UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIAS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
MÉTODOS NUMÉRICOS E OTIMIZAÇÃO

Projeto Computacional:

Solução de Sistemas Não-Lineares via Métodos de Newton-Raphson e Quasi-Newton Usando o **MATLAB®**

Antonio Gabriel Sousa Borralho
Arthur Monteiro Costa Silva
Cesar Eugenio Cunha de Carvalho
Evelyn Cristina de Oliveira Lima
Gilberto Balby Araujo Filho
Lucas Costa Soares

São Luís, MA - Brasil
10 de julho de 2018

Antonio Gabriel Sousa Borralho
Arthur Monteiro Costa Silva
Cesar Eugenio Cunha de Carvalho
Evelyn Cristina de Oliveira Lima
Gilberto Balby Araujo Filho
Lucas Costa Soares

Projeto Computacional: Solução de Sistemas Não-Lineares via Métodos de Newton-Raphson e Quasi-Newton Usando o MATLAB®

Trabalho referente ao desenvolvimento de um projeto computacional para obtenção da terceira nota da disciplina Métodos Numéricos e Otimização no período de 2018.1.

Prof. Anselmo Barbosa Rodrigues.

São Luís, MA - Brasil
10 de julho de 2018

Sumário

1 Introdução

Muitos problemas da engenharia, física e matemática estão associados à solução de sistemas de equações lineares. Nesse contexto, tratamos de técnicas numéricas empregadas para obter a solução desses sistemas. Iniciamos por uma rápida revisão do método de eliminação gaussiana do ponto de vista computacional. No contexto de análise da propagação dos erros de arredondamento, introduzimos o método de eliminação gaussiana com pivotamento parcial, bem como, apresentamos o conceito de condicionamento de um sistema linear. Além disso, exploramos o conceito de complexidade de algoritmos em álgebra linear. Então, passamos a discutir sobre técnicas iterativas, mais especificamente, sobre os métodos de **Newton-Raphson** e **Quasi-Newton** para a solução de sistemas de equações não-lineares.(FRANCO, 2007).

1.1 Solução de sistemas lineares

Considere o sistema de equações lineares (escrito na forma algébrica)

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{1.1}$$

onde m é o número de equações e n é o número de incógnitas. Este sistema pode ser escrito na *forma matricial*

$$Ax = b \tag{1.2}$$

onde:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \text{ e } b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \tag{1.3}$$

onde A é chamada de *matriz dos coeficientes*, x de *vetor das incógnitas* e b de *vetor dos termos constantes*.

Definimos também a *matriz completa* (também chamada de *matriz estendida*) de

um sistema como $Ax = b$ como $[A|b]$, isto é

$$[A|b] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right] \tag{1.4}$$

Salvo especificado ao contrário, assumiremos ao longo deste capítulo que a matriz dos coeficientes A é uma matriz real não singular (isto é, invertível).

1.2 Eliminação de Gauss

A *eliminação gaussiana*, também conhecida como *escalonamento*, é um método para resolver sistemas lineares. Este método consiste em manipular o sistema através de determinadas operações elementares, transformando a matriz estendida do sistema em uma matriz triangular (chamada de *matriz escalonada do sistema*)(UFPB, 2018). Uma vez, triangularizado o sistema, a solução pode ser obtida via substituição regressiva (FRANCO, 2007). Naturalmente estas operações elementares devem preservar a solução do sistema e consistem em:

1. multiplicação de um linha por uma constante não nula.
2. substituição de uma linha por ela mesma somada a um múltiplo de outra linha.
3. permutação de duas linhas.

Resolva o sistema

$$\begin{aligned} x + y + z &= 1 \\ 4x + 4y + 2z &= 2 \\ 2x + y - z &= 0 \end{aligned} \tag{1.5}$$

pelo método de eliminação gaussiana. A matriz estendida do sistema é escrita como

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 4 & 4 & 2 & 2 \\ 2 & 1 & -1 & 0 \end{bmatrix} \tag{1.6}$$

No primeiro passo, subtraímos da segunda linha o quádruplo da primeira e subtraímos da terceira linha o dobro da primeira linha:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & -2 & -2 \\ 0 & -1 & -3 & -2 \end{bmatrix} \tag{1.7}$$

No segundo passo, permutamos a segunda linha com a terceira:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & -3 & -2 \\ 0 & 0 & -2 & -2 \end{bmatrix} \tag{1.8}$$

Neste momento, a matriz já se encontra na forma triangular (chamada de *matriz escalonada do sistema*). Da terceira linha, encontramos $-2z = -2$, ou seja, $z = 1$. Substituindo na segunda equação, temos $-y - 3z = -2$, ou seja, $y = -1$ e finalmente, da primeira linha, $x + y + z = 1$, resultando em $x = 1$.

Neste Exemplo 1.2, o procedimento de eliminação gaussiana foi usado para obtermos um sistema triangular (superior) equivalente ao sistema original. Este, por sua vez, nos permitiu calcular a solução do sistema, isolando cada variável, começando da última linha (última equação), seguindo linha por linha até a primeira.

Alternativamente, podemos continuar o procedimento de eliminação gaussiana, anulando os elementos da matriz estendida acima da diagonal principal. Isto nos leva a uma matriz estendida diagonal (chamada *matriz escalonada reduzida*), na qual a solução do sistema original aparece na última coluna.

No Exemplo 1.2, usamos o procedimento de eliminação gaussiana e obtivemos

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 4 & 4 & 2 & 2 \\ 2 & 1 & -1 & 0 \end{bmatrix}}_{\text{matriz estendida}} \sim \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & -3 & -2 \\ 0 & 0 & -2 & -2 \end{bmatrix}}_{\text{matriz escalonada}}. \tag{1.9}$$

Agora, seguindo com o procedimento de eliminação gaussiana, buscaremos anular os elementos acima da diagonal principal. Começamos dividindo cada elemento da última linha pelo valor do elemento da sua diagonal, obtemos

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & -3 & -2 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{1.10}$$

Então, somando da segunda linha o triplo da terceira e subtraindo da primeira a terceira linha, obtemos

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{1.11}$$

Fixamos, agora, na segunda linha. Dividimos esta linha pelo valor do elemento em sua diagonal, isto nos fornece

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{1.12}$$

Por fim, subtraímos da primeira linha a segunda, obtendo a matriz escalonada reduzida

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{1.13}$$

Desta matriz escalonada reduzida temos, imediatamente, $x = 1$, $y = -1$ e $z = 1$, como no Exemplo 1.2.

1.2.1 Eliminação gaussiana com pivotamento parcial

A eliminação gaussiana com *pivotamento parcial* consiste em fazer uma permutação de linhas de forma a escolher o maior pivô (em módulo) a cada passo (FRANCO, 2007).

Resolva o sistema

$$\begin{aligned} x + y + z &= 1 \\ 2x + y - z &= 0 \\ 2x + 2y + z &= 1 \end{aligned} \tag{1.14}$$

por eliminação gaussiana com pivotamento parcial. A matriz estendida do sistema é

$$\begin{aligned} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & 0 \\ 2 & 2 & 1 & 1 \end{bmatrix} &\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \end{bmatrix} \\ &\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 0 & 1/2 & 3/2 & 1 \\ 0 & 1 & 2 & 1 \end{bmatrix} \\ &\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 1/2 & 3/2 & 1 \end{bmatrix} \\ &\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 1/2 & 1/2 \end{bmatrix} \end{aligned} \tag{1.15}$$

Encontramos $1/2z = 1/2$, ou seja, $z = 1$. Substituímos na segunda equação e temos $y + 2z = 1$, ou seja, $y = -1$ e, finalmente $2x + y - z = 0$, resultando em $x = 1$.

