Laporan Project Metode Numerik

Disusun oleh:

- C14210004 - Andreas Pandu P
- C14210176 - Joy Immanuel K
- C14210007 - Steven Hariyadi
- C12410034 - Kean Siladitya

UNIVERSITAS KRISTEN PETRA

FAKULTAS TEKNOLOGI INDUSTRI

JURUSAN INFORMATIKA

2023/2024

DAFTAR ISI

Soal No. 1

- ### Penjelasan Program

a. Graphical Method

Metode Graphical ,menampilkan grafik fungsi f(x) = x^3 - 6x^2 + 11x - 6.1 pada rentang nilai x dari -10 hingga 10.

Pertama, dari NumPy membuat array x yang berisi 400 titik secara dengan rentang -10 hingga 10. Lalu, kita dihitung nilai f(x) untuk setiap titik dalam array x.

Setelah itu, kita gunakan matplotlib.pyplot untuk membuat plot grafik dengan menggunakan x sebagai sumbu x dan f(x) sebagai sumbu y.

Grafik yang dihasilkan akan menunjukkan bagaimana fungsi f(x) dihasilkan dalam rentang yang ditentukan. Jika terdapat potongan grafik yang memotong sumbu x (sumbu horizontal), maka titik potongan tersebut dapat dianggap sebagai akar persamaan.

b. Bisection Method

Fungsi **bisection_method(a, b)** dalam program tersebut melakukan implementasi metode bisection untuk mencari akar persamaan dalam interval [a, b].

Metode ini bahwa fungsi f(x) adalah kontinu dan memiliki tanda berbeda di ujung-ujung interval [a, b]. Jika ini terpenuhi, metode bisection dapat menemukan akar persamaan dengan melakukan iterasi dan membagi interval menjadi dua bagian secara berulang hingga akurasi yang diinginkan tercapai.

Untuk langkah-langnya yang dilakukan:

1. Deklarasi variabel **max_iter** dengan jumlah maksimum iterasi yang diizinkan.
2. Deklarasi variabel **tolerance** dengan tingkat toleransi yang diinginkan.
3. Lalu cek apakah fungsi f(a) dan f(b) memiliki tanda yang berbeda. Jika tidak, metode bisection tidak dapat diterapkan pada interval ini dan mereturnkan nilai None.
4. Lalu mulai iterasi dengan menginisialisasi variabel iterasi sebagai 1.
5. Selama **iterasi** kurang dari atau sama dengan max_iter, lakukan langkah-langkah berikut:
   a. Hitung titik tengah c dari interval [a, b] sebagai c = (a + b) / 2.
   b. Jika nilai absolut dari f(c) < dari tolerance, maka akar persamaan ditemukan dan mengembalikan nilai c.

c. Jika f(c) dan f(a) memiliki tanda yang berbeda, maka akar persamaan berada di interval [a, c]. Maka, mengupdate nilai b dengan c.
d. Jika f(c) dan f(a) memiliki tanda yang sama, maka akar persamaan berada di interval [c, b]. Maka, mengupdate nilai a dengan c.
e. Meningkatkan nilai **iterasi** dengan 1.

6. Jika iterasi mencapai **max_iter** tanpa mencapai akar dengan tingkat toleransi yang diinginkan, mencetak pesan bahwa metode bisection tidak konvergen setelah jumlah maksimum iterasi dan mengembalikan nilai **None**.

Fungsi **bisection_method** mengembalikan nilai akar persamaan jika ditemukan atau **None** jika metode tidak konvergen pada interval yang diberikan.

c. False Position

Fungsi false_position_method(a, b) dalam implementasikan metode false position (metode regula falsi) untuk mencari akar persamaan dalam interval [a, b].

Metode false position ini mengasumsikan bahwa fungsi f(x) adalah kontinu dan memiliki tanda yang berbeda di akhir interval [a, b]. Metode ini mencari akar persamaan dengan menggantikan garis lurus yang menghubungkan f(a) dan f(b) dengan garis yang menghubungkan f(a) dan f(b) pada titik x yang akan dihitung. Dalam setiap iterasi, metode ini memperbarui interval [a, b] berdasarkan perpotongan garis tersebut dengan sumbu x.

Langkah-langkah yang dapat dilakukan ke dalam fungsi false_position_method adalah:

1. Deklarasikan variabel max_iter dengan jumlah maksimum iterasi yang diizinkan.
2. Deklarasikan variabel tolerance dengan tingkat toleransi yang diinginkan.
3. Memeriksa apakah fungsi f(a) dan f(b) memiliki tanda yang berbeda. Jika tidak, metode false position tidak dapat diterapkan pada interval ini dan mengembalikan nilai None.
4. Memulai iterasi dengan menginisialisasi variabel iterasi sebagai 1
5. Selama iterasi kurang dari atau sama dengan max_iter, dapat dilakukan dengan cara berikut:
    a. Menghitung titik x yang merupakan titik perpotongan garis yang menghubungkan f(a) dan f(b) dengan sumbu x. Dengan rumus: x = (a * f(b) - b * f(a)) / (f(b) - f(a)).

b. Jika nilai absolut dari f(x) lebih kecil dari tolerance, maka akar persamaan ditemukan dan mengembalikan nilai x.
c. Jika f(x) dan f(a) memiliki tanda yang berbeda, maka akar persamaan berada di interval [a, x]. Maka, mengupdate nilai b dengan x.
d. Jika f(x) dan f(a) memiliki tanda yang sama, maka akar persamaan berada di interval [x, b]. Maka, mengupdate nilai a dengan x.
e. Meningkatkan nilai iterasi dengan 1.
f. Jika iterasi mencapai max_iter tanpa mencapai akar dengan tingkat toleransi yang diinginkan, mencetak pesan bahwa metode false position tidak konvergen setelah jumlah maksimum iterasi dan mengembalikan nilai None.

Fungsi false_position_method mengembalikan nilai akar persamaan jika ditemukan atau None jika metode tidak konvergen pada interval yang diberikan.

d. Simple Fixed-Point Iteration

Fungsi fixed_point_iteration_method ini menerima argumen seperti g (fungsi iterasi) dan x0 (tebakan awal).

Variabel max_iter dan tolerance digunakan untuk mengatur jumlah maksimal iterasi dan toleransi konvergensi.

Variabel iterasi diinisialisasi dengan 1 untuk melacak jumlah iterasi yang telah dilakukan.

Dalam loop while dengan kondisi iterasi <= max_iter, nilai x1 dihitung dengan menggunakan fungsi iterasi: x1 = g(x0).

Jika selisih absolut antara x1 dan x0 kurang dari toleransi tolerance, maka ditemukan akar yang memenuhi toleransi. Nilai x1 dikembalikan sebagai hasil akar yang ditemukan.

Jika tidak tercapai konvergensi setelah jumlah maksimal iterasi max_iter, maka pesan kesalahan akan dicetak dan fungsi mengembalikan None.

e. Newton-Raphson

Fungsi newton_raphson_method ini menerima argumen seperti f (fungsi persamaan), f_prime (turunan fungsi persamaan), dan x0 (tebakan awal).

Variabel max_iter dan tolerance digunakan untuk mengatur jumlah maksimal iterasi dan toleransi konvergensi.

Variabel iterasi diinisialisasi dengan 1 untuk melacak jumlah iterasi yang telah dilakukan.

Dalam loop while dengan kondisi iterasi <= max_iter, nilai x1 dihitung dengan menggunakan rumus metode Newton-Raphson: x1 = x0 - f(x0) / f_prime(x0),

Jika selisih absolut antara x1 dan x0 kurang dari toleransi tolerance, maka ditemukan akar yang memenuhi toleransi. Nilai x1 dikembalikan sebagai hasil akar yang ditemukan.

Jika tidak tercapai konvergensi setelah jumlah maksimal iterasi max_iter, maka pesan kesalahan akan dicetak dan fungsi mengembalikan None.

f. **Modified Secant Methods**

Fungsi modified_secant_method menerima argumen seperti f (fungsi persamaan), x0 (tebakan awal), dan p (perubahan delta).

Variabel max_iter dan tolerance digunakan untuk mengatur jumlah maksimal iterasi dan toleransi konvergensi.

Variabel iterasi diinisialisasi dengan 1 untuk melacak jumlah iterasi yang telah dilakukan.

Dalam loop while dengan kondisi iterasi <= max_iter, nilai x1 dihitung dengan menggunakan rumus metode Modified Secant: x1 = x0 - p * f(x0) / (f(x0 + p * x0) - f(x0)).

