TRABALHO MAF271

Nome: Murillo Santhiago Souza Jacob
Matrícula: 4243

Introdução

Foram desenvolvidos códigos para calcular o resultado de expressões, que foram os seguintes:

1. Método Gauss;
2. Método fatoração LU,
3. Método Fatoração Cholesky;
4. Método de Jacobi;
5. Método Gauss Seidel;
6. Método de Newton;
7. Método de Newton Modificado;
8. Método Lagrange;
9. Método de diferenças Finitas e
10. Diferenças Divididas.

Desenvolvimento

O código foi desenvolvido utilizando classes abstratas, que foram para calcular os resultados de sistemas lineares e não lineares, e são as seguintes:

Lineares:

```python
from sympy import symbols, Eq, linear_eq_to_matrix


class AbstractResolver:
    def _solve(self, A, b):
        raise NotImplementedError

    def _run_tests(self):
        x, y, z, w = symbols('x y z w')
        eq1 = Eq(2*x + y, 5)
        eq2 = Eq(x - 3*y, -1)
        A, b = linear_eq_to_matrix([eq1, eq2], [x, y])
        result = self._solve(A.tolist(), list(b))
        eq1_response = int(eq1.args[0].subs({x:result[0], y:result[1]}))
        eq2_response = int(eq2.args[0].subs({x:result[0], y:result[1]}))
        print(f"Compare: expression: {eq1.args[0]}\t Response_data: {result}\t Result:{eq1_response}\t Expected:{eq1.args[1]}\n")
        print(f"Compare: expression: {eq2.args[0]}\t Response_data: {result}\t Result:{eq2_response}\t Expected:{eq2.args[1]}")
        print('----------------------------------')
        eq1 = Eq(3*x - 2*y + z, 4)
        eq2 = Eq(x + y + z, 2)
        eq3 = Eq(2*x - y + 2*z, 1)
        A, b = linear_eq_to_matrix([eq1, eq2, eq3], [x, y, z])
        result = self._solve(A.tolist(), list(b))
        eq1_response = int(eq1.args[0].subs({x:result[0], y:result[1], z:result[2]}))
        eq2_response = int(eq2.args[0].subs({x:result[0], y:result[1], z:result[2]}))
        eq3_response = int(eq3.args[0].subs({x:result[0], y:result[1], z:result[2]}))
        print(f"Compare: expression: {eq1.args[0]}\t Response_data: {result}\t Result:{eq1_response}\t Expected:{eq1.args[1]}\n")
        print(f"Compare: expression: {eq2.args[0]}\t Response_data: {result}\t Result:{eq2_response}\t Expected:{eq2.args[1]}\n")
        print(f"Compare: expression: {eq3.args[0]}\t Response_data: {result}\t Result:{eq3_response}\t Expected:{eq3.args[1]}")
        print('----------------------------------')
        eq1 = Eq(x + y + z + w, 1)
        eq2 = Eq(2*x - y - 3*z + w, 2)
        eq3 = Eq(3*x + y + 2*z - 2*w, 1)
        eq4 = Eq(x - 2*y + 3*z - 4*w, 0)
        A, b = linear_eq_to_matrix([eq1, eq2, eq3, eq4], [x, y, z, w])
        result = self._solve(A.tolist(), list(b))
        eq1_response = int(eq1.args[0].subs({x:result[0], y:result[1], z:result[2], w:result[3]}))
        eq2_response = int(eq2.args[0].subs({x:result[0], y:result[1], z:result[2], w:result[3]}))
        eq3_response = int(eq3.args[0].subs({x:result[0], y:result[1], z:result[2], w:result[3]}))
        eq4_response = int(eq4.args[0].subs({x:result[0], y: result[1], z:result[2], w:result[3]}))
        print(f"Compare: Expression: {eq1.args[0]}\t Response_data: {result}\t Result:{eq1_response}\t Expected:{eq1.args[1]}\n")
        print(f"Compare: Expression: {eq2.args[0]}\t Response_data: {result}\t Result:{eq2_response}\t Expected:{eq2.args[1]}\n")
        print(f"Compare: Expression: {eq3.args[0]}\t Response_data: {result}\t Result:{eq3_response}\t Expected:{eq3.args[1]}\n")
        print(f"Compare: Expression: {eq4.args[0]}\t Response_data: {result}\t Result:{eq4_response}\t Expected:{eq4.args[1]}")
```

