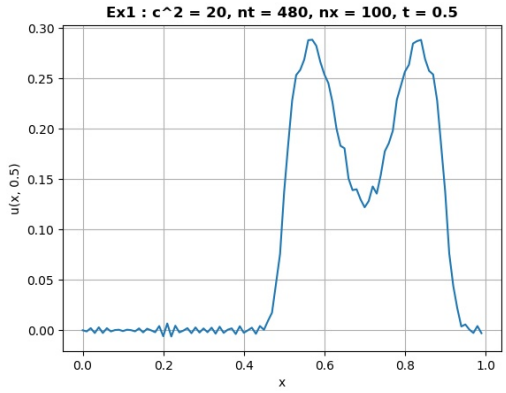


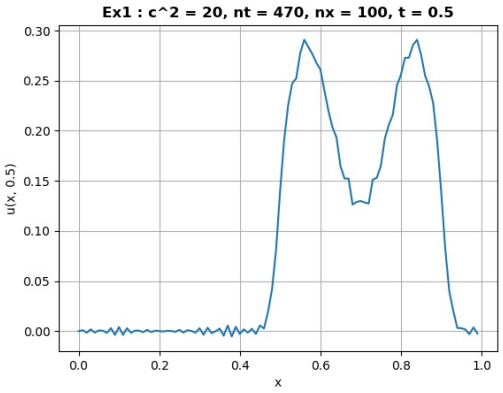


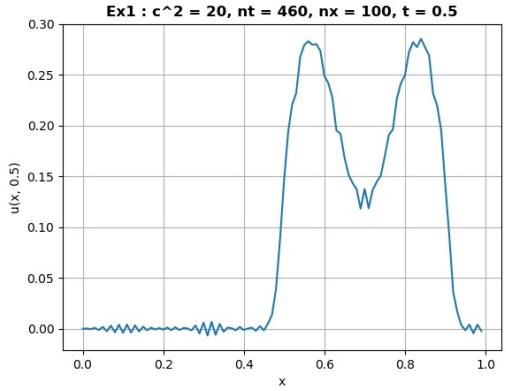
O segundo teste pedido foi definir $c^2 = 20$ e $n_x = 100$, iniciar com $n_t = 500$ e decrementá-lo de dez em dez, buscando novamente a situação na qual o método deixa de convergir para a resposta correta. Essa situação acontece com n_t entre 450 e 440.

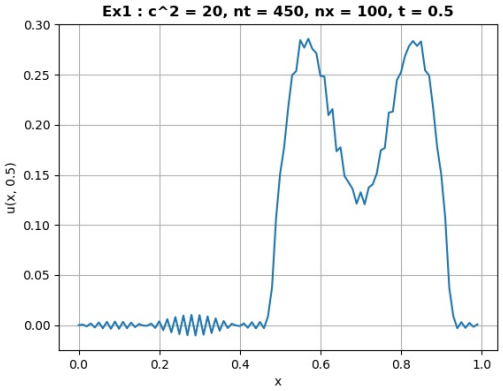


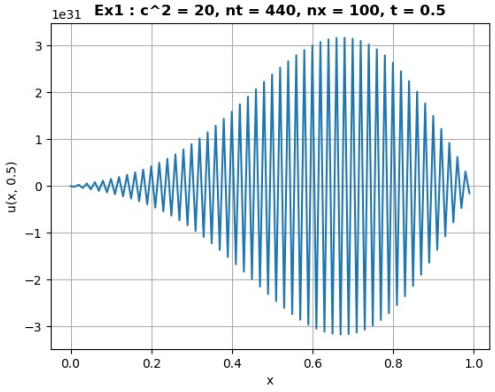








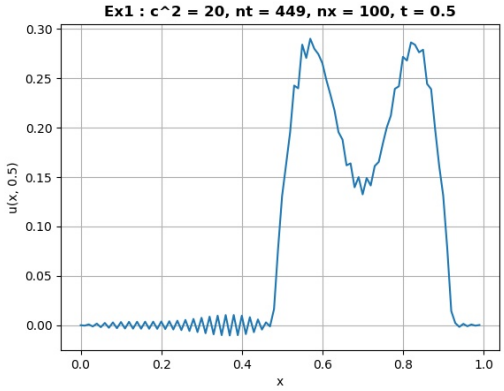


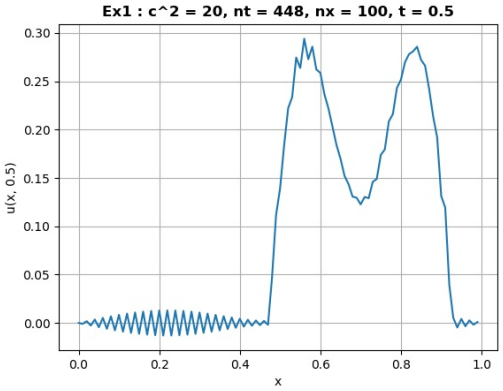


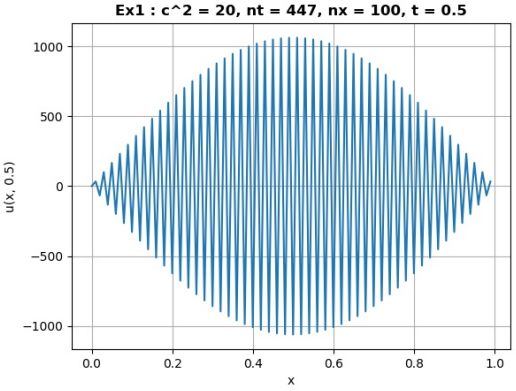
Para obter o momento preciso da perda de estabilidade do método utilizado foram realizados os testes abaixo, chegando a conclusão de que não temos mais convergência para a resposta correta para n_t menor ou igual a 447.

