COMPUTER SCIENCE AND ENGINEERING
Indian Institute of Technology, Palakkad
CS5016: Computational Methods and Applications

Coding Assignment 3
Linear System and Interpolation

30 Jan, 2023

Max points: 100

A few instructions

- Codes should be compatible with *Python3* and should run on Ubuntu.
- Code for each question should be placed in a separate stand-alone files.
- Codes should be well-commented.
- **Appropriate exceptions should be raised and handled.**

1. Create a class `RowVectorFloat`, representing a row vector, such that [20]

   - It is possible to create an object of this class from a list.

```
r = RowVectorFloat([1, 2, 3])
```

```
r = RowVectorFloat([])
```

   - It is possible to `print` it.

```
r = RowVectorFloat([1, 2, 4])
print(r)
```

Expected output:

```
1 2 4
```

   - It is possible to find its `length`.

```
r = RowVectorFloat([1, 2, 4])
print(len(r))
```

Expected output:

```
3
```

```
r = RowVectorFloat([])
print(len(r))
```

Continued on next page. . .

Expected output:

```
0
```

- It is possible to access its i^{th} element.

```
r = RowVectorFloat([1, 2 , 4])
print(r[1])
```

Expected output:

```
2
```

- It is possible to modify its i^{th} element.

```
r = RowVectorFloat([1, 2 , 4])
r[2] = 5
print(r)
```

Expected output:

```
1 2 5
```

- It is possible to create a `RowVectorFloat` object that is a linear combination of other `RowVectorFloat` objects.

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

Expected output:

```
-1 1 5
```

2. Create a class `SquareMatrixFloat`, representing a square matrix, such that [20]

- It is represented as a list of `RowVectorFloat` objects.
- It is possible to create a zero matrix of size $n \times n$

```
# The following code creates a 4 X 4 zero square matrix
s = SquareMatrixFloat(4)
```

- It is possible to `print` it.

```
s = SquareMatrixFloat(3)
print(s)
```

Question 2 continues on the next page...

Expected output:

```
The matrix is:
0 0 0
0 0 0
0 0 0
```

- It should have a method `sampleSymmetric` that samples a random symmetric matrix $\mathbf{A}$ of size $n \times n$ such $a_{ij} = a_{ji} = Uniform(0, 1)$ for $i \neq j$, and $a_{ii} = Uniform(0, n)$.

```python
s = SquareMatrixFloat(4)
s.sampleSymmetric()
print(s)
```

Expected output:

```
The matrix is:
3.30    0.05    0.14    0.67
0.05    2.99    0.29    0.93
0.14    0.29    1.95    0.86
0.67    0.93    0.86    0.23
```

- It should have a method `toRowEchelonForm` that converts the matrix to its *row echelon form*[1].

NOTE: You are expected to use the fact that the matrix is represented as a list of `RowVectorFloat` objects, and linear combinations of such objects are possible.

```python
s = SquareMatrixFloat(4)
s.sampleSymmetric()
print(s)
s.toRowEchelonForm()
print(s)
```

Expected output:

```
The matrix is:
0.20    0.31    0.41    0.47
0.31    1.53    0.42    0.28
0.41    0.42    2.48    0.91
0.47    0.28    0.91    1.07

The matrix is:
1.00    1.56    2.08    2.38
0.00    1.00    -0.21   -0.43
0.00    0.00    1.00    -0.10
0.00    0.00    0.00    1.00
```

[1] `https://en.wikipedia.org/wiki/Row_echelon_form`

Question 2 continues on the next page...

