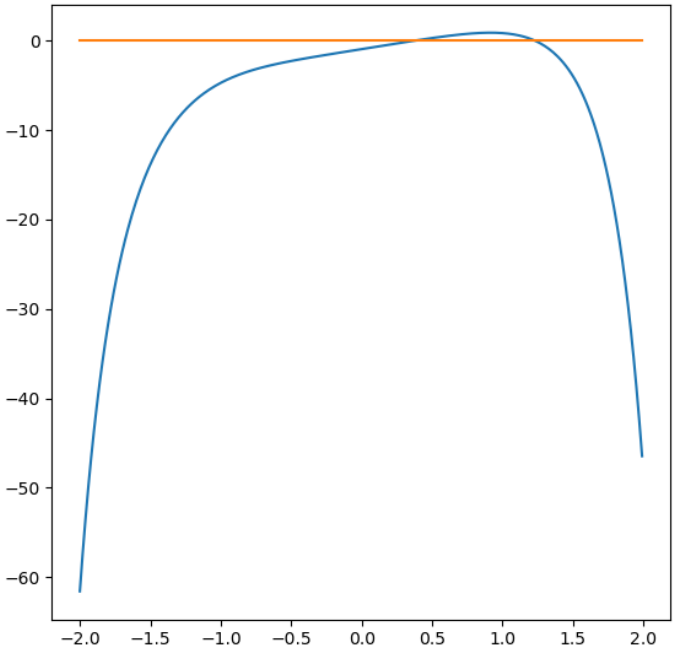
P1 MS211 - Vinícius de Oliveira Peixoto Rodrigues (245294)

Resumo

Todos os programas usados para calcular aproximações numéricas foram implementados em Python, por mim, usando os métodos estudados em sala. As bibliotecas utilizadas (Numpy e Scipy) fornecem estruturas de dados otimizadas para cálculos de Álgebra Linear, além de algumas ferramentas convenientes (multiplicação de matrizes, cálculo de inversa, norma, etc), mas a implementação dos algoritmos em si foi feita por mim. O código fonte pode ser encontrado no meu repositório do Github: https://github.com/nukelets/metodos_numericos/tree/main/p1.

1 Problemas quantitativos

1. Considere a função dada por $f(x) = \sin[(x - 0.23)^2] - e^{x^2} + 3x$ com $x \in [-2, 2]$. Aproxime a raiz ou as raízes justificando sua escolha de método, a precisão de sua aproximação da raiz ou de suas aproximações das raízes.



A partir do gráfico da função, se percebe que ela aparenta possuir duas raízes. Para aproximá-las numericamente, tentei fazer uso de 4 alternativas:

1) Bisecção: como é possível escolher por inspeção dois pontos em que o sinal da função é diferente ($x = 1$ e $x = 2$, por exemplo), o método da bisecção pode ser usado aqui – como $f(x)$ é contínua, é garantido que o método vai eventualmente convergir.

2) Método da posição falsa: pelas mesmas razões do método da bisecção, esse método deve convergir para uma das raízes também.

3) Método do ponto fixo: à primeira vista, parece tentador calcular a raiz usando a iteração $x := \frac{1}{3}(e^{x^2} - \sin[(x-0.23)^2])$; infelizmente, como será discutido mais adiante, essa aproximação se mostrou infeliz.

4) Método de Newton: esse método deve também convergir em teoria, desde que não se selecione por azar o ponto de máximo (com $f'(x) = 0$) como ponto de partida da iteração.

Para encontrar as duas raízes, estipulei visualmente os seguintes parâmetros:

- Para a raiz da esquerda, utilizei o intervalo $a = 0,\ b = 1$ para os métodos da bisecção e posição falsa, e a estimativa inicial $\frac{a+b}{2} = 0.5$ para o método de Newton
- Para a raiz da direita, utilizei o intervalo $a = 1,\ b = 2$ para os métodos da bisecção e posição falsa, e a estimativa inicial $\frac{a+b}{2} = 1.5$ para o método de Newton

Além disso, a tolerância usada foi de 0.001 (de modo que a iteração para quando a diferença entre duas estimativas sucessivas se torna menor do que 0.001).