A técnica de eliminação gaussiana com pivotamento parcial ajuda a evitar a propagação dos erros de arredondamento.

1.3 Sistemas triangulares

Considere um sistema linear onde a matriz é triangular superior, ou seja,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{1.16}$$

tal que todos elementos abaixo da diagonal são iguais a zero.

Podemos resolver esse sistema iniciando pela última equação e isolando x_n obtemos

$$x_n = b_n/a_{nn} \tag{1.17}$$

Substituindo x_n na penúltima equação

$$a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = b_{n-1} \tag{1.18}$$

e isolando x_{n-1} obtemos

$$x_{n-1} = (b_{n-1} - a_{n-1,n}x_n)/a_{n-1,n-1} \tag{1.19}$$

e continuando desta forma até a primeira equação obteremos

$$x_1 = (b_1 - a_{12}x_2 \cdots - a_{1n}x_n)/a_{11}. \tag{1.20}$$

De forma geral, temos que

$$x_i = (b_i - a_{i,i+1}x_{i+1} \cdots - a_{i,n}x_n)/a_{i,i}, \quad i = 2, \dots, n. \tag{1.21}$$

1.4 Gauss-LU

Considere um sistema linear $Ax = b$, onde a matriz A é densa[1]. A fim de resolver o sistema, podemos fatorar a matriz A como o produto de uma matriz L triangular inferior e uma matriz U triangular superior, ou seja, $A = LU$ (FRANCO, 2007).

Sendo assim, o sistema pode ser reescrito da seguinte forma:

$$Ax = b \tag{1.22}$$

$$(LU)x = b \tag{1.23}$$

$$L(Ux) = b \tag{1.24}$$

$$Ly = b \quad \text{e} \quad Ux = y \tag{1.25}$$

[1] Diferentemente de uma matriz esparsa, uma matriz densa possui a maioria dos elementos diferentes de zero.

Isto significa que, ao invés de resolvermos o sistema original, podemos resolver o sistema triangular inferior $Ly = b$ e, então, o sistema triangular superior $Ux = y$, o qual nos fornece a solução de $Ax = b$.

A matriz U da fatoração[2] LU é a matriz obtida ao final do escalonamento da matriz A.

A matriz L é construída a partir da matriz identidade I, ao longo do escalonamento de A. Os elementos da matriz L são os múltiplos do primeiro elemento da linha de A a ser zerado dividido pelo pivô acima na mesma coluna.

Por exemplo, para zerar o primeiro elemento da segunda linha de A, calculamos

$$L_{21} = A_{21}/A_{11} \tag{1.26}$$

e fazemos

$$A_{2,:} \Leftarrow A_{2,:} - L_{21}A_{1,:} \tag{1.27}$$

Note que denotamos $A_{i,:}$ para nos referenciarmos a linha i de A. Da mesma forma, se necessário usaremos $A_{:,j}$ para nos referenciarmos a coluna j de A.

Para zerar o primeiro elemento da terceira linha de A, temos

$$L_{31} = A_{31}/A_{11} \tag{1.28}$$

e fazemos

$$A_{3,:} \Leftarrow A_{3,:} - L_{31}A_{1,:} \tag{1.29}$$

até chegarmos ao último elemento da primeira coluna de A.

Repetimos o processo para as próximas colunas, escalonando a matriz A e coletando os elementos L_{ij} abaixo da diagonal[3].

Use a fatoração LU para resolver o seguinte sistema linear:

$$\begin{aligned} x_1 + x_2 + x_3 &= -2 \\ 2x_1 + x_2 - x_3 &= 1 \\ 2x_1 - x_2 + x_3 &= 3 \end{aligned} \tag{1.30}$$

[2] Não vamos usar pivotamento nesse primeiro exemplo.

[3] Perceba que a partir da segunda coluna para calcular L_{ij} não usamos os elementos de A, mas os elementos da matriz A em processo de escalonamento

Começamos fatorando a matriz A dos coeficientes deste sistema:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & -1 \\ 2 & -1 & 1 \end{bmatrix}. = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{I_{3,3}} \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & -1 \\ 2 & -1 & 1 \end{bmatrix}}_{A} \tag{1.31}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & -3 \\ 0 & -3 & -1 \end{bmatrix} \tag{1.32}$$

$$= \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 3 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & -3 \\ 0 & 0 & 8 \end{bmatrix}}_{U} \tag{1.33}$$

$$\tag{1.34}$$

Completada a fatoração LU, resolvemos, primeiramente, o sistema $Ly = b$:

$$\begin{aligned} y_1 &= -2 \\ 2y_1 + y_2 &= 1 \\ 2y_1 + 3y_2 + y_3 &= 3 \end{aligned} \tag{1.35}$$

o qual no fornece $y_1 = -2$, $y_2 = 5$ e $y_3 = -8$. Por fim, obtemos a solução resolvendo o sistema $Ux = y$:

$$\begin{aligned} x_1 + x_2 + x_3 &= -2 \\ -x_2 - 3x_3 &= 5 \\ 8x_3 &= -8 \end{aligned} \tag{1.36}$$

o qual fornece $x_3 = -1$, $x_2 = -2$ e $x_1 = 1$.

1.4.1 Custo computacional para solução de um sistema linear usando Gauss-LU

Para calcularmos o custo computacional de um algoritmo completo, uma estratégia é separar o algoritmo em partes menores, mais fáceis de analisar.

Para resolver o sistema, devemos primeiro fatorar a matriz A nas matrizes L e U. Vimos que o custo é

$$\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6} \text{ flops.} \tag{1.37}$$

Depois devemos resolver os sistemas $Ly = b$ e $Ux = y$. O custo de resolver os dois sistemas é (devemos contar duas vezes)

$$2n^2 \text{ flops.} \tag{1.38}$$

Somando esses 3 custos, temos que o custo para resolver um sistema linear usando fatoração LU é

$$\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{n}{6} \text{ flops.} \tag{1.39}$$

Quando n cresce, prevalessem os termos de mais alta ordem, ou seja,

$$\mathcal{O}(\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{n}{6}) = \mathcal{O}(\frac{2n^3}{3} + \frac{3n^2}{2}) = \mathcal{O}(\frac{2n^3}{3}) \tag{1.40}$$

1.4.2 Custo computacional para resolver m sistemas lineares

Devemos apenas multiplicar m pelo custo de resolver um sistema linear usando fatoração LU, ou seja, o custo será

$$m(\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{n}{6}) = \frac{2mn^3}{3} + \frac{3mn^2}{2} - \frac{mn}{6} \tag{1.41}$$

e com $m = n$ temos

$$\frac{2n^4}{3} + \frac{3n^3}{2} - \frac{n^2}{6}. \tag{1.42}$$

Porém, se estivermos resolvendo n sistemas com *a mesma matriz* A (e diferente lado direito $\boldsymbol{b}$ para cada sistema) podemos fazer a fatoração LU uma única vez e contar apenas o custo de resolver os sistemas triangulares obtidos.

Custo para fatoração LU de A: $\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}$.

Custo para resolver m sistemas triangulares inferiores: mn^2.

Custo para resolver m sistemas triangulares superiores: mn^2.

