Task 1

I have created the following five routines to compute the following linear algebra operations on vectors, they are all documented here in the Software Manual.

1. Vector Addition

```python
def VecAdd(arr1, arr2):
    sum = []
    for i in range(len(arr1)):
        sum.append(arr1[i] + arr2[i])
    return sum
```

2. Vector Subtraction

```python
def VecSub(arr1, arr2):
    diff = []
    for i in range(len(arr1)):
        diff.append(arr1[i] + arr2[i])
    return diff
```

3. Scalar Multiplication for Vectors

```python
def ScaMul(c, arr):
    ScalarProduct = c * arr
    return ScalarProduct
```

4. Dot Product for Two Vectors of the Same Length

```python
def DotPro(arr1, arr2):
    if len(arr1) != len(arr2):
        print('two vectors are not the same length')
    sum = 0
    for i in range(len(arr1)):
        sum += arr1[i] * arr2[i]
    return sum
```

5. Outer Product for Two Vectors of the Same Length

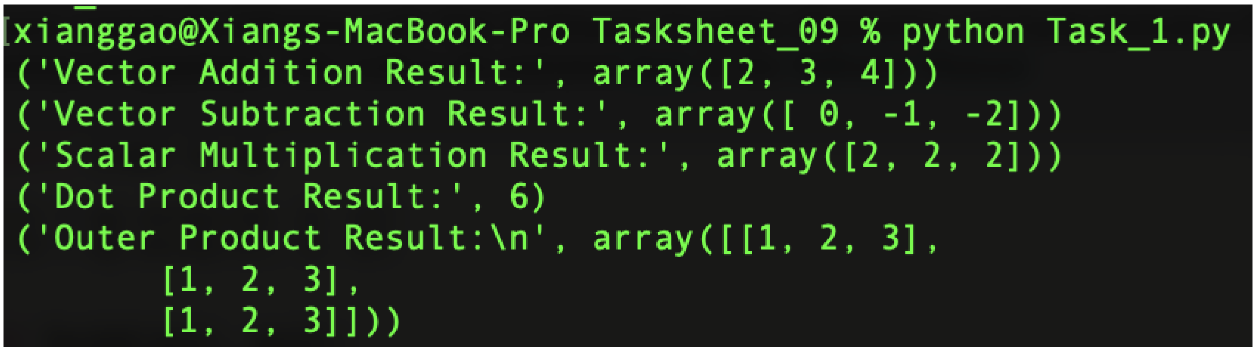
```python
import numpy as np

def OutPro(arr1, arr2):
    if len(arr1) != len(arr2):
        print('two vectors are not the same length')

    OutPro = np.outer(arr1, arr2)
    return OutPro
```

I have also created the following testing code that test each of the routines,

```python
import numpy as np
from VecAdd import VecAdd
from VecSub import VecSub
from ScaMul import ScaMul
from DotPro import DotPro
from OutPro import OutPro

# testing vectors and constant
arr1 = np.array([1, 1, 1])
```

