algoritmos

September 20, 2021

0.1 COC473 - Trabalho 2 - 2021.1

0.1.1 Aluno: Henrique Chaves Magalhães de Menezes

0.1.2 DRE: 119025571

0.2 Algoritmos:

0.2.1 1.1 - Método de Newton para resolução de equações não lineares

[1]:
```
import numpy as np
from sympy import Symbol, diff

def create_F(number_of_functions, functions):
    F = np.array([[functions[i]] for i in range(number_of_functions)])
    return F

def create_J(number_of_functions, number_of_constants, functions):
    J = np.empty((number_of_functions, number_of_constants), dtype=object)
    for i in range(number_of_functions):
        for j in range(number_of_constants):
            J[i][j] = diff(functions[i], f"c{j+2}")
    return J

def update_c_dict(X, extra_dict={}):
    c_dict = {f"c{i+2}":X[i, 0] for i in range(len(X))}
    c_dict.update(extra_dict)
    return c_dict

def sub_F(F, c_dict):
    F = F.copy()
    for i in range(F.shape[0]):
        for j in range(F.shape[1]):
            F[i][j] = F[i][j].subs(c_dict)
    return F.astype(float)

def sub_J(J, c_dict):
    J = J.copy()
    for i in range(J.shape[0]):
        for j in range(J.shape[1]):
            J[i][j] = J[i][j].subs(c_dict)
    return J.astype(float)

def calculate_nl_newton(max_iter, max_tol, X0, theta_dict, F, J):
    X_list = []
    X_list.append(X0)

    for k in range(1, max_iter+1):
        c_dict = update_c_dict(X_list[k-1], theta_dict)
        F_k = sub_F(F, c_dict)
        J_k = sub_J(J, c_dict)
        J_k_inv = np.linalg.inv(J_k)
        delta_X = np.dot(-J_k_inv, F_k)
        X_k = X_list[k-1] + delta_X
        X_list.append(X_k.copy())
        tolk = np.linalg.norm(delta_X)/np.linalg.norm(X_k)
        if tolk < max_tol:
            return X_k

        if k == max_iter:
            raise Exception("Não convergiu")


def run_newton(max_iter, max_tol, X0, theta_dict):
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
     1 = Symbol(' 1')
     2 = Symbol(' 2')

    f1 = 2*c3**2+c2**2+6*c4**2-1
    f2 = 8*c3**3+6*c3*c2**2+36*c3*c2*c4+108*c3*c4**2- 1
    f3 =␣
 ↪60*c3**4+60*c3**2*c2**2+576*c3**2*c2*c4+2232*c3**2*c4**2+252*c4**2*c2**2+1296*c4**3*c2+3348

    fns = [f1, f2, f3]

    number_of_functions = 3
    number_of_constants = 3


    F = create_F(number_of_functions, fns)
    J = create_J(number_of_functions, number_of_constants, fns)

    X = calculate_nl_newton(max_iter, max_tol, X0, theta_dict, F, J)
    return np.round(X.ravel(), 3).tolist()
```

Exemplo:

```
[22]: c2 = Symbol('c2')
      c3 = Symbol('c3')
      c4 = Symbol('c4')
       1 = Symbol(' 1')
       2 = Symbol(' 2')

      f1 = 2*c3**2+c2**2+6*c4**2-1
      f2 = 8*c3**3+6*c3*c2**2+36*c3*c2*c4+108*c3*c4**2- 1
      f3 =␣
       ↪60*c3**4+60*c3**2*c2**2+576*c3**2*c2*c4+2232*c3**2*c4**2+252*c4**2*c2**2+1296*c4**3*c2+3348
      display(f1)
      display(f2)
      display(f3)
```

$c_2^2 + 2c_3^2 + 6c_4^2 - 1$

$6c_2^2c_3 + 36c_2c_3c_4 + 8c_3^3 + 108c_3c_4^2 - {}_1$

$24c_2^3c_4 + 60c_2^2c_3^2 + 252c_2^2c_4^2 + 576c_2c_3^2c_4 + 1296c_2c_4^3 + 3c_2 + 60c_3^4 + 2232c_3^2c_4^2 + 3348c_4^4 - {}_2$

```
[7]: theta_dict = {" 1": 0.75, " 2": 6.5}
     max_iter = 100
     max_tol = 1e-5

     X0 = np.array([[1], [0], [4]])


     c2, c3, c4 = run_newton(max_iter, max_tol, X0, theta_dict)
     print("Constantes encontradas:")
     print(f"c2: {c2}")
     print(f"c3: {c3}")
     print(f"c4: {c4}")
```

```
Constantes encontradas:
c2: 0.784
c3: 0.25
c4: -0.208
```