Somando esses custos obtemos

$$\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6} + 2mn^2 \tag{1.43}$$

que quando $m = n$ obtemos

$$\frac{8n^3}{3} - \frac{n^2}{2} - \frac{n}{6} \text{ flops.} \tag{1.44}$$

2 Métodos iterativos para sistemas lineares

Na seção anterior, tratamos de métodos diretos para a resolução de sistemas lineares. Em um *método direto* (por exemplo, solução via fatoração LU) obtemos uma aproximação da solução depois de realizarmos um número finito de operações (só teremos a solução ao final do processo).

Veremos nessa seção dois *métodos iterativos* básicos para obter uma aproximação para a solução de um sistema linear. Geralmente em um método iterativo, iniciamos com uma aproximação para a solução (que pode ser ruim) e vamos melhorando essa aproximação através de sucessivas iterações.

2.1 Método de Newton-Raphson

Nesta seção, apresentamos o *método de Newton-Raphson*[1,2] para calcular o zero de funções reais de uma variável real.

Consideramos que x^* seja um zero de uma dada função $f(x)$ continuamente diferenciável, isto é, $f(x^*) = 0$. A fim de usar a iteração do ponto fixo, observamos que, equivalentemente, x^* é um ponto fixo da função:

$$g(x) = x + \alpha(x)f(x), \quad \alpha(x) \neq 0, \tag{2.1}$$

onde $\alpha(x)$ é uma função arbitrária, a qual escolheremos de forma que a iteração do ponto fixo tenha ótima taxa de convergência.

Do *teorema do ponto fixo*, a taxa de convergência é dada em função do valor absoluto da derivada de $g(x)$. Calculando a derivada temos:

$$g'(x) = 1 + \alpha(x)f'(x) + \alpha'(x)f(x). \tag{2.2}$$

No ponto $x = x^*$, temos:

$$g'(x^*) = 1 + \alpha(x^*)f'(x^*) + \alpha'(x^*)f(x^*). \tag{2.3}$$

Como $f(x^*) = 0$, temos:

$$g'(x^*) = 1 + \alpha(x^*)f'(x^*). \tag{2.4}$$

Sabemos que o processo iterativo converge tão mais rápido quanto menor for $|g'(x)|$ nas vizinhanças de x^*. Isto nos leva a escolher:

$$g'(x^*) = 0, \tag{2.5}$$

[1] Joseph Raphson, 1648 - 1715, matemático inglês.
[2] Também chamado apenas de método de Newton.

e, então, temos:

$$\alpha(x^*) = -\frac{1}{f'(x^*)}, \tag{2.6}$$

se $f'(x^*) \neq 0$.

A discussão acima nos motiva a introduzir o método de Newton, cujas iterações são dada por:

$$x^{(n+1)} = x^{(n)} - \frac{f\left(x^{(n)}\right)}{f'\left(x^n\right)}, \quad n \geq 1, \tag{2.7}$$

sendo $x^{(1)}$ uma aproximação inicial dada.

2.1.1 Interpretação geométrica

Seja uma dada função $f(x)$ conforme na Figura 1. Para tanto, escolhemos uma aproximação inicial $x^{(1)}$ e computamos:

$$x^{(2)} = x^{(1)} - \frac{f(x^{(1)})}{f'(x^{(1)})}. \tag{2.8}$$

Geometricamente, o ponto $x^{(2)}$ é a interseção da reta tangente ao gráfico da função $f(x)$ no ponto $x = x^{(1)}$ com o eixo das abscissas. Com efeito, a equação desta reta é:

$$y = f'(x^{(1)})(x - x^{(1)}) + f(x^{(1)}). \tag{2.9}$$

Assim, a interseção desta reta com o eixo das abscissas ($y = 0$) ocorre quando:

$$f'(x^{(1)})(x - x^{(1)}) + f(x^{(1)}) = 0 \Rightarrow x = x^{(1)} - \frac{f(x^{(1)})}{f'(x^{(1)})}. \tag{2.10}$$

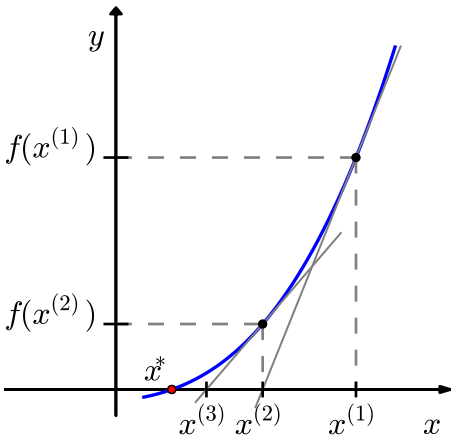

Figura 1 – Interpretação do método de Newton.

Ou seja, dada aproximação $x^{(n)}$, a próxima aproximação $x^{(n+1)}$ é o ponto de interseção entre o eixo das abscissas e a reta tangente ao gráfico da função no ponto $x = x^{(n)}$. Observe a Figura 1.

2.1.2 Análise de convergência

Seja $f(x)$ um função com derivadas primeira e segunda contínuas tal que $f(x^*) = 0$ e $f'(x^*) \neq 0$. Seja também a função $g(x)$ definida como:

$$g(x) = x - \frac{f(x)}{f'(x)}. \tag{2.11}$$

Expandimos em série de Taylor em torno de $x = x^*$, obtemos:

$$g(x) = g(x^*) + g'(x^*)(x - x^*) + \frac{g''(x^*)}{2}(x - x^*)^2 + O\left((x - x^*)^3\right). \tag{2.12}$$

Observamos que:

$$g(x^*) = x^* \tag{2.13}$$

$$g'(x^*) = 1 - \frac{f'(x^*)f'(x^*) - f(x^*)f''(x^*)}{\left(f'(x^*)\right)^2} = 0 \tag{2.14}$$

Portanto:

$$g(x) = x^* + \frac{g''(x^*)}{2}(x - x^*)^2 + O\left((x - x^*)^3\right) \tag{2.15}$$

Com isso, temos:

$$x^{(n+1)} = g(x^{(n)}) = x^* + \frac{g''(x^*)}{2}(x^{(n)} - x^*)^2 + O\left((x - x^*)^3\right), \tag{2.16}$$

ou seja:

$$\left|x^{(n+1)} - x^*\right| \leq C\left|x^{(n)} - x^*\right|^2, \tag{2.17}$$

com constante $C = |g''(x^*)/2|$. Isto mostra que o método de Newton tem *taxa de convergência quadrática*. Mais precisamente, temos o seguinte teorema.

