Assignment

Exercise sheet 2

Name TN 1: Tim Peinkofer

tim.peinkofer@student.uni-tuebingen.de

Name TN 2: Fabian Kostow

fabian.kostow@student.uni-tuebingen.de

Tutor: Jose Carlos Olvera Meneses

Date: 14. November 2024

Inhaltsverzeichnis

1 Problem I

1.1 Matrix-Vektor-Multiplication

We want to calculate the product of a Matrix with a vector. The algorithm works as follows:

- Take the first row of the Matrix
- Multiply the j-column of this row with the j-row of the Vektor
- Add up all the values from the multiplication together
- As a result we get the value for the Vektor in the first row.
- Repeat this for all rows.

The Python code which executes this algorithm is in the following:

```
import numpy as np


Matrix = np.array([[1,2,3],
                   [0,5,6],
                   [0,0,9]])
Vektor = [10,11,12]



def matrix_vektor (N):
    Solution = np.zeros(N)
    for i in range(N):
      for j in range(N):
         Solution[i] = Solution[i]+ Matrix[i][j]*Vektor[j]
    return Solution

print(matrix_vektor(3))
```

1.2 Solving a System with an upper diagonal matrix

We want to solve a system in the following form:

$$\begin{pmatrix} a & b & c \\ 0 & e & f \\ 0 & 0 & i \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} j \\ k \\ l \end{pmatrix}$$

Because the Matrix is in an upper triangular form the algorithm to find solutions for x, y, z becomes easier, because we only have to use back substitution. The Algorithm is the following:

- go to the last row j
- calculates all the non-diagonal values with the corresponding, known value of (x, y, z), with non-diagonal we only consider the part of the matrix on the right side of the diagonal because the rest is zero.
- subtracts all the non-diagonal values from the corresponding k-row of the vector
- divide the value in the vector with the value [k][k] in the matrix
- go to row k-1 and repeat

The python code that executes this algorithm is the following:

```
import numpy as np


Matrix = np.array([[1,2,3],
                   [0,5,6],
                   [0,0,9]])
Vektor = [10,11,12]


def upper_matrix(n):
    solution = np.zeros(n)
    for k in range(n-1, -1, -1):
      non_diag = 0
      for l in range(k+1,n):
        non_diag = non_diag + Matrix[k][l] * solution[l]
      solution[k] =(Vektor[k]-non_diag)/Matrix[k][k]
    return solution

print(upper_matrix(3))
```

2 Problem II

2.1 Gauss Elimination

In the next step, we want to extend our linear system solver from Problem I. The reason is that we normally have no triangular matrix. However, we can extract these from any matrix using the Gauss elimination. If we add this method to our code we get the following result:

```
import numpy as np

# Initialize matrix and vector via numpy
matrix = np.array([[1, 2, 3], [4, 5, 7], [7, 8, 9]], dtype=np
                                    .float64)
vector = np.array([10, 11, 12], dtype=np.float64).reshape(-1,
                                     1)

print("Original Matrix:") # print the original matrix as a
                                   reference
print(matrix)
print("Original Vector:")
print(vector)

# Get dimensions of our matrix for later use
rows, columns = matrix.shape


def gauss(): # Function for gau  elimination

    # Generate a copy of the vector and the matrix for our
                                       gau  algorithm
    U_Matrix = np.copy(matrix)
    U_vector = np.copy(vector)
```

```
    for i in range(rows - 1):

        if U_Matrix[i][i] == 0:
            for k in range(i + 1, rows):
                if U_Matrix[k][i] != 0:
                    # Swap the rows in both U_Matrix and
                                                U_vector
                                                if the
                                                a_ii
                                                component
                                                is zero
                    U_Matrix[[i, k]] = U_Matrix[[k, i]]
                    U_vector[[i, k]] = U_vector[[k, i]]
                    break

        # Continue with elimination if a_ii != 0
        for j in range(i + 1, rows):
            if U_Matrix[i][i] != 0:
                factor = U_Matrix[j][i] / U_Matrix[i][i]
                U_Matrix[j] = U_Matrix[j] - factor * U_Matrix
                                                [i]
                U_vector[j] = U_vector[j] - factor * U_vector
                                                [i]
    return U_Matrix, U_vector

def solver(mat, vec): # Solver from our first program (a
                                little bit modified)
    x = np.zeros((rows, 1)) # Generate a solution vektor
                                    based on the number of
                                    rows of our matrix
    for i in range(rows):
        index = rows - i - 1
        b_new = vec[index] / mat[index, index]

        for r in range(index + 1, rows):
            b_new -= mat[index, r] * x[r] / mat[index, index]

        x[index] = b_new
    return x

# Get the triangular matrix and solve the linear equation
m, v = gauss()
solution = solver(m, v)
```

```
print("Upper Triangular Matrix:")
print(m)
print("Modified Vector after Gaussian elimination:")
print(v)
print("Solution Vector:")
print(solution)
```