0.2.2 1.2 - Método de Broyden para resolução de equações não lineares

```
[8]: import numpy as np
     from sympy import Symbol, diff

     def create_F(number_of_functions, functions):
         F = np.array([[functions[i]] for i in range(number_of_functions)])
         return F
```

```python
def create_J(number_of_functions, number_of_constants, functions):
    J = np.empty((number_of_functions, number_of_constants), dtype=object)
    for i in range(number_of_functions):
        for j in range(number_of_constants):
            J[i][j] = diff(functions[i], f"c{j+2}")
    return J

def update_c_dict(X, extra_dict={}):
    c_dict = {f"c{i+2}":X[i, 0] for i in range(len(X))}
    c_dict.update(extra_dict)
    return c_dict

def sub_F(F, c_dict):
    F = F.copy()
    for i in range(F.shape[0]):
        for j in range(F.shape[1]):
            F[i][j] = F[i][j].subs(c_dict)
    return F.astype(float)

def sub_J(J, c_dict):
    J = J.copy()
    for i in range(J.shape[0]):
        for j in range(J.shape[1]):
            J[i][j] = J[i][j].subs(c_dict)
    return J.astype(float)

def calculate_nl_broyden(max_iter, max_tol, theta_dict, X0, F, J):
    X_arr = np.empty((max_iter, ), dtype="object")
    B_arr = np.empty((max_iter, ), dtype="object")

    X_arr[0] = X0

    B_arr[0] = sub_J(J, {"c2": X_arr[0][0][0], "c3": X_arr[0][1][0], "c4":␣
 ↪X_arr[0][2][0]} | theta_dict)

    for k in range(1, max_iter+1):
        if k == max_iter:
            raise Exception("Não convergiu")
        J_ = B_arr[k-1]
        delta_X = - np.matmul(np.linalg.inv(J_), sub_F(F, {"c2":␣
 ↪X_arr[k-1][0][0], "c3": X_arr[k-1][1][0], "c4": X_arr[k-1][2][0]} |␣
 ↪theta_dict))
        X_arr[k] = X_arr[k-1] + delta_X
        Y_k = sub_F(F, {"c2": X_arr[k][0][0], "c3": X_arr[k][1][0], "c4":␣
 ↪X_arr[k][2][0]} | theta_dict) - sub_F(F, {"c2": X_arr[k-1][0][0], "c3":␣
 ↪X_arr[k-1][1][0], "c4": X_arr[k-1][2][0]} | theta_dict)
        tolk = np.linalg.norm(delta_X)/np.linalg.norm(X_arr[k])
        if tolk < max_tol:
            return X_arr[k]
        else:
            B_arr[k] = B_arr[k-1] + ((Y_k - (B_arr[k-1] @ delta_X)) @ (delta_X.
↪T))/((delta_X.T) @ delta_X)

def run_broyden(max_iter, max_tol, X0, theta_dict):
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
     1 = Symbol(' 1')
     2 = Symbol(' 2')

    f1 = 2*c3**2+c2**2+6*c4**2-1
    f2 = 8*c3**3+6*c3*c2**2+36*c3*c2*c4+108*c3*c4**2- 1
    f3 =␣
↪60*c3**4+60*c3**2*c2**2+576*c3**2*c2*c4+2232*c3**2*c4**2+252*c4**2*c2**2+1296*c4**3*c2+3348

    fns = [f1, f2, f3]

    number_of_functions = 3
    number_of_constants = 3

    F = create_F(number_of_functions, fns)
    J = create_J(number_of_functions, number_of_constants, fns)

    X = calculate_nl_broyden(max_iter, max_tol, theta_dict, X0, F, J)
    return np.round(X.ravel(), 3).tolist()
```

Exemplo:

[21]:
```
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
 1 = Symbol(' 1')
 2 = Symbol(' 2')

f1 = 2*c3**2+c2**2+6*c4**2-1
f2 = 8*c3**3+6*c3*c2**2+36*c3*c2*c4+108*c3*c4**2- 1
f3 =␣
↪60*c3**4+60*c3**2*c2**2+576*c3**2*c2*c4+2232*c3**2*c4**2+252*c4**2*c2**2+1296*c4**3*c2+3348
display(f1)
display(f2)
display(f3)
```

$c_2^2 + 2c_3^2 + 6c_4^2 - 1$

$6c_2^2c_3 + 36c_2c_3c_4 + 8c_3^3 + 108c_3c_4^2 - {}_1$

$$24c_2^3c_4 + 60c_2^2c_3^2 + 252c_2^2c_4^2 + 576c_2c_3^2c_4 + 1296c_2c_4^3 + 3c_2 + 60c_3^4 + 2232c_3^2c_4^2 + 3348c_4^4 - 2$$

[14]:
```python
theta_dict = {" 1": 0, " 2": 3}
max_iter = 100
max_tol = 1e-5

X0 = np.array([[1], [0.5], [-1]])


c2, c3, c4 = run_broyden(max_iter, max_tol, X0, theta_dict)
print("Constantes encontradas:")
print(f"c2: {c2}")
print(f"c3: {c3}")
print(f"c4: {c4}")
```

```
Constantes encontradas:
c2: 0.891
c3: 0.0
c4: -0.185
```

0.2.3 2.1.1 - Método da Bisseção para encontrar uma raiz em um intervalo

[15]:
```python
import numpy as np
from sympy import Symbol, exp

def run_bisseccao(constants, a, b, max_tol, max_iter):
    x = Symbol('x')
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    f = c1*exp(c2*x)+c3*x**c4

    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
 ↪constants[3]}
    count = 0
    while np.abs(b-a) > max_tol:
        x_i = (a+b)/2
        f_i = f.subs(c_dict | {"x": x_i})
        if (f_i > 0):
            b = x_i
        else:
            a = x_i
        if count == max_iter:
            raise Exception("Não convergiu!")

    return x_i
```

Exemplo:

[19]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
display(f)
```

$c_1e^{c_2x} + c_3x^{c_4}$

[20]:
```
#c1, c2, c3, c4
constants = [1, 1, 1, 0]

#Intervalo entre 'a' e 'b'
a = 0
b = 10

max_iter = 100
max_tol = 1e-5

raiz = run_bisseccao(constants, a, b, max_tol, max_iter)
print(f"Raiz encontrada: {raiz}")
```

```
Raiz encontrada: 9.5367431640625e-06
```

0.2.4 2.1.2 - Método de Newton para encontrar uma raiz a partir de um x0

[28]:
```
import numpy as np
from sympy import Symbol, exp, diff

def run_newton_root(constants, x_0, max_tol, max_iter):
    x = Symbol('x')
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    f = c1*exp(c2*x)+c3*x**c4
    f_deriv = diff(f, x)

    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
 ↪constants[3]}
    for k in range(1, max_iter+1):
        if k == 1:
            x_old = x_0
        x_k = x_old - float(f.subs({"x": x_old} | c_dict))/float(f_deriv.
↪subs({"x": x_old} | c_dict))
        tolk = np.abs(x_k - x_old)
        x_old = x_k
        if tolk < max_tol:
            return x_k
        if k == max_iter:
            raise Exception("Não convergiu!")
```

Exemplo:

[29]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
display(f)
```

$c_1 e^{c_2 x} + c_3 x^{c_4}$

[30]:
```
#c1, c2, c3, c4
constants = [1, 1, 1, 1]

#Chute inicial x0
x_0 = 1

max_iter = 100
max_tol = 1e-5

raiz = run_newton_root(constants, x_0, max_tol, max_iter)
print(f"Raiz encontrada: {raiz}")
```

```
Raiz encontrada: -0.5671432904097811
```

0.2.5 2.2.1 - Quadratura de Gauss para integrar uma função em um intervalo

[33]:
```
import numpy as np
from sympy import Symbol, exp

def run_gauss_quadrature(constants, a, b, N):
    x = Symbol("x")
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
↪constants[3]}
```