E, para as funções não lineares, têm-se a seguinte classe abstrata:

```python
from sympy import symbols, Eq, Matrix
from sympy import diff, cos, sin
import numpy as np


class AbstractNoLinearSolver():
    def _solver(self):
        raise NotImplementedError

    def _run_tests(self):
        x, y, z = symbols('x y z')
        x_sym = [x, y, z]

        # Equations 1 and 2
        F = [cos(x) + y - z - 1, x**2 + y**2 - 4, x + y + z - 2]
        x0 = [1, 1, 1]  # Initial guess for the solution
        result = self._solve(F, x_sym, x0)
        F_val = [F[i].subs(list(zip(x_sym, result))) for i in range(len(F))]
        print(F_val)
```

Gauss:

```python
class GaussMethod(AbstractResolver):
    def _solve(A, b):
        n = len(A)
        # Etapa de eliminação
        for i in range(n-1):
            # Verifica se o pivô é zero e realiza um pivoteamento parcial
            if A[i][i] == 0:
                for j in range(i+1, n):
                    if A[j][i] != 0:
                        A[i], A[j] = A[j], A[i]
                        b[i], b[j] = b[j], b[i]
                        break

            for j in range(i+1, n):
                factor = A[j][i] / A[i][i]
                for k in range(i, n):
                    A[j][k] -= factor * A[i][k]
                b[j] -= factor * b[i]

        x = [0] * n
        x[n-1] = b[n-1] / A[n-1][n-1]

        for i in range(n-2, -1, -1):
            sum = b[i]
            for j in range(i+1, n):
                sum -= A[i][j] * x[j]
            x[i] = sum / A[i][i]

        return x
```

```
murillossj@murillo-ssj:~/Documentos/ufv/tp02-maf271$ python3 main.py
Compare: expression: 2*x + y     Response_data: [2, 1]    Result:5        Expected:5

Compare: expression: x - 3*y     Response_data: [2, 1]    Result:-1       Expected:-1
-----------------------------------
Compare: expression: 3*x - 2*y + z       Response_data: [5/2, 1, -3/2]   Result:4        Expected:4

Compare: expression: x + y + z   Response_data: [5/2, 1, -3/2]   Result:2        Expected:2

Compare: expression: 2*x - y + 2*z       Response_data: [5/2, 1, -3/2]   Result:1        Expected:1
-----------------------------------
Compare: Expression: w + x + y + z       Response_data: [47/63, -10/21, 11/63, 5/9]      Result:1        Expected:1

Compare: Expression: w + 2*x - y - 3*z   Response_data: [47/63, -10/21, 11/63, 5/9]      Result:2        Expected:2

Compare: Expression: -2*w + 3*x + y + 2*z        Response_data: [47/63, -10/21, 11/63, 5/9]      Result:1        Expe
cted:1

Compare: Expression: -4*w + x - 2*y + 3*z        Response_data: [47/63, -10/21, 11/63, 5/9]      Result:0        Expe
cted:0
```

Método fatoração LU

```python
from scipy.linalg import lu_factor, lu_solve
from methods.abstract_resolver import AbstractResolver

class LuDecompose(AbstractResolver):
    def _solve(self, A, b):
        LU, piv = lu_factor(A)
        x = lu_solve((LU, piv), b)
        return x
```

```
murillossj@murillo-ssj:~/Documentos/ufv/tp02-maf271$ python3 main.py
Compare: expression: 2*x + y     Response_data: [2. 1.]  Result:5        Expected:5

Compare: expression: x - 3*y     Response_data: [2. 1.]  Result:-1       Expected:-1
----------------------------------
Compare: expression: 3*x - 2*y + z       Response_data: [ 2.5  1.  -1.5]         Result:4        Expected:4

Compare: expression: x + y + z   Response_data: [ 2.5  1.  -1.5]         Result:2        Expected:2

Compare: expression: 2*x - y + 2*z       Response_data: [ 2.5  1.  -1.5]         Result:1        Expected:1
----------------------------------
Compare: Expression: w + x + y + z       Response_data: [ 0.74603175 -0.47619048  0.17460317  0.55555556]        Resu
lt:1     Expected:1

Compare: Expression: w + 2*x - y - 3*z   Response_data: [ 0.74603175 -0.47619048  0.17460317  0.55555556]        Resu
lt:2     Expected:2

Compare: Expression: -2*w + 3*x + y + 2*z        Response_data: [ 0.74603175 -0.47619048  0.17460317  0.55555556]
Result:1         Expected:1

Compare: Expression: -4*w + x - 2*y + 3*z        Response_data: [ 0.74603175 -0.47619048  0.17460317  0.55555556]
Result:0         Expected:0
```