[Método de Newton] Sejam $f \in C^2([a, b])$ com $x^* \in (a, b)$ tal que $f(x^*) = 0$ e:

$$m := \min_{x\in[a,b]} |f'(x)| > 0 \quad \text{e} \quad M := \max_{x\in[a,b]} |f''(x)|. \tag{2.18}$$

Escolhendo $\rho > 0$ tal que:

$$q := \frac{M}{2m}\rho < 1, \tag{2.19}$$

definimos a *bacia de atração* do método de Newton pelo conjunto:

$$K_\rho(x^*) := \{x \in \mathbb{R};\ |x - x^*| \leq \rho\} \subset [a, b]. \tag{2.20}$$

Então, para qualquer $x^{(1)} \in K_\rho(x^*)$ a iteração do método de Newton:

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})}, \tag{2.21}$$

fornece uma sequência $x^{(n)}$ que converge para x^*, isto é, $x^{(n)} \to x^*$ quando $n \to \infty$. Além disso, temos a seguinte estimativa de erro *a priori*:

$$|x^{(n)} - x^*| \leq \frac{2m}{M} q^{(2^{n-1})}, \quad n \geq 2, \tag{2.22}$$

e a seguinte estimativa de erro *a posteriori*:

$$|x^{(n)} - x^*| \leq \frac{M}{2m}|x^{(n)} - x^{(n-1)}|^2, \quad n \geq 2. \tag{2.23}$$

Para $n \in \mathbb{N}$, $n \geq 2$, temos:

$$x^{n+1} - x^* = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})} - x^* = -\frac{1}{f(x^{(n)})}\left[f(x^{(n)}) + (x^* - x^{(n)})f'(x^{(n)}\right]. \tag{2.24}$$

Agora, para estimar o lado direito desta equação, usamos o polinÃ´mio de Taylor de grau 1 da função $f(x)$ em torno de $x = x^{(n)}$, isto é:

$$f(x^*) = f(x^{(n)}) + (x^* - x^{(n)})f'(x^{(n)}) + \int_{x^{(n)}}^{x^*} f''(t)(x^* - t)\, dt. \tag{2.25}$$

Pela mudança de variável $t = x^{(n)} + s(x^{(n)} - x^*)$, observamos que o resto deste polinÃ´mio de Taylor na forma integral é igual a:

$$R(x^*, x^{(n)}) := (x^* - x^{(n)})^2 \int_0^1 f''\left(x^{(n)} + s(x^* - x^{(n)})\right)(1 - s)\, ds. \tag{2.26}$$

Assim, da cota da segunda derivada de $f(x)$, temos:

$$|R(x^*, x^{(n)})| \leq M|x^* - x^{(n)}|^2 \int_0^1 (1 - s)\, ds = \frac{M}{2}|x^* - x^{(n)}|^2. \tag{2.27}$$

Se $x^{(n)} \in K_\rho(x^*)$, então de (2.24) e (2.27) temos:

$$|x^{(n+1)} - x^*| \leq \frac{M}{2m}|x^{(n)} - x^*|^2 \leq \frac{M}{2m}\rho^2 < \rho. \tag{2.28}$$

Isto mostra que se $x^{(n)} \in K_\rho(x^*)$, então $x^{(n+1)} \in K_\rho(x^*)$, isto é, $x^{(n)} \in K_\rho(x^*)$ para todo $n \in \mathbb{R}$.

Agora, obtemos a estimativa *a priori* de (2.1.2), pois:

$$|x^{(n)} - x^*| \leq \frac{2m}{M}\left(\frac{M}{2m}|x^{(n-1)} - x^*|\right)^2 \leq \cdots \leq \frac{2m}{M}\left(\frac{M}{2m}|x^{(1)} - x^*|\right)^{2^{n-1}}. \tag{2.29}$$

Logo:

$$|x^{(n)} - x^*| \leq \frac{2m}{M}q^{2^{n-1}}, \tag{2.30}$$

donde também vemos que $x^{(n)} \to x^*$ quando $n \to \infty$, pois $q < 1$.

Por fim, para provarmos a estimativa *a posteriori* tomamos a seguinte expansão em polinÃ´mio de Taylor:

$$f(x^{(n)}) = f(x^{(n-1)}) + (x^{(n)} - x^{(n-1)})f'(x^{(n-1)}) + R(x^{(n)}, x^{(n-1)}). \tag{2.31}$$

Aqui, temos:

$$f(x^{(n-1)}) + (x^{(n)} - x^{(n-1)})f'(x^{(n-1)}) = 0 \tag{2.32}$$

e, então, conforme acima:

$$|f(x^{(n)})| = |R(x^{(n)}), x^{(n-1)}| \leq \frac{M}{2}|x^{(n)} - x^{(n-1)}|^2. \tag{2.33}$$

Com isso e do teorema do valor médio, concluímos:

$$|x^{(n)} - x^*| \leq \frac{1}{m}|f(x^{(n)}) - f(x^*)| \leq \frac{M}{2m}|x^{(n)} - x^{(n-1)}|^2. \tag{2.34}$$

Estime o raio ρ da bacia de atração $K_\rho(x^*)$ para a função $f(x) = \cos(x) - x$ restrita ao intervalo $[0, \pi/2]$. O raio da bacia de atração é tal que:

$$\rho < \frac{2m}{M} \tag{2.35}$$

onde $m := \min |f'(x)|$ e $M := \max |f''(x)|$ com o mínimo e o máximo tomados em um intervalo $[a, b]$ que contenha o zero da função $f(x)$. Aqui, por exemplo, podemos tomar $[a, b] = [0, \pi/2]$. Como, neste caso, $f'(x) = -\sin(x) - 1$, temos que $m = 1$. Também, como $f''(x) = -\cos x$, temos $M = 1$. Assim, concluímos que $\rho < 2$ (lembrando que $K_\rho(x^*) \subset [0, \pi/2]$). Ou seja, neste caso as iterações de Newton convergem para o zero de $f(x)$ para qualquer escolha da aproximação inicial $x^{(1)} \in [0, \pi/2]$.

2.2 Método Quasi-Newton

Para descrever este método, suponha que uma aproximação inicial $x^{(0)}$ seja dada para a solução p de $F(x) = 0$. Calculamos a aproximação seguinte x^1 do mesmo modo que no método de Newton, ou, se for inconveniente determinar $J\left(x^{(0)}\right)$ exatamente, utilizamos as equações de diferença dadas pela Equação abaixo:

$$\frac{\partial f_j}{\partial x_k}\left(x^{(i)}\right) \approx \frac{f_j\left(x^{(i)} + e_k h\right) - f_j\left(x^{(i)}\right)}{h}$$

Para aproximar as derivadas parciais. Para calcular x^2,'contudo, saímos do método de Newton e examinamos o método as Secantes para uma única equação não-linear. O método das Secantes usa a aproximação:

$$f'(x_1) \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

Como uma substituição para $f'(x_1)$ no método de Newton. Para sistemas não-lineares, $x^1 - x^{(0)}$ é um vetor e o quociente correspondente não é definido. Todavia, o método prossegue analogamente quanto a substituirmos na matriz $J\left(x^{(1)}\right)$ no método de Newton para sistemas por uma matriz A_1 com a propriedade que:

$$A_1\left(x^{(1)} - x^{(0)}\right) = F\left(x^{(1)}\right) - F\left(x^{(0)}\right)$$

Qualquer vetor diferente de zero em $\mathbb{R}^n$ pode ser escrito como a soma de um múltiplo de $x^1 - x^{(0)}$ com um múltiplo de um vetor no complemento ortogonal de $x^1 - x^{(0)}$. Assim, para definirmos de $x^1 - x^{(0)}$, já que nenhuma informação está disponível a respeito da variação em F em uma direção ortogonal a $x^1 - x^{(0)}$, exigimos que:

$$A_1 z = J\left(x^{(0)}\right) z, \text{ sempre que } \left(x^{(1)} - x^{(0)}\right)^t z = 0.$$

Assim, qualquer vetor ortogonal a $x^1 - x^{(0)}$ não é afetado pela atualização de $J\left(x^{(0)}\right)$, que foi usada para calcular x^1, para A_1, que foi usada na determinação de x^2.