Resultados:

A iteração $x := \frac{1}{3}(e^{x^2} - \sin[(x-0.23)^2])$ acabou crescendo de forma extremamente rápida, de modo que após a quarta ou quinta iteração passava a extrapolar o limite de precisão de ponto flutuante com o qual a biblioteca de funções matemáticas do Python consegue trabalhar. Por essa razão, são considerados aqui só os resultados dos outros métodos.

Para a primeira raiz, obteve-se:

```
❯ python3 root-finding/EstimadorRaizes.py
Método: bissecção
estimativa: 0.377048673902209, iteracoes: 40, valor: -1.2101430968414206e-12
Método: ponto falso
estimativa: 0.3770486739027259, iteracoes: 9, valor: 4.3298697960381105e-14
Método: Newton-Raphson
estimativa: 0.37704867390270813, iteracoes: 4, valor: 2.220446049250313e-16
```

E para a segunda:

```
❯ python3 root-finding/EstimadorRaizes.py
Método: bissecção
estimativa: 1.2288100394171124, iteracoes: 40, valor: 2.9967139880682225e-12
Método: ponto falso
estimativa: 1.2288100394174089, iteracoes: 241, valor: 9.090506125630782e-13
Método: Newton-Raphson
estimativa: 1.228810039417539, iteracoes: 7, valor: -6.661338147750939e-15
```

Como normalmente acontece, a estimativa do método de Newton foi a mais precisa. Ademais, a estimativa parece ser boa, visto que o erro absoluto é da ordem de 10^{-14}.

2. Considere a função $f(x) = x + e^{-x^2}$ na reta real. Como identificar se esta função $f(x)$ tem raiz ou raízes reais? Qual seria o método mais adequado para se identificar rapidamente uma boa aproximação dessas raízes? Qual controle de aproximação você usaria?

Para identificar se a função tem raízes reais, é possível observar primeiramente que

$$x + e^{-x^2} = 0 \Rightarrow -x = e^{-x^2} \Rightarrow x < 0$$

$$\ln(-x) = -x^2 \Rightarrow 0 < (-x) < 1 \Rightarrow -1 < x < 0$$

de modo que as raízes da função estão necessariamente no intervalo $(-1, 0)$. Suponha que $f(x)$ possui ao menos duas raízes distintas a e b. Pelo teorema de Rolle, deve haver algum ponto $c \in [a, b]$ tal que $f'(c) = 0$. Contudo, como $x < 0$,

$$f'(x) = 1 - 2xe^{-x^2} > 0$$

o que é uma contradição. Isso implica que $f(x)$ tem no máximo uma raiz, localizada no intervalo $(-1, 0)$. Além disso, observando que $f(-1) = -1 + 1/e < 0$ e $f(0) = 1 > 0$, é possível concluir que $f(x)$ possui exatamente uma raiz real.

Como i) $f(x)$ e suas derivadas são contínuas (em especial $f'(x) \neq 0$) ii) $f(x)$ não tende assintoticamente a 0 e iii) $f(x)$ tem uma só raiz, o critério de convergência quadrática garante que o método de Newton eventualmente converge para a raiz da função, de modo que esse parece ser o método de aproximação mais adequado.

Resultados

```
Método: Newton-Raphson
estimativa: -0.6529186404192048, iteracoes: 7, valor: -1.1102230246251565e-16
```

Foi utilizada uma tolerância de 0.001 (i.e., o método para de iterar quando a diferença absoluta entre duas estimativas seguidas passa a ser menor que 0.001). Percebe-se que o método convergiu após apenas 7 iterações, atingindo um erro final da ordem de 10^{-16}, o que parece ser um bom resultado.