},

```
    f = c1*exp(c2*x)+c3*x**c4

    gauss_weights = {
    2: {
        1: 1,
        2: 1
    },

    3: {
        1: 0.8888888888888888,
        2: 0.5555555555555556,
        3: 0.5555555555555556
    },

    4: {
        1: 0.6521451548625461,
        2: 0.6521451548625461,
        3: 0.3478548451374538,
        4: 0.3478548451374538
    },

    5: {
        1: 0.5688888888888889,
        2: 0.4786286704993665,
        3: 0.4786286704993665,
        4: 0.2369268850561891,
        5: 0.2369268850561891
    },

    6: {
        1: 0.3607615730481386,
        2: 0.3607615730481386,
        3: 0.4679139345726910,
        4: 0.4679139345726910,
        5: 0.1713244923791704,
        6: 0.1713244923791704
    },

    7: {
        1: 0.4179591836734694,
        2: 0.3818300505051189,
        3: 0.3818300505051189,
        4: 0.2797053914892766,
        5: 0.2797053914892766,
        6: 0.1294849661688697,
        7: 0.1294849661688697
    },
    8: {
        1: 0.3626837833783620,
        2: 0.3626837833783620,
        3: 0.3137066458778873,
        4: 0.3137066458778873,
        5: 0.2223810344533745,
        6: 0.2223810344533745,
        7: 0.1012285362903763,
        8: 0.1012285362903763
    },

    9: {
        1: 0.3302393550012598,
        2: 0.1806481606948574,
        3: 0.1806481606948574,
        4: 0.0812743883615744,
        5: 0.0812743883615744,
        6: 0.3123470770400029,
        7: 0.3123470770400029,
        8: 0.2606106964029354,
        9: 0.2606106964029354
    },

    10: {
        1: 0.2955242247147529,
        2: 0.2955242247147529,
        3: 0.2692667193099963,
        4: 0.2692667193099963,
        5: 0.2190863625159820,
        6: 0.2190863625159820,
        7: 0.1494513491505806,
        8: 0.1494513491505806,
        9: 0.0666713443086881,
        10: 0.0666713443086881
    }
    }

    gauss_abscissas = {
    2: {
        1: -0.5773502691896257,
        2: 0.5773502691896257
    },

    3: {
        1: 0,
        2: -0.7745966692414834,
        3: 0.7745966692414834
    },

    4: {
        1:        -0.3399810435848563,
        2:         0.3399810435848563,
        3: -0.8611363115940526,
        4: 0.8611363115940526
    },

    5: {
        1: 0,
        2: -0.5384693101056831,
        3: 0.5384693101056831,
        4: -0.906179845938664,
        5: 0.9061798459386640
    },

    6: {
        1: 0.6612093864662645,
        2: -0.6612093864662645,
        3: -0.2386191860831969,
        4: 0.2386191860831969,
        5: -0.9324695142031521,
        6: 0.9324695142031521
    },

    7: {
        1: 0,
        2: 0.4058451513773972,
        3: -0.4058451513773972,
        4: -0.7415311855993945,
        5: 0.7415311855993945,
        6: -0.9491079123427585,
        7: 0.9491079123427585
    },

    8: {
        1: -0.1834346424956498,
        2: 0.1834346424956498,
        3: -0.5255324099163290,
        4: 0.5255324099163290,
        5: -0.7966664774136267,
        6: 0.7966664774136267,
        7: -0.9602898564975363,
        8: 0.9602898564975363
    },
    9: {
        1: 0,
        2: -0.8360311073266358,
        3: 0.8360311073266358,
        4: -0.9681602395076261,
        5: 0.9681602395076261,
        6: -0.3242534234038089,
        7: 0.3242534234038089,
        8: -0.6133714327005904,
        9: 0.6133714327005904
    },

    10: {
        1: -0.1488743389816312,
        2: 0.1488743389816312,
        3: -0.4333953941292472,
        4: 0.4333953941292472,
        5: -0.6794095682990244,
        6: 0.6794095682990244,
        7: -0.8650633666889845,
        8: 0.8650633666889845,
        9: -0.9739065285171717,
        10: 0.9739065285171717
    }
    }

    L = b - a
    area = 0

    for i in range(1, N+1):
        w_i = gauss_weights[N][i]
        z_i = gauss_abscissas[N][i]
        x_i = (a + b + gauss_abscissas[N][i]*L)/2
        f_i = float(f.subs({"x": x_i} | c_dict))
        area += f_i*w_i

    return area*L/2
```