As condições acima definem A_1 de modo único, como:

$$A_1 = J\left(x^{(0)}\right) + \frac{\left[F\left(x^{(1)}\right) - F\left(x^{(0)}\right) - J\left(x^{(0)}\right)\left(x^{(1)} - x^{(0)}\right)\right]\left(x^{(1)} - x^{(0)}\right)^t}{\|x^{(1)} - x^{(0)}\|_2^2}$$

é esta matriz que é utilizada no lugar de $J\left(x^{(1)}\right)$ para se determinar $x^{(2)}$ como:

$$x^{(2)} = x^{(1)} - A_1^{-1} F\left(x^{(1)}\right).$$

Uma vez que $x^{(2)}$ tenha sido determinado, o método é repetido para determinar $x^{(3)}$, usando A_1 no lugar de $A_0 = J\left(x^{(0)}\right)$, e como $x^{(2)}$ e $x^{(1)}$ no lugar de $x^{(1)}$ e $x^{(0)}$. Em geral, uma vez que $x^{(i)}$ tenha sido determinado, $x^{(i+1)}$ é calculado por:

$$A_i = A_{i-1} + \frac{y_i - A_{i-1} s_i}{\|s_i\|_2^2} {s_i}^t$$

$$x^{(i+1)} = x^{(i)} - A_1^{-1} F\left(x^{(i)}\right),$$

Em que a notação $y_i = F\left(x^{(i)}\right) - F\left(x^{(i-1)}\right)$ e $s_i = x^{(i)} - x^{(i-1)}$ é introduzida para simplificar as equações.

Se o método for realizado como delineado nas equações aimca, a quantidade de cálculos de funções escalares ficará reduzida de $n^2 + n$ para n(aquelas necessárias para calcular $F\left(x^{(i)}\right)$), mas cálculos de $O\left(n^3\right)$ ainda são necessários para resolver o sistema linear n x n associado.

$$A_i s_{i+1} = -F\left(x^{(i)}\right)$$

O emprego do método nesta forma não seria justificado por causa da redução para convergência super linear a partir da convergência quadrática do método de Newton.

3 Resultados

Sobre os testes com o sistema tridiagonal de Broyden, temos os resultados para os seguintes tópicos:

3.1 O grau e o padrão de esparsidade da matriz jacobiana

O grau e o padrão de esparsidade da matriz jacobiana para $p = 1000$. Com base nestes resultados identificando se a matriz jacobiana é densa ou esparsa.

Primeiramente, iremos discutir um pouco sobre esparsidade, mostrando como se calcula seu grau e seu padrão:

3.1.1 GRAU DE ESPARSIDADE

Para calcular o grau de esparsidade, levemos em consideração que este grau é uma métrica que informa a percentagem dos elementos nulos da matriz. Daí, seja G_S grau de esparsidade. Pode-se escrever:

$$G_S = \frac{p^2 - N}{p^2}$$

Onde N é o número de elementos não-nulos e p é a ordem da matriz quadrada cujo grau de esparsidade será extraído.

Para o jacobiano do sistema de Broyden, tem-se 2 elementos não nulos na primeira e na última linha. Nas $p - 2$ linhas intermediárias, tem-se 3 elementos não nulos. Daí:

$$N = 3 \cdot (p - 2) + 4 = 3p - 2$$

Portanto,

$$G_S = \frac{p^2 - (3p - 2)}{p^2} = \frac{p^2 - 3p + 2}{p^2}$$

Para o caso particular em que $p = 1000$ tem-se:

$$G_S = \frac{1000^2 - (3 \cdot 1000 + 2)}{1000^2} \cdot 100\% = 99.7002\%$$

Desta forma, como a maior parte dos elementos do jacobiano é nula, esta matriz é esparsa.

3.2 Solução do sistema de Broyden - primeiro caso

A solução do sistema de Broyden para p = 5, 10 e 20 usando o Método de Newton-Raphson. Apresente o vetor solução, os valores finais de $||F(x)||_\infty$ e também o número de iterações realizadas.

3.2.1 Para p = 5:

Iterações: 3

$||F(x)||_\infty = 6.6411 \times 10^{-9}$

Vetor Solução:

$$X = \begin{bmatrix} -0.9684 \\ -1.1870 \\ -0.9590 \\ -0.5942 \end{bmatrix}. \tag{3.1}$$

3.2.2 Para p = 10:

Iterações: 3

$||F(x)||_\infty = 6.3098 \times 10^{-7}$

Vetor Solução:

$$X = \begin{bmatrix} -1.0301 \\ -1.3104 \\ -1.3799 \\ -1.3907 \\ -1.3796 \\ -1.3499 \\ -1.2907 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}. \tag{3.2}$$

3.2.3 Para p = 20:

Iterações: 4

$||F(x)||_\infty = 1.8145 \times 10^{-12}$

Vetor Solução:

$$X = \begin{bmatrix} -1.03024 \\ -1.3150 \\ -1.3887 \\ -1.4076 \\ -1.4125 \\ -1.4137 \\ -1.4139 \\ -1.4139 \\ -1.4136 \\ -1.4130 \\ -1.4119 \\ -1.4098 \\ -1.4055 \\ -1.3973 \\ -1.3813 \\ -1.3504 \\ -1.2908 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}. \tag{3.3}$$

3.3 Solução do sistema de Broyden - segundo caso

Para $p = 5$, foram necessário 8 iterações, sendo $||F_x||_\infty = 3,2873e - 007$, e encontrado a seguinte solução:

Tabela 1 – Valores de X para $p = 5$.

X	Resultado
$X(2)$	$-1,1870$
$X(3)$	$-1,1485$
$X(4)$	$-0,9590$
$X(5)$	$-0,5942$

Para $p = 10$, foram necessário 10 iterações, sendo $||F_x||_\infty = 6,2736e - 007$, e encontrado a seguinte solução:

Tabela 2 – Valores de X para $p = 10$.

X	Resultado
$X(1)$	$-1,0301$
$X(2)$	$-1,3104$
$X(3)$	$-1,3799$
$X(4)$	$-1,3907$
$X(5)$	$-1,3796$
$X(6)$	-1.3499
$X(7)$	-1.2907
$X(8)$	-1.1775
$X(9)$	-0.9675
$X(10)$	$-0,5965$

Para $p = 20$, foram necessário 10 iterações, sendo $||F_x||_\infty = 7.9266e - 007$, e encontrado a seguinte solução:

Tabela 3 – Valores de X para $p = 20$.

X	Resultado
$X(1)$	$-1,0324$
$X(2)$	$-1,3150$
$X(3)$	$-1,3887$
$X(4)$	$-1,4076$
$X(5)$	$-1,4125$
$X(6)$	$-1,4137$
$X(7)$	$-1,4139$
$X(8)$	$-1,4138$
$X(9)$	$-1,4136$
$X(10)$	$-1,4130$
$X(11)$	$-1,4119$
$X(12)$	$-1,4098$
$X(13)$	$-1,4055$
$X(14)$	$-1,3973$
$X(15)$	$-1,3813$
$X(16)$	$-1,3504$
$X(17)$	$-1,2908$
$X(18)$	$-1,1775$
$X(19)$	$-0,9675$
$X(20)$	$-0,5965$

3.4 Comparação dos resultados com o MATLAB®

Compare os resultados obtidos nas seções 2 e 3 com aqueles obtidos pela função nativa do MATLAB® para a solução de sistemas não-lineares. Apresente uma descrição resumida do método usado pela função nativa do MATLAB® para solução de sistemas não-lineares.