4. Considere a matriz e o vetor dados pelo programa abaixo e resolva um sistema do tipo m.x = b. Justifique cuidadosamente sua escolha de método e critique avaliativamente seu resultado.

Primeiramente, fiz uma implementação das matrizes m e b em Python:

```
n = 251

m = np.zeros((n, n))

for i in range(n):

    m[i, i] = 2.35

for i in range(n-1):

    m[i, i+1] = -0.78

    m[i+1, i] = -0.41

for i in range(n-25):

    m[i, i+25] = -0.51

    m[i+25, i] = -0.28


b = np.zeros(n)

for i in range(0, n, 2):

    b[i] = 1.5

for i in range(1, n-1, 2):

    b[i] = 0.75
```

Em seguida, contemplei qual dos métodos estudados em sala deveria ser utilizado para resolver o problema; a minha primeira impressão foi que o método de Gauss-Seidel seria o menos computacionalmente custoso para esse problema (visto que a matriz é esparsa, de modo que a), mas para ter uma confirmação empírica decidi implementar e testar três métodos diferentes de solução de sistemas lineares: decomposição LU e Jacobi/Gauss-Seidel:

i) Decomposição LU

A ideia utilizada é relativamente simples: para se resolver o sistema $\boldsymbol{A}x = \boldsymbol{B}$, se decompõe a matriz $\boldsymbol{A} = \mathbf{PLU}$, onde $\boldsymbol{L}, \boldsymbol{U}$ são matrizes triangulares inferiores e superiores respectivamente e $\boldsymbol{P}$ é uma matriz de permutação de linhas. Tem-se que $\mathbf{PLU}x = \boldsymbol{B} \Rightarrow \boldsymbol{L}(\boldsymbol{U}x) = \boldsymbol{P^{-1}B}$; tendo em mente que $\boldsymbol{P^{-1}} = \boldsymbol{P^T}$ e fazendo a substituição $y = \boldsymbol{U}x$, resolve-se primeiramente $\boldsymbol{L}y = \boldsymbol{P^T B}$ e, em seguida, $\mathbf{U}x = y$; a vantagem é que para os dois casos só é preciso fazer uma substituição direta e uma substituição reversa, em vez de fazer o procedimento completo da eliminação de Gauss (apesar de, por debaixo dos panos, a obtenção da decomposição PLU da matriz $\boldsymbol{A}$ potencialmente fazer uso da eliminação de Gauss ou de algum outro algoritmo otimizado).

ii) Jacobi/Gauss-Seidel

A minha implementação foi praticamente igual à apresentada pelo professor nas notas de aula: no sistema

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ &\cdots \\ a_{n1}x_1 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

é possível observar que $x_i = \left(-\sum_{j \neq i} \frac{a_{\mathrm{ij}}}{a_{\mathrm{ii}}} x_j \right) + \frac{b_i}{a_{\mathrm{ii}}}$, de modo que isso sugere a iteração de ponto fixo

$$x_i^{(k+1)} = \left(-\sum_{j \neq i} \frac{a_{\mathrm{ij}}}{a_{\mathrm{ii}}} x_j^{(k)} \right) + \frac{b_i}{a_{\mathrm{ii}}}$$

Esse é o método de Jacobi. Uma forma alternativa de se escrever esse método, em notação matricial, é

$$x^{(k+1)} = \boldsymbol{P}x + d,$$

$$\boldsymbol{P} = \begin{pmatrix} 0 & \frac{-a_{12}}{a_{11}} & \cdots & \frac{-a_{1n}}{a_{11}} \\ \frac{-a_{21}}{a_{22}} & 0 & \cdots & \frac{-a_{2n}}{a_{22}} \\ & \cdots & & \\ \frac{-a_{n1}}{a_{nn}} & \frac{-a_{n2}}{a_{nn}} & \cdots & 0 \end{pmatrix}$$

$$d = \begin{pmatrix} \frac{b_i}{a_{ii}} \end{pmatrix}$$

Uma otimização possível, que visa a acelerar a convergência das estimativas, é "reutilizar" as estimativas iniciais: utilizam-se os $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ para se calcular a estimativa de $x_i^{(k+1)}$. Essa otimização é o método de Gauss-Seidel.

