MAP2212

Aplicações de Álgebra Linear

Enzo Valentim Cappelozza
Lucas Panfilo Donaire

Professor:
Alexandre Roma

IME-USP

10 de junho de 2022

1 Modelos de condição de contorno

Abaixo, listados estão alguns modelos para condições de contorno para equações diferenciais ordinárias.

1.1 Condição de contorno de Dirichlet

Dentre todos aqui listados, provavelmente é o mais simples de se entender e de se usar. No caso onde a nossa fução u está definida num intervalo compacto (isto é, fechado e limitado) da reta real, temos que as condições de contorno são relativas aos extremos da reta. Isto é, se tiver uma função $u : [a, b] \to \mathbb{R}$, então teremos as condições de contorno definidas por $u(a) = \alpha$ e $u(b) = \beta$ onde α, β são dois números reais.

Agora, em termos mais gerais, poderíamos tomar uma Equação Diferencial Parcial e, assim, suas condições de contorno seriam relativas ao bordo do conjunto em que a função está definida.

1.2 Condição de contorno de Neumann

Aqui, a condição de contorno é baseada no valor da função $u'(x)$ nos extremos do intervalo, isto é, a primeira derivada da função u acima citada. Por exemplo, para a equação diferencial ordinária abaixo, definida no intervalo compacto da reta $[0, 1]$

$$y'' + 3y = 1,$$

teríamos as seguinte condições de contorno:

$$y'(0) = \alpha \text{ e } y'(1) = \beta$$

Onde α, β são ambos números reais.

1.3 Condição de contorno de Robin

Esse pode ser entendido como uma "forma mista" entre os dois acima. Em geral, quando definida para uma Equação Diferencial Ordinária, em um intervalo compacto da reta, $[0, 1]$, por exemplo, temos que, para a *EDO* da forma

$$y'' + 3y = 1$$

teríamos as seguintes condições de contorno:

$$ay(0) - by'(0) = f(0)$$
$$ay(1) + by'(1) = f(1)$$

Onde f é uma função definida no nosso intervalo fechado e limitado da reta $[0, 1]$ e os números a, b são reais.

O motivo dos valores + e − multiplicando, respectivamente, os termos by' é devido pelo conceito de "derivada normal". Tal conceito, em dimensões maiores, é uma derivada direcional que é ortogonal à superfície no espaço que delimita o domínio da função. No caso, como o conjunto $[0, 1]$ é um intervalo da reta, então a derivada normal no ponto $x = 1$ aponta para a "direita" e portanto adquire um valor positivo, enquanto a derivada normal no ponto $x = 0$ aponta para a "esquerda" adotando um valor negativo. Em outras palavras, a normal em $x = 0$ aponta (para fora) com a orientação negativa e em $x = 1$ aponta (para fora) com a orientação positiva.

1.4 Condições de contorno periódicas

Aqui, temos que, quando definida num intervalo compacto da reta, os valores da aplicação de u nos extremos do intervalo resultam nos mesmos valores da imagem. Em geral, podemos elucidar a ocorrência de tais condições em equações diferenciais cujas funções resposta são periódicas. Por exemplo, as funções seno e cosseno. Em outras palavras, caso $u : [a, b] \to \mathbb{R}$ seja uma função contínua, então podemos afirmar que $u(a) = u(b) = \alpha$, onde α é um número real. Isso se dá não por mera coincidência, é claro. Ocorre que a função que é a solução para a equação diferencial é, por si só, uma função periódica, isto é, para todo x no domínio de u (ou até mesmo para extensões do domínio) se c = b - a, então $u(x) = u(x + c)$.

2 Algoritmo a ser desenvolvido

Sabemos de antemão por "solução manufaturada" que a nossa função resposta da equação diferencial dada é periódica. Ou seja, poderíamos ter, sem problemas, tomado um método de aproximação por valores centrais e utilizado a periodicidade da função. Assim, poderíamos dizer que k mod(2π)= j mod$(2\pi) \Rightarrow u_k = u_j$ e simplificar a resolução. No entanto, tentamos ao máximo fazer uma solução "o mais geral possível" para que, caso necessário, pudéssemos usar a atual implementação para resolver numericamente outros tipos de equações diferenciais. Sendo assim, tentamos implementar um processo misto de aproximação pelas laterais e pelo centro. Fora usado também o pacote *csc matrix* do *Scipy*, cuja finalidade fora a de armazenar as

matrizes esparsas resultantes do sistema linear.