Nas três tabelas a baixos estão as comparações das soluções do sistema de Broyden, com quatro casas de precisão, para p = 5, 10 e 20. Foram utilizados os métodos de Newton-Raphson, Newton Modificado (Quase-Newton) e a função nativa do MATLAB® fsolve (MATHWORKS, 1993).

A função fsolve é uma função nativa do MATLAB® robusta que implementa três algorítimos diferentes: trust region dogleg, trust region e Levenberg-Marquardt (MATSUMOTO, 2008). Ela possui variações de entradas, porém a utilizada foi X = fsolve (FUN, X0).

Os resultados para quatro casas de precisão e com tolerância 1*10^-6 foram os mesmos para as três tabelas. Contudo as diferenças mais acentuadas foram no número de interações, sendo a QuasiNewton nas três tabelas apresentou um numero de interações maior que o dobro das outras duas. Para os tamanhos utilizados o método de Newton-Raphson e a função fsolve obtiveram o mesmo número de interações. Nos algoritmos implementados também foi utilizado a cputime para medir o tempo. Embora o tempo não tenha se mantido constante para ser posto na tabela foi possível observar que a conseguia os resultados de forma mais rápida que o método Newton-Raphson. Em relação as normas que mostram o quanto foi preciso os cálculos. é possível observar que utilizando Newton Modificado a norma nas três tabelas foi maior que os outros dois métodos. E entre a função nativa do MATLAB® e Newton-Raphson obtiveram bons resultados, variando em qual se saiu melhor nas três tabelas.

Tabela 4 – Primeira comparação com o MATLAB®.

$p = 5$	Newton_Raphson	Quasi_Newton	*fsolve*
Solução x	$\begin{bmatrix} -0.9684 \\ -1.1870 \\ -1.1485 \\ -0.9590 \\ -0.5942 \end{bmatrix}$	$\begin{bmatrix} -0.9684 \\ -1.1870 \\ -1.1485 \\ -0.9590 \\ -0.5942 \end{bmatrix}$	$\begin{bmatrix} -0.9684 \\ -1.1870 \\ -1.1485 \\ -0.9590 \\ -0.5942 \end{bmatrix}$
Nº de iterações	3	9	3
Solução inicial	$-\text{ones}(5,1)$	$-\text{ones}(5,1)$	$-\text{ones}(5,1)$
Tolerância	1.10^{-6}	1.10^{-6}	1.10^{-6}
$\|\|F(x)\|\|_\infty$	$6,6411 \cdot 10^{-9}$	$3,3131.10^{-7}$	$6.6392 \cdot 10^{-9}$

Tabela 5 – Segunda comparação com o MATLAB®.

$p = 10$	Newton_Raphson	Quasi_Newton	*fsolve*
Solução x	$\begin{bmatrix} -1.0301 \\ -1.3104 \\ -1.3799 \\ -1.3907 \\ -1.3796 \\ -1.3499 \\ -1.2907 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}$	$\begin{bmatrix} -1.0301 \\ -1.3104 \\ -1.3799 \\ -1.3907 \\ -1.3796 \\ -1.3499 \\ -1.2907 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}$	$\begin{bmatrix} -1.0301 \\ -1.3104 \\ -1.3799 \\ -1.3907 \\ -1.3796 \\ -1.3499 \\ -1.2907 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}$
Nº de iterações	3	10	3
Solução inicial	$-\text{ones}(10, 1)$	$-\text{ones}(10, 1)$	$-\text{ones}(10, 1)$
Tolerância	1.10^{-6}	1.10^{-6}	1.10^{-6}
$\|\|F(x)\|\|_\infty$	$6,3098 \cdot 10^{-7}$	$1.9219.10^{-7}$	$3.0453 \cdot 10^{-8}$

Tabela 6 – Terceira comparaçao com o MATLAB®.

$p = 20$	Newton_Raphson	Quasi_Newton	*fsolve*
Solução x	$\begin{bmatrix} -1.0324 \\ -1.3150 \\ -1.3887 \\ -1.4076 \\ -1.4125 \\ -1.4137 \\ -1.4139 \\ -1.4139 \\ -1.4136 \\ -1.4130 \\ -1.4119 \\ -1.4098 \\ -1.4055 \\ -1.3973 \\ -1.3813 \\ -1.3504 \\ -1.2908 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}$	$\begin{bmatrix} -1.0324 \\ -1.3150 \\ -1.3887 \\ -1.4076 \\ -1.4125 \\ -1.4137 \\ -1.4139 \\ -1.4139 \\ -1.4136 \\ -1.4130 \\ -1.4119 \\ -1.4098 \\ -1.4055 \\ -1.3973 \\ -1.3813 \\ -1.3504 \\ -1.2908 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}$	$\begin{bmatrix} -1.0324 \\ -1.3150 \\ -1.3887 \\ -1.4076 \\ -1.4125 \\ -1.4137 \\ -1.4139 \\ -1.4139 \\ -1.4136 \\ -1.4130 \\ -1.4119 \\ -1.4098 \\ -1.4055 \\ -1.3973 \\ -1.3813 \\ -1.3504 \\ -1.2908 \\ -1.1775 \\ -0.9675 \\ -0.5965 \end{bmatrix}$
Nº de iterações	4	10	4
Solução inicial	$-\text{ones}(20, 1)$	$-\text{ones}(20, 1)$	$-\text{ones}(20, 1)$
Tolerância	1.10^{-6}	1.10^{-6}	1.10^{-6}
$\|\|F(x)\|\|_\infty$	$1.8145 \cdot 10^{-12}$	7.926610^{-7}	$8.5744 \cdot 10^{-10}$

3.5 Avaliação do custo computacional

Avaliação do custo computacional (tempo de processamento) para o obter a solução do sistema de broyden com $p = 1000$ para os seguintes métodos: método de Newton-Raphson, método de Newton modificado e o método usado pela função nativa do MATLAB®.

Na tabela, são computados os tempos de processamento de cada um dos métodos e da função *fsolve*(MATHWORKS, 2018), onde notamos um contrabalanceamento por parte do método de Quasi-Newton, que obteve um tempo de processamento muito menor que do método de Newton-Raphson, devido a maior complexidade deste. Apesar da função *fsolve* também ser implementada de forma complexa, seu tempo de processamento foi bem próximo ao do método de Quasi-Newton, pois fsolve resolveu o sistema através do método *trust-region dogleg.*

$P = 1000$	Tempo de Processamento (segundos)
Newton-Raphson	130.4324
Quase-Newton	42.2919
fsolve	4.2276

Para entender como a função nativa é tão mais rápida quanto a Newton-Raphson e Quase-Newton, basta entender que a solução que o MATLAB® impõe. Isto é, mescla-se o método de mínimos quadrados para adaptar e reduzir a função por uma menor, além de usar série de Taylor também para manipular a função e ocasionar novas aproximações e o método de gradiente.

Toda esta adaptação é para que o MATLAB® convirja os cálculos para algo matricial. A adaptação final da aproximação para uma dada função $F_i(x) = 0$ com todas as técnicas de minimização, resultando em $\min\{\frac{1}{2}s^T Hs + s^T g \; such \; that \; ||Ds|| \leq \Delta$ onde H é a matriz Hessiana, g é o gradiente da função e D a matriz diagonal, Δ é um valor numérico escalar. Ademais, é utilizado os autovalores e autovetores da função $F_i(x)$ definida na forma matricial. Concluindo, então, usando o próprio método de Newton.

