Algebra Linear Computacional COC473 - Lista 4

Bruno Dantas de Paiva
DRE: 118048097

October 18, 2020

1 Programas

1.1 Método da Bissecção

```
STEPS       = 100
TOL         = 10**-4

def bissection_method(function, a, b):
    if function(a)*function(b) > 0:
        print("Error")
        return;

    c = a
    while((b - a) >= TOL):

        c = (a+b)/2
        if(function(c) == 0):
            break

        if (function(c)*function(a) < 0):
            b = c
        else:
            a = c

    print("Root: " + str(round(c, 3)))
```

1.2 Método de Newton Original

```
STEPS          = 100
TOL            = 10**-4

def newton_method(function, x0):
    x = x0

    for i in range(STEPS):
        xi      = x - function(x) / Matrix_Utils.derivate(function, x)
        residue = math.fabs(xi - x)

        if(residue < TOL):
            print("Root: " + str(xi))
            return;

        x = xi

    print("Convergence not reached")
```

Note que esta função possui rotinas secundárias para auxiliar nos cálculos, onde estas funções podem ser encontradas abaixo:

```
def derivate(function, value):
    delta       = 10**(-10)
    numerator   = function(value + delta) - function(value)
    denominator = delta
    result      = numerator/denominator

    return result
```

1.3 Método de Newton Original

```
STEPS        = 100
TOL          = 10**-4

def secant_method(function, x0):
    delta_x        = 0.001
    first_x        = x0
    actual_x       = first_x + delta_x
    first_function = function(first_x)

    for i in range(STEPS):
        actual_function = function(actual_x)
        next_x          = actual_x - float((actual_function*(actual_x - first_x))/(actua
        residue         = abs(next_x - actual_x)

        if (residue < TOL):
            print("Root: " + str(actual_x))
            return

        first_function = actual_function
        first_x        = actual_x
        actual_x       = next_x

    print("Convergence not reached")
```

1.4 Método da Interpolação Inversa

```
STEPS        = 100
TOL          = 10**-4

def inverse_interpolation_method(function, x1, x2, x3):
    x0        = 10**(36)

    y1         = function(x1)
    y2         = function(x2)
    y3         = function(x3)

    x          = [x1, x2, x3]
    y          = [y1, y2, y3]

    for i in range(STEPS):
        xi      = Matrix_Utils.inverse_interpolation_helper(x[0], x[1], x[2], y[0], y[1]
        residue = abs(xi - x0)

        if (residue < TOL):
            print("Root: " + str(xi))
            return

        y_max = max(y)
        i     = y.index(y_max)
        x[i]  = xi
        y[i]  = function(xi)
        x0    = xi

        y.sort()
        x.sort()

    print("Convergence not reached")
```

Note que esta função possui rotinas secundárias para auxiliar nos cálculos, onde estas funções podem ser encontradas abaixo:

```
def inverse_interpolation_helper(x1, x2, x3, y1, y2, y3):
    a = (y2*y3*x1) / ((y1 - y2)*(y1 - y3))
    b = (y1*y3*x2) / ((y2 - y1)*(y2 - y3))
    c = (y1*y2*x3) / ((y3 - y1)*(y3 - y2))
    return a + b + c
```

1.5 Método de Newton para Sistema de Equações não lineares

```
STEPS       = 100
TOL         = 10**-4

def non_linear_newton_method(functions_list, first_solution):
    x = first_solution

    for i in range(STEPS):
        jacobian_matrix  = Matrix_Utils.get_jacobian_matrix(functions_list, x)
        vector_f         = Matrix_Utils.get_f_vector(functions_list, x)
        jacobian_inverse = Matrix_Utils.get_inverse_matrix(jacobian_matrix)
        if(jacobian_inverse == 0):
            print("Error")
            break

        delta_x = Matrix_Utils.multiply_matrix_vector(jacobian_inverse, vector_f)
        delta_x = [x*(-1) for x in delta_x]
        x       = Matrix_Utils.sum_vectors(x, delta_x)

        residue = Matrix_Utils.vector_norm(delta_x)/Matrix_Utils.vector_norm(x)

        if (residue < TOL):
            print("Solution: " + str(x))
            return

    print("Convergence not reached")
```