É interessante observar como para um mesmo n_x um valor maior de c acarreta em um n_t de corte maior também. Isso pode ser explicado pela condição CFL (Courant–Friedrichs–Lewy) para estabilidade de soluções de equações diferenciais parciais.

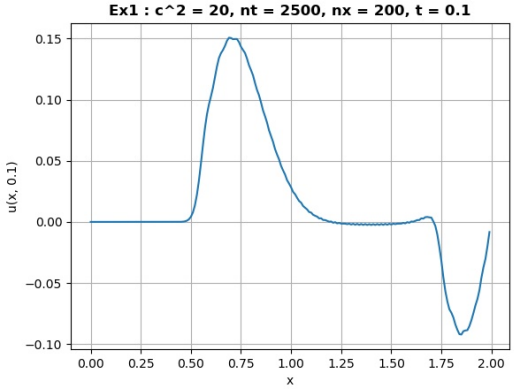
A condição $c\frac{\Delta t}{\Delta x} \leq C_{max}$ (CFL) estabelece claramente que, para mesmos valores de n_t e C_{max} (depende do tipo de método utilizado e da implementação do algoritmo), a dependência entre o valor correspondente de n_t e o de c é linear e diretamente proporcional. Uma outra forma de ver : $n_t \geq \frac{cn_x}{C_{max}}$.

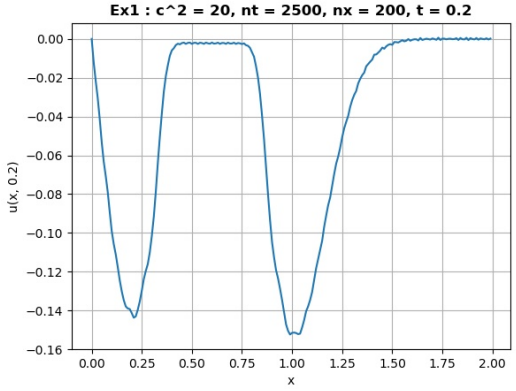


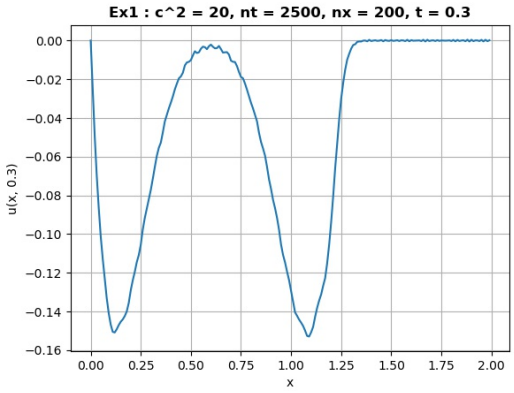


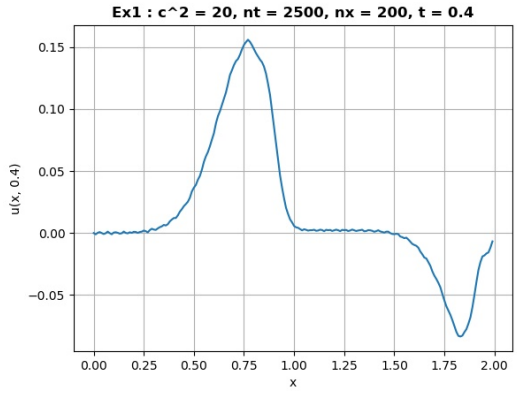


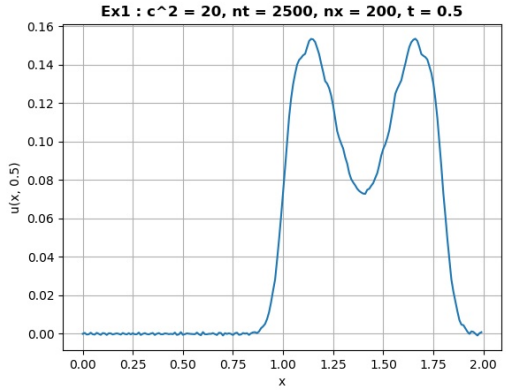
Como sugerido, a seguir tem-se gráficos para $n_t = 2500, n_x = 200$ e $c^2 = 20$ com t no intervalo fechado [0.1, 0.6].