Exemplo:

[34]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
```

```
display(f)
```

$c_1 e^{c_2 x} + c_3 x^{c_4}$

[36]:
```
#c1, c2, c3, c4
constants = [1, 1, 1, 0]

#Intervalo a-b a ser integrado
a = 0
b = 1

#Número de pontos de integração (entre 2 a 10)
N = 5


area = run_gauss_quadrature(constants, a, b, N)
print(f"Área: {area}")
```

```
Área: 2.7182818284583914
```

0.2.6 2.2.2 - Quadratura Polinomial para integrar uma função em um intervalo

[38]:
```
import numpy as np
from sympy import Symbol, exp

def get_polinomial_x(a, b):
    polinomial_x = {}

    for N in range(2, 11):
        delta = delta = (b-a)/(N-1)
        polinomial_x[N] = {}

        for i in range(1, N+1):
            if i == 1:
                polinomial_x[N][i] = a
            elif i == N:
                polinomial_x[N][i] = b
            else:
                polinomial_x[N][i] = a + (i-1)*delta

    return polinomial_x

def get_polinomial_weights():
    polinomial_weights = {}

    for N in range(2, 11):
        polinomial_weights[N] = {}
        A = np.empty((N, N))
        B = np.empty((N, 1))
        x = np.empty((N, 1))
        delta = 1/(N-1)

        for i in range(1, N+1):
            x[i-1][0] = (i-1)*delta

        for i in range(1, N+1):
            for j in range(1, N+1):
                A[i-1][j-1] = x[j-1]**(i-1)
            B[i-1][0] = 1/i

        w = np.dot(np.linalg.inv(A), B)

        for i in range(1, N+1):
            polinomial_weights[N][i] = w[i-1][0]
    return polinomial_weights

def run_polinomial_quadrature(constants, a, b, N):
    x = Symbol("x")
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
↪constants[3]}
    f = c1*exp(c2*x)+c3*x**c4
    polinomial_x = get_polinomial_x(a, b)
    polinomial_weights = get_polinomial_weights()

    area = 0
    for i in range(1, N+1):
        w_i = polinomial_weights[N][i]
        x_i = polinomial_x[N][i]
        f_i = float(f.subs({"x": x_i} | c_dict))
        area += f_i*w_i

    return area
```

Exemplo:

[39]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
```

```
display(f)
```

$c_1e^{c_2x} + c_3x^{c_4}$

[40]:
```
#c1, c2, c3, c4
constants = [1, 1, 1, 0]

#Intervalo a-b a ser integrado
a = 0
b = 1

#Número de pontos de integração (entre 2 a 10)
N = 5


area = run_polinomial_quadrature(constants, a, b, N)
print(f"Área: {area}")
```

```
Área: 2.718282687924767
```

0.2.7 2.3.1 - Derivada pela diferença central a partir de um valor inicial a e um delta x

[41]:
```
def deriv_central(constants, x_value, delta_x):
    x = Symbol("x")
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
 ↪constants[3]}
    f = c1*exp(c2*x)+c3*x**c4
    x_mais = x_value + delta_x
    f_mais = float(f.subs({"x": x_mais} | c_dict))
    x_menos = x_value - delta_x
    f_menos = float(f.subs({"x": x_menos} | c_dict))

    f_deriv = (f_mais - f_menos)/(2*delta_x)
    return f_deriv
```

Exemplo:

[42]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
```

```
display(f)
```

$c_1 e^{c_2 x} + c_3 x^{c_4}$

[44]:
```
#c1, c2, c3, c4
constants = [1, 1, 1, 0]

#Ponto a
a = 2

#Valor de delta x
delta_x = 0.02


deriv = deriv_central(constants, a, delta_x)
print(f"Derivada no ponto a: {deriv}")
```

```
Derivada no ponto a: 7.389548712522753
```

0.2.8 2.3.2 - Derivada passo a frente a partir de um valor inicial a e um delta x

[45]:
```
def deriv_frente(constants, x_value, delta_x):
    x = Symbol("x")
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
 ↪constants[3]}
    f = c1*exp(c2*x)+c3*x**c4
    x_mais = x_value + delta_x
    f_mais = float(f.subs({"x": x_mais} | c_dict))
    f_normal = float(f.subs({"x": x_value} | c_dict))

    f_deriv = (f_mais - f_normal)/delta_x
    return f_deriv
```

Exemplo:

[46]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
display(f)
```