Método fatoração Cholesky

```python
from methods.abstract_resolver import AbstractResolver
from scipy.linalg import cholesky, solve_triangular
from numpy.linalg import LinAlgError


class CholeskyDecomposer(AbstractResolver):
    def _solve(self, A, b):
        try:
            L = cholesky(A)
        except LinAlgError:
            print('Matrix is not positive definite')
            return [0 for _ in range(len(A))]
        y = solve_triangular(L, b, lower=True)
        x = solve_triangular(L.T, y)
        return x
```

```
Matrix is not positive definite
Compare: expression: 2*x + y     Response_data: [0, 0]    Result:0       Expected:5

Compare: expression: x - 3*y     Response_data: [0, 0]    Result:0       Expected:-1
----------------------------------
Matrix is not positive definite
Compare: expression: 3*x - 2*y + z       Response_data: [0, 0, 0]        Result:0        Expected:4

Compare: expression: x + y + z   Response_data: [0, 0, 0]        Result:0        Expected:2

Compare: expression: 2*x - y + 2*z       Response_data: [0, 0, 0]        Result:0        Expected:1
----------------------------------
Matrix is not positive definite
Compare: Expression: w + x + y + z       Response_data: [0, 0, 0, 0]     Result:0        Expected:1

Compare: Expression: w + 2*x - y - 3*z   Response_data: [0, 0, 0, 0]     Result:0        Expected:2

Compare: Expression: -2*w + 3*x + y + 2*z        Response_data: [0, 0, 0, 0]     Result:0        Expected:1

Compare: Expression: -4*w + x - 2*y + 3*z        Response_data: [0, 0, 0, 0]     Result:0        Expected:0
```

OBS: Neste método, nenhuma matriz gerada atendeu ao pré-requisito do método, por isso, caiu nas exceções e o resultado não foi condizente.

Método de Jacobi

```python
from methods.abstract_resolver import AbstractResolver
import numpy as np

class Jacobi(AbstractResolver):
    def _solve(self, A, b, tolerance=1e-6):
        n = len(A)
        x = np.zeros(n)

        while True:
            x_new = np.zeros(n)

            for i in range(n):
                sum_term = 0
                for j in range(n):
                    if j != i:
                        sum_term += A[i][j] * x[j]

                x_new[i] = (b[i] - sum_term) / A[i][i]
            x = x_new

            if np.linalg.norm(x_new - x) < tolerance:
                break

        return x
```

```
murillossj@murillo-ssj:~/Documentos/ufv/tp02-maf271$ python3 main.py
Compare: expression: 2*x + y     Response_data: [1.9999997 1.0000002]    Result:4        Expected:5

Compare: expression: x - 3*y     Response_data: [1.9999997 1.0000002]    Result:-1       Expected:-1
----------------------------------
Compare: expression: 3*x - 2*y + z       Response_data: [-203050.74545594  216816.92543986 -147260.21592553]     Resu
lt:-1190046       Expected:4

Compare: expression: x + y + z   Response_data: [-203050.74545594  216816.92543986 -147260.21592553]     Result:-1334
94       Expected:2

Compare: expression: 2*x - y + 2*z       Response_data: [-203050.74545594  216816.92543986 -147260.21592553]     Resu
lt:-917438       Expected:1
----------------------------------
Compare: Expression: w + x + y + z       Response_data: [ 9.79371954e+28  1.01798263e+29  1.19792543e+29 -3.28579352e
+28]     Result:286670066239218995558208241664   Expected:1

Compare: Expression: w + 2*x - y - 3*z   Response_data: [ 9.79371954e+28  1.01798263e+29  1.19792543e+29 -3.28579352e
+28]     Result:-298159437147625197244273131520  Expected:2

Compare: Expression: -2*w + 3*x + y + 2*z        Response_data: [ 9.79371954e+28  1.01798263e+29  1.19792543e+29 -3.2
8579352e+28]     Result:700910806081241497766589366272   Expected:1

Compare: Expression: -4*w + x - 2*y + 3*z        Response_data: [ 9.79371954e+28  1.01798263e+29  1.19792543e+29 -3.2
8579352e+28]     Result:385150041470503856309218050048   Expected:0
```

Obs: É importante lembrar que o jacobi é um método iterativo, por isso, vemos alguns valores que não convergem.