Uma observação curiosa é que, ao tentarmos calcular 100.000 pontos usando *arrays* do *Numpy* ao invés do pacote do *Scipy*, fora acusado que seriam necessários cerca de 74.5gb de memória RAM. Claramente, a implementação de uma maneira mais eficiente de se armazenar uma matriz se mostrou necessária caso quiséssemos prosseguir a fim de obter maior grau de precisão.

3 Considerações sobre o algoritmo

Julgamos que o algoritmo é relativamente eficiente em tempo principalmente devido aos métodos utilizados. Um deles, o pacote do *spsolve* destinado à solução de matrizes esparsas e/ou de bandas. Uma outra opção que tínhamos era usar a *solveh* também do *Scipy* que é destinada à resolução de matrizes de banda simétricas por meio de fatoração LU. O uso do *spsolve* foi ultimamente por fins de conveniência.

Também utilizamos o pacote de matrizes esparsas do *Scipy* pelo simples motivo de que, dado um número suficientemente de pontos a serem inseridos (algo em torno de 100.000), ambas máquinas não dispunham de memória RAM o suficiente para comportar a quantidade de dados.

Utilizamos também o *Numpy* através de arrays.

3.1 Detalhando o sistema linear

Usamos as seguintes aproximações de diferenças finitas:

$$f_k = -\frac{1}{h^2}(-u_{k-1} + 2u_k + 1u_{k+1}) \tag{1}$$

$$f_k = -\frac{1}{12h^2}(-u_{k-2} + 16u_{k-1} - 30u_k + 16u_{k+1} - u_{k+2}) \tag{2}$$

A aproximação (1) tem erro da ordem $\mathrm{O}(h^2)$, e (2) tem ordem $\mathrm{O}(h^4)$. Usamos (1) somente nas expressões de f_1, f_2, f_{n-1} e f_{n-2}, e (2) em todo o resto
A partir disso, e sabendo que $u_n = u_0 = e$, temos o seguinte sistema de equações:

$$\begin{cases} (-h^2)f_1 = e - 2u_1 + u_2 \\ (-12h^2)f_2 = u_1 - 2u_2 + u_3 \\ (-12h^2)f_3 = -u_1 + 16u_2 - 30u_3 + 16u_4 - u_5 \\ \vdots \\ (-12h^2)f_i = -u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2} \\ \vdots \\ (-12h^2)f_{n-3} = -u_{n-5} + 16u_{n-4} - 30u_{n-3} + 16u_{n-2} - u_{n-1} \\ (-h^2)f_{n-2} = u_{n-3} - 2u_{n-2} + un - 1 \\ (-h^2)f_{n-1} = u_{n-2} - 2u_{n-1} + e \end{cases}$$

Que, organizado na forma matricial, nos dá:

$$\begin{bmatrix} 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & \dots \\ 0 & -1 & 16 & -30 & 16 & -1 & 0 & 0 & \dots \\ 0 & 0 & -1 & 16 & -30 & 16 & -1 & 0 & \dots \\ & & & & \vdots & & & & \\ \dots & 0 & 0 & 0 & -1 & 16 & -30 & 16 & -1 \\ \dots & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 1 & \dots & 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} e \\ u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-3} \\ u_{n-2} \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} -h^2 f_1 \\ -h^2 f_2 \\ -12h^2 f_3 \\ -12h^2 f_4 \\ \vdots \\ -12h^2 f_{n-3} \\ -h^2 f_{n-2} \\ -h^2 f_{n-1} 1 \end{bmatrix}$$

Como e não deve ficar na nossa matriz de incógnitas, rearranjamos o sistema para obter:

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & \dots \\ -1 & 16 & -30 & 16 & -1 & 0 & 0 & \dots \\ 0 & -1 & 16 & -30 & 16 & -1 & 0 & \dots \\ & & & \vdots & & & & \\ \dots & 0 & 0 & -1 & 16 & -30 & 16 & -1 \\ \dots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ \dots & 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{n-3} \\ u_{n-2} \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} -h^2 f_1 \\ -h^2 f_2 \\ -12h^2 f_3 \\ -12h^2 f_4 \\ \vdots \\ -12h^2 f_{n-3} \\ -h^2 f_{n-2} \\ -h^2 f_{n-1} \end{bmatrix} - \begin{bmatrix} e \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ e \end{bmatrix}$$

Que é o sistema da forma Ax=b que formaremos no python, e usaremos o scipy para resolver.