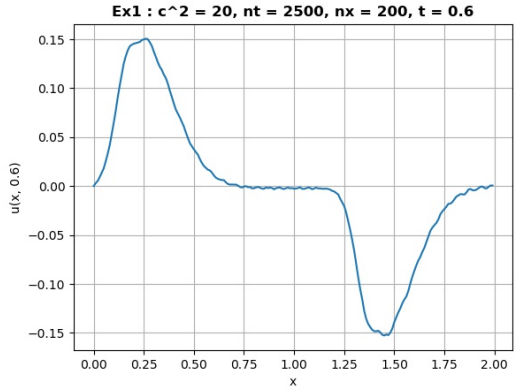












Exercício 2

Nesta seção, a equação do exercício 1 foi modificada, para contemplar outro problema. Agora o lado direito dela se tornou uma soma de funções f, ou seja, uma combinação linear de K *fontes* diferentes, ponderadas por constantes a_k^* (*intensidades*), cujos valores são desconhecidos a priori.

$$\partial_{tt}^2 u^*(t,x) - c^2 \partial_{xx}^2 u^*(t,x) = \sum_{k=1}^{K} a_k^* f_k(t,x) \; em \; [0,1] \times [0,1]$$

$$u^*(0,x) = 0 \; em \; [0,1]$$

$$\partial_t u^*(0,x) = 0 \; em \; [0,1]$$

$$u^*(t,0) = 0 \; em \; [0,1]$$

$$u^*(t,1) = 0 \; em \; [0,1]$$

O problema inverso consiste, justamente, em definir o valor das intensidades a_k^* a partir de medições de $u^*(x_r, t)$ em um único ponto $x_r \in [0,1]$.

As medições são feitas por um receptor, que será denotado como a função $d_r(t)$ a partir de agora.

O problema agora é definir uma aproximação para os valores de a_k^*, que serão chamadas de intensidades a_k, a partir dos valores medidos de $u^*(x_r, t) = d_r(t)$.

Neste caso, sabemos os valores das fontes f_k e podemos dividir a nossa equação diferencial em K partes independentes, graças ao fato de que ela é linear. Para um a_j específico, temos:

$$\partial^2_{tt} u_k(t,x) - c^2 \partial^2_{xx} u_k(t,x) = a_j f_j(t,x)\ [0,1] \times [0,1]$$
$$u_k(0,x) = 0\ em\ [0,1]$$
$$\partial_t u_k(0,x) = 0\ em\ [0,1]$$
$$u_k(t,0) = 0\ em\ [0,1]$$
$$u_k(t,1) = 0\ em\ [0,1]$$

A título de revisão, a seguir vem a fórmula que define como é feito o cálculo de f_k.

$$f_k(t,x) = \begin{cases} f_k(t,x) = 1000c^2(1 - 2\beta^2 t^2 \pi^2)e^{-\beta^2 t^2 \pi^2}, & x = x_k \\ 0, & x \neq x_k \end{cases}$$

Para que o problema inverso seja de fato resolvido, é definida uma função custo, que depende dos parâmetros a_j (desconhecidos) resultando em um problema dos Mínimos Quadrados.

Ao analisarmos a equação diferencial deste exercício, é fácil concluir que $u(t,x) = \sum_{k=1}^{K} a_k u_k(t,x)$ e podemos, consequentemente, definir uma função de custo com a seguinte expressão:

$$E(a_1, ..., a_K) = \frac{1}{2(t_f - t_i)} \int_{t_i}^{t_f} (\sum_{k=1}^{K} a_k u_k(t, x_r) - d_r(t))^2\ dt$$

Busca-se minimizar a função custo, logo devem ser calculadas suas derivadas parciais em relação a cada uma das intensidades a_j, com $j = 1, 2...K$.

$$\frac{\partial E(a_1, ..., a_K)}{\partial a_j} = \frac{1}{2(t_f - t_i)} \int_{t_i}^{t_f} 2u_j(t, x_r)(\sum_{k=1}^{K} a_k u_k(t, x_r) - d_r(t))\ dt$$

$$\frac{\partial E(a_1, ..., a_K)}{\partial a_j} = \frac{1}{t_f - t_i} \int_{t_i}^{t_f} u_j(t, x_r)(\sum_{k=1}^{K} a_k u_k(t, x_r)) - u_j(t, x_r) d_r(t)\ dt$$

Igualando a expressão da derivada a zero (Método dos Mínimos Quadrados), tem-se:

$$\frac{1}{t_f - t_i} \int_{t_i}^{t_f} u_j(t, x_r)(\sum_{k=1}^{K} a_k u_k(t, x_r)) - u_j(t, x_r) d_r(t)\ dt = 0$$

$$\int_{t_i}^{t_f} u_j(t, x_r)(\sum_{k=1}^{K} a_k u_k(t, x_r))\ dt - \int_{t_i}^{t_f} u_j(t, x_r) d_r(t)\ dt = 0$$

$$\sum_{k=1}^{K} a_k (\int_{t_i}^{t_f} u_j(t, x_r) u_k(t, x_r)\ dt) = \int_{t_i}^{t_f} u_j(t, x_r) d_r(t)\ dt$$