Note que esta função possui rotinas secundárias para auxiliar nos cálculos, onde estas funções podem ser encontradas abaixo:

```
def get_jacobian_matrix(functions_list, first_solution):
    first_dimention  = len(functions_list)
    second_dimention = len(first_solution)

    result = [[0 for _ in range(second_dimention)] for _ in range(first_dimention)]

    for i in range(first_dimention):
        for j in range(second_dimention):
            result[i][j] = partial_derivate(functions_list[i], first_solution, j)
    return result


def get_f_vector(functions_list, first_solution):
    dimension = len(functions_list)
    result    = [0 for _ in range(dimension)]

    for i in range(dimension):
        result[i] = functions_list[i](first_solution)

    return result
```

```python
def get_inverse_matrix(matrix):
    cofactors   = []
    determinant = matrix_determinant(matrix)
    if(determinant == 0):
        return 0

    for r in range(len(matrix)):
        cofactorRow = []

        for c in range(len(matrix)):
            minor = inverse_auxiliar_function(matrix,r,c)
            cofactorRow.append(((-1)**(r+c)) * matrix_determinant(minor))

        cofactors.append(cofactorRow)

    cofactors = get_transposed_matrix(cofactors)

    for r in range(len(cofactors)):
        for c in range(len(cofactors)):

            cofactors[r][c] = cofactors[r][c]/determinant

    return cofactors


def inverse_auxiliar_function(matrix, i, j):
    return [row[:j] + row[j+1:] for row in (matrix[:i]+matrix[i+1:])]


def vector_norm(vector):
    dimension = len(vector)
    result    = 0

    for i in range(dimension):
        result += vector[i] * vector[i]

    return result**(0.5)


def sum_vectors(vector_a , vector_b):
    dimension = len(vector_a)
    result    = [0 for i in range(dimension)]

    for i in range(dimension):
        result[i] = vector_a[i] + vector_b[i]

    return result
```

1.6 Método de Broyden para Sistema de Equações não lineares

```
STEPS        = 100
TOL          = 10**-4

def broyden_method(functions_list, first_solution):
    dimension      = len(first_solution)
    x              = first_solution
    jacobian_b     = Matrix_Utils.get_jacobian_matrix(functions_list, x)
    jacobian_a     = [[0 for i in range(dimension)] for j in range(dimension)]

    for i in range(STEPS):

        for k in range(dimension):
            for j in range(dimension):
                jacobian_a[k][j] = jacobian_b[k][j]

        vector_f         = Matrix_Utils.get_f_vector(functions_list, x)
        jacobian_inverse = Matrix_Utils.get_inverse_matrix(jacobian_a)
        if(jacobian_inverse == 0):
            print("Error")
            break

        delta_x = Matrix_Utils.multiply_matrix_vector(jacobian_inverse, vector_f)
        delta_x = [x*(-1) for x in delta_x]

        x = Matrix_Utils.sum_vectors(x, delta_x)

        vector_f2 = Matrix_Utils.get_f_vector(functions_list, x)

        yK      = Matrix_Utils.sum_vectors(vector_f2, [x*(-1) for x in vector_f])
        residue = Matrix_Utils.vector_norm(delta_x)/Matrix_Utils.vector_norm(x)
        if (residue < TOL):
            print("Solution: " + str(x))
            return

        product = Matrix_Utils.multiply_matrix_vector(jacobian_b, delta_x)
        product = [x*(-1) for x in product]

        denominator = Matrix_Utils.vector_multiplication(delta_x, delta_x)
        top         = Matrix_Utils.sum_vectors(yK, product)
        numerator   = Matrix_Utils.broyden_method_helper(top, delta_x)

        for k in range(len(jacobian_b)):
            for j in range(len(jacobian_b)):
                jacobian_b[k][j] = numerator[k][j]/denominator

        jacobian_b = Matrix_Utils.sum_matrixes(jacobian_a, jacobian_b)

    print("Convergence not reached")
```

Note que esta função possui rotinas secundárias para auxiliar nos cálculos, onde estas funções podem ser encontradas abaixo:

```
def broyden_method_helper(vector_a, vector_b):
    dimension = len(vector_a)
    result    = [[0 for i in range(dimension)] for j in range(dimension)]

    for i in range(dimension):
        for j in range(dimension):
            result[i][j] = vector_a[i]*vector_b[j]

    return result
```

1.7 Método dos Mínimos Quadrados para Sistema de Equações não lineares

```
STEPS         = 100
TOL           = 10**-4

def non_linear_mmq(functions_list, vector_x, vector_y, first_solution):
    x = first_solution

    for i in range (STEPS):
        jacobian            = Matrix_Utils.get_jacobian_matrix(functions_list, x)
        transposed_jacobian = Matrix_Utils.get_transposed_matrix(jacobian)
        vector_f            = Matrix_Utils.get_f_vector(functions_list,x)

        a = Matrix_Utils.get_inverse_matrix(Matrix_Utils.multiply_matrixes(transposed_jac
        if(a == 0):
            print("Error")
            break

        b       = Matrix_Utils.multiply_matrix_scalar(a, -1)
        c       = Matrix_Utils.multiply_matrix_vector(transposed_jacobian,vector_f)
        delta_b = Matrix_Utils.multiply_matrix_vector(b,c)
        x       = Matrix_Utils.sum_vectors(x, delta_b)

        residue = Matrix_Utils.vector_norm(delta_b)/Matrix_Utils.vector_norm(x)

        if (residue < TOL):
            print("Solution: " + str(x))
            return

    print("Convergence not reached")
```