Resultados

Nota: para os métodos de Jacobi/Gauss-Seidel, foram realizadas 100 iterações.

Os resultados (que não se encontram escritos aqui por serem numerosos, mas que estão disponíveis em arquivos .csv na pasta do meu repositório do Github, assim como em anexo na submissão do Classroom) tiveram os seguintes erros:

```
❯ python3 linear_systems.py
erro maximo lu: 3.1086244689504383e-15
erro maximo jacobi: -4.475253501112775e-11
erro maximo gauss-seidel: 5.551115123125783e-16
```

onde o erro é calculado como

$$\max\{e_i\}, \begin{pmatrix} e_1 \\ e_2 \\ \dots \\ e_n \end{pmatrix} = \boldsymbol{A}x - b$$

Como era de se esperar, o método de Jacobi teve o erro mais alto dos três. É importante observar que os erros do resultado com decomposição LU e pelo método de Gauss-Seidel são um pouco enganosos, porque são muito pequenos e entram no limite da precisão de ponto flutuante da biblioteca de álgebra linear numérica utilizada (numpy).

O método de Gauss-Seidel converge de forma extremamente rápida (passando dos limites de ponto flutuante antes da décima iteração) e realiza muito menos operações (complexidade $O(n^2)$) que a decomposição LU ou Gauss-Jordan (complexidade $O(n^3)$), de modo que é o melhor método para esse problema (onde se tem uma matriz esparsa e diagonal dominante).

5. Considere o sistema não linear dado por

$$\begin{aligned} f(x,y,z) &= 3x^2y^2 - z + 2 \\ g(x,y,z) &= x - 2y^2 - 3z^2 + 12 \\ h(x,y,z) &= x + y - 0.5 \end{aligned}$$

Identifique uma terna (x,y,z) para a qual f, g e h se anulam simultaneamente. Avalie seu resultado e comente a precisão.

Para se encontrar o ponto (x, y, z) que anula as três equações simultaneamente, escolhi utilizar o método de Newton não linear, que se trata basicamente de uma generalização do método de Newton em uma variável. Primeiramente, se reescreve o problema em notação matricial:

$$F(x,y,z) = \begin{pmatrix} 3x^2y^2 - z + 2 \\ x - 2y^2 - 3z^2 + 12 \\ x + y - 0.5 \end{pmatrix}$$

O método de Newton em si consiste em fazer a iteração

$$x_{n+1} = x_n - J_F^{-1}(x_n)F(x_n)$$

onde J_F é o jacobiano de $F(x, y, z)$, definido como o operador

$$J_F = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} \\ \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} & \frac{\partial h}{\partial z} \end{pmatrix}$$

Calculando-se explicitamente as derivadas parciais, encontra-se

$$J_F = \begin{pmatrix} 6\mathrm{xy}^2 & 6\mathrm{yx}^2 & -1 \\ 1 & -4y & -6z \\ 1 & 1 & 0 \end{pmatrix}$$

Resultados

Utilizando uma tolerância de 0.001 (de modo que a iteração para assim que $\|x_{n+1} - x_n\| < 0.001$), obteve-se o seguinte resultado:

```
❯ python3 nonlinear_newton.py
Metodo: Newton nao-linear
estimativa: [0.25793895 0.24206105 2.01169513]
iteracoes: 96
erro: [-9.82587345e-10 -7.24912574e-09  0.00000000e+00]
```

O erro, da ordem de 10^{-9}, parece ter uma precisão aceitável; o número de iterações necessárias para atingir a tolerância, contudo, parece um pouco alto, mas acredito que isso seja causado por causa do número maior de dimensões do problema, que deve fazer com que o método leve mais tempo para convergir. Em particular, como há $n = 3$ dimensões e $91^{1/3} = 4.57$, um número que é compatível com o número de iterações observado nos outros problemas da prova (que têm a mesma tolerância), especulo que talvez o número de iterações cresça exponencialmente com o número de dimensões (complexidade $O(e^n)$).