Jika selisih absolut antara x1 dan x0 kurang dari toleransi tolerance, maka ditemukan akar yang memenuhi toleransi. Nilai x1 dikembalikan sebagai hasil akar yang ditemukan.

Jika tidak tercapai konvergensi setelah jumlah maksimal iterasi max_iter, maka pesan kesalahan akan dicetak dan fungsi mengembalikan None.

- **Listing program beserta penjelasan isi programnya**

a. Graphical Method

```python
# Metode Graphical
def graphical_method():
    # Buat rentang arraynya
    x = np.linspace(-10, 10, 400)
    y = f(x) # lalu masukin hasil array yang ada

    # mulai gambar grafiknya
    plt.plot(x, y)
    plt.axhline(0, color='black', linewidth=0.5)
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title('Graphical Method')
    plt.grid(True)
    plt.show()
```

b. Bisection Method

```python
# Metode Bisection
def bisection_method(a, b):
    max_iter = 100
    tolerance = 0.0005

    # cek apakah interval dapat diaplikasikan
    if f(a) * f(b) >= 0:
        print("Metode bisection tidak dapat diaplikasikan
pada interval ini.")
        return None

    # mulai melakukan iterasi
    iterasi = 1
    while iterasi <= max_iter:
        c = (a + b) / 2

        if abs(f(c)) < tolerance:
            return c

        if f(c) * f(a) < 0:
```

```
            b = c
        else:
            a = c

        iterasi += 1

    print("Metode bisection tidak konvergen setelah",
max_iter, "iterasi.")
    return None
```

c. False Position

```
# Metode False Position
def false_position_method(a, b):
    max_iter = 100
    tolerance = 0.0005

    # Cek false positionnya
    if f(a) * f(b) >= 0:
        print("Metode false position tidak dapat
diaplikasikan pada interval ini.")
        return None

    # buat iterasi selama iterasi <= max_iter
    iterasi = 1
    while iterasi <= max_iter:
        # cek dengan rumus
        c = (a * f(b) - b * f(a)) / (f(b) - f(a))

        if abs(f(c)) < tolerance:
            return c

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

```
        iterasi += 1

    print("Metode false position tidak konvergen setelah",
max_iter, "iterasi.")
    return None
```

d. Simple Fixed-Point Iteration

```
# Metode Simple Fixed-Point Iteration
def fixed_point_iteration_method(g, x0):
    max_iter = 100
    tolerance = 0.0005

    iterasi = 1
    while iterasi <= max_iter:
        x1 = g(x0)
        if abs(x1 - x0) < tolerance:
            return x1
        x0 = x1
        iterasi += 1

    print("Metode simple fixed-point iteration tidak konvergen setelah", max_iter, "iterasi.")
    return None

    # Metode Simple Fixed-Point Iteration
    g = lambda x: (x**3 - 6*x**2 + 6.1) / 11
    root_fixed_point_iteration = fixed_point_iteration_method(g, initial_guess)
    print("Metode Simple Fixed-Point Iteration: x =", root_fixed_point_iteration)
```

e. Newton-Raphson

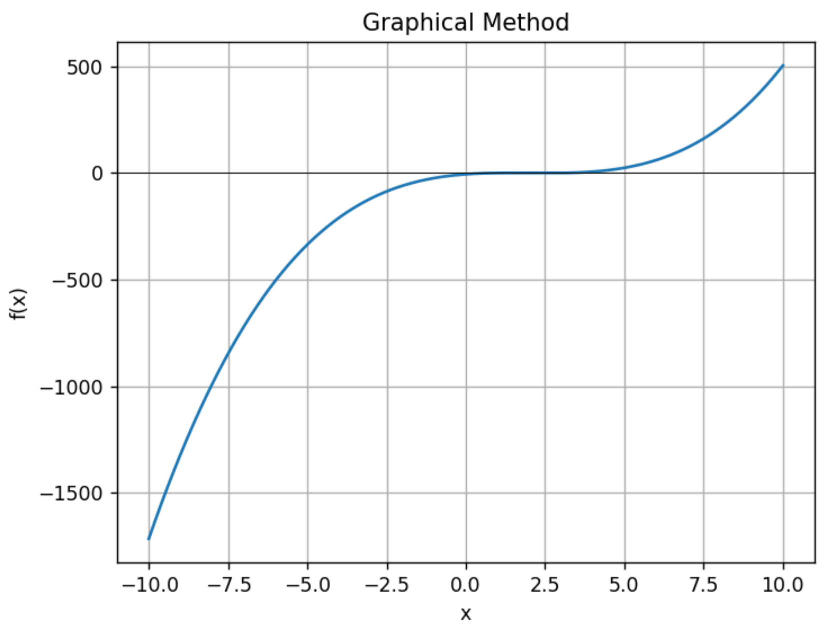
```python
# Metode Newton-Raphson
def newton_raphson_method(f, f_prime, x0):
    max_iter = 100
    tolerance = 0.0005

    iterasi = 1
    while iterasi <= max_iter:
        x1 = x0 - f(x0) / f_prime(x0)
        if abs(x1 - x0) < tolerance:
            return x1
        x0 = x1
        iterasi += 1

    print("Metode Newton-Raphson tidak konvergen setelah", max_iter, "iterasi.")
    return None

    # Metode Newton-Raphson
    f_prime = lambda x: 3*x**2 - 12*x + 11
    root_newton_raphson = newton_raphson_method(f, f_prime, initial_guess)
    print("Metode Newton-Raphson: x =", root_newton_raphson)
```

f. Modified Secant Methods

```python
# Metode Modified Secant
def modified_secant_method(f, x0, p):
    max_iter = 100
    tolerance = 0.0005

    iterasi = 1
    while iterasi <= max_iter:
        x1 = x0 - p * f(x0) / (f(x0 + p * x0) - f(x0))
        if abs(x1 - x0) < tolerance:
            return x1
        x0 = x1
        iterasi += 1

    print("Metode modified secant tidak konvergen setelah", max_iter, "iterasi.")
    return None

    # Metode Modified Secant
    p = 0.01
    root_modified_secant = modified_secant_method(f, initial_guess, p)
    print("Metode Modified Secant: x =", root_modified_secant)
```

- **Input dan hasil output program yang dibuat**

  a. **Graphical Method**



  b. **Bisection Method**

  Metode bisection tidak dapat diaplikasikan pada interval ini.
  Metode Bisection: x = None

  c. **False Position**

  Metode false position tidak dapat diaplikasikan pada interval ini.
  Metode False Position: x = None

  d. **Simple Fixed-Point Iteration**

  Metode Simple Fixed-Point Iteration: x = 0.45178701525691006

  e. **Newton-Raphson**

  Metode Newton-Raphson: x = 1.0543507260588052

  f. **Modified Secant Methods**

Metode Modified Secant: x = 1.8986534046464376

- **Penjelasan atas hasil yang didapatkan dari semua solusi**

  Dari hasil yang didapat metode yang paling cepat prosesnya adalah Newton-Raphson, dikarenakan metode ini memiliki konvergensi cepat untuk persamaan non linear, jika initial guess yang baik diberikan.

Soal No. 2

1. Naive Gauss Elimination
   a. Nama Program: NaiveGaussElimination.py
   b. Penjelasan Program

      Pada program ini, program akan menerima inputan berupa soal yang diubah dalam bentuk matrix lalu mengolah matrix tersebut menggunakan metode NaiveGaussElimination untuk mencari nilai X1, X2, dan X3.