O MATLAB®, portanto, através do $fsolve$ consegue misturar métodos de funções não lineares (método de Newton) com os métodos lineares (gradientes e uso de autovalores e autovetores, diagonalização de matriz, etc.). Tanto o método de Newton-Raphson quanto de Broyden apresentam uma complexidade superior à O(n∧2). Isso porque além dos laços para varrimento de linhas e colunas, apresentam lógicas de condições dentro destes, acarretando numa lentidão natural por parte do MATLAB®. Entretanto, evita-se o tratamento do método com matrizes inversas cujo custo poderia ser ainda maior. O mesmo pensamento de vetorização e tratamento matricial do código diminuí o custo computacional.

4 Conclusões

A aprendizagem e o domínio dos métodos de Newton-Raphson e Quasi-Newton são de grande importância para determinar com eficiência a resolução de problemas que envolvam a solução de sistemas não lineares. O método de Newton-Raphson apresenta convergência do processo iterativo mais rápida. Isso o torna mais vantajoso em relação aos métodos anteriormente estudados. O método de Newton modificado tem a vantagem de calcular uma única vez a matriz Jacobiana. No caso de resolver por fatoração LU, os fatores L e U serão calculados uma única vez. Esse método se mostrou ineficaz para sistemas com $p \geq 10$. Por isso foi implementado o método de Quasi-Newton proposto por Broyden. É importante destacar que a ferramenta MATLAB® foi de grande auxílio na resolução dos problemas propostos porém estes métodos podem ser implementado de forma semelhante em outras linguagens como *C++, Julia, Phyton*, entre outras. Como o programa foi dividido em funções, a resolução da matriz jacobiana e o cálculo das normas foram feitas de forma satisfatória, bem como o sistema de equações não-lineares testadas no sistema tridiagonal de Broyden. Na grande maioria das resoluções de sistemas não-lineares o método de Quasi-Newton pode assim apresentar um melhor desempenho.

Referências

FRANCO, N. M. B. *Cálculo Numérico*. 10. ed. São Paulo: Pearson, 2007. Citado 4 vezes nas páginas 4, 5, 7 e 8.

MATHWORKS. *Equation Solving Algorithms*. 2018. Disponível em: <https://www.mathworks.com/help/optim/ug/equation-solving-algorithms.html>. Acesso em: 26 jun 2018. Citado na página 26.

MATHWORKS, I. T. *Matlab Reference Guide: High-Performance Numeric Computation and Vizualization Software*. 2. ed. Massachusetts, USA: The MathWorks, Inc, 1993. Citado na página 23.

MATSUMOTO Élia Y. *Matlab 7: Fundamentos*. 2. ed. São Paulo: Érica, 2008. Citado na página 23.

UFPB. *Servidor online da Produção Virtual da UFPB - Cálculo Numérico*. 2018. Disponível em: <http://producao.virtual.ufpb.br/books/reamat/CalculoNumerico/>. Acesso em: 25 jun 2018. Citado na página 5.

Apêndices

APÊNDICE A – Códigos Fonte

Listings

A seguir temos os códigos fonte feito neste trabalho:

```
function [F] = broyden(x)

n = length(x); %Definição da ordem do sistema tridiagonal de
   Broyden
if(n == 1)
    disp('O sistema tridiagonal de Broyden não está definido
       para ordem unitária!');
return
end
F = zeros(n,1); %Alocação de memória

%Definição do sistema
F(1) = x(1)*(3-.5*x(1))-2*x(2)+1;
for i = 2:n-1
    F(i) = x(i)*(3-.5*x(i))-x(i-1)-2*x(i+1)+1;
end
F(n) = x(n)*(3-.5*x(n))-x(n-1)+1;
end
```

Listing A.1 – BroydenFunção

```
function [B] = norm_inf(x)

B = max(abs(x));
end
```

Listing A.2 – Função Norm-inf

```
function [J] = jacobian(x)
%Retorna o Jacobino do Sistema Tridiagonal de Broyden
%Escolhido como Sistema Teste

n = length(x); %Definição da ordem do Jacobiano
if(n == 1)
    disp('O Jacobiano do sistema teste não está definido para
        ordem unitária!');
return
end
J = zeros(n); %Alocação de memória para o Jacobiano

J(1,1) = 3-x(1); J(1,2) = -2; %Definição da primeira linha do
    Jacobiano

%Definição da i-ésima linha do Jacobiano para i $\in$ \{2,..n
   -1\}
for i=2:n-1
    J(i,i-1) = -1;
    J(i,i) = 3-x(i);
    J(i,i+1) = -2;
end

J(n,n-1) = -1; J(n,n) = 3-x(n); %Definição da última linha do
    Jacobiano

%Return
end
```

Listing A.3 – Função Jacobian

```
function [J] = jacobian_broyden(x)
%Retorna o Jacobino do Sistema Tridiagonal de Broyden
%Escolhido como Sistema Teste

n = length(x); %Definição da ordem do Jacobiano
if(n == 1)
    disp('O Jacobiano do sistema de Broyden não está definido
        para ordem unitária!');
return
end
J = zeros(n); %Alocação de memória para o Jacobiano

J(1,1) = 3-x(1); J(1,2) = -2; %Definição da primeira linha do
    Jacobiano

%Definição da i-ésima linha do Jacobiano para i $\in$ \{2,..n
   -1\}
for i = 2:n-1
    J(i,i-1) = -1;
    J(i,i) = 3-x(i);
    J(i,i+1) = -2;
end

J(n,n-1) = -1; J(n,n) = 3-x(n); %Definição da última linha do
    Jacobiano

%Return
end
```

Listing A.4 – Função Jacobian-Broyden

```
1
2  %Decomposição LU através de pivotamento parcial com retorno
       de coeficientes
3  %em esquema de armazenamento compacto e esquema de permutação
4
5  function [A,P] = lugauss_fac(A)
6
7  n = length(A);
8  P = zeros(n,1); % Guarda as permutações do pivotamento
       parcial
9
10 for i = 1:n
11     P(i) = i;
12 end
13
14 tol_piv = sqrt(eps); % Pivo deve ser maior que a raiz
       quadrada do menor
15 %número interpretado como não nulo pelo PC.
16
17 for k = 1:(n-1)
18     pv = abs(A(k,k));
19     r = k; % Guarda a linha do pivo;
20 fori = (k+1):n
21 if(abs(A(i,k))>pv)
22 pv = abs(A(i,k)); % Bloco que define o pivô
23 r = i;
24 end
25 end
26
27 if(pv<= tol_piv)
28 fprintf('\n A matriz é singular!');
29         fprintf('\n Parando...');
30 return
31 else
32 if (r~=k)
33 aux = P(k);
34             P(k) = P(r);
35             P(r) = aux;
```

```
for j = 1:n
                    aux = A(k,j);
                    A(k,j) = A(r,j); % Permuta as linhas r e j
A(r,j) = aux;
end
end
end
fori = (k+1):n
mik = (A(i,k))/A(k,k);
A(i,k) = mik; %Coeficientes da matriz L
for j = (k+1):n
              A(i,j) = A(i,j)-mik*A(k,j);
end
end
```

Listing A.5 – Função LUGauss-Fac

```
function [ A ] = mult( X, Y )
[m,n]=size(X);
[j,k]=size(Y);

if(n~=j)
    error('Dimensões de Matrizes Incompatíveis');
end

A=zeros(m,k);
for a=1:m
for b=1:k
for v=1:j
         A(a,b)=A(a,b)+(X(a,v)*Y(v,b));
end

end
end
end
```