We can now use the code above to solve the following system of equations:

$$\begin{pmatrix} 2 & 0.1 & -0.2 \\ 0.05 & 4.2 & 0.032 \\ 0.12 & -0.07 & 5 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$$

We obtain the following solution:

$$\vec{x} \approx \begin{pmatrix} 5.10427 \\ 2.54066 \\ 2.31307 \end{pmatrix}$$

Let's now check this solution:

$$\begin{aligned} 2 \cdot 5.10427 + 0.1 \cdot 2.54066 - 0.2 \cdot 2.31307 &\approx 10 \\ 0.05 \cdot 5.10427 + 4.2 \cdot 2.54066 + 0.032 \cdot 2.31307 &\approx 11 \\ 0.12 \cdot 5.10427 - 0.07 \cdot 2.54066 + 5 \cdot 2.31307 &\approx 12 \end{aligned}$$

As we see our code calculates the results correctly.

2.2 Gauss elimination with pivoting

We now try to calculate the solution of the following equation by adding pivoting to our above code:

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 2 & -2 \\ 0 & 3 & 15 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 1 \\ -2 \\ 33 \end{pmatrix}$$

The customized code looks like this:

```
import numpy as np

# Initialize matrix and vector via numpy
matrix = np.array([[1, 1, 0], [2 ,2, -2], [0, 3, 15]], dtype=
                                np.float64)
vector = np.array([1, -2, 33], dtype=np.float64).reshape(-1,
                                1)

print("Original Matrix:")  # print the original matrix as a
                                reference
print(matrix)
print("Original Vector:")
print(vector)

# Get dimensions of our matrix for later use
rows, columns = matrix.shape

# Generate a copy of the vector and the matrix for our
                                Gaussian elimination algorithm
U_Matrix = np.copy(matrix)
U_vector = np.copy(vector)
x = np.zeros((rows, 1))  # Generate a solution vector based
                                on the number of rows of our
                                matrix
```

```
def gauss():
    for i in range(rows - 1):
        if U_Matrix[i][i] == 0:
            for k in range(i + 1, rows):
                if U_Matrix[k][i] != 0:
                    # Swap the rows in both U_Matrix and
                                                U_vector
                                                if the
                                                a_ii
                                                component
                                                is zero
                    U_Matrix[[i, k]] = U_Matrix[[k, i]]
                    U_vector[[i, k]] = U_vector[[k, i]]
                    break

        # Continue with elimination if a_ii != 0
        for j in range(i + 1, rows):
            if U_Matrix[i][i] != 0:
                factor = U_Matrix[j][i] / U_Matrix[i][i]  #
                                                Calculate the
                                                factor
                U_Matrix[j] -= factor * U_Matrix[i]
                U_vector[j] -= factor * U_vector[i]

    return U_Matrix, U_vector

def pivoting(m, v):
    U_Matrix = m
    U_vector = v
    for i in range(rows - 1, -1, -1):
        if U_Matrix[i][i] != 0:
            # Normalize the pivots if a_ii != 0
            factor = U_Matrix[i][i]
            U_Matrix[i] /= factor #both vector and matrix!
            U_vector[i] /= factor

        # Get the matrix with only pivots
        for j in range(i - 1, -1, -1):
            if U_Matrix[j][i] != 0:
                factor = U_Matrix[j][i]
                U_Matrix[j] -= factor * U_Matrix[i]
                U_vector[j] -= factor * U_vector[i]

    return U_Matrix, U_vector
```

```
# Get the triangular matrix and solve the linear equation
m, v = gauss()
m, v = pivoting(m, v)
solution = v

print("Modified Matrix generated via Pivoting:")
print(m)
print("Solution Vector:")
print(solution)
```

With pivoting our code gives us the following solution:

$$\vec{x} \approx \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}$$

If we use the gauss elimination without pivoting we get:

$$\vec{x} \approx \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}$$

As we see both solutions are equal, but the code for the second program is much longer and less time-efficient.

3 Problem III

3.1 Finding the largest and smallest Eigenvalue/Eigenvektor

The algorithm to find the biggest eigenvector of a matrix A is

- choose an vector x and the maximum number of iterations
- multiply the vector x with the matrix A
- repeat this with the resulting vector until you reach max iteration
- the resulting vector is the eigenvector
- multiply the (j+1)-vector by the j-vector and divide it by the length of the j-vector to find the eigenvalue

