retselis_set2

January 30, 2021

1 Problem Set #2

Computational Mathematics (ΥΦΥ101)

Implemented by: **Anastasios-Faidon Retselis (AEM: 4394)**

2 Exercise 1

2.1 Problem Statement:

Represent the linear system:

$$\begin{aligned} x_1 - x_2 + 2x_3 - x_4 &= -8 \\ 2x_1 - 2x_2 + 3x_3 - 3x_4 &= -20 \\ x_1 + x_2 + x_3 &= -2 \\ x_1 - x_2 + 4x_3 + 3x_4 &= 4 \end{aligned}$$

as an augmented matrix and use Gaussian Elimination to find its solution.

2.2 Solution

[1]:
```
import numpy as np

def gauss_elimination(matrix):
    # Performs gauss elimination
    # Input is a linear system as an augmented matrix
    # Returns the solution vector

    # Forward Elimination
    x = np.zeros(4)
    n = len(matrix)
    for k in range(0,n):
        aug_mat = pivoting(matrix,k)
        for i in range(k+1,n):
            factor = matrix[i][k]/matrix[k][k]
            for j in range(0,n):
                matrix[i][j] = matrix[i][j] - factor * matrix[k][j]
```

```
            matrix[i][n] = matrix[i][n] - matrix[k][n]*factor

    # Backward substitution
    for i in range(n-1,-1,-1):
        summation = 0
        for j in range(i,n):
            summation = summation + matrix[i][j]*x[j]
        x[i] = (matrix[i][n]-summation)/matrix[i][i]
    return(x)

def pivoting(matrix,k):
    # Performs pivoting for gauss elimination
    # Input is a linear system as an augmented matrix and counter k
    # Returns the matrix after pivoting is performed
    n = len(matrix)
    p = k
    big = np.absolute(matrix[k][k])
    for i in range(k+1,n):
        dummy = np.absolute(matrix[i][k])
        if dummy>big:
            big = dummy
            p = i
    if p!=k:
        for j in range(k,n):
            dummy = matrix[p][j]
            matrix[p][j] = matrix[k][j]
            matrix[k][j] = dummy
        dummy = matrix[p][n]
        matrix[p][n] = matrix[k][n]
        matrix[k][n] = dummy
    return(matrix)

# Driving code

aug_mat = np.array([[1.,-1.,2.,-1.,-8.],
                    [2.,-2.,3.,-3.,-20.],
                    [1.,1.,1.,0.,-2.],
                    [1.,-1.,4.,3.,4.]])

solution = gauss_elimination(aug_mat)

print('\nSolution vector')
print(solution)

print('\nThe solution is:\n')
for i in range(len(solution)):
    print("x_", i+1, "=",round(solution[i]))
```

```
Solution vector
[-7.  3.  2.  2.]

The solution is:

x_ 1 = -7.0
x_ 2 = 3.0
x_ 3 = 2.0
x_ 4 = 2.0
```

3 Exercise 2

3.1 Problem Statement:

Values for $f(x) = xe^x$ are given in the following table. Use numerical differentiation (aim for errors of $O(h^2)$) to complete the table and compare your results with the actual values. Then numerically integrate the values using Simpson's h/3 rule (The simple and the multiple form if possible).

x	f(x)	f'(x)
1.8	10.889365	
1.9	12.703199	
2.0	14.77112	
2.1	17.14957	
2.2	19.855030	

