Q1.
The RowVectorFloat class is a Python class that implements basic operations on vectors of floating-point numbers. such as addition, subtraction, and scalar multiplication.

The constructor method of the RowVectorFloat class takes a list of floating-point numbers as an argument and initializes a new vector object. If the input argument is not a list, or if any of the elements in the list are not floating-point numbers, an exception is raised. The vector object is stored internally as an attribute of the class.
Added these methods:

- __str__(self): Returns a string representation of the vector object.
- __len__(self): Returns the length of the vector object.
- __getitem__(self, i): Allows the vector object to be indexed like a list.
- __setitem__(self, i, value): Allows the value at a particular index in the vector to be set.
- __mul__(self, r1): Allows two vectors to be multiplied together or a scalar to be multiplied by a vector.
- __rmul__(self, scalar): Allows a vector to be multiplied by a scalar.
- __add__(self, r1): Allows two vectors to be added together or a scalar to be added to a vector.
- __radd__(self, scalar): Allows a vector to be added to a scalar.

To make object that is a linear combination of other RowVectorFloat objects

Example usage

```
if __name__ == "__main__":

  r1 = RowVectorFloat([1, 2 , 4])
  r2 = RowVectorFloat([1, 1 , 1])
  r3 = 2*r1 + (-3)*r2
  print(r3)
```

Output :-

```
kaustubh@kaustubh:~/CMA$ /bin/python3 /home/kaustubh/CMA/week3/Q1.py
-1 1 5
```

Q2
SquareMatrixFloat have methods for generating a random symmetric matrix ,converting a matrix to its row echelon form, and checking whether the matrix is diagonally dominant and methods to solve the linear system using Jacobi and Gauss-Siedel methods.

The SquareMatrixFloat class has the following methods:

- sampleSymmetric(self) method that generates a random symmetric matrix with positive diagonal elements and stores it in the object.
- toRowEchelonForm(self) method that converts the matrix to its row echelon form.
- isDRDominant(self) method that checks if the matrix is diagonally dominant.
- jSolve(self, givenlist, NumIterations) method that solves the linear system using Jacobi method. It takes a list givenlist of values for the right-hand side of the equation and the number of iterations NumIterations as inputs. It returns the error and the values of the variables.
- gsSolve(self, givenlist, NumIterations) method that solves the linear system using Gauss-Siedel method. It takes a list givenlist of values for the right-hand side of the equation and the number of iterations NumIterations as inputs. It returns the error and the values of the variables.

The jSolve and gsSolve methods use the isDRDominant method to ensure that the matrix is diagonally dominant, which is a condition for the convergence of these methods. They also use. The jacobi and gsSolve method uses two lists itrs and prev_itr to store the values of the variables at the current and previous iterations, respectively.

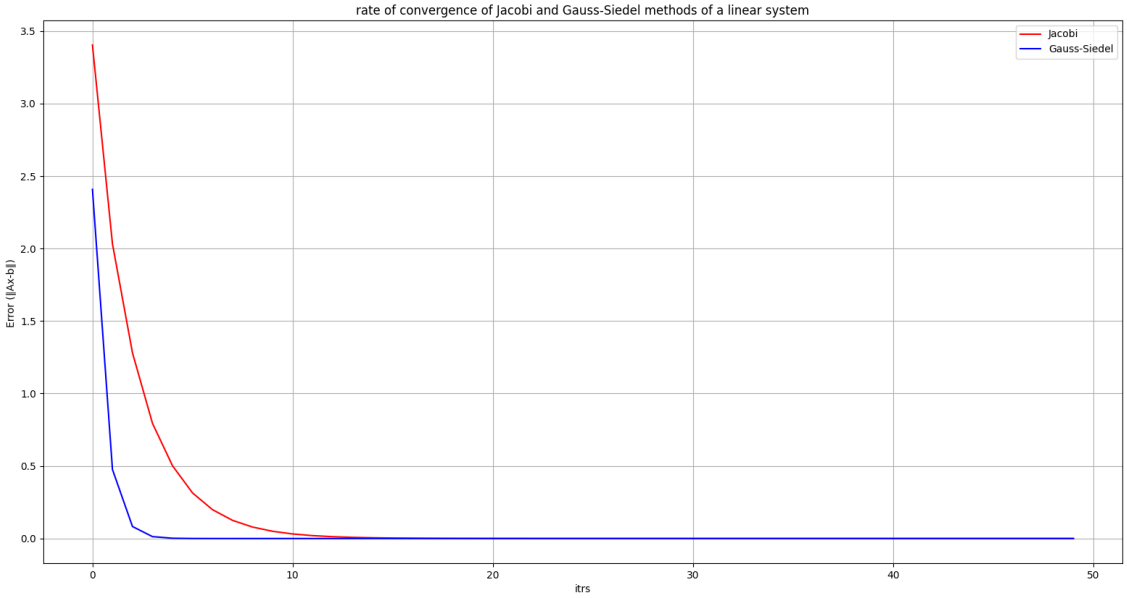
```
if __name__ == "__main__":
    print("1...")
    s = SquareMatrixFloat(3)
    print(s)
    print("\n2''''")
    s = SquareMatrixFloat(4)
    s.sampleSymmetric()
    print(s)
    s.toRowEchelonForm()
    print(s)
    print("\n3..")
    s = SquareMatrixFloat(4)
    s.sampleSymmetric()
    print(s.isDRDominant())
    (e, x) = s.gsSolve([1, 2, 3, 4], 10)
    print(x)
    print(e)
```