To find the smallest eigenvalue we can make use of that the inverse of the biggest eigenvalue of the inverse matrix is the smallest of the original matrix, so we can find the smallest eigenvalue/vector analogously.

```
import numpy as np

M = np.array([[6, 5, -5],
              [2, 6, -2],
              [2, 5, -1]])
V = np.array([10, 11, 12])
size,size = M.shape
iteration = 100

def matrix_vector(N, Matrix, Vector):
    Solution = np.zeros(N)
    for i in range(N):
        for j in range(N):
            Solution[i] += Matrix[i][j] * Vector[j]
    return Solution

def Eigenvalue(mat,vec, Iteration):
    for i in range(Iteration):
        vec = matrix_vector(size, mat, vec)
    eigenvalue = np.dot(matrix_vector(size, mat, vec),vec) /
                                    np.dot(vec, vec)
    eigenvector = vec / np.linalg.norm(vec)
    return eigenvalue,eigenvector
```

```
eigenvalue_1,eigenvector_1 = Eigenvalue(M,V, iteration)
eigenvalue_2,eigenvector_2 = Eigenvalue(np.linalg.inv(M),V,
                               iteration)
eigenvalue_2 = 1/eigenvalue_2
print("Largest eigenvalue and corresponding eigenvektor:",
                               eigenvalue_1, eigenvector_1)
print("Smallest eigenvalue and corresponding eigenvektor:",
                               eigenvalue_2, eigenvector_2)
```

3.2 Aitken Acceleration

To apply Aitkens acceleration we have to store all the approximated eigenvalues. At the moment we have more than three eigenvalues we can apply the acceleration. The following code extends the code from 3.1 with Aitkens acceleration.

```
import numpy as np

M = np.array([[6, 5, -5],
              [2, 6, -2],
              [2, 5, -1]])
V = np.array([10, 11, 12])
size,size = M.shape
iteration = 10

def Aitken(Eigenvalues):
    if len(Eigenvalues) < 3:
        return Eigenvalues[-1]
    else:
        x_n, x_n1, x_n2 = Eigenvalues[-3], Eigenvalues[-2],
                                           Eigenvalues[-1]
        return x_n - (x_n1 - x_n) ** 2 / (x_n2 - 2 * x_n1 +
                                           x_n)

def matrix_vector(N, Matrix, Vector):
    Solution = np.zeros(N)
    for i in range(N):
        for j in range(N):
            Solution[i] += Matrix[i][j] * Vector[j]
    return Solution

def Eigenvalue(mat,vec, Iteration):
    EVArray=[]
    for i in range(Iteration):
        vec = matrix_vector(size, mat, vec)
        eigenvector = vec / np.linalg.norm(vec)
        eigenvalue = np.dot(matrix_vector(size, mat, vec),vec
                                           ) / np.dot(vec, vec)
        EVArray.append(eigenvalue)
        Aitken_Eigenvalue = Aitken(EVArray)
        if i >= 3 and  Aitken_Eigenvalue-EVArray[-1]< 1e-8  :
            eigenvalue = Aitken_Eigenvalue
            break

    return eigenvalue,eigenvector
```

```
eigenvalue_1,eigenvector_1 = Eigenvalue(M,V, iteration)
eigenvalue_2,eigenvector_2 = Eigenvalue(np.linalg.inv(M),V,
                               iteration)
eigenvalue_2 = 1/eigenvalue_2
print("Largest eigenvalue and corresponding eigenvektor:",
                               eigenvalue_1, eigenvector_1)
print("Smallest eigenvalue and corresponding eigenvektor:",
                               eigenvalue_2, eigenvector_2)
```

4 Problem IV

The following code calculates the solution for a system of equations like given in the task.

```
import numpy as np

# Generate a random NxN matrix and vector
n = 2
A = np.round(np.random.rand(n, n), 2)
b = np.round(np.random.rand(n, 1), 2)


print("Matrix:")
print(A)
print("Vector:")
print(b)
print("\n")

Max_iterations = 15000

# Get dimensions of our NxN matrix
rows, columns = A.shape


def overrel(m, vector, iterations, w, tol=1e-6):  #
                                Overrelaxation function
    x = vector.copy()  # Use a copy of the initial guess to
                                    avoid modifying the
                                    original
    for j in range(iterations):
        x_old = x.copy()
        for i in range(rows):
            if m[i][i] != 0:
                # Calculate the factor for iteration
                factor = w / m[i][i]

                # Calculate the sums for the iteration part
                sum1 = sum(m[i][l] * x[l] for l in range(i))
                sum2 = sum(m[i][l] * x_old[l] for l in range(
                                                i, rows))

                # Update x based on our sums and the factor
                x[i] = x_old[i] + factor * (vector[i] - sum1
                                                - sum2)
```

```
        # Check for convergence
        if np.linalg.norm(x - x_old, ord=np.inf) < tol:
            print(f"Convergence achieved after {j + 1}
                                                iterations with w
                                                = {w}")
            return x

    print(f"No convergence after {iterations} iterations with
                                        w = {w}")
    return None

# Calculate the function with different overrelaxation
                                    factors
for factor in np.arange(1, 2, 0.1):
    result = overrel(A, b, Max_iterations, factor)
    if result is not None:
        print(f"Solution vector after over-relaxation (w = {
                                         factor}):")
        print(result)
        print(" ")
```

However, we encountered a problem with this code for which we were unable to find a quick solution. With higher-dimensional matrices, our computer was no longer able to calculate the solutions to our problem if they have been too large.