```
#MATRIX NORMAL
matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]
```

      Program ini dibagi menjadi 2 bagian proses yaitu Proses 1 adalah forward elimination dan proses 2 adalah back substitution.

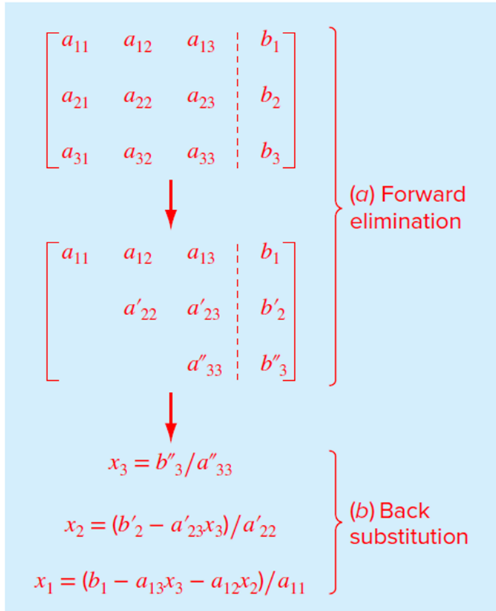


FIGURE 9.3
The two phases of Gauss elimination: (*a*) forward elimination and (*b*) back substitution.

Kita mulai dari proses 1 : Forward Elimination. Pada proses 1.1 kita akan mencari pivot yang akan digunakan untuk me-nolkan lower triangle. Pivot yang digunakan adalah bagian diagonal dari matrix. Setelah itu pada proses 1.2 mencari faktor yang akan digunakan dalam proses menolkan lower triangle pada matrix. Setelah mendapat faktor, pada proses 1.3 kita tinggal menolkan lower triangle dengan dengna mengurangi lower triangle dengan hasil perkalian dari faktor * lower triangle.

Lalu brikutnya ada proses 2 : Back Subtitution. Pada proses ini cukup singkat karena hanya perlu mencari nilai x1,x2,x3 dengan cara mengkalikan nilai hasil row pada matrix A dengan nilai di row yang sama pada matrix B. Setelah itu, nilai x1,x2,x3 akan disimpan ke dalam matrix X lalu di return menjadi nilai.

c. Coding

```
def gaussian_elimination(A, b):
    n = len(A)
    # Proses 1: forward elimination
    for i in range(n):
        # Proses 1.1: Menentukan pivot yang digunakan untuk
me-nolkan lower triangle
```

```
        pivot = A[i][i]
        for j in range(i + 1, n):
            # Proses 1.2: Mencari faktor pembagi.
            factor = A[j][i] / pivot
            for k in range(i, n):
                # Proses 1.3: Melakukan pengurangan dari hasil
kali faktor
                A[j][k] = A[j][k] - factor * A[i][k]
            b[j] = b[j] - factor * b[i]
    # Proses 2: back substitution
    x = [0] * n
    for i in range(n - 1, -1, -1):
        # Porses 2.1 : Mencari nilai x1,x2,x3 dengan membagi
semua nilai pada row dengan nilai pada matrix B di row yang
sama
        x[i] = b[i] / A[i][i]
        for j in range(i - 1, -1, -1):
            b[j] = b[j] - A[j][i] * x[i]
    # Proses 2.2 : Mereturn nilai x1,x2,x3 yang di simpan
dalam array x
    return x

#MATRIX NORMAL
matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]


y = gaussian_elimination(matrix,matrixB)
print("Output: ",y)
```

d. Input dan Output

Input :

```
matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]
```

Output:

```
Output:  [0.5, 8.0, -6.0]
```

2. LU Decomposition
    a. Nama Program: LUFactorization.py
    b. Penjelasan Program

Pada program ini akan menerima input berupa matrix. Yang nanti akan memproses menggunakan metode LU Factorization atau LU Decomposition untuk mencari nilai dari X1, X2, dan X3.

Seperti Naive Gauss Elimination, LU Decomposition juga dibagi menjadi dua proses yaitu proses Factorization dan proses Substitution. Lalu di dalam Substitution dibagi menjadi dua proses yaitu forward dan back.

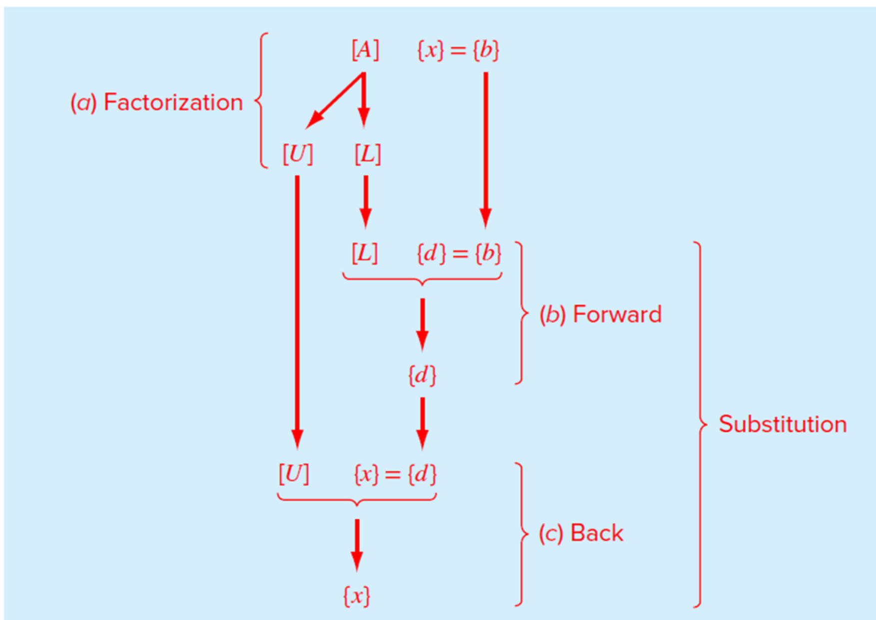


FIGURE 10.1
The steps in *LU* factorization.

Pada progam sudah dipisahkan dua function yaitu function lu_factorization dan lu_substitution. Kita mulai dari penjelasan function lu_factorization. Pada function lu_factorization, kita akan menerima inputan berupa matrix sisi kiri atau dalam kasus ini adalah matrix A dan mencari lower dan upper (mencari L dan U) dari matrix A. Pada proses 1.1 kita akan mencari Upper triangle dengan cara yang sama yang kita lakukan pada Naive Gauss Elimination. Sedangkan pada proses 1.2 kita akan mencatat semua faktor pada proses 1.1 dan digunakan untuk membentuk Lower Triangular Matrix.

Setelah mendapatkan Lower Triangular dan Upper Triangular dari matrix, kita kemudian masuk ke function yang kedua yaitu lu_substitution atau masuk ke proses substitution. Pada proses 2.1 kita akan mencari d menggunakan Lower Triangular matrix. Setelah mendapatkan matrix d, kita kemudian pindah ke proses 2.2 dimana kita akan menggunakan matrix d tadi untuk mendapatkan nilai X1,X2, dan X3. Prosesnya kira - kira sama untuk part substitution di Naive Gauss Elimination.

c. Coding

```
import numpy as np
def lu_factorization(matrix):
    n = len(matrix)
    lower = [[0.0] * n for _ in range(n)]
    upper = [[0.0] * n for _ in range(n)]

    for i in range(n):
        # Upper Triangular
        for k in range(i, n):
            sum_uk = sum(lower[i][j] * upper[j][k] for j in
range(i))
            upper[i][k] = matrix[i][k] - sum_uk

        # Lower Triangular
        for k in range(i, n):
            if i == k:
                lower[i][i] = 1.0
            else:
                sum_lk = sum(lower[k][j] * upper[j][i] for j
in range(i))
                lower[k][i] = (matrix[k][i] - sum_lk) /
upper[i][i]

    return lower, upper


def lu_substitution(lower, upper, b):
    n = len(lower)
    y = [0.0] * n
    x = [0.0] * n
```

```
    # Forward Substitution: Ly = b
    for i in range(n):
        sum_ly = sum(lower[i][j] * y[j] for j in range(i))
        y[i] = (b[i] - sum_ly) / lower[i][i]

    # Backward Substitution: Ux = y
    for i in range(n - 1, -1, -1):
        sum_ux = sum(upper[i][j] * x[j] for j in range(i + 1,
n))
        x[i] = (y[i] - sum_ux) / upper[i][i]

    return x

def LUSolve(A,B):
    lower,upper = lu_factorization(A)
    x = lu_substitution(lower,upper,B)
    return x


matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]

x = LUSolve(matrix,matrixB)

print(x)
```

d. Input and Output

Input:
```
matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]
```

Output :
```
[0.5, 8.0, -6.0]
```

3. Cholesky
   a. Nama Program : NaiveGaussElimination.py
   b. Penjelasan Program

      Untuk program ini, sebenarnya tidak bisa digunakan untuk menghitung soal dikarenakan matrix yang digunakan tidak simetris. Tapi secara keseluruhan, input yang diterima juga sama yaitu berupa matrix dari soal. Setelah itu, matrix akan diproses menggunakan dua function yaitu function cholesky_factorization dan function solve_linear_equation

Kita mulai dari function pertama yaitu cholesky_factorization. Pada function ini kita akan mencari upper triangular matrix. Cara carinya sama seperti LU Factorization hanya saja kita tidak mencari Lowernya dan hanya mencari Upper Triangularnya saja.

Setelah mendapatkan Upper Triangular, kita akan menggunakan Upper Triangular matrix untuk mencari nilai matrix d. Sama seperti LU Factorization, kita menggunakan U untuk mencari matrix d dengan persamaan Ud = b dimana U, d, dan b adalah matrix.

Setelah mendapatkan matrix d, kita akan menggunakan matrix d untuk mendapatkan x1,x2 dan x3. Cara carinya adalah dengan menggunakan rumus U^T x = d. Dimana U^T adalah Transpose matrix U, x adalah nilai x yang ingin dicari dan d adalah matrix d yang dicari sebelumnya.

c. Coding