Output :-
One of the output

```
False
Traceback (most recent call last):
  File "/home/kaustubh/CMA/week3/Q2.py", line 146, in <module>
    (e, x) = s.gsSolve([1, 2, 3, 4], 10)
  File "/home/kaustubh/CMA/week3/Q2.py", line 108, in gsSolve
    raise Exception("<class 'Exception'>\nNot solving because convergence is not guranteed.")
Exception: <class 'Exception'>
Not solving because convergence is not guranteed.
```

```
1..
the matrix is:
0 0 0
0 0 0
0 0 0


2'''
the matrix is:
3.798 0.324 0.595 0.633
0.324 2.343 0.632 0.859
0.595 0.632 1.783 0.795
0.633 0.859 0.795 1.165

the matrix is:
1.0 0.09 0.16 0.17
0.0 1.0 0.25 0.35
0.0 0.0 1.0 0.32
-0.0 -0.0 -0.0 1.0


3..
True
the matrix is:
3.17 0.558 0.013 0.108
0.558 2.743 0.565 0.964
0.013 0.565 3.087 0.017
0.108 0.964 0.017 2.24

[0.2802902471984706, -0.17989288055201258, 0.9934176786163226, 1.8420791471509945]
[1.9799605642511358, 0.4807382891775617, 0.1029963277872403, 0.0212026699786583 63, 0.004352314694137798, 0.0008932297112939774, 0.0001833158250593 5935, 3.762151935721169e-05, 7.720984381117173e-06, 1.5845611915485
163e-06]
```