2 Questões qualitativas

3. Dos métodos apresentados para aproximar a raiz de uma função, qual ou quais os que lhe parecem mais úteis e por quê? Há algum método que você acredita que nunca vai usar? Justifique cuidadosamente todas as suas opiniões.

O método de Newton me pareceu o mais útil, visto que ele proporciona convergência quadrática para uma classe razoável de funções. Além disso, ele apresenta um custo computacional relativamente baixo – o custo de se calcular a derivada é compensado pelo menor número de iterações. O método também possui aplicações interessantes em problemas de otimização, onde se procuram os pontos críticos de uma função (ou, equivalentemente, os zeros da derivada) através da iteração

$$x_{n+1} = x_n - \frac{f'(x)}{f''(x)}$$

ou também, de forma mais geral, para equações de várias variáveis,

$$x_{n+1} = x_n - H_f^{-1} \nabla f(x_n)$$

onde H_f é o hessiano de f. Um dos problemas mais graves, contudo, é que esse método é atraído não somente para os pontos de máximo/mínimo de funções, mas também para os pontos de inflexão – não é incomum que funções envolvidas em problemas de otimização em aprendizado de máquina tenham pontos de inflexão, e essa desvantagem do método de Newton é uma das razões pelas quais nessa área normalmente se opta por outros métodos (como o *gradient descent*). Ainda assim, o método de Newton fornece uma ferramenta poderosa para a otimização de funções convexas, que também são uma classe de problemas relevantes.

Quanto a um método que eu acho que nunca usaria: acredito que talvez eu não chegue a ter que usar o método da bisecção, a não ser que seja preciso calcular uma estimativa inicial para o método de Newton (ou da secante); e ainda assim, acredito que o método do ponto falso seja uma alternativa melhor (visto que frequentemente converge mais rápido que a bisecção).

4. Se, em seu trabalho profissional, você tiver que orientar a compra de um software para resolver sistemas lineares, quais métodos de solução você gostaria que ter disponíveis no software? Justifique...

Gostaria que o software tivesse basicamente duas ferramentas de solução de sistemas lineares:

i) Uma ferramenta iterativa, como o método de Gauss-Seidel, a ser usada para resolver matrizes esparsas/com condições especiais (diagonal dominante, por exemplo) que permitam resolver sistemas em complexidade menor que $O(n^3)$;

ii) Uma implementação da eliminação de Gauss-Jordan que seja otimizada (fazendo uso de vetorização e tomando vantagem máxima do processador – usando de forma esperta o tamanho das palavras de cache, por exemplo) e que tenha suporte a paralelização.

7. Comente o que lhe pareceu mais útil no que foi visto até aqui de MS211.

Certamente os algoritmos de solução de sistemas de equações. Nas minhas áreas de interesse em Computação (processamento de imagens, visão computacional e aprendizado de máquina), operações com matrizes são extremamente importantes e muito frequentemente surge a necessidade de se calcular a inversa de matrizes.