Note que esta função possui rotinas secundárias para auxiliar nos cálculos, onde estas funções podem ser encontradas abaixo:

```
def multiply_matrix_scalar(matrix_a, scalar):
    n      = len(matrix_a)
    result = [[0.0 for _ in range(n)] for _ in range(n)]

    for j in range(n):
        for i in range(n):
            result[i][j] = matrix_a[i][j] * scalar

    return result
```

2 Questão 1

2.1 Método da Bissecção

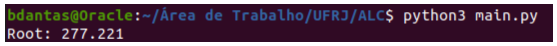


Figure 1: Imagem contendo uma das raizes para a questão 1 usando o método da bissecção

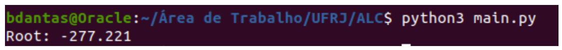


Figure 2: Imagem contendo uma das raizes para a questão 1 usando o método da bissecção

2.2 Método de Newton Original

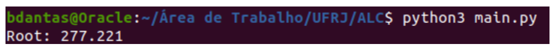


Figure 3: Imagem contendo uma das raizes para a questão 1 usando o método da newton original

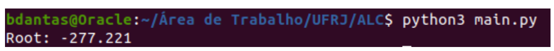


Figure 4: Imagem contendo uma das raizes para a questão 1 usando o método da newton original

2.3 Método de Newton Secante

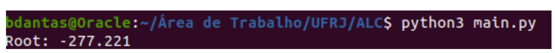


Figure 5: Imagem contendo uma das raizes para a questão 1 usando o método da newton secante

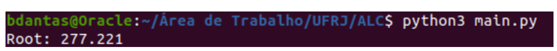


Figure 6: Imagem contendo uma das raizes para a questão 1 usando o método da newton secante

2.4 Método da Interpolação Inversa

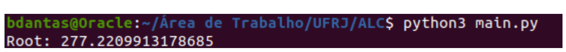


Figure 7: Imagem contendo uma das raizes para a questão 1 usando o método da interpolação inversa

3 Questão 2

Observação: Como esta função possui cosseno em seu termo, é notável que esta função terá um número infinito de raízes, contudo, foram considerados nesta lista somente 2 raízes.

3.1 Método da Bissecção

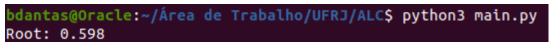


Figure 8: Imagem contendo uma das raizes para a questão 2 usando o método da bissecção

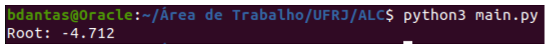


Figure 9: Imagem contendo uma das raizes para a questão 2 usando o método da bissecção

3.2 Método de Newton Original

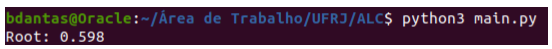


Figure 10: Imagem contendo uma das raizes para a questão 2 usando o método de newton original

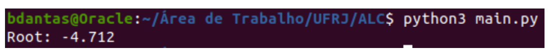


Figure 11: Imagem contendo uma das raizes para a questão 2 usando o método de newton original

3.3 Método de Newton Secante

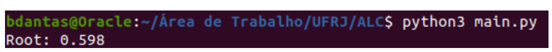


Figure 12: Imagem contendo uma das raizes para a questão 2 usando o método de newton secante

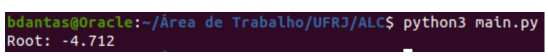


Figure 13: Imagem contendo uma das raizes para a questão 2 usando o método de newton secante

3.4 Método da Interpolação Inversa

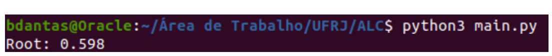


Figure 14: Imagem contendo uma das raizes para a questão 2 usando o método da interpolação inversa

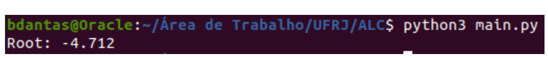


Figure 15: Imagem contendo uma das raizes para a questão 2 usando o método da interpolação inversa

4 Questão 3

Note que para esta função, podemos possuir mais que uma solução, com isso, pode-se observar com os resultados abaixo 2 possíveis soluções dependendo do chute inicial adotado para atender tal equação.