```
import numpy as np
def cholesky_factorization(A):
    n = len(A)
    L = [[0.0] * n for _ in range(n)]

    for i in range(n):
        for j in range(i + 1):
            if i == j:
                L[i][j] = np.sqrt(A[i][i] - sum(L[i][k] ** 2
for k in range(j)))
            else:
                L[i][j] = (1.0 / L[j][j]) * (A[i][j] -
sum(L[i][k] * L[j][k] for k in range(j)))

    return L



def solve_linear_equation(A, b):
    L = cholesky_factorization(A)
    n = len(A)
    y = [0.0] * n
    x = [0.0] * n
```

```
    # Proses 2.1: Forward substitution: Ly = b
    for i in range(n):
        y[i] = (b[i] - sum(L[i][j] * y[j] for j in range(i))) / L[i][i]

    # Proses2.2: Backward substitution: L^T x = y
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - sum(L[j][i] * x[j] for j in range(i + 1, n))) / L[i][i]

    return x

matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]


z = solve_linear_equation(matrix,matrixB)

print(z)
```

d. Input dan Output

Input:

```
matrix = [[10,2,-1],[-3,-6,2],[1,1,5]]
matrixB = [27,-61.5,-21.5]
```

Output : Tidak ada karena matrix tidak simetris.

4. Inverse Matrix Menggunakan LU Decomposition
   a. Nama Program : InverseLUDecomposition
   b. Penjelasan Program

      Pada program ini akan mengambil input berupa matrix dari soal dan nanti akan digunakan untuk mencari inverse dari matrix tersebut menggunakan LU Decomposition.

      Pada program ini, dibagi menjadi dua function yaitu lu_decomposition dan inverse_matrix. Saya akan mulai menjelaskan dari lu_decomposition. Pada function lu_decomposition, kita akan mencari lower dan Upper Triangular dari matrix menggunakan function tersebut.

Cara yang digunakan sama seperti pada program LU Decomposition sebelumnya. Setelah mendapatkan lower triangular dan upper triangular, kita kemudian melakukan inverse dari matrix menggunakan function inverse_matrix.

c. Coding

```
import numpy as np
def lu_decomposition(matrix):
    n = len(matrix)
    lower = np.zeros((n, n))
    upper = np.zeros((n, n))

    for i in range(n):
        lower[i, i] = 1.0

        for j in range(i + 1):
            sum_upper = sum(lower[i, k] * upper[k, j] for k in 
range(j))
            upper[i, j] = matrix[i, j] - sum_upper

        for j in range(i, n):
            sum_lower = sum(lower[i, k] * upper[k, j] for k in 
range(i))
            lower[i, j] = (matrix[i, j] - sum_lower) / 
upper[i, i]

    return lower, upper


def inverse_matrix(matrix):
    n = len(matrix)
    identity = np.eye(n)
    lower, upper = lu_decomposition(matrix)

    inv_matrix = np.zeros((n, n))
    for i in range(n):
        y = np.zeros(n)
        x = np.zeros(n)
        y[0] = identity[i, 0] / lower[0, 0]

        for j in range(1, n):
```

```
            sum_lower = sum(lower[j, k] * y[k] for k in
range(j))
            y[j] = (identity[i, j] - sum_lower) / lower[j, j]

        x[n - 1] = y[n - 1] / upper[n - 1, n - 1]
        for j in range(n - 2, -1, -1):
            sum_upper = sum(upper[j, k] * x[k] for k in
range(j + 1, n))
            x[j] = (y[j] - sum_upper) / upper[j, j]

        inv_matrix[i] = x

    return inv_matrix


matrix = np.array([[10,2,-1],[-3,-6,2],[1,1,5]])
matrixB = np.array([27,-61.5,-21.5])

print(inverse_matrix(matrix))
```

d. Input dan Output

Input :

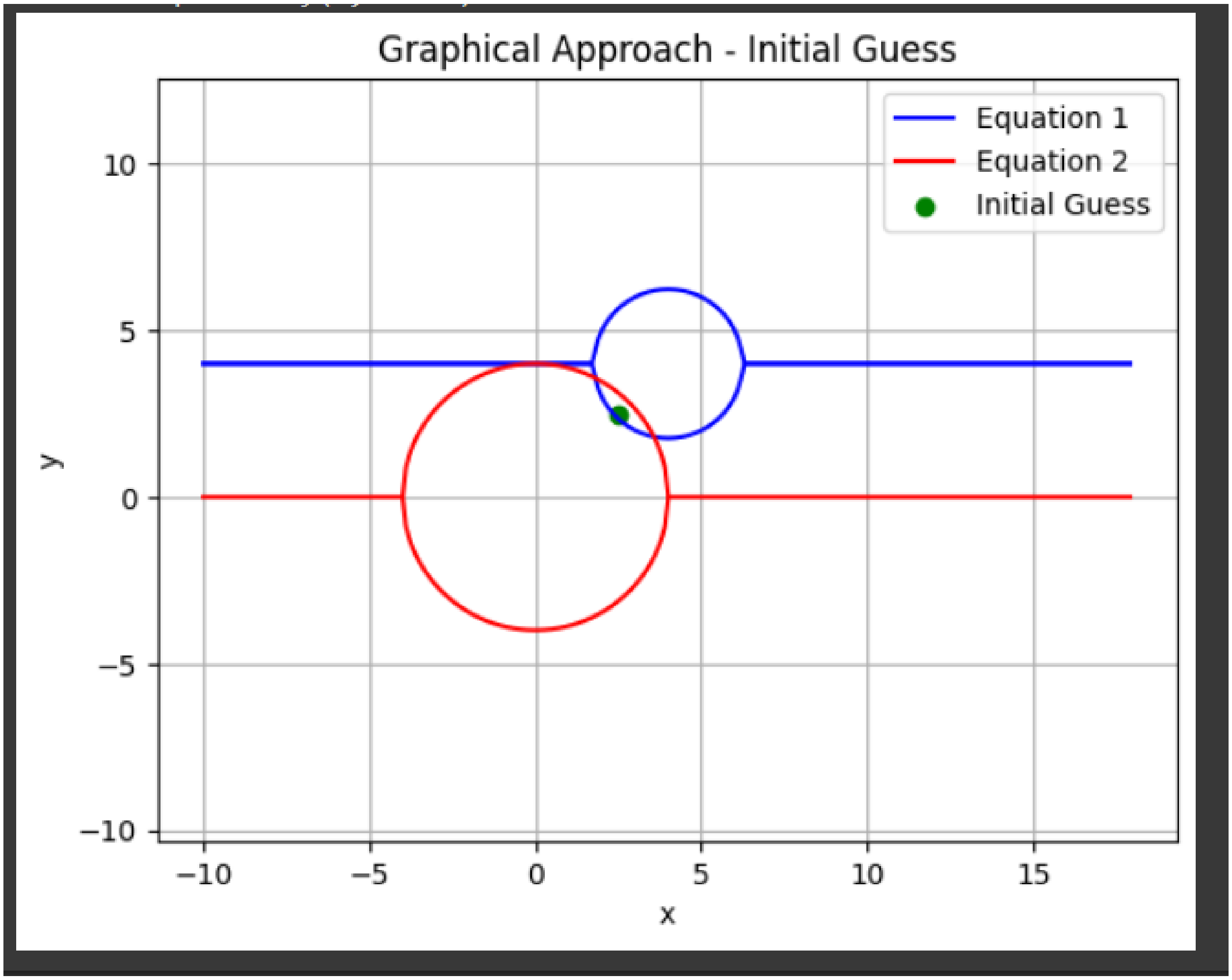
```
matrix = np.array([[10,2,-1],[-3,-6,2],[1,1,5]])
matrixB = np.array([27,-61.5,-21.5])
```

Output:

```
[[ 0.1         -0.          0.        ]
 [ 0.         -0.16666667  0.        ]
 [ 0.         -0.          0.2       ]]
```

Soal No. 3

3 A1. Grafik




```
import matplotlib.pyplot as plt

# Persamaan 1: (x - 4)^2 + (y - 4)^2 = 5
def equation1(x):
    return ((5 - (x - 4)**2)**0.5) + 4, -((5 - (x - 4)**2)**0.5) + 4

# Persamaan 2: x^2 + y^2 = 16
def equation2(x):
    return ((16 - x**2)**0.5), -((16 - x**2)**0.5)

# Plot persamaan-persamaan
x = [i/10 for i in range(-100, 180)]  # Menghasilkan nilai x dari
-10 hingga 17 dengan increment 0.1
y1_pos = []
y1_neg = []
y2_pos = []
y2_neg = []
```

```python
for i in range(len(x)):
    y1_pos_val, y1_neg_val = equation1(x[i])
    y1_pos.append(y1_pos_val)
    y1_neg.append(y1_neg_val)
    y2_pos_val, y2_neg_val = equation2(x[i])
    y2_pos.append(y2_pos_val)
    y2_neg.append(y2_neg_val)

plt.plot(x, y1_pos, 'b', label='Equation 1')
plt.plot(x, y1_neg, 'b')
plt.plot(x, y2_pos, 'r', label='Equation 2')
plt.plot(x, y2_neg, 'r')

# Titik perkiraan awal
initial_guess_x = 2.5
initial_guess_y = 2.5
plt.scatter(initial_guess_x, initial_guess_y, color='green',
label='Initial Guess')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Graphical Approach - Initial Guess')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.show()
```

Penjelasan Hasil Program :

1. Import library matplotlib:
   - **import matplotlib.pyplot as plt** digunakan untuk mengimport modul pyplot dari library matplotlib. Modul ini digunakan untuk membuat plot grafik.
2. Definisi persamaan nonlinear:
   - Persamaan pertama didefinisikan oleh fungsi **equation1(x)** yang menghitung nilai y berdasarkan persamaan (x - 4)^2 + (y - 4)^2 = 5.
   - Persamaan kedua didefinisikan oleh fungsi **equation2(x)** yang menghitung nilai y berdasarkan persamaan x^2 + y^2 = 16.
3. Plot persamaan:

    - Menggunakan perulangan **for** untuk mengiterasi nilai x dari -10 hingga 17 dengan increment 0.1.
    - Setiap nilai x digunakan untuk menghitung nilai y1_pos, y1_neg, y2_pos, dan y2_neg menggunakan fungsi persamaan yang telah didefinisikan.
    - Kurva persamaan pertama diplot dalam warna biru dengan **plt.plot(x, y1_pos, 'b')** dan **plt.plot(x, y1_neg, 'b')**.
    - Kurva persamaan kedua diplot dalam warna merah dengan **plt.plot(x, y2_pos, 'r')** dan **plt.plot(x, y2_neg, 'r')**.
4. Titik perkiraan awal:
    - Nilai perkiraan awal x dan y ditentukan oleh **initial_guess_x** dan **initial_guess_y**.
    - Scatter plot ditambahkan pada titik perkiraan awal dengan **plt.scatter(initial_guess_x, initial_guess_y, color='green')**.
5. Konfigurasi plot:
    - Label sumbu x dan y ditentukan dengan **plt.xlabel('x')** dan **plt.ylabel('y')**.
    - Judul plot ditentukan dengan **plt.title('Graphical Approach - Initial Guess')**.
    - Legenda persamaan ditambahkan dengan **plt.legend()**.
    - Grid ditampilkan dengan **plt.grid(True)**.
    - Skala sumbu x dan y diset agar sama dengan **plt.axis('equal')**.
6. Tampilkan plot:
    - Plot yang telah dikonfigurasi ditampilkan dengan **plt.show()**.

3 A2. Coding dan hasil

```
# Fungsi persamaan
def equation1(x, y):
    return (x - 4)**2 + (y - 4)**2 - 5

def equation2(x, y):
    return x**2 + y**2 - 16

# Metode Gaus-Seidel
def gauss_seidel():
    x = 1.0  # Perkiraan awal x
    y = 1.0  # Perkiraan awal y
    error = 1.0  # Kesalahan awal

    while error > 1e-4:
        x_new = (4 + y - (5 - (x - 4)**2)**0.5)**0.5  # Perhitungan x baru
        y_new = (16 - x**2)**0.5  # Perhitungan y baru

        error = max(abs(x_new - x), abs(y_new - y))  # Menghitung
kesalahan
        x = x_new
        y = y_new

    return x, y

# Solusi
solution_x, solution_y = gauss_seidel()
print("Solusi:")
print("x =", solution_x)
print("y =", solution_y)
```

```
Solusi:
x = (2.3814157927306123+5.067866649808871e-06j)
y = (3.2138798667514186+3.084978104329975e-05j)
```

Penjelasan Program :

1. Definisi fungsi persamaan:
    - Fungsi **equation1(x, y)** mengimplementasikan persamaan (x - 4)^2 + (y - 4)^2 - 5.
    - Fungsi **equation2(x, y)** mengimplementasikan persamaan x^2 + y^2 - 16.

2. Metode Gaus-Seidel:
    - Fungsi **gauss_seidel()** mengimplementasikan metode Gaus-Seidel untuk menyelesaikan persamaan nonlinear.
    - Langkah-langkah metode Gaus-Seidel dilakukan dalam loop while.
    - Variabel x dan y diinisialisasi dengan perkiraan awal.
    - Kesalahan awal diatur sebagai 1.0.
    - Selama kesalahan lebih besar dari 1e-4 (batas kesalahan yang ditentukan), langkah-langkah metode Gaus-Seidel dilakukan.
    - Dalam setiap iterasi, x dan y baru dihitung menggunakan rumus iterasi Gaus-Seidel.
    - Kesalahan relatif antara nilai baru dan nilai sebelumnya diukur dan dijadikan sebagai kesalahan saat ini.
    - Nilai x dan y diperbarui dengan nilai baru yang dihitung.
    - Iterasi dilanjutkan sampai kesalahan turun di bawah batas yang ditentukan.
3. Solusi:
    - Setelah loop selesai, nilai x dan y yang konvergen menjadi solusi akhir dicetak.

3 A3. Coding dan Hasil

```
# Fungsi persamaan

def equation1(x, y):

    return (x - 4)**2 + (y - 4)**2 - 5



def equation2(x, y):

    return x**2 + y**2 - 16



# Metode Gaus-Seidel dengan relaksasi

def gauss_seidel_relaxation(relaxation):

    x = 1.0  # Perkiraan awal x

    y = 1.0  # Perkiraan awal y

    error = 1.0  # Kesalahan awal
```

```
    while error > 1e-4:

        x_new = relaxation * ((4 + y - (5 - (x - 4)**2)**0.5)**0.5) + (1 -
relaxation) * x  # Perhitungan x baru dengan relaksasi

        y_new = relaxation * (16 - x**2)**0.5 + (1 - relaxation) * y  #
Perhitungan y baru dengan relaksasi


        error = max(abs(x_new - x), abs(y_new - y))  # Menghitung
kesalahan

        x = x_new

        y = y_new


    return x, y


# Relaksasi yang ditentukan

relaxation_factor = 0.5


# Solusi

solution_x, solution_y = gauss_seidel_relaxation(relaxation_factor)

print("Solusi:")

print("x =", solution_x)

print("y =", solution_y)
```

```
Solusi:
x = (2.3813996342461947-5.194147175466448e-06j)
y = (3.2137702663348895-5.562700713436985e-06j)
```

Penjelasan Program :

1. Definisi fungsi persamaan:
   - Fungsi **equation1(x, y)** mengimplementasikan persamaan (x - 4)^2 + (y - 4)^2 - 5.
   - Fungsi **equation2(x, y)** mengimplementasikan persamaan x^2 + y^2 - 16.
2. Metode Gaus-Seidel dengan relaksasi:
   - Fungsi **gauss_seidel_relaxation(relaxation)** mengimplementasikan metode Gaus-Seidel dengan relaksasi untuk menyelesaikan persamaan nonlinear.
   - Langkah-langkah metode Gaus-Seidel dengan relaksasi dilakukan dalam loop while.
   - Variabel x dan y diinisialisasi dengan perkiraan awal.
   - Kesalahan awal diatur sebagai 1.0.
   - Selama kesalahan lebih besar dari 1e-4 (batas kesalahan yang ditentukan), langkah-langkah metode Gaus-Seidel dengan relaksasi dilakukan.
   - Dalam setiap iterasi, x dan y baru dihitung menggunakan rumus iterasi Gaus-Seidel dengan relaksasi.
   - Kesalahan relatif antara nilai baru dan nilai sebelumnya diukur dan dijadikan sebagai kesalahan saat ini.
   - Nilai x dan y diperbarui dengan nilai baru yang dihitung.
   - Iterasi dilanjutkan sampai kesalahan turun di bawah batas yang ditentukan.
3. Nilai relaksasi:
   - Nilai relaksasi (relaxation factor) ditentukan oleh variabel **relaxation_factor**.
   - Nilai ini dapat diubah untuk mengatur seberapa besar pengaruh iterasi baru terhadap nilai yang ada sebelumnya.
   - Nilai relaksasi yang lebih besar dari 0 hingga kurang dari 2 dapat digunakan, dengan 1 sebagai nilai tanpa relaksasi.
4. Solusi:
   - Setelah loop selesai, nilai x dan y yang konvergen menjadi solusi akhir dicetak.

3 A4. Coding dan Hasil