Método Gauss Seidel

```python
from methods.abstract_resolver import AbstractResolver
import numpy as np

class GaussSeidSolver(AbstractResolver):
    def _solve(self, A, b, tolerance=1e-6):
        n = len(A)
        x = np.zeros(n)

        while(True):
            x_new = np.zeros(n)

            for i in range(n):
                sum_term = 0
                for j in range(n):
                    if j != i:
                        sum_term += A[i][j] * x_new[j]

                x_new[i] = (b[i] - sum_term) / A[i][i]

            if np.linalg.norm(x_new - x) < tolerance:
                break

            x = x_new

        return x
```

```
[Running] python -u "/home/murillossj/Documentos/ufv/tp02-maf271/main.py"
Compare: expression: 2*x + y     Response_data: [2.5        1.16666667]  Result:6    Expected:5

Compare: expression: x - 3*y     Response_data: [2.5        1.16666667]  Result:-1   Expected:-1
---------------------------------
Compare: expression: 3*x - 2*y + z   Response_data: [ 1.33333333  0.66666667 -0.5       ]   Result:2    Expected:4

Compare: expression: x + y + z   Response_data: [ 1.33333333  0.66666667 -0.5       ]   Result:1    Expected:2

Compare: expression: 2*x - y + 2*z   Response_data: [ 1.33333333  0.66666667 -0.5       ]   Result:1    Expected:1
---------------------------------
Compare: Expression: w + x + y + z   Response_data: [ 1.   0.  -1.  -0.5]   Result:0    Expected:1

Compare: Expression: w + 2*x - y - 3*z   Response_data: [ 1.   0.  -1.  -0.5]   Result:4    Expected:2

Compare: Expression: -2*w + 3*x + y + 2*z    Response_data: [ 1.   0.  -1.  -0.5]   Result:2    Expected:1

Compare: Expression: -4*w + x - 2*y + 3*z    Response_data: [ 1.   0.  -1.  -0.5]   Result:0    Expected:0
```

Método de Newton

Nesta parte, já passamos para funções não lineares, e é importante ressaltar que, os resultados, quanto mais perto de 0, significa que chegou mais próximo da resposta:

```
import numpy as np
from sympy import Matrix
from sympy import diff


class NewtonSolver(AbstractNoLinearSolver):
    def _solve(self, F, x, x0, max_iterations=100, tolerance=1e-6):
        n = len(x)
        J = Matrix([[diff(F[i], x[j]) for j in range(n)] for i in range(n)])

        x_vals = np.array(x0, dtype=float)

        for _ in range(max_iterations):
            F_val = np.array([F[i].subs(list(zip(x, x_vals))) for i in range(n)], dtype=float)
            J_val = np.array([[J[i, j].subs(list(zip(x, x_vals))) for j in range(n)] for i in range(n)], dtype=float)

            delta_x = np.linalg.solve(J_val, -F_val)
            x_vals = x_vals + delta_x

            if np.linalg.norm(delta_x) < tolerance:
                break

        return x_vals
```

```
[Running] python -u "/home/murillossj/Documentos/ufv/tp02-maf271/main.py"
[ 1.86881725  0.71240585 -0.5812231 ]
[3.99680288865056e-15, 2.79776202205539e-14, 0]
```