O resultado acima é realizado para cada a_j do problema, de modo que pode-se construir o sistema linear a seguir com as expressões obtidas:

$$\begin{bmatrix} \int_{t_i}^{t_f} u_1(t,x_r)u_1(t,x_r)\,dt & \int_{t_i}^{t_f} u_1(t,x_r)u_2(t,x_r)\,dt & \dots & \int_{t_i}^{t_f} u_1(t,x_r)u_K(t,x_r)\,dt \\ \int_{t_i}^{t_f} u_2(t,x_r)u_1(t,x_r)\,dt & \int_{t_i}^{t_f} u_2(t,x_r)u_2(t,x_r)\,dt & \dots & \int_{t_i}^{t_f} u_2(t,x_r)u_K(t,x_r)\,dt \\ \vdots & \vdots & \vdots & \vdots \\ \int_{t_i}^{t_f} u_K(t,x_r)u_1(t,x_r)\,dt & \int_{t_i}^{t_f} u_K(t,x_r)u_2(t,x_r)\,dt & \dots & \int_{t_i}^{t_f} u_K(t,x_r)u_K(t,x_r)\,dt \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_K \end{bmatrix} = \begin{bmatrix} \int_{t_i}^{t_f} u_1(t,x_r)d_r(t)\,dt \\ \int_{t_i}^{t_f} u_2(t,x_r)d_r(t)\,dt \\ \vdots \\ \int_{t_i}^{t_f} u_K(t,x_r)d_r(t)\,dt \end{bmatrix}$$

Método Cholesky

Considerando um sistema linear representado por matrizes $Ba = c$, sendo a a matriz das incógnitas, temos:

$$B = LL^t$$

Sendo l_{ij} os termos da matriz L e b_{ij} da matriz B e $i = 1, 2, ..., n$, podemos calcular l_{ij} da seguinte maneira:

$$l_{ii} = \sqrt{b_{ii} - \sum_{k=1}^{i-1}(l_{ik})^2}\ com\ l_{11} = \sqrt{b_{11}}$$

$$l_{ij} = \frac{1}{l_{jj}}(b_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk})\ para\ i \neq j$$

Para elementos a esquerda da diagonal principal temos que $l_{ij} = 0$.

Com isso obtemos a matriz L, então, pode-se rescrever o sistema assim:

$$LL^T a = c$$

Podemos definir a matriz y como $y = L^T a$ e o sistema fica:

$$Ly = c$$

Como$L$ e uma matriz triangular inferior obtêm-se facilmente a matriz y por substituição, então a substituímos em:

$$L^T a = y$$

Finalmente resolve-se esse sistema por substituição retroativa, obtendo-se o vetor de incógnitas a.

Método SOR

Dado o sistema $Ba = c$ obtemos a solução dele realizando a iteração apresentada a seguir 10000 vezes para cada a_i,$i = 1, 2, ..., n$:

$$a_i^k = (1-\omega)a_i^{k-1} + \frac{\omega}{b_{ii}}\left(c_i - \sum_{j=i+1}^{i-1} b_{ij}a_j^k - \sum_{j=1}^{n} b_{ij}a_j^{k-1}\right)$$

Para este exercício-programa $\omega = 1.6$ e $a_i^0 = 0$, nas primeiras iterações usou-se um ω menor para evitar que o vetor inicial não convirja para a resposta correta.

Integração

Considerando a utilização de um computador para a resolução deste problema, foi utilizada a Fórmula dos Trapézios para aproximar as integrais:

$$\int_{s_0}^{s_n} g(s)\ ds \approx \frac{\Delta t}{2}\left[g(s_0) + g(s_n) + 2\sum_{i=1}^{n-1} g(s_i)\right]$$

com $s_i = s_0 + i\Delta t$ sendo $i = 0, 1, .., n$ e $n = (s_n - s_0)/\Delta t$, para uma função g : $\mathbb{R} \to \mathbb{R}$.

Solução técnica do exercício 2

Discorrer-se-á agora sobre como o sistema linear foi resolvido tecnicamente (usando a linguagem Python):

Para se montar o sistema linear descrito, foi necessário definir uma função chamada *linearSystem()*, a qual deve montar uma matriz numpy com os valores das integrais que compõem a matriz B.

Foi, portanto, criada a função *(*integral()) toma como parâmetros duas funções u1 e u2 e calcula a integral do produto entre elas no intervalo $[s_0, s_N]$ usando a Fórmula dos Trapézios, já descrita anteriormente. O cálculo é discretizado em intervalos iguais de tempo Δt.

Devido à necessidade de funções como entrada da função *integral*, em vários momentos o código necessitou ser descrito por funções lambda, o que foi feito possível graças à sintaxe do Python. O uso das lambdas pode ser visto na seção Apêndice deste relatório.

Para juntar todas essas funções, foi criado um método *createFuncs()*, o qual compila em um numpyndarray cada uma das funções lambda u(t) soluções da equação diferencial deste exercício, a partir do método EDO() descrito no exercício 1, mas agora com valores variáveis de x_c.