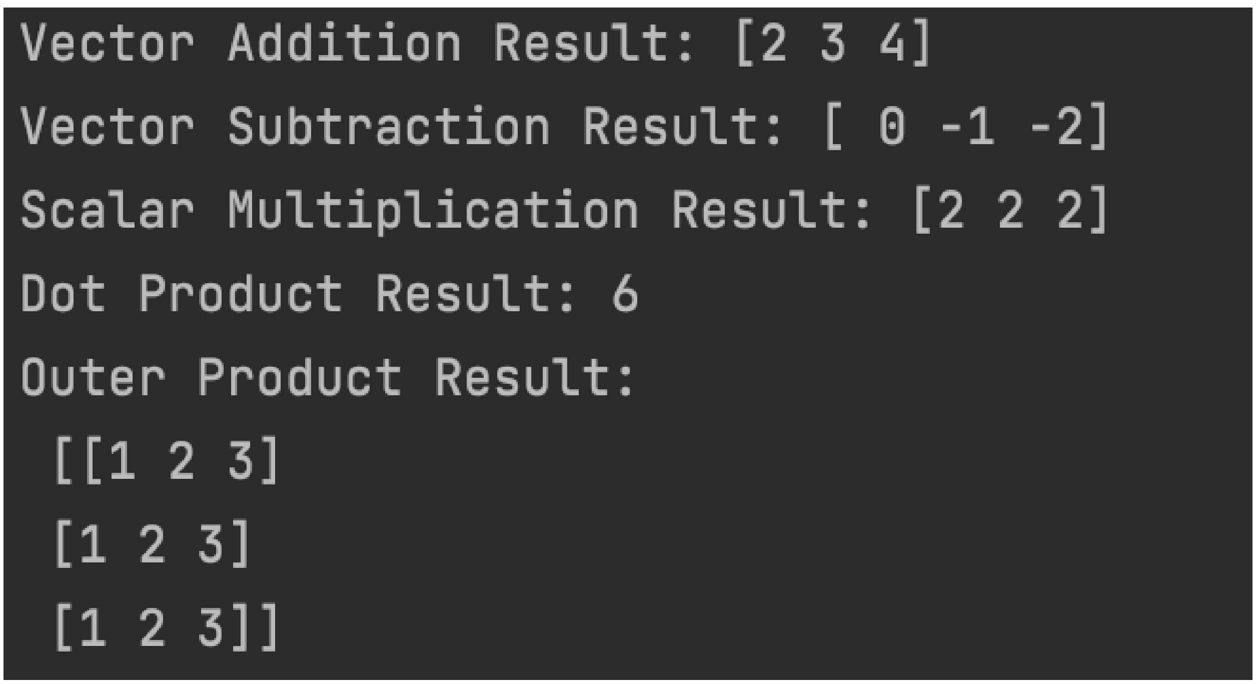
```python
arr2 = np.array([1, 2, 3])
C = 2

print('Vector Addition Result:', VecAdd(arr1, arr2))
print('Vector Subtraction Result:', VecSub(arr1, arr2))
print('Scalar Multiplication Result:', ScaMul(C, arr1))
print('Dot Product Result:', DotPro(arr1, arr2))
print('Outer Product Result:\n', OutPro(arr1, arr2))
```

Which has the following output:



```
[xianggao@Xiangs-MacBook-Pro Tasksheet_09 % python Task_1.py
('Vector Addition Result:', array([2, 3, 4]))
('Vector Subtraction Result:', array([ 0, -1, -2]))
('Scalar Multiplication Result:', array([2, 2, 2]))
('Dot Product Result:', 6)
('Outer Product Result:\n', array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]))
```

Figure 1. The Test Result of Each Routine from Terminal.



```
Vector Addition Result: [2 3 4]
Vector Subtraction Result: [ 0 -1 -2]
Scalar Multiplication Result: [2 2 2]
Dot Product Result: 6
Outer Product Result:
 [[1 2 3]
 [1 2 3]
 [1 2 3]]
```

Figure 1.5. The Test Result of Each Routine from IDE.

Task 2

I have created the following six routines to compute the following linear algebra operations on vectors, they are all documented here in the Software Manual.

1. The magnitude of a vector - (l_1) norm version

```python
import numpy as np
```

```python

def L1_Norm (arr1):
    norm = np.linalg.norm(arr1, ord=1)
    return norm
```

2. The magnitude of a vector - (l_2) norm version

```python
import numpy as np

def L2_Norm (arr1):
    norm = np.linalg.norm(arr1, ord=2)
    return norm
```

3. The magnitude of a vector - (l_∞) norm version

```python
import numpy as np

def Linf_Norm (arr1):
    norm = np.linalg.norm(arr1, ord=np.inf)
    return norm
```

4. The error between vectors - (l_1) norm version.

```python
import numpy as np

def L1_Norm_Error (arr1, arr2):
    norm = np.linalg.norm((arr1 - arr2), ord=1)
    return norm
```

5. The error between vectors - (l_2) norm version.

```python
import numpy as np

def L2_Norm_Error (arr1, arr2):
    norm = np.linalg.norm((arr1 - arr2), ord=2)
    return norm
```