$c_1 e^{c_2 x} + c_3 x^{c_4}$

```
[47]: #c1, c2, c3, c4
constants = [1, 1, 1, 0]

#Ponto a
a = 2

#Valor de delta x
delta_x = 0.02


deriv = deriv_frente(constants, a, delta_x)
print(f"Derivada no ponto a: {deriv}")
```

```
Derivada no ponto a: 7.463441736563592
```

0.2.9 2.3.3 - Derivada passo atrás a partir de um valor inicial a e um delta x

```
[48]: def deriv_tras(constants, x_value, delta_x):
    x = Symbol("x")
    c1 = Symbol('c1')
    c2 = Symbol('c2')
    c3 = Symbol('c3')
    c4 = Symbol('c4')
    c_dict = {"c1": constants[0], "c2": constants[1], "c3": constants[2], "c4":␣
 ↪constants[3]}
    f = c1*exp(c2*x)+c3*x**c4
    f_normal = float(f.subs({"x": x_value} | c_dict))
    x_menos = x_value - delta_x
    f_menos = float(f.subs({"x": x_menos} | c_dict))

    f_deriv = (f_normal - f_menos)/delta_x
    return f_deriv
```

Exemplo:

```
[49]: x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
display(f)
```

$c_1e^{c_2x} + c_3x^{c_4}$

```
[50]: #c1, c2, c3, c4
constants = [1, 1, 1, 0]
```

```
#Ponto a
a = 2

#Valor de delta x
delta_x = 0.02


deriv = deriv_tras(constants, a, delta_x)
print(f"Derivada no ponto a: {deriv}")
```

```
Derivada no ponto a: 7.315655688481915
```

0.2.10 2.4 - Derivada por extrapolação de Richard a partir de um ponto a e dois delta x

[51]:
```
def deriv_richard(constants, x_value, delta_x_1, delta_x_2):
    d_1 = deriv_frente(constants, x_value, delta_x_1)
    d_2 = deriv_frente(constants, x_value, delta_x_2)
    q = delta_x_1/delta_x_2

    f_deriv = d_1 + (d_1-d_2)/(np.power(q, -1) - 1)
    return f_deriv
```

Exemplo:

[52]:
```
x = Symbol('x')
c1 = Symbol('c1')
c2 = Symbol('c2')
c3 = Symbol('c3')
c4 = Symbol('c4')
f = c1*exp(c2*x)+c3*x**c4
display(f)
```

$c_1 e^{c_2 x} + c_3 x^{c_4}$

[ ]:
```
#c1, c2, c3, c4
constants = [1, 1, 1, 0]

#Ponto a
a = 2

#Valores de delta x
delta_x_1 = 0.5
delta_x_2 = 0.25


deriv = deriv_richard(constants, a, delta_x_1, delta_x_2)
print(f"Derivada no ponto a: {deriv}")
```

```
Derivada no ponto a: 7.202562175877361
```