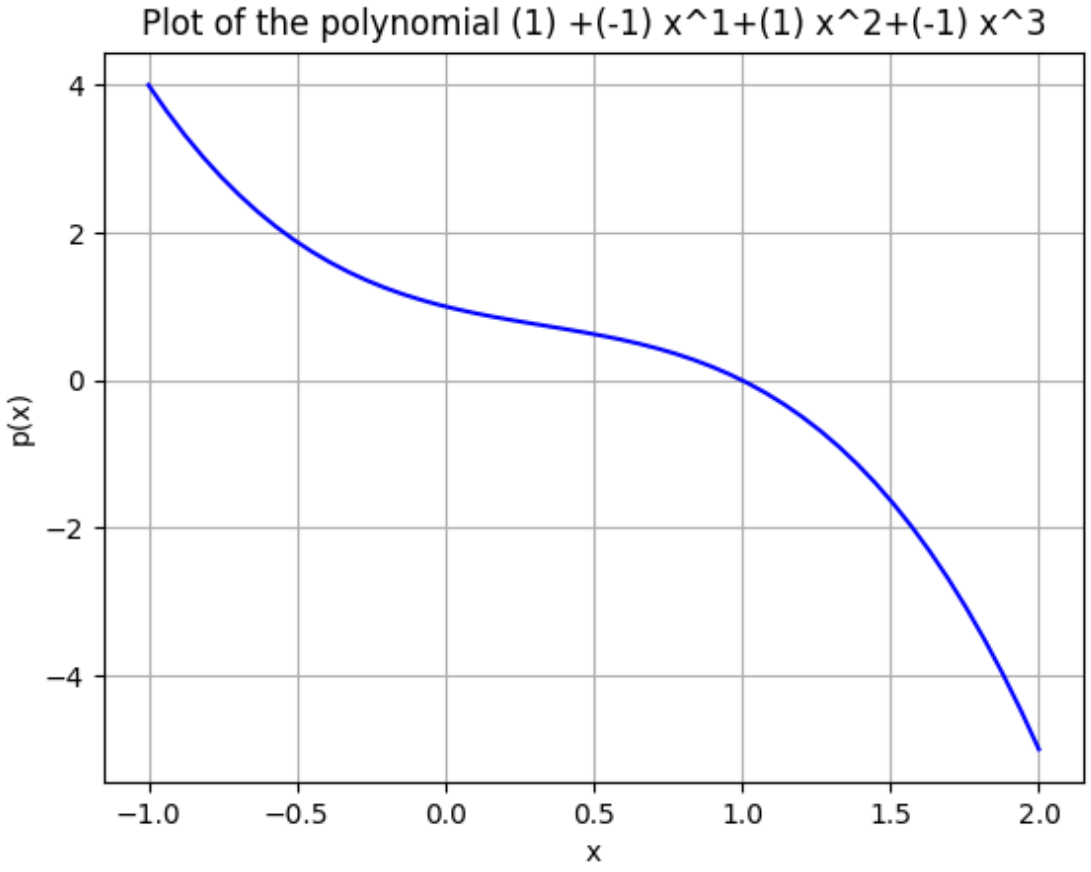
Q3



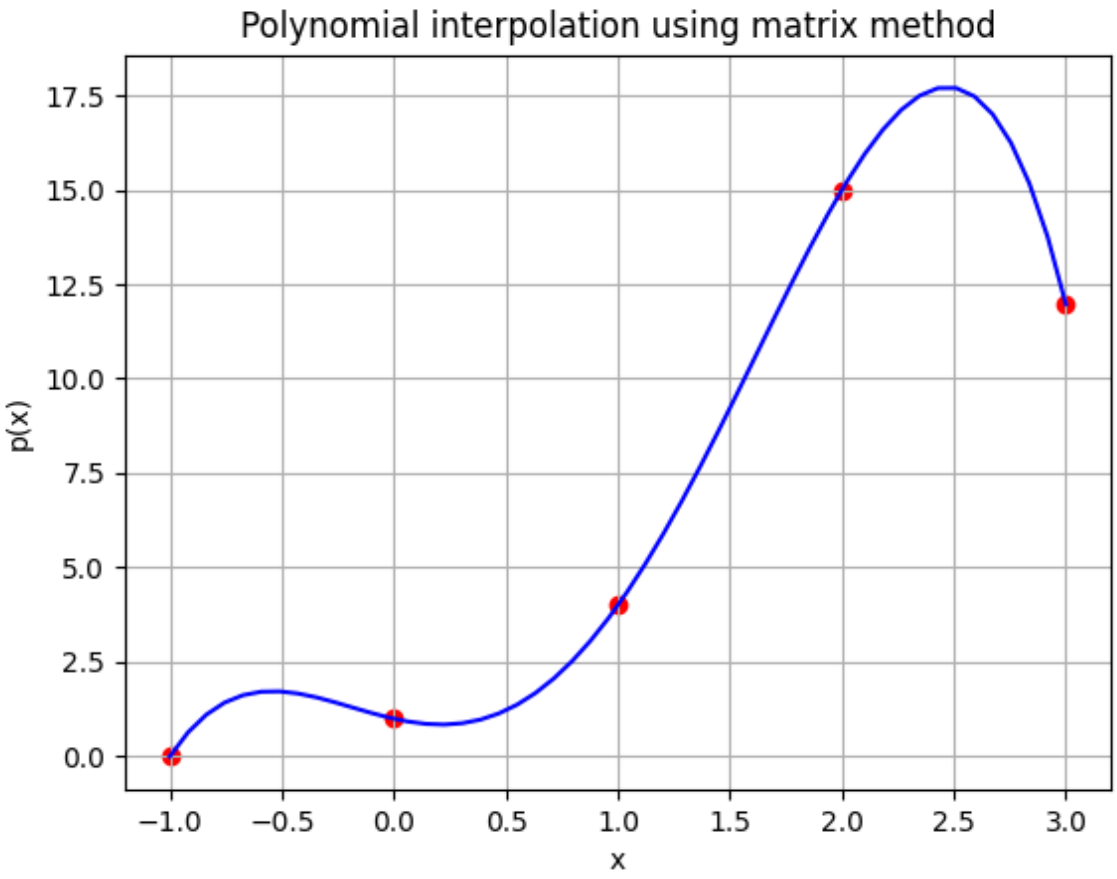
Q4

```
kaustubh@kaustubh:~/CMA$ /bin/python3 /home/kaustubh/CMA/week3/Q
1..
Coefficients of the polynomial are:
1 2 3

2..
Coefficients of the polynomial are:
4 4 4

3..
Coefficients of the polynomial are:
-2 0 2

4..
Coefficients of the polynomial are:
-0.5 -1.0 -1.5

5..
Coefficients of the polynomial are:
-1 0 0 1

6..
17
```