```
# Fungsi persamaan
def equation1(x, y):
    return -x**2 + x + 0.75 - y


def equation2(x, y):
    return y + 5*x*y - x**2


# Turunan parsial
def partial_derivative_x(f, x, y, h=1e-6):
    return (f(x + h, y) - f(x - h, y)) / (2 * h)


def partial_derivative_y(f, x, y, h=1e-6):
    return (f(x, y + h) - f(x, y - h)) / (2 * h)


# Metode Newton-Raphson
def newton_raphson():
    x = 1.2  # Perkiraan awal x
    y = 1.2  # Perkiraan awal y
    error = 1.0  # Kesalahan awal


    while error > 1e-4:
        f1 = equation1(x, y)
```

```python
        f2 = equation2(x, y)

        df1_dx = partial_derivative_x(equation1, x, y)
        df1_dy = partial_derivative_y(equation1, x, y)
        df2_dx = partial_derivative_x(equation2, x, y)
        df2_dy = partial_derivative_y(equation2, x, y)

        determinant = df1_dx * df2_dy - df1_dy * df2_dx

        x_new = x - (f1 * df2_dy - f2 * df1_dy) / determinant
        y_new = y - (f2 * df1_dx - f1 * df2_dx) / determinant

        error = max(abs(x_new - x), abs(y_new - y))  # Menghitung
kesalahan
        x = x_new
        y = y_new

    return x, y

# Solusi
solution_x, solution_y = newton_raphson()
print("Solusi:")
print("x =", solution_x)
print("y =", solution_y)
```

```
Solusi:
x = 1.3720654058273418
y = 0.23950192795920694
```

Penjelasan Program :

1. Definisi fungsi persamaan:
    - Fungsi **equation1(x, y)** mengimplementasikan persamaan -x^2 + x + 0.75 - y.
    - Fungsi **equation2(x, y)** mengimplementasikan persamaan y + 5xy - x^2.
2. Turunan parsial:
    - Fungsi **partial_derivative_x(f, x, y, h)** menghitung turunan parsial f terhadap x dengan menggunakan rumus perbedaan maju.
    - Fungsi **partial_derivative_y(f, x, y, h)** menghitung turunan parsial f terhadap y dengan menggunakan rumus perbedaan maju.
    - Nilai h adalah pilihan Anda dan digunakan untuk mengaproksimasi turunan.
3. Metode Newton-Raphson:
    - Fungsi **newton_raphson()** mengimplementasikan metode Newton-Raphson untuk menyelesaikan persamaan nonlinear.
    - Langkah-langkah metode Newton-Raphson dilakukan dalam loop while.
    - Variabel x dan y diinisialisasi dengan perkiraan awal.
    - Kesalahan awal diatur sebagai 1.0.
    - Selama kesalahan lebih besar dari 1e-4 (batas kesalahan yang ditentukan), langkah-langkah metode Newton-Raphson dilakukan.
    - Dalam setiap iterasi, nilai persamaan dan turunan parsial persamaan terhadap x dan y dihitung.
    - Determinan dari matriks turunan parsial dihitung.
    - Nilai x dan y baru dihitung menggunakan rumus iterasi Newton-Raphson.
    - Kesalahan relatif antara nilai baru dan nilai sebelumnya diukur dan dijadikan sebagai kesalahan saat ini.
    - Nilai x dan y diperbarui dengan nilai baru yang dihitung.
    - Iterasi dilanjutkan sampai kesalahan turun di bawah batas yang ditentukan.
4. Solusi:
    - Setelah loop selesai, nilai x dan y yang konvergen menjadi solusi akhir dicetak.

No 3A5. Coding dan Hasil

```
function F = equations(x)
    F = zeros(2, 1);
    F(1) = (x(1) - 4)^2 + (x(2) - 4)^2 - 5;
    F(2) = x(1)^2 + x(2)^2 - 16;
endfunction

initial_guess = [1.2; 1.2];
options = optimset('fsolve');
options.TolFun = 1e-4;
[x, ~, exitflag] = fsolve(@equations, initial_guess, options);

if exitflag == 1
    fprintf('Solusi:\n');
    fprintf('x = %f\n', x(1));
    fprintf('y = %f\n', x(2));
else
    fprintf('Tidak ada solusi yang ditemukan.\n');
end
```

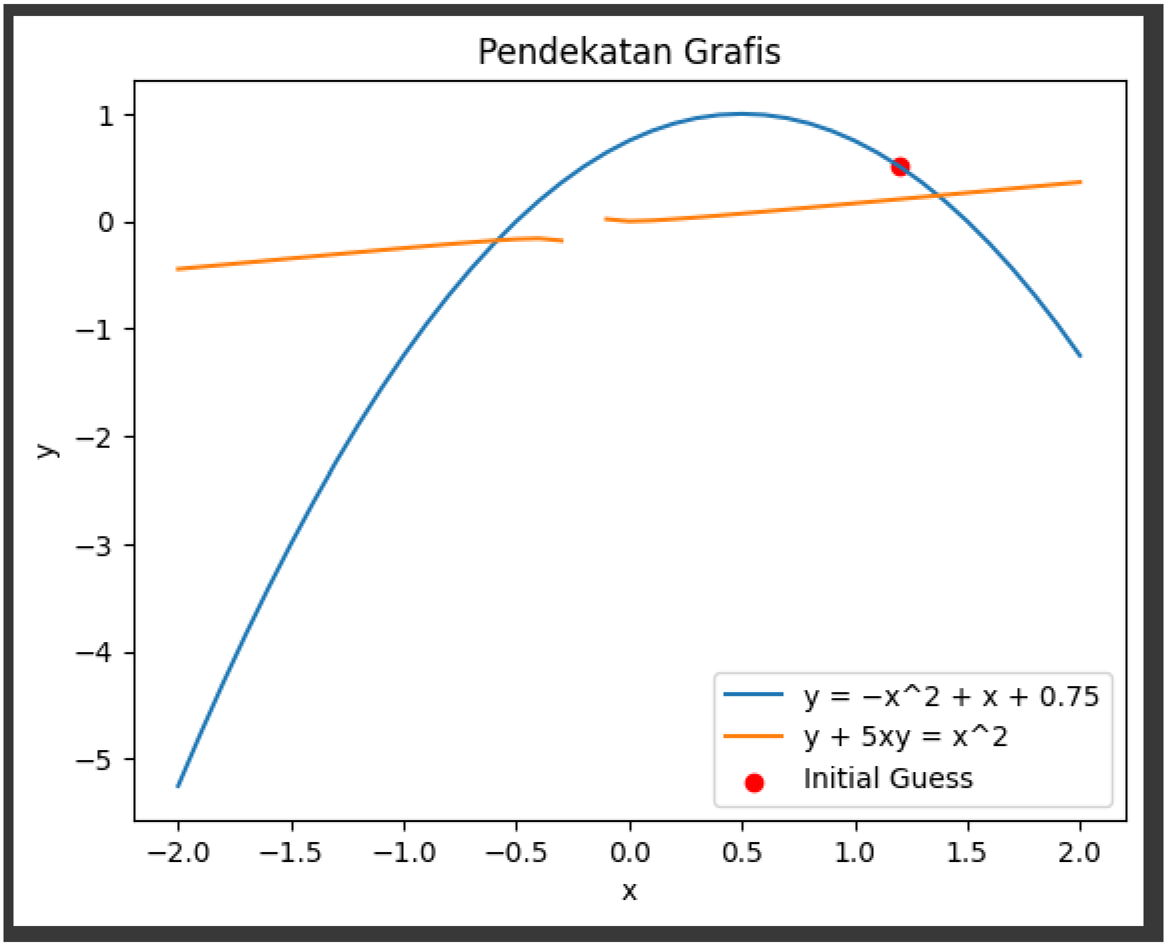
Name	Class	Dimension	Value	Attribute
options	struct	1x1	...	
exitflag	double	1x1	3	
initial_guess	double	2x1	[1.2000; 1.2000]	
x	double	2x1	[2.7517; 2.7517]	

Penjelasan Program :

1. Definisi fungsi **equations(x)**:
    - Fungsi ini mengimplementasikan persamaan nonlinear yang diberikan.
    - Fungsi ini mengambil argumen x, yang merupakan vektor kolom dengan elemen x dan y.
    - Dalam fungsi ini, kita menghitung dua persamaan: (x - 4)^2 + (y - 4)^2 - 5 dan x^2 + y^2 - 16.
    - Kedua persamaan tersebut diberikan nilai nol.
2. Inisialisasi perkiraan awal dan opsi:
    - **initial_guess** adalah vektor kolom dengan perkiraan awal untuk x dan y, yaitu [1.2; 1.2].

    - **options** digunakan untuk mengatur opsi untuk fungsi **fsolve**.
    - Dalam contoh ini, kita menggunakan **optimset** untuk membuat objek opsi, dan kemudian mengatur **TolFun** (batas toleransi kesalahan fungsi) menjadi 1e-4.
3. Penggunaan fungsi **fsolve**:
    - Fungsi **fsolve** digunakan untuk mencari solusi numerik dari persamaan nonlinear.
    - Kami memberikan fungsi **equations** sebagai argumen pertama, yang akan dipecahkan oleh **fsolve**.
    - Argumen kedua adalah perkiraan awal **initial_guess**, dan argumen ketiga adalah opsi **options** yang telah diatur sebelumnya.
    - Hasil dari **fsolve** disimpan dalam variabel **x**.
4. Pemeriksaan **exitflag**:
    - Setelah menjalankan **fsolve**, kita memeriksa nilai **exitflag**.
    - Jika **exitflag** sama dengan 1, ini berarti **fsolve** berhasil menemukan solusi.
    - Dalam hal ini, kita mencetak solusi yang ditemukan dengan memformat dan mencetak nilai x dan y.
    - Jika **exitflag** tidak sama dengan 1, maka tidak ada solusi yang ditemukan, dan pesan yang sesuai dicetak.

No 3B1. Coding dan Hasil