3.2 Solution

[2]:
```
import numpy as np

def forward_num_derivative(matrix, h, rows, cols):
    # Forward finite-divided-difference first derivative
    # Achieves O(h^2) accuracy
    matrix[0][cols-1] =␣
 ↪(-matrix[0+2][cols-2]+4*matrix[0+1][cols-2]-3*matrix[0][cols-2])/(2*h)
    return(matrix)

def backward_num_derivative(matrix, h, rows, cols):
    # Backward finite-divided-difference first derivative
    # Achieves O(h^2) accuracy
    matrix[rows-1][cols-1] =␣
 ↪(3*matrix[rows-1][cols-2]-4*matrix[rows-2][cols-2]+matrix[rows-3][cols-2])/
 ↪(2*h)
    return(matrix)
```

```
def centered_num_derivative(matrix, h, rows, cols):
    # Centered finite-divided-difference first derivative
    # Achieves O(h^2) accuracy
    min_row = 1
    max_row = rows - 2
    for i in range(min_row, max_row+1):
        matrix[i][cols-1] = (matrix[i+1][cols-2]-matrix[i-1][cols-2])/(2*h)
    return(matrix)

def num_differentiation(matrix):
    # Achieves O(h^2) accuracy
    # Input is a matrix with first column being x value, second column being␣
 ↪f(x) values
    # Returns the matrix with an additional column containing the corresponding␣
 ↪f'(x) values
    dim = matrix.shape
    rows = dim[0]
    cols = dim[1]
    if (rows < 3):
        raise ValueError('Less than three data points, cannot perform forward/
 ↪backward/centered num differentiation')
    zero_v = np.zeros(rows)
    matrix = np.insert(matrix,cols,zero_v,axis=1)
    cols += 1
    h = matrix[1][0]-matrix[0][0]
    for i in range(1,rows):
        if (matrix[i][0]-matrix[i-1][0]-h<0):
            print('Warning: Data are not equally spaced!')
    matrix = forward_num_derivative(matrix, h, rows, cols)
    matrix = backward_num_derivative(matrix, h, rows, cols)
    matrix = centered_num_derivative(matrix, h, rows, cols)
    return(matrix)

def simpson_simple_form(matrix):
    # Simple form of Simpson's h/3 rule
    # Input is a matrix with first column being x value, second column being␣
 ↪f(x) values
    # Computes the integral at interval [x_min, x_max]
    # Returns integral value I
    dim = matrix.shape
    rows = dim[0]
    if ((rows % 2) == 0):
        raise ValueError('Simpson\'s h/3 rule only works for an even number of␣
 ↪segments!')
    cols = dim[1]
    middle = int(np.ceil(rows/2))
```

```
    h = matrix[rows-1][cols-2] - matrix[0][cols-2]
    I =␣
 ↪h*(matrix[0][cols-1]+(4*matrix[middle-1][cols-1])+matrix[rows-1][cols-1])/6
    return(I)

def simpson_multiple_form(matrix):
    # Multiple form of Simpson's h/3 rule
    # Input is a matrix with first column being x value, second column being␣
 ↪f(x) values
    # Computes the integral at interval [x_min, x_max]
    # Returns integral value I
    dim = matrix.shape
    rows = dim[0]
    if ((rows % 2) == 0):
        raise ValueError('Simpson\'s h/3 rule only works for an even number of␣
 ↪segments!')
    cols = dim[1]
    middle = int(np.ceil(rows/2))
    h = matrix[rows-1][cols-2] - matrix[0][cols-2]
    odd_sum = 0
    even_sum = 0
    for i in range(1,rows,2): # stops at n-1
        odd_sum = odd_sum + matrix[i][cols-1]
    for i in range(2,rows-1,2): # stops at n-2
        even_sum = even_sum + matrix[i][cols-1]
    f_x0 = matrix[0][cols-1]
    f_xn = matrix[rows-1][cols-1]
    I = h*(f_x0 + 4*odd_sum + 2*even_sum + f_xn)/(3*(rows-1))
    return(I)


input_matrix = np.array([[1.8,10.889365],
                         [1.9,12.703199],
                         [2.0,14.778112],
                         [2.1,17.148957],
                         [2.2,19.855030],])

matrix_with_first_derivative = num_differentiation(input_matrix)
print('Numerical differentiation results:\n')
print(matrix_with_first_derivative)

integral_simple = simpson_simple_form(input_matrix)
integral_multiple = simpson_multiple_form(input_matrix)
print('\n Numerical integration results:')
print('\n Using simple form of Simpson\'s h/3 rule, I = %.7f' % integral_simple)
print('\n Using multiple form of Simpson\'s h/3 rule, I = %.7f' %␣
 ↪integral_multiple)
```

```
Numerical differentiation results:

[[ 1.8      10.889365 16.832945]
 [ 1.9      12.703199 19.443735]
 [ 2.       14.778112 22.22879 ]
 [ 2.1      17.148957 25.38459 ]
 [ 2.2      19.85503  28.73687 ]]

 Numerical integration results:

 Using simple form of Simpson's h/3 rule, I = 5.9904562

 Using multiple form of Simpson's h/3 rule, I = 5.9903081
```

3.3 Conclusion

Based on the output above, we can complete the table below. The actual values of $f'(x)$ are also included.

x	f(x)	f'(x) (computed)	f'(x) (actual)
1.8	10.889365	16.832945	16.939013
1.9	12.703199	19.443735	19.389094
2.0	14.77112	22.22879	22.167168
2.1	17.14957	25.38459	25.315127
2.2	19.855030	28.7367	28.880043

We notice that for the centered finite-divided-difference formulas, the error is indeed $O(h^2)$. However, for $x = 1.8$ and $x = 2.2$ we notice that the forward and backward finite-divided-difference formulas do no to achieve $O(h^2)$, instead they manage to achieve an error of $O(h)$.

4 Exercise 3

4.1 Problem Statement:

Use the power method to determine the largest eigenvalue of

$$\begin{bmatrix} 4 & 1 & 2 & 1 \\ 1 & 7 & 1 & 0 \\ 2 & 1 & 4 & -1 \\ 1 & 0 & -1 & 3 \end{bmatrix}$$

What is the corresponding eigenvector?