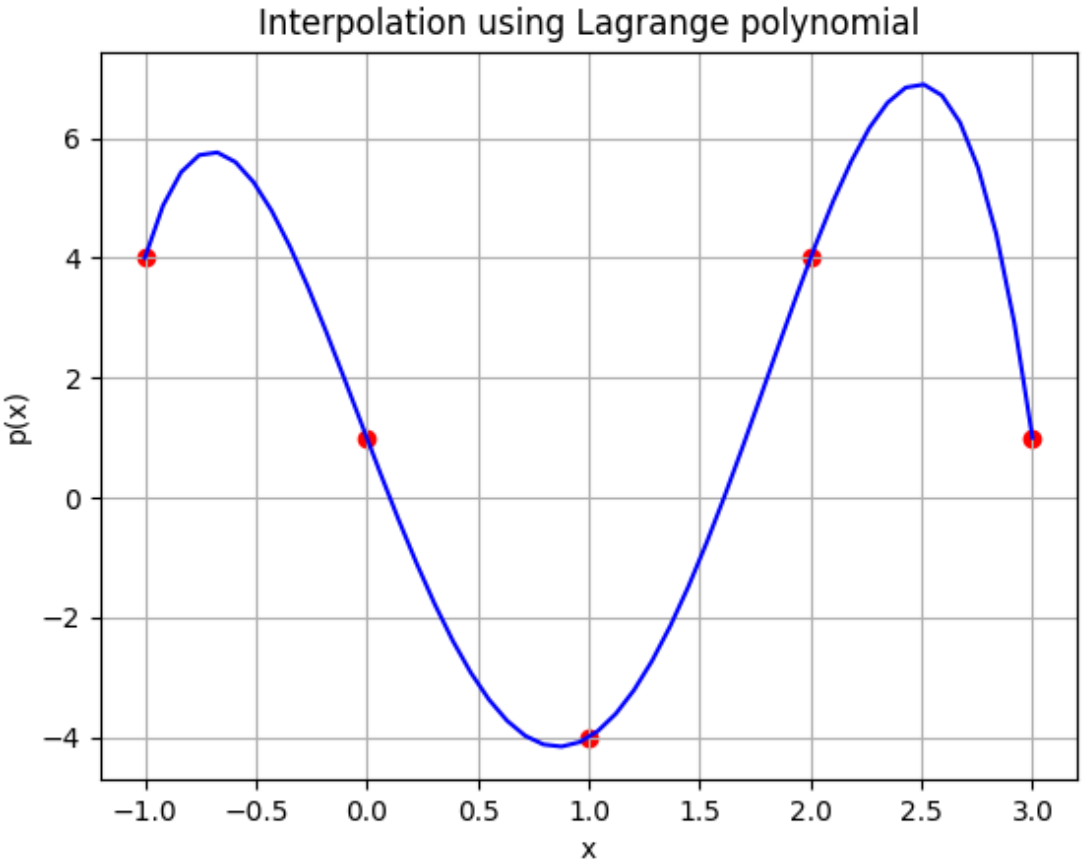
```
p = Polynomial([1, -1, 1, -1])
p.show(-1, 2)
```




```
p = Polynomial([])
p.fitViaMatrixMethod([(1,4), (0,1), (-1, 0), (2, 15), (3,12)])
# p.fitViaMatrixMethod([(0,1), (1,4), (-1,0), (2,15)])
```




```
p = Polynomial([])
p.fitViaLagrangePoly([(1,-4), (0,1), (-1, 4), (2, 4), (3,1)])
```



Q5
Run the code to see animation