6. The error between vectors - (l_∞) norm version.

```python
import numpy as np

def Linf_Norm_Error (arr1, arr2):
    norm = np.linalg.norm((arr1 - arr2), ord=np.inf)
    return norm
```

I have also created the following testing code that test each of the routines,

```python
import numpy as np
from L1_Norm import L1_Norm
from L2_Norm import L2_Norm
from Linf_Norm import Linf_Norm
from L1_Norm_Error import L1_Norm_Error
from L2_Norm_Error import L2_Norm_Error
from Linf_Norm_Error import Linf_Norm_Error

# testing vectors
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([2, 3, 4, 5])

print('L1 Norm Result:', L1_Norm(arr1))
```

```
print('L2 Norm Result:', L2_Norm(arr1))
print('L_infinity Norm Result:', Linf_Norm(arr1))
print('L1 Norm Result for Errors:', L1_Norm_Error(arr1, arr2))
print('L2 Norm Result for Errors:', L2_Norm_Error(arr1, arr2))
print('L_infinity Norm Result for Errors:', Linf_Norm_Error(arr1, arr2))
```

Which has the following output:

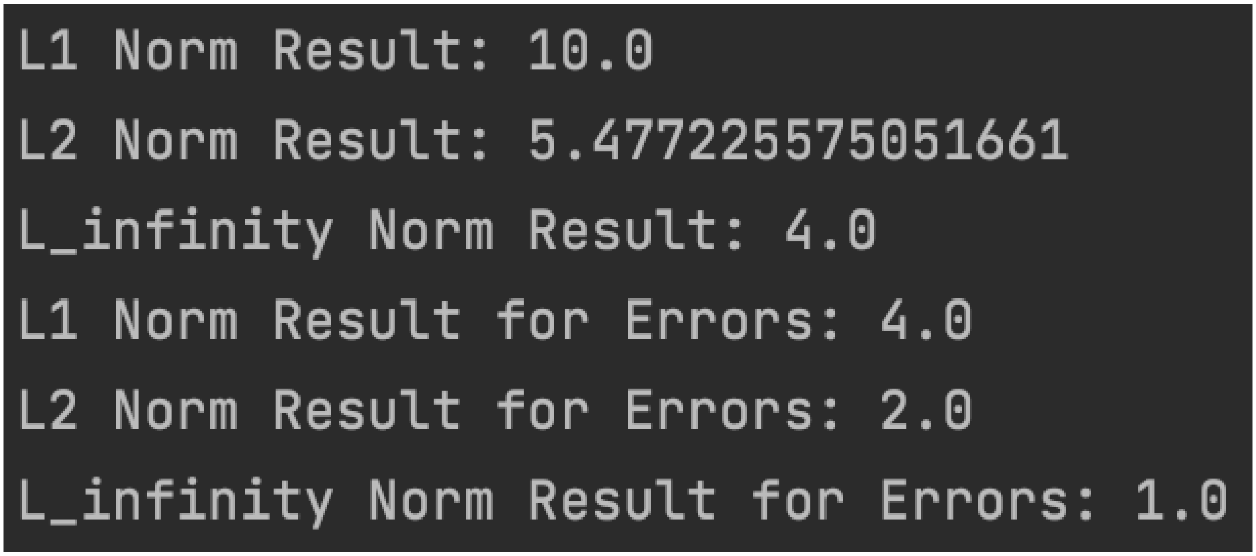


Figure 2. The Test Result of Each Routine from IDE.

Task 3

I have created the following six routines to compute the following linear algebra operations on vectors, they are all documented here in the Software Manual.