Os valores de x_c estão, neste caso, em um vetor *xcs[]*, determinado em cada uma das subseções deste exercício.

Com o sistema linear descrito, pôde-se calcular os valores das intensidades a_k^* usando os métodos SOR e Cholesky descritos anteriormente. Isso foi possível pois, como pode ser facilmente observado na dedução deste sistema linear, a matriz B é simétrica. Nenhuma verificação é feita, em contrapartida, para garantir se essa matriz é positiva, então é possível que sob circunstâncias diferentes das fornecidas neste EP os métodos SOR e Cholesky podem não convergir e/ou apresentar erro de execução caso a matriz B não seja positiva.

Cálculos residuais e de erro

A partir dos valores das intensidades calculadas a_k, foi possível fazer uma análise comparativa entre o que se esperava e o que foi obtido (nos casos K = 3 e K = 10) e outra de resíduo (para qualquer um dos casos). A seguir a fórmula usada para o cálculo residual (função custo).

$$E(\tilde{a}_1, ..., \tilde{a}_K) = \frac{1}{2(t_f - t_i)} \int_{t_i}^{t_f} (\sum_{k=1}^{K} \tilde{a}_k u_k(t, x_r) - d_r(t))^2\ dt$$

Sabendo-se a priori que a_k é o valor encontrado da resolução do sistema linear, o erro pode ser calculado da seguinte forma (levar em conta que $a\ k$ é o valor calculado e a_k^* é o valor esperado):

$$erro = \sqrt{\sum_{k=1}^{K} (a\ - a_k^*)^2}$$

Resultados obtidos

Conforme mencionado brevemente até agora, foram propostos 3 subproblemas neste exercício, os quais procuram resolver o problema inverso a partir dos valores de K fontes e K medições.

K=3

A resposta esperada era [0.1,40.0,7.5]. Com o método Cholesky, obteve-se esse mesmo vetor considerando um arredondamento de uma casa decimal e o valor do erro foi $3.48 * 10^{-12}$ com um resíduo de $1.62 * 10^{-23}$. Utilizando-se o método SOR também foi obtido o vetor esperado, sendo o erro e o resíduo iguais a $3.48 * 10^{-12}$ e $1.62 * 10^{-23}$, respectivamente.

K=10

A resposta esperada era [7.3,2.4,5.7,4.7,0.1,20.0,5.1,6.1,2.8,15.3]. Novamente com o método Cholesky, obteve-se esse mesmo vetor considerando um arredondamento de uma casa decimal e o valor do erro foi $4.82 * 10^{-11}$ com um resíduo de $1.64 * 10^{-24}$. Utilizando-se o método SOR também foi obtido o vetor esperado, sendo o erro e o resíduo iguais a $4.88 * 10^{-11}$ e $1.64 * 10^{-24}$, respectivamente.

K=20

Neste caso não temos a resposta esperada e, portanto, pode-se apenas calcular o resíduo. A resposta obtida com as respostas apresentadas em ordem crescente é [-17.2,-11.7, -11.5, -6.2, -5.0, -3.9, -2.4, 0.5, 6.5, 9.6, 11.1, 11.5, 12.3, 13.0, 14.7, 20.3, 21.1, 22.6, 37.4, 65.2] com arredondamento de uma casa decimal e o resíduo vale $6.13 * 10^{-4}$.

Vale notar que tanto o erro quanto o resíduo foram praticamente iguais nos dois métodos, com o Cholesky obtendo esses parâmetros um pouco menores se forem observadas mais casas decimais na comparação.

Exercício 3

Nesta seção, a função $d_{r_{10}}$ foi modificada para representar de forma um pouco mais fidedigna a verdadeira natureza do problema que foi resolvido.

As medições feitas pelos *receptores* d_r não são sempre precisas e pequenas mudanças nos valores fornecidos por esses detectores acarretam em soluções bastante diferentes para o sistema linear das intensidades a_k. Justamente por esse motivo é que este problema inverso é considerado *mal-posto*.

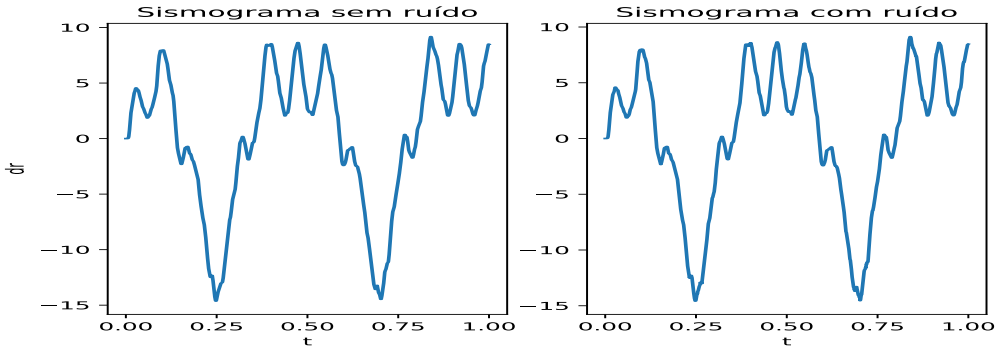
Para simular as imprecisões nas medidas, foi necessária a criação de uma função $d_r^{ruído}$(t), a qual distorce cada um dos termos do vetor $d_{r_{10}}(t)$ aleatoriamente, segundo um nível de ruído desejado. Os valores escolhidos para η foram de $10^{-3}, 10^{-4}$ e 10^{-5}.