```
import matplotlib.pyplot as plt


# Persamaan pertama: y = −x^2 + x + 0.75

x = []

y1 = []

for i in range(-20, 21):

    x_val = i / 10

    y1_val = -x_val**2 + x_val + 0.75

    x.append(x_val)
```

```
    y1.append(y1_val)


# Persamaan kedua: y + 5xy = x^2
y2 = []
for x_val in x:
    if 1 + 5 * x_val != 0:
        y2_val = x_val**2 / (1 + 5 * x_val)
        y2.append(y2_val)
    else:
        y2.append(None)


# Titik perkiraan awal
initial_guess_x = 1.2
initial_guess_y = -initial_guess_x**2 + initial_guess_x + 0.75


# Gambar grafik
plt.plot(x, y1, label='y = -x^2 + x + 0.75')
plt.plot(x, y2, label='y + 5xy = x^2')
plt.scatter(initial_guess_x, initial_guess_y, color='red', label='Initial Guess')


# Konfigurasi grafik
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.title('Pendekatan Grafis')

plt.legend()



# Tampilkan grafik

plt.show()
```

Penjelasan Program :

1. Mengimpor modul matplotlib.pyplot:
    - Modul ini digunakan untuk membuat plot grafik.
2. Persamaan pertama: $y = -x^2 + x + 0.75$:
    - Kita menggunakan loop **for** untuk menghasilkan serangkaian nilai x dalam rentang -2 hingga 2 dengan langkah 0.1.
    - Setiap nilai x dihitung dengan membagi nilai loop **i** dengan 10.
    - Nilai y1 dihitung menggunakan persamaan $y = -x^2 + x + 0.75$.
    - Array nilai x dan y1 disusun menggunakan metode **.append()**.
3. Persamaan kedua: $y + 5xy = x^2$:
    - Kita menggunakan loop **for** untuk menghitung nilai y2 dengan menggunakan persamaan $y + 5xy = x^2$.
    - Setiap nilai x diambil dari array x yang telah dibentuk sebelumnya.
    - Sebelum menghitung nilai y2, kita melakukan pengujian untuk memeriksa apakah penyebut dalam persamaan (1 + 5 * x_val) tidak nol.
    - Jika penyebut tidak nol, nilai y2 dihitung dengan membagi x_val^2 dengan penyebut.
    - Jika penyebut nol, nilai y2 diatur sebagai None untuk menunjukkan bahwa tidak ada nilai yang valid pada titik tersebut.
    - Array nilai y2 disusun menggunakan metode **.append()**.
4. Titik perkiraan awal:
    - Variabel **initial_guess_x** dan **initial_guess_y** menyimpan nilai perkiraan awal yang telah diberikan.
5. Gambar grafik:
    - Menggunakan fungsi **plt.plot** untuk menggambar persamaan pertama dan kedua.
    - Menggunakan fungsi **plt.scatter** untuk menandai titik perkiraan awal dengan warna merah.
6. Konfigurasi grafik:

    ○ Menggunakan fungsi **plt.xlabel**, **plt.ylabel**, dan **plt.title** untuk memberikan label sumbu x, sumbu y, dan judul grafik.
7. Tampilkan grafik:
    ○ Menggunakan fungsi **plt.show()** untuk menampilkan grafik yang telah dibuat.

No 3B2. Coding dan Hasil

```
def gauss_seidel(x0, y0, error=1e-4, max_iter=100):

    x = x0

    y = y0

    iterasi = 0

    while True:

        x_new = (-y + (x**2)) / (5 * x)

        y_new = -x**2 + x + 0.75

        dx = abs(x_new - x)

        dy = abs(y_new - y)

        x = x_new

        y = y_new

        iterasi += 1

        if dx < error and dy < error:

            break

        if iterasi > max_iter:

            break

    return x, y
```

```
# Titik perkiraan awal

initial_guess_x = 1.2

initial_guess_y = 1.2


# Solusi menggunakan metode Gauss-Seidel

solusi = gauss_seidel(initial_guess_x, initial_guess_y, error=1e-4)


# Menampilkan solusi

print("Solusi:")

print("x =", solusi[0])

print("y =", solusi[1])
```

```
Solusi:
x = -9.20151777199217
y = -1976.1518643669126
```

Penjelasan Program :

1. Fungsi **gauss_seidel**:
    - Fungsi ini mengimplementasikan metode Gauss-Seidel untuk menyelesaikan sistem persamaan non-linear.
    - Fungsi menerima parameter **x0** dan **y0** sebagai titik perkiraan awal, **error** sebagai nilai toleransi kesalahan (default: 1e-4), dan **max_iter** sebagai jumlah maksimum iterasi (default: 100).
    - Variabel **x** dan **y** diinisialisasi dengan nilai **x0** dan **y0**.
    - Pada setiap iterasi, variabel **x_new** dan **y_new** dihitung berdasarkan rumus iterasi Gauss-Seidel.
    - Selisih antara nilai **x_new** dengan **x** (dx) dan nilai **y_new** dengan **y** (dy) dihitung untuk menentukan kapan iterasi harus berhenti.
    - Jika selisih dx dan dy lebih kecil dari nilai **error**, iterasi dihentikan dan nilai **x** dan **y** dikembalikan sebagai solusi.

    - Jika jumlah iterasi melebihi **max_iter**, iterasi dihentikan dan nilai **x** dan **y** dikembalikan sebagai solusi terbaik yang ditemukan.
2. Titik perkiraan awal:
    - Variabel **initial_guess_x** dan **initial_guess_y** menyimpan nilai perkiraan awal yang telah diberikan.
3. Solusi menggunakan metode Gauss-Seidel:
    - Fungsi **gauss_seidel** dipanggil dengan menggunakan titik perkiraan awal dan nilai toleransi kesalahan yang diinginkan.
    - Hasil solusi disimpan dalam variabel **solusi**.
4. Menampilkan solusi:
    - Nilai solusi **x** dan **y** ditampilkan di layar.

No 3B3. Coding dan Hasil