1. Matrix Addition

```
import numpy as np

def MatAdd (mat1, mat2):
    sum = np.add(mat1, mat2)
    return sum
```

2. Matrix Subtraction

```
import numpy as np

def MatSub (mat1, mat2):
    diff = np.subtract(mat1, mat2)
    return diff
```

3. Scalar Multiplication for a Matrix

```
def MatScaMul(c, mat1):
    ScalarProduct = c * mat1
    return ScalarProduct
```

4. The Transpose of a Matrix.

```python
import numpy as np

def MatTran (mat1):
    transpose = np.transpose(mat1)
    return transpose
```

5. The Product of a Rectangular Matrix and Vector.

```python
import numpy as np

def MatVecPro (mat, vec):
    product = np.dot(mat, vec)
    return product
```

6. The Product of Two Rectangular Matrices.

```python
def MatMatPro (mat1, mat2):
    res = [[0 for x in range(len(mat1))] for y in range(len(mat2))]
    for i in range(len(mat1)):
        for j in range(len(mat2[0])):
            for k in range(len(mat2)):
                # resulted matrix
                res[i][j] += mat1[i][k] * mat2[k][j]
    return res
```

I have also created the following testing code that test each of the routines,

```python
import numpy as np
from matAdd import MatAdd
from matSub import MatSub
from matScaMul import MatScaMul
from matTran import MatTran
from matVecPro import MatVecPro
from matMatPro import MatMatPro

# testing matrices, vector, and constant
mat1 = np.array([[2, -7, 5], [-6, 2, 0]])
mat2 = np.array([[5, 8, -5], [3, 6, 9]])
matR = np.array([[5, 1, 3], [1, 1, 1], [1, 2, 1]])
arr = np.array([1, 2, 3])
C = 2

print('Matrix Addition Result:\n', MatAdd(mat1, mat2))
print('Matrix Subtraction Result:\n', MatSub(mat1, mat2))
print('Scalar Multiplication for a Matrix Result:\n', MatScaMul(C, mat1))
print('The Transpose of a Matrix Result:\n', MatTran(mat1))
print('The Product of a Rectangular Matrix and Vector Result:\n', MatVecPro(matR, arr))
print('The Product of Two Rectangular Matrices Result:\n', MatMatPro(matR, matR))
```

Which has the following output:

```
Matrix Addition Result:
 [[ 7  1  0]
 [-3  8  9]]
Matrix Subtraction Result:
 [[ -3 -15  10]
 [ -9  -4  -9]]
Scalar Multiplication for a Matrix Result:
 [[  4 -14  10]
 [-12   4   0]]
The Transpose of a Matrix Result:
 [[ 2 -6]
 [-7  2]
 [ 5  0]]
The Product of a Rectangular Matrix and Vector Result:
 [16  6  8]
The Product of Two Rectangular Matrices Result:
 [[29, 12, 19], [7, 4, 5], [8, 5, 6]]
```

Figure 3. The Test Result of Each Routine from IDE.

Task 4

I have written the following code to implement Jacobi Iteration for solving linear systems of equations, it's documented here in the Software Manual.

```python
# Defining equations to be solved
# in diagonally dominant form
f1 = lambda x, y, z: (17 - y + 2 * z) / 20
f2 = lambda x, y, z: (-18 - 3 * x + z) / 20
f3 = lambda x, y, z: (25 - 2 * x + 3 * y) / 20

# Initial setup
x0 = 0
y0 = 0
z0 = 0
count = 1

# Reading tolerable error
e = float(input('Enter tolerable error: '))

# Implementation of Jacobi Iteration
```

```python
print('\nCount\tx\ty\tz\n')

condition = True

while condition:
    x1 = f1(x0, y0, z0)
    y1 = f2(x0, y0, z0)
    z1 = f3(x0, y0, z0)
    print('%d\t%0.4f\t%0.4f\t%0.4f\n' % (count, x1, y1, z1))
    e1 = abs(x0 - x1);
    e2 = abs(y0 - y1);
    e3 = abs(z0 - z1);

    count += 1
    x0 = x1
    y0 = y1
    z0 = z1

    condition = e1 > e and e2 > e and e3 > e

print('\nSolution: x=%0.3f, y=%0.3f and z = %0.3f\n' % (x1, y1, z1))
```

Which has the following output:

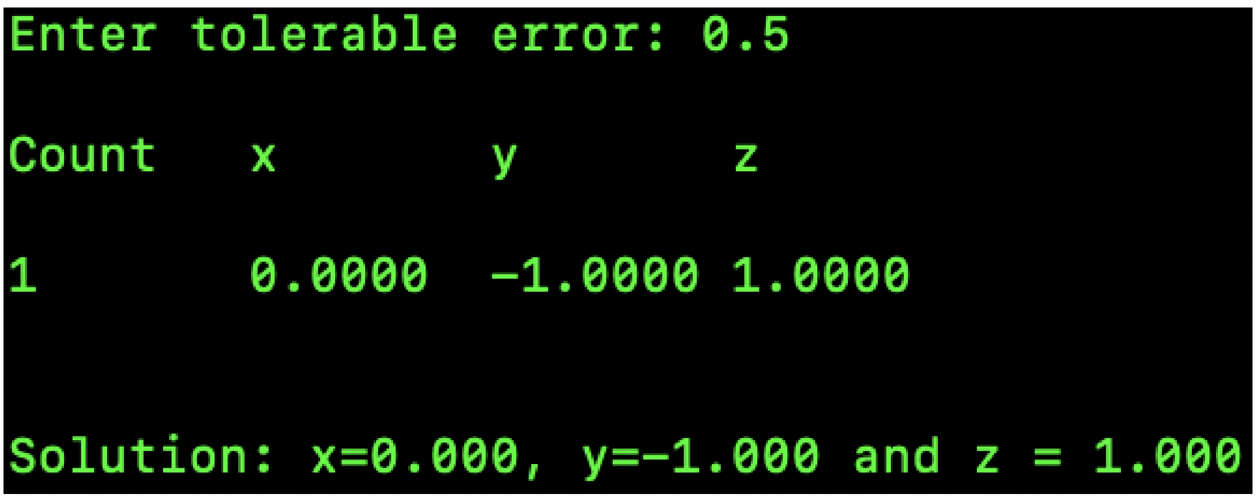


Figure 4. The Test Result of the Jacobi Method from Terminal.

Task 5

I have written the following code to compare the results for the Gaussian elimination to the results from Jacobi Iteration for solving linear systems of equations, it's documented here in the Software Manual.

```python
import numpy as np

ones = [1 for i in range(100)]
solution = np.array(1 for i in range(100))

def jacobi(A, b, tolerance=1e-10, max_iterations=10000):
    x = np.zeros_like(b, dtype=np.double)
    T = A - np.diag(np.diagonal(A))
    for k in range(max_iterations):
        x_old = x.copy()
        x[:] = (b - np.dot(T, x)) / np.diagonal(A)
        error = np.linalg.norm(x - x_old, ord=np.inf) / np.linalg.norm(x, ord=np.inf)
        if error < tolerance:
            break
    return error
```

```python


def gauss(A, b, tolerance=1e-10, max_iterations=10000):
    x = np.zeros_like(b, dtype=np.double)
    # Iterate
    for k in range(max_iterations):
        x_old = x.copy()
        # Loop over rows
        for i in range(A.shape[0]):
            x[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, (i + 1):], x_old[(i + 1):])) / A[i, i]
        # Stop condition
        error = np.linalg.norm(x - x_old, ord=np.inf) / np.linalg.norm(x, ord=np.inf)
        if error < tolerance:
            break
    return error

print('The Error for the Jacobi Iteration is:', jacobi(ones, solution))
print('The Error for the Gaussian Method is:', gauss(ones, solution))
```

Which has the following output:

```
The Error for the Jacobi Iteration is: 1.67e-11
The Error for the Gaussian Method is: 0.0

Process finished with exit code 0
```

Figure 5. The Comparison of the Results.

Task 6

In this website[1] I found, the Gauss-Seidel method has a faster rate of convergence than the Jacobi Iteration.

The element-wise formula for the Gauss–Seidel method is extremely similar to that of the Jacobi method. However, in this website[2] I found, unlike the Jacobi method, the computations for each element are generally much harder to implement in parallel, since they can have a very long critical path, and are thus most feasible for sparse matrices. Furthermore, the values at each iteration are dependent on the order of the original equations.

[1] https://www.math-linux.com/mathematics/linear-systems/article/gauss-seidel-method
[2] https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_IterativeSolvers.html