Um exemplo comum em processamento de imagens é o caso da aplicação de kernels. Seja uma imagem representada por uma função $f\colon R^2 \to R,\ f(x, y)$ que mapeia os valores x e y (linha e coluna) a um valor numérico (o valor do pixel naquela posição). Um kernel ω é uma matriz (normalmente 3x3) que representa uma transformação a ser aplicada em cada pixel da imagem; essa matriz representa como o pixel vai ser modificado pelos seus pixels vizinhos. Mais formalmente, ao se realizar uma operação de convolução entre $f(x, y)$ e ω, gera-se uma nova imagem $g(x, y)$:

$$g(x, y) = \omega * f(x, y) = \sum_{\mathrm{dx}=-a}^{a} \sum_{\mathrm{dy}=-b}^{b} f(x + \mathrm{dx}, y + \mathrm{dy})\, \omega\,(\mathrm{dx}, \mathrm{dy})$$

Por exemplo, o kernel $\omega = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$, conhecido como *box blur*, aplica uma transformação que faz cada pixel da imagem se tornar a média dos seus vizinhos. Essa transformação tem o efeito de "borrar" a imagem:

É fácil ver que essa operação de aplicação do kernel é uma operação linear – é possível, inclusive, ao se representar uma imagem de resolução m x n como um vetor grande $\boldsymbol{x}$ de tamanho mn, determinar uma matriz $\boldsymbol{A}$ de tamanho $(mn)^2$ que representa essa transformação.

Muito frequentemente se tem interesse em aplicar kernels e, após aplicar uma sequência de transformações na imagem, "reverter" a transformação do kernel – aplicar a transformação inversa, ou, equivalentemente, a matriz $\boldsymbol{A}^{-1}$. É aqui que os métodos que estudamos nesse curso se tornam importantes, em especial os métodos de aproximação iterativos: em aplicações no mundo real (em visão computacional de robôs, por exemplo), é preciso fazer esse processo de aplicação-reversão de kernels várias vezes por segundo, então existe a necessidade de uma forma rápida de se calcular inversas. Inclusive, a matriz da questão 4 da prova (grande, esparsa e com valores distribuídos ao longo de diagonais) é muito parecida com as matrizes que se encontram nesse tipo de aplicação.

Essa necessidade de se calcular inversas de matrizes grandes é um problema que aparece muito frequentemente também em aprendizado de máquina, assim como em várias outras áreas da Computação. Por essa razão, eu acredito que os métodos de solução de sistemas lineares são as técnicas mais úteis para mim até agora.

8. Comente algum tópico que não foi abordado nas questões anteriores e justifique o(s) motivo(s) de sua escolha.

Um ponto que não foi abordado nas questões, mas que foi abordado no começo da disciplina e se mostrou presente durante as minhas implementações dos algoritmos, foi a questão da precisão de ponto flutuante. Em todos os meus resultados utilizando a biblioteca Numpy, os menores valores de erro absoluto foram sempre

da ordem de 10^{-15}. A explicação para isso é fácil: utilizando-se a função `numpy.finfo()`, que retorna informações a respeito da precisão de tipos de pontos flutuantes (nesse caso o `numpy.float64`, o tipo float de 64 bits),

```
>>> np.finfo(np.float64)
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=float64)
```

percebe-se que o tipo flutuante utilizado tem resolução de 10^{-15}. Para os problemas da prova, isso não foi relevante – os resultados estavam várias ordens de magnitude acima da resolução do float de 64 bits. Mas em outras aplicações onde os tipos float têm tamanho menor (como é comum em sistemas embarcados, por exemplo), isso pode ser um problema: floats de 32 bits têm resolução de 10^{-6}, e floats de 16 bits têm resolução de apenas 0.001! Isso já seria suficiente para potencialmente quebrar a checagem de tolerância que utilizei na prova.

Por essa razão, eu acho que esse tópico é relevante: nos computadores modernos que nós usamos no conforto de nossas casas isso não faz diferença, mas em dispositivos como microcontroladores, que têm poder modesto de armazenamento, isso pode ser até perigoso – ninguém gostaria de ter o seu carro esmagado por um portão que fechou com toda a força porque um desenvolvedor desatento esqueceu de checar a precisão de ponto flutuante na hora de fazer a leitura dos sensores de distância.