Listing A.6 – Função Mult

```
1  % Função que resolve um sistema linear, dada a matriz de
       decomposição
2  % LU compacta A, o vetor de permutações e o vetor de
       coeficientes b
3
4  function [x] = lugauss_sol(A,P,b)
5
6      n = length(b);
7      y = zeros(n,1);
8      x = zeros(n,1);
9
10     for i = 1:n
11         r = P(i);
12         c(i) = b(r);
13     end
14
15     for i = 1:n
16         acum = 0;
17         for j = 1:(i-1)
18             acum = acum + A(i,j)*y(j);
19         end
20         y(i) = c(i)-acum;
21     end
22
23     for i = n:-1:1
24         acum = 0;
25         for j = (i+1):n
26             acum = acum + A(i,j)*x(j);
27         end
28         x(i) = (y(i)-acum)/A(i,i);
29     end
30
31     end
```

Listing A.7 – Função LUGauss-Sol

```
clc
clear all

%% Script base para execução dos testes e comparações do sistema de Broyden resolvido por Newton-Raphson e pela função fsolve do MATLAB
p=input('Digite a ordem do sistema: ');%Leitura da ordem do sistema de Broyden definido pelo usuário;
tol = 1e-6; % Define a tolerência (norma infinita do vetor de resíduos)
iter_max = 800; %Define o número máximo de iterações realizadas pelo método de Newton-Raphson
time_fsolve = 0; %Conterá o tempo de processamento para solução do sistema com fsolve
time_newtonraphson = 0; %Conterá o tempo de processamento para solução do sistema com Newton-Raphson
aux = 0; %variável auxiliar;

%% Cálculo do grau e padrão de esparsidade da matriz para p
x0 = -ones(p,1);
spy(jacobian_broyden(x0),'+');
title('Padrão de Esparsidade do Jacobiano')
sparsity_degree = ((p^2-3*p+2)/p^2)*100;

%% Mede o tempo utilizado pelo método de Newton-Raphson
aux = cputime;
[x_nr,norm_res,iter] = newton_raphson(@broyden,@jacobian_broyden,x0,tol,iter_max);
time_newtonraphson = cputime-aux;

%% Mede o tempo utilizado pelo fsolve
aux = cputime;
[x_fs] = fsolve(@broyden,x0);
time_fsolve = cputime-aux;

%% Compara a solução encontrada pelo fsolve e o método de Newton-Raphson
[x_nr,x_fs]
```

Listing A.8 – Função main

```
clc
clear all

p=input('Digite a ordem do sistema: ');
x0=-ones(1, p); %Define a solução inicial;
tol = 1e-6; % Define a tolerência (norma infinita do vetor de
    resíduos)
inter_max = 800; %Define o número máximo de iterações
   realizadas pelo método de Quasi-Newton
time_fsolve = 0; %Contém o tempo de processamento para soluçã
   o do sistema com fsolve
time_quasinewton = 0; %Conterá o tempo de processamento para
   solução do sistema com Quasi-Newton
aux = 0; %variável auxiliar;

clc

aux = cputime;
[x_qn,iter, n] = quasi2(x0, tol, inter_max);
time_quasinewton = cputime-aux;

aux = cputime;
[x_fs] = fsolve(@broyden,x0);
time_fsolve = cputime-aux;
```

Listing A.9 – Função mainQuasiNewton

```
function [x,norm_res,iter] = newton_raphson(system,jacobian,
   x0,tol,iter_max)

%Descrição da função: retorna a solução de um sistema não-
   linear F(x)=0
%utilizando o método de Newton-Raphson

%Entrada:
% system: handle para o sistema base a ser resolvido F(x)=0
% jacobian: handle para o jacobiano do sistema base
% x0: estimativa inicial de solução
% tol: tolerância esperada, medida através da norma infinita
   do vetor F de resíduos
% iter_max: número máximo de iterações que podem ser
   realizadas pela função

%Saída:
% x: solução do sistema
% norm_res: norma do vetor de resíduos
% iter: número de iterações realizadas

n = length(x0);
x = x0;
iter = 0;
res = feval(system,x);
norm_res = norm_inf(res);

while(iter<iter_max&&norm_res>tol)
J = feval(jacobian,x);
    [A,P] = lugauss_fac(J);%Fatora o jacobiano em LU com
       pivotamento parcial em armazenamento compacto
    dx = lugauss_sol(A,P,-res);
    x = sum_mat(x,dx);
res = feval(system,x);
norm_res = norm_inf(res);
iter = iter+1;
end
```

Listing A.10 – Newton-Raphson

```
function [x,k, n]=quasi2(x, tol , ninter_max)
% O metodo da secante tem por finalidade encontrar
% a solucao do sistema nao linear de equacoes da
% forma F(x)=0

%o numero de iteracoes inicia com valor zero
k=0;
%atribuicao de valor zero a variavel de controle
control=0;
%Bk recebe o valor da jacobiana de F(x) em x
B=jacobian(x);
[B,P] = lugauss_fac(B);
%especificando quando o loop finaliza
while (control==0)&&(k<=ninter_max)
%determinando o valor de F(x) no ponto x em questao
f=broyden(x);
%Bks=-f;
s = lugauss_sol(B,P,-f);
%armazenando o valor de x_k
 a=transp(s);
 xp=sum_mat(x,a); % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% soma(x,a) %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5
%determinando o valor de F(x) no ponto xp
 f2=broyden(xp);
%a variavel y recebe a diferenca de F(xp)-F(x)
 y=sub(f2, f);
%atualizacao da matriz Bk
%B=B+((y-B*s)*a)/(a*s)
 dum1=mult(B, s);
 dum2=sub(y, dum1);
 dum3=mult(dum2, a);
 dum4=mult(a,s);
 dum=dum3/dum4;
 B=sum_mat(B, dum);% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% soma(B,dum) %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5

```

```
35 %testando a variavel de controle
36 if norm_inf(xp-x)<=tol
37 control=1;
38 end
39  n = norm_inf(xp-x);
40 %atualizando x
41  x=xp;
42 %atualizando o numero de iteracoes
43  k=k+1;
44 end
```

Listing A.11 – Função Quasi2

```
1  function [ A ] = sub( X, Y )
2  [m,n]=size(X);
3  [j,k]=size(Y);
4
5  if(m~=j)||(n~=k)
6      error('Dimensões de Matrizes Incompatíveis');
7  end
8
9  A=zeros(m,n);
10
11 for a=1:m
12 for b=1:n
13 A(a,b)=X(a,b)-Y(a,b);
14 end
15 end
16 end
```

Listing A.12 – Função Sub

```
% Função que soma duas matrizes

function C = sum_mat(A,B);
[M,N] = size(A);
[O,P] = size(B);

if(N~=P || M ~=O)
error('As matrizes não podem ser somadas! Dimensões incompatí
   veis');
else
    C = zeros(M,N);
for i=1:M
for j=1:N
            C(i,j)=A(i,j)+B(i,j);
end
end
end
```

Listing A.13 – Função Sum-Mat

```

function [ b ] = transp(v)
[m,n]=size(v);
b=zeros(n,m);

for i=1:n
for j=1:m
b(i,j)=v(j,i);
end
end
end
```

Listing A.14 – Função Transp