- It should have a method `isDRDominant` that checks if the matrix is diagonally row dominant.

```
s = SquareMatrixFloat(4)
s.sampleSymmetric()
print(s.isDRDominant())
print(s)
```

Expected output:

```
False
The matrix is:
3.64    0.01    0.44    0.26
0.01    1.74    0.98    0.91
0.44    0.98    0.71    0.22
0.26    0.91    0.22    2.97
```

```
s = SquareMatrixFloat(4)
s.sampleSymmetric()
print(s.isDRDominant())
print(s)
```

Expected output:

```
True
The matrix is:
2.31    0.11    0.02    0.84
0.11    2.87    0.19    0.41
0.02    0.19    2.55    0.84
0.84    0.41    0.84    3.71
```

- It should have a method `jSolve` that takes a list (denoting vector $\mathbf{b}$) and number of iterations m as its arguments, and performs m iterations of the *Jacobi* iterative procedure. This method should return the final iteration value, and value of the term $\|\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}\|_2$ from all the m iterations

  NOTE: If the matrix is not diagonally row dominant, the above method should throw an appropriate exception.

```
s = SquareMatrixFloat(4)
s.sampleSymmetric()
(e, x) = s.jSolve([1, 2, 3, 4], 10)
print(x)
print(e)
```

Expected output:

```
<class 'Exception'>
Not solving because convergence is not guranteed.
```

Continued on next page...

```
s = SquareMatrixFloat(4)
s.sampleSymmetric()
(e, x) = s.jSolve([1, 2, 3, 4], 10)
print(x)
print(e)
```

Expected output:

```
[0.06774123259846261, 0.0855117382520666, 1.8397261159567733,
3.0792129145462313]
[2594.5169746032284, 572.1329123800882, 166.745598116459,
42.64875125783601, 11.849210941928957, 3.117406566153758,
0.8524386967513456, 0.22681769812818547, 0.06155621008246957,
0.016470289259301205]
```

- It should have a method `gsSolve` that takes a list (denoting vector $\mathbf{b}$) and number of iterations m as its arguments, and performs m iterations of the *Gauss-Siedel* iterative procedure. This method should return the final iteration value, and value of the term $\|\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}\|_2$ from all the m iterations

```
s = SquareMatrixFloat(4)
s.sampleSymmetric()
(err, x) = s.gsSolve([1, 2, 3, 4], 10)
print(x)
print(err)
```

Expected output:

```
[-0.20147823121926625, 0.5383185155290146, 1.4197620577119878,
1.3612863007301668]
[1.7005986908310777, 0.5619941428201292, -2.114828835232932,
-4.440969442496047, -6.587740791522815, -8.680715095063702,
-10.76131879180163, -12.839445614308259, -14.917126914627547,
-16.994737345012226]
```

3. Write a function to visualize rate of convergence of *Jacobi* and *Gauss-Siedel* methods of a linear system with a diagonally dominant square symmetric matrix. [10]

4. Create a class `Polynomial`, representing a algebraic polynomial, such that [40]

   - It is possible to create a polynomial by specifying its coefficients.

```
# The following code creates the polynomial 1 + 2x + 3x^2
p = Polynomal([1, 2, 3])
```

Question 4 continues on the next page...

- It is possible to `print` it.

```
p = Polynomal([1, 2, 3])
print(p)
```

Expected output:

```
Coefficients of the polynomial are:
1 2 3
```

- It is possible to add (subtract resp.) two polynomial using the + (- resp.) operators.

```
p1 = Polynomal([1, 2, 3])
p2 = Polynomal([3, 2, 1])
p3 = p1 + p2
print(p3)
```

Expected output:

```
Coefficients of the polynomial are:
4 4 4
```

```
p1 = Polynomal([1, 2, 3])
p2 = Polynomal([3, 2, 1])
p3 = p1 - p2
print(p3)
```

Expected output:

```
Coefficients of the polynomial are:
-2 0 2
```

- It is possible to pre-multiply the polynomial by a real number using the * operator.

```
p1 = Polynomal([1, 2, 3])
p2 = (-0.5)*p1
print(p3)
```

Expected output:

```
Coefficients of the polynomial are:
-0.5 -1 -1.5
```

- It is possible to multiple two polynomials using the * operator.

```
p1 = Polynomal([-1, 1])
p2 = Polynomal([1, 1, 1])
p3 = p1 * p2
print(p3)
```

Question 4 continues on the next page. . .