$$d_r^{ruido}(t) = (1 + \delta(t))d_r^{10}(t) \; para \; t \in [t_i, t_f]$$

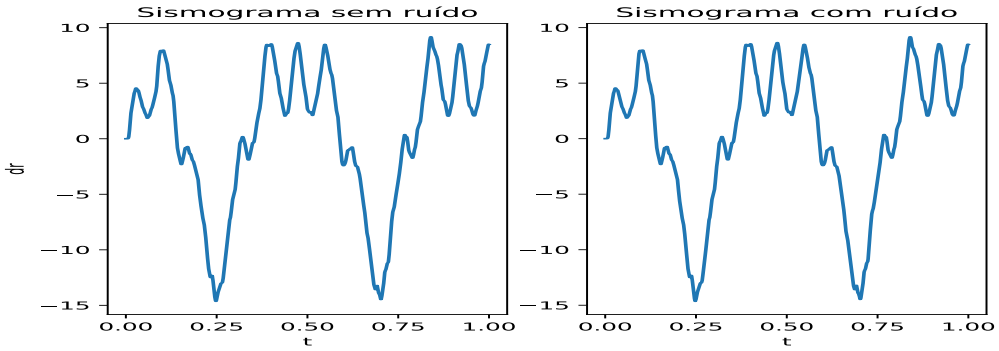
$$\delta(t) = \left(\eta \max_{s \in [t_i, t_f]} |d_r^{10}(s)| \right) v(t)$$

Neste programa foi utilizada a função *random* do *numpy* para realizar o papel de $v(t)$, ajustando-a para que forneça valores de -1.0 a 1.0.

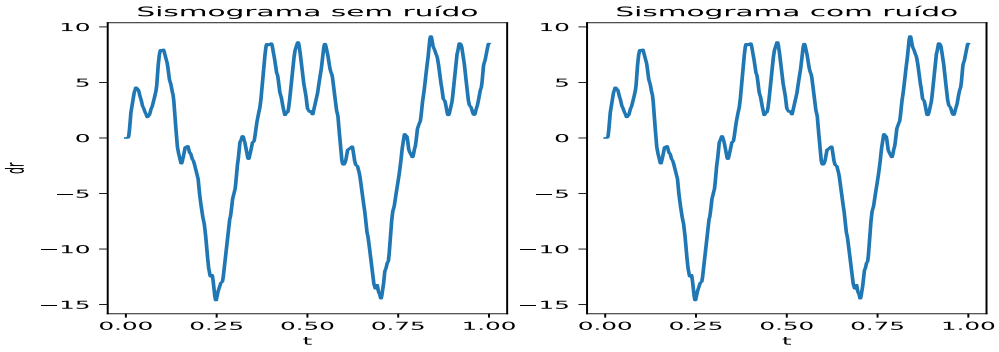
$$ruido = 100 \frac{\int_{t_i}^{t_f} |d_r^{10}(t) - d_r^{ruido}(t)| \; dt}{\int_{t_i}^{t_f} |d_r^{10}(t)| \; dt}$$



$\eta = 10^{-3}$



$\eta = 10^{-4}$



$\eta = 10^{-5}$

Resultados obtidos com K = 10 e ruído

Foram realizados os testes para os três valores de η(ruído) a seguir:

$\eta = 10^{-3}$

A resposta esperada era [7.3,2.4,5.7,4.7,0.1,20.0,5.1,6.1,2.8,15.3]. Com o método Cholesky, obteve-se [8.2, 1.3, 5.8, 5.5, 0.5, 19.6, 5.0, 6.2, 2.8, 15.3] considerando um arredondamento de uma casa decimal e o valor do erro foi 1.79 com um resíduo de $3.66 * 10^{-4}$. Utilizando-se o método SOR também foi obtido o mesmo vetor de respostas que com Cholesky, sendo o erro e o resíduo iguais a 1.79 e $3.66 * 10^{-4}$, respectivamente. O nível de ruido para este caso foi 4.80%.

$\eta = 10^{-4}$

A resposta esperada era [7.3,2.4,5.7,4.7,0.1,20.0,5.1,6.1,2.8,15.3]. Com o método Cholesky, obteve-se [7.2, 2.4, 5.6, 4.6, 0.1, 20.0, 5.1, 6.1, 2.8, 15.3] considerando um arredondamento de uma casa decimal e o valor do erro foi 0.16 com um resíduo de $4.66 * 10^{-6}$. Utilizando-se o método SOR também foi obtido o vetor esperado, sendo o erro e o resíduo iguais a 0.16 e $4.66 * 10^{-6}$, respectivamente. O nível de ruido para este caso foi 5.11%.