4.1 Método de Newton para Sistema de Equações não lineares

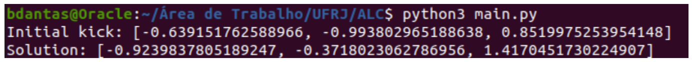


Figure 16: Imagem contendo uma das soluções para a questão 3 usando o método de newton para sistema de equações não lineares

4.2 Método de Broyden para Sistema de Equações não lineares

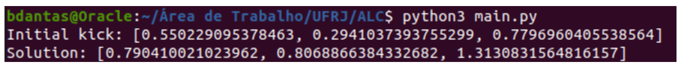


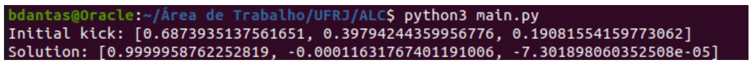
Figure 17: Imagem contendo uma das soluções para a questão 3 usando o método de broyden para sistema de equações não lineares

5 Questão 4

Note que para esta função, podemos possuir mais que uma solução, com isso, pode-se observar com os resultados abaixo várias possíveis soluções dependendo do chute inicial adotado assim como a questão anterior.

5.1 $\theta_1 = 0.00$, $\theta_2 = 3.0$

5.1.1 Método de Newton para Sistema de Equações não lineares

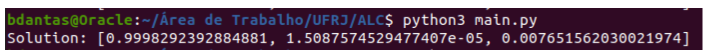


```
bdantas@Oracle:~/Área de Trabalho/UFRJ/ALC$ python3 main.py
Initial kick: [0.6873935137561651, 0.39794244359956776, 0.19081554159773062]
Solution: [0.9999958762252819, -0.00011631767401191006, -7.301898060352508e-05]
```

Figure 18: Imagem contendo uma das soluções para a questão 4 usando o método de newton para sistema de equações não lineares

5.1.2 Método de Broyden para Sistema de Equações não lineares

Chute inicial: $[0.4784, 0.5966, -0.8031]$

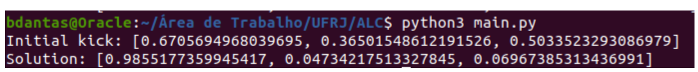


```
bdantas@Oracle:~/Área de Trabalho/UFRJ/ALC$ python3 main.py
Solution: [0.9998292392884881, 1.5087574529477407e-05, 0.007651562030021974]
```

Figure 19: Imagem contendo uma das soluções para a questão 4 usando o método de broyden para sistema de equações não lineares

5.2 $\theta_1 = 0.75$, $\theta_2 = 6.5$

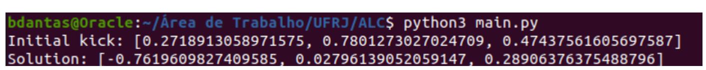
5.2.1 Método de Newton para Sistema de Equações não lineares



```
bdantas@Oracle:~/Área de Trabalho/UFRJ/ALC$ python3 main.py
Initial kick: [0.6705694968039695, 0.36501548612191526, 0.5033523293086979]
Solution: [0.9855177359945417, 0.04734217513327845, 0.06967385313436991]
```

Figure 20: Imagem contendo uma das soluções para a questão 4 usando o método de newton para sistema de equações não lineares

5.2.2 Método de Broyden para Sistema de Equações não lineares



```
bdantas@Oracle:~/Área de Trabalho/UFRJ/ALC$ python3 main.py
Initial kick: [0.2718913058971575, 0.7801273027024709, 0.47437561605697587]
Solution: [-0.7619609827409585, 0.02796139052059147, 0.28906376375488796]
```

Figure 21: Imagem contendo uma das soluções para a questão 4 usando o método de broyden para sistema de equações não lineares

5.3 $\theta_1 = 0.00$, $\theta_2 = 11.667$

5.3.1 Método de Newton para Sistema de Equações não lineares

```
Initial kick: [0.46990325929067445, -0.674996277484541, -0.11530113984281498]
Solution: [0.5877635865320168, -0.569215163813701, -0.03611692377849667]
```

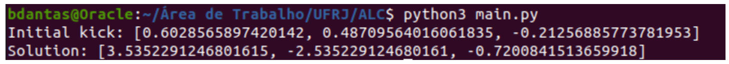
Figure 22: Imagem contendo uma das soluções para a questão 4 usando o método de newton para sistema de equações não lineares

5.3.2 Método de Broyden para Sistema de Equações não lineares

```
bdantas@Oracle:~/Área de Trabalho/UFRJ/ALC$ python3 main.py
Initial kick: [0.27184473638109075, 0.0953240585212276, -0.5608342019912129]
Solution: [0.7550584047222728, -1.5997115660523281e-07, -0.29321909689421605]
```

Figure 23: Imagem contendo uma das soluções para a questão 4 usando o método de newton para sistema de equações não lineares

6 Questão 5



```
bdantas@Oracle:~/Área de Trabalho/UFRJ/ALC$ python3 main.py
Initial kick: [0.6028565897420142, 0.48709564016061835, -0.21256885773781953]
Solution: [3.5352291246801615, -2.535229124680161, -0.7200841513659918]
```

Figure 24: Imagem contendo o ajuste da função não linear da questão 5 usando o método do mmq não linear