Expected output:

```
Coefficients of the polynomial are:
-1 0 0 1
```

- It is possible to evaluate the polynomial at any real number using the [] operator.

```python
p = Polynomal([1, 2, 3])
print(p[2])
```

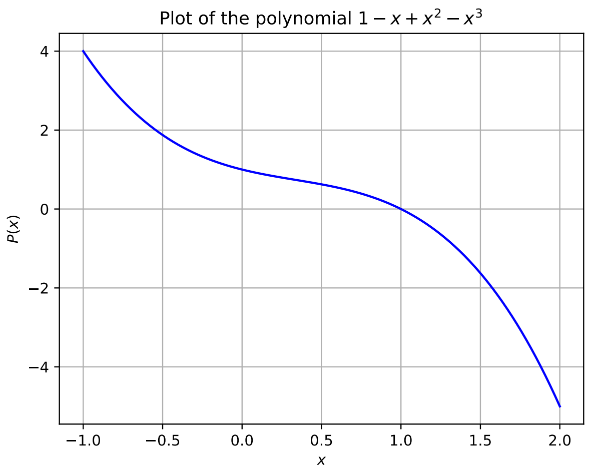
Expected output:

```
17
```

- It is possible to visualize the polynomial in any interval of the type $[a, b]$

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

Expected Output:



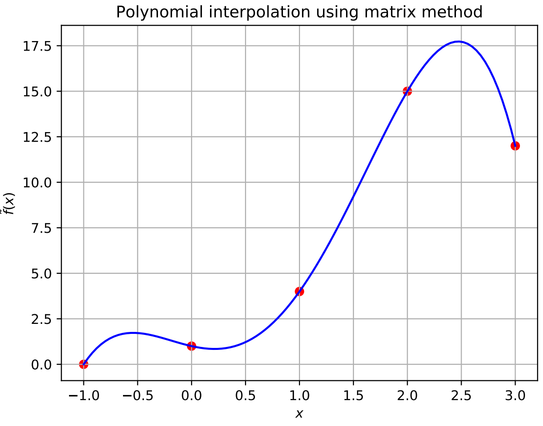
- It should have a method `fitViaMatrixMethod` that fits, using the idea of linear systems, a polynomial to the points passed as its argument. This method should display a plot with the given points and the computed polynomial.

  NOTE: You are expected to use Python's `numpy.linalg` module to solve the linear system.

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

Continued on next page...

Expected Output:

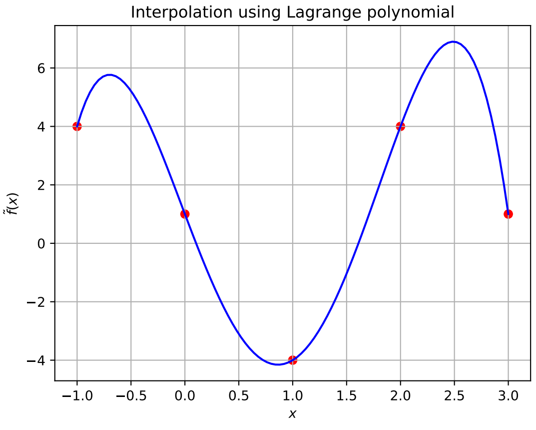


- It should have a method `fitViaLagrangePoly` that computes the *Lagrange polynomial* for the points passed as argument to this method. This method should display a plot with the given points and the computed polynomial.

  NOTE: You are expected to use the `Polynomial` class along with the overloaded operators + and * to compute the Lagrange polynomial.

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

Expected Output:



5. Using the FuncAnimation function of *matplotlib*, create an animation to demonstrate the various interpolations available in Python's `scipy.interpolate` module. Your animation should also demonstrate how sampling more points of a function will create better interpolations. A reference animation is available at animation link. Try to match the reference animation as much as possible. [10]