$\eta = 10^{-5}$

A resposta esperada era [7.3,2.4,5.7,4.7,0.1,20.0,5.1,6.1,2.8,15.3]. Com o método Cholesky, obteve-se esse mesmo vetor considerando um arredondamento de uma casa decimal e o valor do erro foi 0.02 com um resíduo de $3.92 * 10^{-8}$. Utilizando-se o método SOR também foi obtido o vetor esperado, sendo o erro e o resíduo iguais a 0.02 e $3.92 * 10^{-8}$, respectivamente. O nível de ruido para este caso foi 4.76%.

Conclusão

A partir dos dados analisados até aqui foi possível perceber a dificuldade de resolver o problema inverso uma vez que mesmo com um η de 10^{-3} obtiveram-se resultados diferentes dos esperados comparados ao caso ideal. É importante ressaltar que na realidade esse tipo de problema é mais complexo, pois passa-se a trabalhar num espaço de duas ou três dimensões, dificultando-se tanto da perspectiva matemática quanto computacional. Além disso, a função custo não é tão simples quanto a utilizada aqui.

Pode-se observar também neste exercício a importância do método Cholesky como resolutor de sistemas lineares com matriz B simétrica e positiva, bem como a do SOR, que converge iterativamente para a solução certa. Tudo isso possível, é claro, graças ao Método de Euler, o qual permite a aproximação de equações diferenciais sem a necessidade de achar sua solução analítica, o que é muito útil para a engenharia em geral.

Apêndice

Aqui ficam algumas partes-chave dos arquivos .py do EP inteiro.

```python
# Função da fonte
# Recebe os indices i e j onde se deseja calcular f
def funcaoF(i, j, c2, xc, nx, nt):

    t = i / nt

    betha2 = 10 ** 2
    pi2 = (np.pi) ** 2

    if j != int(xc * nx):
        return 0

    aux1 = 1000 * c2
    aux2 = 1 - 2 * betha2 * pi2 * (t ** 2)
    aux3 = np.exp(- betha2 * pi2 * (t ** 2))

    return aux1 * aux2 * aux3
```

Ex1: Função de fonte.

```python
# Soluciona u(ti, xj) para os parâmetros dados
def EDO(i, j, matrix, nt, nx, c2, T, firstTime, xc):

    deltaT = T / nt
    deltaX = 1 / nx
    alpha = math.sqrt(c2) * (deltaT / deltaX)

    if firstTime:
        matrix = m.criarMatriz(nt + 1, nx + 1)
        matrix = fillEDOmatrix(nt, nx, alpha, T, matrix, c2, xc)

    return matrix[i][j], matrix
```

Ex1: Função que popula *matrix* com os valores da solução da EDO.

```python
# Plota o gráfico de u(x, t) para um t dado
# e depende dos parâmetros nt, nx e c^2 (únicos que podem variar)
def plotArt(t):
    valoresx = np.array([])
    valoresy = np.array([])
    i = int(t*nt)

    matrix = []

    firstTime = True
    for j in range (nx):
        valoresx = np.append(valoresx, j*deltaX)

        aux = EDO(i, j, matrix, nt, nx, c2, T, firstTime, xc=0.7)
        valoresy = np.append(valoresy, aux[0])
        matrix = aux[1]

        firstTime = False

    plt.plot(valoresx, valoresy)
    nome = "Ex1 : c^2 = " + str(c2) + \
                    ", nt = " + str(nt) + \
                    ", nx = " + str(nx) + \
                    ", t = " + str(round(t,1))
    plt.title(nome,fontweight="bold")
    plt.grid(True)
    plt.xlabel("x")
    plt.ylabel("u(x, " + str(round(t,1)) + ")")

    plt.savefig("ImagensEx1/" + nome + ".jpeg")
    plt.clf()
```

Ex1: PlotArt é a função que plota os gráficos do exercício 1, dependendo dos valores atuais de c^2, n_t, n_x e t.

```
15  # Calcula a integral do produto das funções u1 e u2
16  # Os limites da integral são s0 e sN, e o passo entre
17  # as áreas somadas é deltaT
18  # Aqui a aproximação é feita usando a Fórmula dos Trapézios Composta
19  def integral(u1, u2, s0, sN, deltaT):
20      n = int( (sN - s0) / deltaT )
21      sum = 0
22      for i in range(1, n):
23          si = s0 + i * deltaT
24          sum += (u1(si) * u2(si))
25
26      sum *= 2
27      sum += (u1(s0) * u2(s0)) + (u1(sN) * u2(sN))
28
29      return sum * deltaT / 2
30
31  # Cria um sistema linear a partir das integrais
32  # das funções Ui obtidas e o dr fornecido
33  def linearSystem(K, funcs, dr, deltaT, ti, tf):
34      B = m.criarMatriz(K, K)
35      c = m.criarMatriz(K, 1)
36
37      for i in range(K):
38          for j in range(i):
39              B[i][j] = integral(funcs[i], funcs[j], ti, tf, deltaT)
40              B[j][i] = B[i][j]
41          B[i][i] = integral(funcs[i], funcs[i], ti, tf, deltaT)
42
43      for i in range(K):
44          c[i][0] = integral(funcs[i], dr, ti, tf, deltaT)
45
46      return B, c
```