4 Testes de erro

Abaixo, alguns testes da nossa implementação do método de elementos finitos:

n	h (passo)	$\|\|e(h)\|\|_2$	$\|\|e(h)\|\|_\infty$	p_2	p_∞
128	0.049087	1.843915	1.646933e-01	-	-
256	0.024544	0.664607	4.175452e-02	5.464696	1.979778
512	0.012272	0.236412	1.047516e-02	5.987458	1.99496
1024	0.006136	0.083766	2.621076e-03	6.495	1.998741
2048	0.003068	0.029641	6.554118e-04	6.997817	1.999685
4096	0.001534	0.010484	1.638620e-04	7.498987	1.999921
8192	0.000767	0.003707	4.096635e-05	7.999503	1.99997
16384	0.000383	0.001311	1.024285e-05	8.499586	1.999823
32768	0.000192	0.000464	2.565505e-06	8.997136	1.997302
65536	0.000096	0.000168	6.616525e-07	9.454278	1.955097
131072	0.000048	0.000081	2.461473e-07	9.414498	1.426552
262144	0.000024	0.000161	3.909072e-07	7.702515	-0.667305
524288	0.000012	0.000817	1.426254e-06	6.820337	-1.867333
1048576	0.000006	0.004379	5.394678e-06	7.242908	-1.919306

Onde temos que $p_2 = log_2(||h(e)||_2/||h(e/2)||_2)$ e $p_\infty = log_2(||h(e)||_\infty/||h(e/2)||_\infty)$

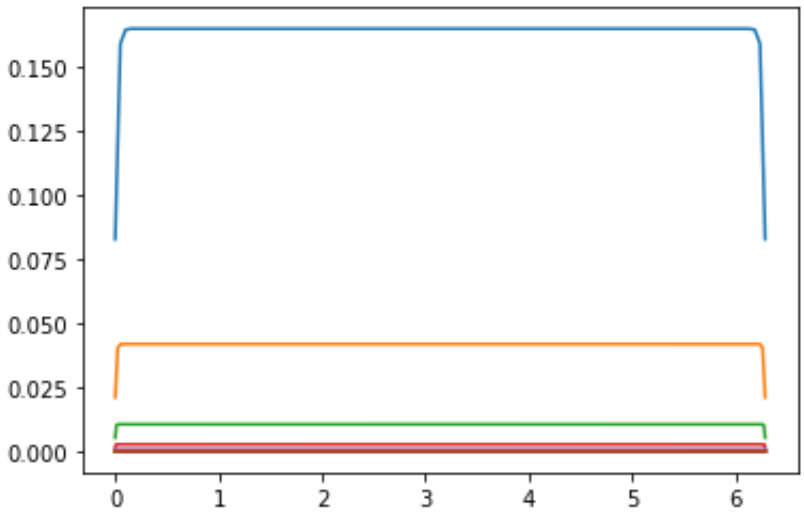

Figura 1: Erros de aproximação em valores absolutos

5 Assunto concernentes à convergência

De maneira curiosa, conseguimos aproximar a nossa função com um erro relativamente aceitável. Por vezes, o nosso erro máximo pela norma

$$||.||_\infty$$

não era superior a 10^{-5}, no entanto, o algoritmo não convergia. Assim, tentamos mudar nossa implementação e a nossa lógica. A implementação antiga envolvia uma discretização por método centrado de ordem $O(h^4)$, isto é, de ordem $p = 4$. No entanto, tal método convergia menos ainda (por mais que o erro ainda fosse tão pequeno quanto, se não menor) de maneira que, consistentemente, nosso coeficiente $log_2(||h(e)||_\infty/||h(e/2)||_\infty)$ não era superior a de 0.5. De tal maneira, supusemos que, talvez, o problema fosse justamente a implementação centrada no domínio todo.
Assim, trocamos para um regime misto de cálculo de diferenças finitas: no extremo esquerdo do domínio usamos um método de *aproximação central* cuja ordem é de $O(h^2)$, no centro (após um número de passos) utilizamos o código que havíamos feito anteriormente com ordem de $O(h^4)$ e, ao final, no extremo direito do domínio, utilizamos o método de *aproximação central* com ordem de $O(h^2)$. Notamos uma certa "melhora" na convergência do algoritmo, isto é: não mais tínhamos a ordem p convergindo para 0.5. No entanto, como se pode notar pela tabela acima,valores negativos começam a aparecer para o teste de p. Não sabemos, no entanto, o motivo que isso ocorre