4.2 Solution

[3]:
```
import numpy as np

def power_method(matrix, desired_acc, max_iter):
    # Computes the largest eigenvalue of a matrix using the power method
    # Input is the desired matrix, desired accuracy and maximum amount of␣
 ↪iterations allowed
    # Returns the largest eigenvalue
    print('Desired accuracy = ', desired_acc)
    print('Now computing max eigenvalue using power method...')
    n = len(matrix)
    x = np.zeros(n)
    for i in range(0,n):
        summation = 0
        for j in range(0,n):
            summation = summation + matrix[i][j]
        x[i] = summation
    # Normalize x and compute first eigenvalue
    lambda_previous = np.max(x)
    for i in range(0,n):
        x[i] = x[i]/lambda_previous
    # Main loop
    for iteration in range(0,max_iter):
        temp = np.matmul(matrix,x)
        x = temp
        lambda_now = np.max(x)
        for i in range(0,n):
            x[i] = x[i]/lambda_now
        e_a = np.abs((lambda_now-lambda_previous)/lambda_now)*100
        if e_a < desired_acc:
            print('Accuracy achieved!')
            break
        if iteration == max_iter-1:
            print('Maximum iteration reached without achieving accuracy!')
            break
        lambda_previous = lambda_now

    print('Largest eigenvalue is %.2f' % lambda_now)
    return(lambda_now)

def compute_eigenvector(matrix, eigenvalue):
    # Computes the eigenvector of a matrix for a given eigenvalue of the matrix
    # Input is the desired matrix and the corresponding eigenvalue
    # Returns the eigenvector
    n = len(matrix)
    ev = np.zeros(n)
```

```
    eigenvalue = int(eigenvalue)
    for i in range(0,n):
        matrix[i][i] = matrix[i][i] - eigenvalue
    # Since the problem of finding eigenvectors is like solving a linear␣
↪system, we will use the gauss elimination method to compute the matrix
    # We will transform our matrix to an augmented one and we will use the code␣
↪from exercise 1
    # Due to the fact that we are dealing with a singular matrix, assume x=1
    matrix = np.insert(matrix,n,ev,axis=1)
    k = 1  # Assume x=1
    for i in range(0,n):
        matrix[i][n] = matrix[i][n] - matrix[i][k-1]
        matrix[i][k-1] = matrix[i][k-1] - matrix[i][k-1]
    matrix = np.delete(matrix,k-1,1)
    matrix = np.delete(matrix,k-1,0)
    ev = gauss_elimination(matrix)
    ev = np.insert(ev,0,1) # Attach first element (x=1) to the eigenvector
    print('\nThe corresponding eigenvector is:')
    print(ev)

def gauss_elimination(matrix):
    # Performs gauss elimination
    # Input is a linear system as an augmented matrix
    # Returns the solution vector

    # Forward Elimination
    n = len(matrix)
    x = np.zeros(n)
    for k in range(0,n):
        aug_mat = pivoting(matrix,k)
        for i in range(k+1,n):
            factor = matrix[i][k]/matrix[k][k]
            for j in range(0,n):
                matrix[i][j] = matrix[i][j] - factor * matrix[k][j]
            matrix[i][n] = matrix[i][n] - matrix[k][n]*factor
    # Backward substitution
    for i in range(n-1,-1,-1):
        summation = 0
        for j in range(i,n):
            summation = summation + matrix[i][j]*x[j]
        x[i] = (matrix[i][n]-summation)/matrix[i][i]
    return(x)

def pivoting(matrix,k):
    # Performs pivoting for gauss elimination
    # Input is a linear system as an augmented matrix and counter k
    # Returns the matrix after pivoting is performed
```

```
    n = len(matrix)
    p = k
    big = np.absolute(matrix[k][k])
    for i in range(k+1,n):
        dummy = np.absolute(matrix[i][k])
        if dummy>big:
            big = dummy
            p = i
    if p!=k:
        for j in range(k,n):
            dummy = matrix[p][j]
            matrix[p][j] = matrix[k][j]
            matrix[k][j] = dummy
        dummy = matrix[p][n]
        matrix[p][n] = matrix[k][n]
        matrix[k][n] = dummy
    return(matrix)


# Driving code

alpha =  np.array([[4.,1.,2.,1.],
                   [1.,7.,1.,0.],
                   [2.,1.,4.,-1.],
                   [1.,0.,-1.,3.]])
accuracy = pow(10,-3)
maximum_iterations = 100
max_eigenvalue = power_method(alpha, accuracy, maximum_iterations)
compute_eigenvector(alpha, max_eigenvalue)
```

```
Desired accuracy =  0.001
Now computing max eigenvalue using power method…
Accuracy achieved!
Largest eigenvalue is 8.00

The corresponding eigenvector is:
[ 1.  2.  1. -0.]
```