Ex2: A função integral() calcula a integral do produto de duas funções u1(s) e u2(s) no intervalo $[s_0, s_N]$. A função linearSystem() usa os valores das integrais entre os produtos de u_ks para criar a matriz B do sistema linear.

```
116  # Cria uma lista de funcoes a partir de um dos parametros
117  # do vetor xc fornecido, com K posições
118  def createFuncs(xcs, K):
119      porFora = []
120      funcs = []
121      for i in range(K):
122          porFora.append(ukx(xr=xr, xc=xcs[i]))
123          funcs.append(lambda t, k = i: porFora[k][int(t * nt)])
124      return funcs
```

Ex2: Função createFuncs(), ela é quem cria as funções lambda usadas na função de integral para a resolução do sistema linear.

```
172      # Lê o arquivo dr correspondente
173      def readNPY(name):
174          return np.load(name)
175
176      # Definimos aqui a função dr
177      dr = readNPY("dr" + str(K) + ".npy")
178      def funDr(t):
179          pos = int(t * nt)
180          return dr[pos]
```

Ex2: Aqui é definida a função d_r dependendo do nome do arquivo .npy a ser consultado.

```python
# Função que calcula o dr com ruído.
# Precisa que o dr carregue o arquivo dr10
def drRuido(eta):
    dr10 = np.load("dr10.npy")
    ruido = np.array([])
    v = (lambda t : (2 * np.random.rand() - 1) )
    maximo = max(dr10)

    delta = (lambda t, eta=eta: eta * maximo * v(t))

    for i in range(len(dr10)):
        ruido = np.append(ruido, (1 + delta(i / nt)) * dr10[i])

    return ruido
```

Ex3: Função que cria artificialmente o ruído na função $d_{r_{10}}$

```python
# Calcula o nível de ruído da função dr distorcida
def nivelRuido(eta, ti, tf):
    um = (lambda t : 1)
    ruido = drRuido(eta)
    dr10 = np.load("dr10.npy")

    numerador = (lambda t : abs(dr10[int(t * nt)] -  ruido[int(t * nt)]))
    denominador = (lambda t : dr10[int(t * nt)])

    num = ex2.integral(numerador, um, ti, tf, T / nt)
    den = ex2.integral(denominador, um, ti, tf, T / nt)

    return 100 * num / den

# Retorna a função que contém os valores de dr ruidosos
def drRuidoLambda(eta):
    ruido = drRuido(eta)
    return (lambda t : ruido[int(t * nt)])
```

Ex3: A função nivelRuido() calcula o nível de ruído da nova função d_r(t). A drRuidoLambda retorna a função que retorna os valores discretos da função d_r(t)

```python
def metodoSOR (matrizA,matrizb,n):
    numIteracoes = 10000
    omega = 1.6
    iteracoes = 0
    respostas = criarMatriz(n, 1)

    while(iteracoes < 10):
        for i in range(n):
            somatorias = 0
            for j in range(i):
                somatorias = matrizA[i][j]*respostas[j][0] + somatorias
            for j in range(i+1,n):
                somatorias = matrizA[i][j]*respostas[j][0] + somatorias
            respostas[i][0] = (1/matrizA[i][i])*(matrizb[i][0]-somatorias)
        iteracoes += 1

    iteracoes = 0

    while(iteracoes < numIteracoes):
        for i in range(n):
            somatorias = 0
            for j in range(i):
                somatorias = matrizA[i][j]*respostas[j][0] + somatorias
            for j in range(i+1,n):
                somatorias = matrizA[i][j]*respostas[j][0] + somatorias
            respostas[i][0] = (1-omega)*respostas[i][0] + (omega/matrizA[i][i])*(matrizb[i][0]-somatorias)
        iteracoes += 1
    return respostas
```

Matrizes: O método SOR resolve um sistema de equações lineares quaisquer e tem garantia de convergência quando a matiz B da equação é determinada positiva.

```python
# A * x = b
# (ch * chT) * x = b
#
# ch * y  = b
# chT * x = y
def cholesky(A, b):
    ch = matrixCholesky(A)
    chT = transpostaMatriz(ch)

    # ch * y = b
    y = sistemaLouU(ch, b, isL=True)

    # chT * x = y
    x = sistemaLouU(chT, y, isL=False)


    return x
```

Matrizes: O método Cholesky fatora a matriz B em um produto de matrizes LL^T e resolve em seguida um sistema L (matriz triangular inferior) e outro U (matriz triangular superior) para chegar à solução do sistema.

Referências

[1] https://numpy.org/

[2] https://matplotlib.org/3.1.1/contents.html

[3] https://en.wikipedia.org/wiki/Courant–Friedrichs–Lewy_condition

[4] https://www.telecom.uff.br/pet/petws/downloads/apostilas/LaTeX.pdf

[5] https://edisciplinas.usp.br/pluginfile.php/5104579/mod_resource/content/3/EP2.pdf