Método de Newton Modificado

```python
from methods.abstract_no_linear import AbstractNoLinearSolver
import numpy as np
from sympy import diff

class NewtonModifiedSolver(AbstractNoLinearSolver):
    def _solve(self, F, x, x0, max_iterations=100, tolerance=1e-6):
        n = len(x)
        J = np.zeros((n, n), dtype=float)

        x_vals = np.array(x0, dtype=float)

        for _ in range(max_iterations):
            F_val = np.array([F[i].subs(list(zip(x, x_vals))) for i in range(n)], dtype=float)

            for i in range(n):
                for j in range(n):
                    J[i, j] = diff(F[i], x[j]).subs(list(zip(x, x_vals)))

            delta_x = np.linalg.solve(J, -F_val)
            x_vals = x_vals + delta_x

            if np.linalg.norm(delta_x) < tolerance:
                break

        return x_vals
```

```
[Running] python -u "/home/murillossj/Documentos/ufv/tp02-maf271/main.py"
[ 1.86881725  0.71240585 -0.5812231 ]
[3.99680288865056e-15, 2.79776202205539e-14, 0]
```

Agora, entra-se nos métodos de interpolação, que seguem a seguinte classe abstrata e para testes:

```python
class AbstractInterpolate():
    def _run_tests(self):
        # Define os pontos conhecidos
        x_known = [1, 2, 3, 4]
        y_known = [2, 1, 3, 2]

        # Interpolação para encontrar o valor em x = 2.5
        x_interp = 2.5
        y_interp = self._solve(x_known, y_known, x_interp)

        print(f"Interpolação para x = {x_interp}: y = {y_interp}")

    def _solve(self, x_known, y_known, x_interp):
        raise NotImplementedError
```

Método Lagrange

```python
from methods.interpolate_abstract import AbstractInterpolate


class Lagrange(AbstractInterpolate):
    def _solve(self, x_known, y_known, x_interp):
        n = len(x_known)
        y_interp = 0.0

        for i in range(n):
            term = y_known[i]
            for j in range(n):
                if j != i:
                    term *= (x_interp - x_known[j]) / (x_known[i] - x_known[j])
            y_interp += term

        return y_interp
```

```
[Running] python -u "/home/murillossj/Documentos/ufv/tp02-maf271/main.py"
Interpolação para x = 2.5: y = 2.0
```

Método das diferenças finitas

```python
from methods.interpolate_abstract import AbstractInterpolate
import numpy as np


class DiffFinites(AbstractInterpolate):
    def _solve(self, x_known, y_known, x_interp):
        n = len(x_known)

        # Calcula as diferenças finitas
        diff_y = np.diff(y_known)
        diff_x = np.diff(x_known)

        # Encontra o intervalo no qual o x_interp está
        interval = None
        for i in range(n-1):
            if x_known[i] <= x_interp <= x_known[i+1]:
                interval = i
                break

        # Realiza a interpolação usando diferenças finitas
        y_interp = y_known[interval] + (x_interp - x_known[interval]) * (diff_y[interval] / diff_x[interval])

        return y_interp
```

```
[Running] python -u "/home/murillossj/Documentos/ufv/tp02-maf271/main.py"
Interpolação para x = 2.5: y = 2.0
```

Diferenças divididas

```python
from methods.interpolate_abstract import AbstractInterpolate
import numpy as np


class DiffDivision(AbstractInterpolate):
    def _solve(self, x_known, y_known, x_interp):
        n = len(x_known)
        # Calcula as diferenças divididas
        divided_diff = np.zeros((n, n))
        divided_diff[:, 0] = y_known

        for j in range(1, n):
            for i in range(n-j):
                divided_diff[i, j] = (divided_diff[i+1, j-1] - divided_diff[i, j-1]) / (x_known[i+j] - x_known[i])

        # Realiza a interpolação usando diferenças divididas
        y_interp = divided_diff[0, 0]
        prod = 1
        for j in range(1, n):
            prod *= (x_interp - x_known[j-1])
            y_interp += divided_diff[0, j] * prod

        return y_interp
```

```
[Running] python -u "/home/murillossj/Documentos/ufv/tp02-maf271/main.py"
Interpolação para x = 2.5: y = 2.0
```

OBS: OS CASOS TESTADOS, ESTÃO NAS FUNÇÕES ABSTRATAS, DEVIDAMENTE CITADAS, SENDO QUE OS MÉTODOS DE INTERPOLAÇÃO TESTAM COM A CLASSE ABSTRATA DE INTERPOLAÇÃO, OS DE FUNÇÕES LINEARES COM A ABSTRAÇÃO DE FUNÇÕES LINEARES, E ASSIM SUCESSIVAMENTE.