```
def gauss_seidel_relaxation(x0, y0, relaxation_factor, error=1e-4,
max_iter=100):

    x = x0

    y = y0

    iterasi = 0

    while True:

        x_new = (-y + (x**2)) / (5 * x)

        y_new = -x**2 + x + 0.75

        dx = abs(x_new - x)

        dy = abs(y_new - y)

        x = relaxation_factor * x_new + (1 - relaxation_factor) * x

        y = relaxation_factor * y_new + (1 - relaxation_factor) * y

        iterasi += 1

        if dx < error and dy < error:

            break
```

```
        if iterasi > max_iter:
            break
    return x, y


# Titik perkiraan awal
initial_guess_x = 1.2
initial_guess_y = 1.2


# Nilai relaksasi
relaxation_factor = 0.8


# Solusi menggunakan metode Gauss-Seidel dengan relaksasi
solusi = gauss_seidel_relaxation(initial_guess_x, initial_guess_y, relaxation_factor, error=1e-4)


# Menampilkan solusi
print("Solusi:")
print("x =", solusi[0])
print("y =", solusi[1])
```

```
Solusi:
x = -0.7664214300520493
y = -1.4003854271358758
```

Penjelasan Program :

1. Fungsi **gauss_seidel_relaxation**:

- Fungsi ini mengimplementasikan metode Gauss-Seidel dengan relaksasi untuk menyelesaikan sistem persamaan non-linear.
- Fungsi menerima parameter **x0** dan **y0** sebagai titik perkiraan awal, **relaxation_factor** sebagai faktor relaksasi yang ditentukan, **error** sebagai nilai toleransi kesalahan (default: 1e-4), dan **max_iter** sebagai jumlah maksimum iterasi (default: 100).
- Variabel **x** dan **y** diinisialisasi dengan nilai **x0** dan **y0**.
- Pada setiap iterasi, variabel **x_new** dan **y_new** dihitung berdasarkan rumus iterasi Gauss-Seidel.
- Selisih antara nilai **x_new** dengan **x** (dx) dan nilai **y_new** dengan **y** (dy) dihitung untuk menentukan kapan iterasi harus berhenti.
- Nilai **x** dan **y** diperbarui dengan menggunakan rumus relaksasi yang melibatkan faktor relaksasi.
- Jika selisih dx dan dy lebih kecil dari nilai **error**, iterasi dihentikan dan nilai **x** dan **y** dikembalikan sebagai solusi.
- Jika jumlah iterasi melebihi **max_iter**, iterasi dihentikan dan nilai **x** dan **y** dikembalikan sebagai solusi terbaik yang ditemukan.

2. Titik perkiraan awal dan nilai relaksasi:
    - Variabel **initial_guess_x** dan **initial_guess_y** menyimpan nilai perkiraan awal yang telah diberikan.
    - Variabel **relaxation_factor** menyimpan nilai faktor relaksasi yang telah ditentukan.
3. Solusi menggunakan metode Gauss-Seidel dengan relaksasi:
    - Fungsi **gauss_seidel_relaxation** dipanggil dengan menggunakan titik perkiraan awal, faktor relaksasi, dan nilai toleransi kesalahan yang diinginkan.
    - Hasil solusi disimpan dalam variabel **solusi**.
4. Menampilkan solusi:
    - Nilai solusi **x** dan **y** ditampilkan di layar.

No 3B4. Coding dan Hasil

```
def newton_raphson(x0, y0, error=1e-4, max_iter=100):

    x = x0

    y = y0

    iterasi = 0

    while True:
```

```
        f1 = -x**2 + x + 0.75 - y

        f2 = y + 5 * x * y - x**2

        jacobian = [[-2 * x + 1, -1], [-2 * x + 5 * y, 5 * x + 1]]

        delta_x = (jacobian[0][0] * f1 + jacobian[0][1] * f2) /
(jacobian[0][0]**2 + jacobian[0][1]**2)

        delta_y = (jacobian[1][0] * f1 + jacobian[1][1] * f2) /
(jacobian[1][0]**2 + jacobian[1][1]**2)

        x -= delta_x

        y -= delta_y

        iterasi += 1

        if abs(delta_x) < error and abs(delta_y) < error:

            break

        if iterasi > max_iter:

            break

    return x, y


# Titik perkiraan awal

initial_guess_x = 1.2

initial_guess_y = 1.2


# Solusi menggunakan metode Newton-Raphson

solusi = newton_raphson(initial_guess_x, initial_guess_y, error=1e-4)


# Menampilkan solusi
```

```
print("Solusi:")

print("x =", solusi[0])

print("y =", solusi[1])
```

```
Solusi:

x = 1.3721256746967123
y = 0.239510363761202
```

Penjelasan Program :

1. Fungsi **newton_raphson** mengambil titik perkiraan awal **x0** dan **y0**, serta parameter opsional **error** yang menentukan toleransi kesalahan dan **max_iter** yang menentukan jumlah maksimum iterasi yang diizinkan. Variabel **x** dan **y** diinisialisasi dengan nilai titik perkiraan awal, sedangkan **iterasi** digunakan untuk menghitung jumlah iterasi yang dilakukan.
2. Dalam setiap iterasi, fungsi **newton_raphson** menghitung nilai fungsi-fungsi **f1** dan **f2** berdasarkan persamaan yang diberikan. Kemudian, matriks Jacobian dihitung menggunakan turunan parsial persamaan-persamaan tersebut.
3. Selanjutnya, **delta_x** dan **delta_y** dihitung dengan menggunakan rumus iterasi metode Newton-Raphson. Untuk menghindari OverflowError, perhitungan ini telah dimodifikasi dengan membagi setiap elemen dengan kuadrat elemen matriks Jacobian.
4. Nilai **x** dan **y** diperbarui dengan mengurangi **delta_x** dan **delta_y** dari nilai sebelumnya.
5. Proses iterasi dilanjutkan hingga selisih antara **delta_x** dan **delta_y** lebih kecil dari toleransi kesalahan yang ditentukan, atau jika jumlah iterasi sudah mencapai batas maksimum yang ditentukan.
6. Setelah iterasi selesai, solusi akhir berupa nilai **x** dan **y** dikembalikan oleh fungsi **newton_raphson**.
7. Solusi akhir dicetak dalam bentuk "x = ..." dan "y = ..." menggunakan perintah print.

No 3B5. Coding dan Hasil

```
function F = myFunction(x)
  F(1) = -x(1)^2 + x(1) + 0.75 - x(2);
  F(2) = x(2) + 5 * x(1) * x(2) - x(1)^2;
endfunction

initial_guess = [1.2; 1.2];
solution = fsolve(@myFunction, initial_guess);

disp("Solusi:");
disp(["x = ", num2str(solution(1))]);
disp(["y = ", num2str(solution(2))]);
```

```
>> Solusi:
x = 1.3721
y = 0.2395
```

Penjelasan Program :

1. Fungsi **myFunction(x)** didefinisikan untuk menghitung persamaan-persamaan non-linear yang ingin diselesaikan. Fungsi ini mengambil vektor **x** sebagai argumen dan mengembalikan vektor **F** yang berisi nilai fungsi untuk setiap persamaan. Dalam contoh ini, terdapat dua persamaan yang didefinisikan: **-x(1)^2 + x(1) + 0.75 - x(2)** dan **x(2) + 5 * x(1) * x(2) - x(1)^2**. Persamaan-persamaan ini diatur dalam vektor **F** sesuai dengan urutan yang ditentukan.
2. Variabel **initial_guess** diinisialisasi dengan vektor **[1.2; 1.2]**, yang merupakan perkiraan awal untuk nilai **x** dan **y**.
3. Fungsi **fsolve(@myFunction, initial_guess)** digunakan untuk menyelesaikan sistem persamaan non-linear dengan menggunakan metode numerik yang disediakan oleh **fsolve**. Argumen pertama **@myFunction** menyatakan fungsi yang ingin diselesaikan, dan argumen kedua **initial_guess** adalah perkiraan awal untuk solusi sistem.
4. Solusi yang ditemukan disimpan dalam variabel **solution**.
5. Perintah **disp** digunakan untuk mencetak solusi yang ditemukan. Solusi ditampilkan dengan format "x = ..." dan "y = ..." menggunakan fungsi **num2str** untuk mengonversi nilai numerik menjadi string.