0.2.11 3 - Rugen-Kutta-Nystrom para calcular EDO de Segunda Ordem

[57]:
```
import numpy as np
import pandas as pd
from sympy import Symbol, diff, sin


def get_runge_kutta_nystrom(f, delta, y_0, dy_0, maximum_t = 1):
    y_arr = []
    y_arr.append(y_0)

    dy_arr = []
    dy_arr.append(dy_0)

    t_ = 0
    i = 0

    results = []
    results.append([t_, y_0, dy_0])

    while (t_ < maximum_t):
        K_1 = delta/2 * f.subs({"t": t_, "y": y_arr[i], "dy": dy_arr[i]})
        Q = delta/2 * (dy_arr[i] + K_1/2)

        K_2 = delta/2 * f.subs({"t": t_ + delta/2, "y": y_arr[i] + Q, "dy":␣
↪dy_arr[i] + K_1})
        K_3 = delta/2 * f.subs({"t": t_ + delta/2, "y": y_arr[i] + Q, "dy":␣
↪dy_arr[i] + K_2})
        L = delta * (dy_arr[i] + K_3)

        K_4 = delta/2 * f.subs({"t": t_ + delta, "y": y_arr[i] + L, "dy":␣
↪dy_arr[i] + 2*K_3})

        y_new = y_arr[i] + delta * (dy_arr[i] + (K_1 + K_2 + K_3)/3)
        y_arr.append(y_new)

        dy_new = dy_arr[i] + (K_1 + 2*K_2 + 2*K_3 + K_4)/3
        dy_arr.append(dy_new)

        i += 1
        t_ = delta * i

        results.append([t_, y_new, dy_new])
    return results
```

```
def run_runge_kutta_nystrom(delta, maximum_t, c_dict):
    m = Symbol("m")
    c = Symbol("c")
    k = Symbol("k")

    a1 = Symbol("a1")
    a2 = Symbol("a2")
    a3 = Symbol("a3")
    w1 = Symbol("w1")
    w2 = Symbol("w2")
    w3 = Symbol("w3")

    t = Symbol("t")
    y = Symbol("y")
    dy = Symbol("dy")

    f = (a1*sin(w1*t)+a2*sin(w2*t)+a3*sin(w3*t) - c*dy + k*y)/m
    f = f.subs(c_dict)

    y_0 = 0
    dy_0 = 0

    results = get_runge_kutta_nystrom(f, delta, y_0, dy_0, maximum_t)
    results_new = [[a[0], a[1], a[2], f.subs({"t": a[0], "y": a[1], "dy":␣
↪a[2]})] for a in results]
    df = pd.DataFrame(results_new, columns=["tempo", "deslocamento",␣
↪"velocidade", "aceleração"])
    return df
```

Exemplo:

[58]:
```
m = Symbol("m")
c = Symbol("c")
k = Symbol("k")

a1 = Symbol("a1")
a2 = Symbol("a2")
a3 = Symbol("a3")
w1 = Symbol("w1")
w2 = Symbol("w2")
w3 = Symbol("w3")

t = Symbol("t")
y = Symbol("y")
dy = Symbol("dy")

f = (a1*sin(w1*t)+a2*sin(w2*t)+a3*sin(w3*t) - c*dy + k*y)/m
```

```
f
```

[58]:

$$\frac{a_1 \sin(tw_1) + a_2 \sin(tw_2) + a_3 \sin(tw_3) - cdy + ky}{m}$$

[59]:

```
#Definindo as constantes
c_dict = {"a1": 1,
          "a2": 2,
          "a3": 1.5,
          "w1": 0.05,
          "w2": 1,
          "w3": 2,
          "m": 1,
          "c": 0.1,
          "k": 2}
#Passo
delta = 0.05

#t máximo
maximum_t = 1

df = run_runge_kutta_nystrom(delta, maximum_t, c_dict)
df
```

[59]:

```
    tempo          deslocamento          velocidade          aceleração
0    0.00                     0                   0                   0
1    0.05  0.000105046482065132  0.00630097358159336  0.251788456513404
2    0.10  0.000839201013054396   0.0251497931162853  0.501834231367421
3    0.15   0.00282747984799277   0.0564474935240383  0.749666714970487
4    0.20   0.00668937064980470    0.100072081654548  0.994837541521419
5    0.25    0.0130377142299364    0.155879968292860   1.23693333252765
6    0.30    0.0224776736874462    0.223708029299934   1.47558810536646
7    0.35    0.0356058219516773    0.303376277171284   1.71049526873818
8    0.40    0.0530093777017213    0.394691120489022   1.94141913101446
9    0.45    0.0752656184191856    0.497449185166706   2.16820585259594
10   0.50     0.102941497942054    0.611441668088841   2.39079377941019
11   0.55     0.136593494341711    0.736459190771660   2.60922310154495
12   0.60     0.176767712259631    0.872297118059546   2.82364478865671
13   0.65     0.224000262033918     1.01876130465954   3.03432876214774
14   0.70     0.278817936039663     1.17567423053809   3.24167127308793
15   0.75     0.341739200683444     1.34288148488927   3.44620146438617
16   0.80     0.413275520455004     1.52025855755665   3.64858710570228
17   0.85     0.493933028373121     1.70771789647329   3.84963949994280
18   0.90     0.584214555093909     1.90521618989091   4.05031757080865
19   0.95     0.684622026904607     2.11276183291249   4.25173115166361
20   1.00     0.795659240831549     2.33042253912751   4.45514350687534
```