CS5016: Computational Methods and Applications

Lab Report - 3

111901030
Mayank Singla

First of all, we need to handle exceptions in all the questions. So to avoid repetition of code, I made a Decorator Factory function for class methods called `handleError`, which simply wraps the class method in a `try-except` block and handles and prints the error as in the expected format. This decorator forwards all the arguments it receives to the actual method and also returns the value returned by the method so that the functionality of the method doesn't break. I am using the same decorator factory function in all of my questions for all the class methods.

Q1.

I am building `RowVectorFloat` class for this question.

First of all, I am creating the `_validateListValues` helper method to validate that the input is a list of float or integers and also `_validateIndex` helper method to validate the input index values is a valid integer within the range (also allowing negative indexing).

Then I am creating a constructor method to initialize the `RowVectorFloat` class.

I am also implementing the `__str__` method so that any instance of this class could be printed in the expected format.

I am implementing the `__len__` method so that any instance of this class could be passed to the `len` function to get the length of the vector.

Next, I am implementing `__getitem__` and `__setitem__` dunder methods so that we can access and modify the *ith* element of the vector.

To overload the `+` operator, I am implementing the `__add__` and `__radd__` dunder methods to allow both *post* and *pre* addition with any other row vector.

Similarly to allow both *post* and *pre* multiplication with any scalar and overloading of `*` operator, I am implementing the `__mul__` and `__rmul__` dunder methods.

Q2.

I am building `SqaureMatrixFloat` class for this question which represents a square matrix.

I am implementing the `__init__` constructor method which takes an argument *n* and builds a square matrix using `RowVectorFloat` class. I am implementing the `__str__` dunder method so that any instance of this class could be printed using the `print` function.

Next, I am implementing the `sampleSymmetric` method to sample this square matrix as per the criteria mentioned in the question. I am using `random.uniform` built-in method to sample a random number uniformly between *(0, 1)*.

Next, I am implementing the `toRowEchelonForm` method to convert the square matrix to its row echelon form using elementary matrix row operations. The method implemented is the naive method from scratch to convert a matrix to its row echelon form as taught to us in our 1st year for which a summary is written below:

- Pivot the matrix
    - Find the pivot, the first non-zero entry in the first column of the matrix.
    - Interchange rows, moving the pivot row to the first row.
    - Multiply each element in the pivot row by the inverse of the pivot, so the pivot equals 1.
    - Add multiples of the pivot row to each of the lower rows, so every element in the pivot column of the lower rows equals 0.
- Repeat the pivot
    - Repeat the procedure from Step 1 above, ignoring previous pivot rows.
    - Continue until there are no more pivots to be processed.

Next, I am implementing the `isDRDominant` method to check if the matrix diagonally rows dominant or not. I am here implementing for **Strictly Diagonally Row Dominance**, though it was not mentioned in the question, we need to satisfy the convergence criteria for the Jacobi method.

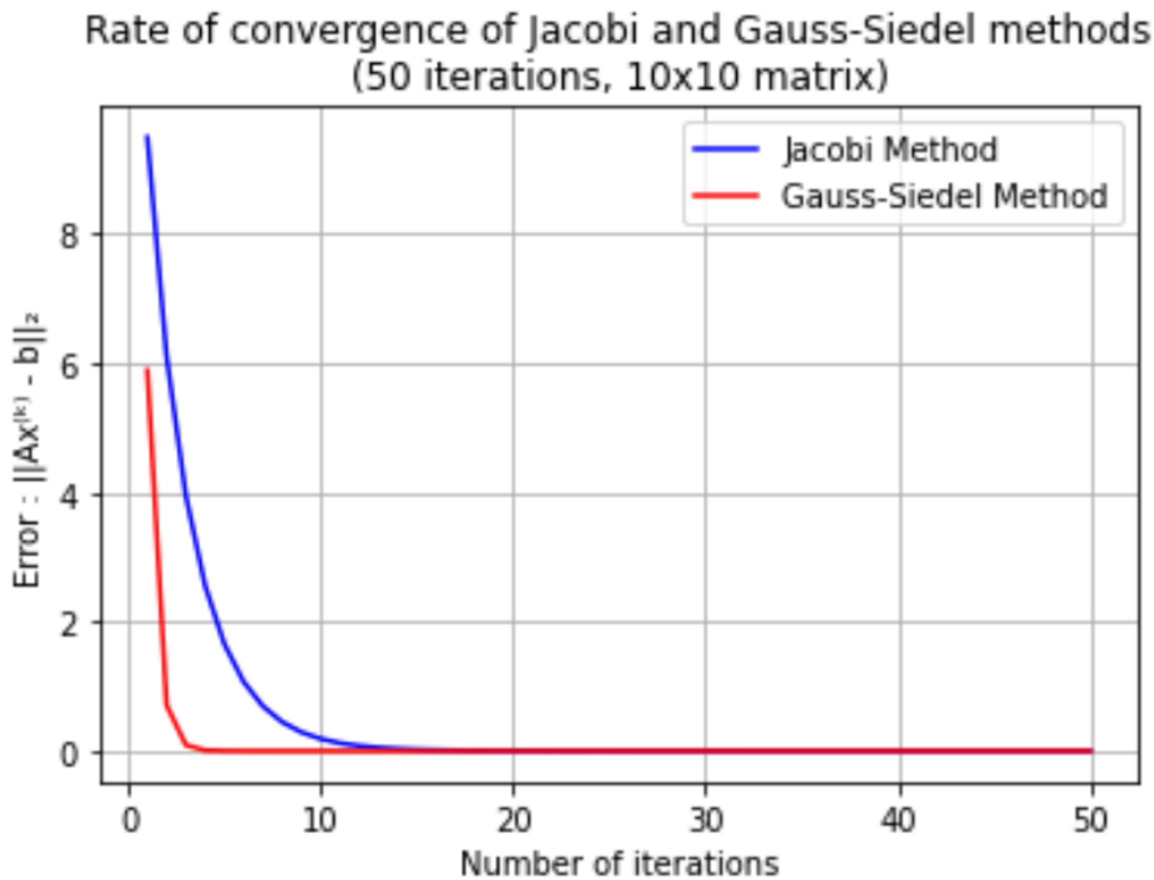
Next, I am implementing the `_validateListValues` helper method which is the same as implemented for the 1st question.

Next, I am implementing the `_iterativeMethod` helper method which performs iterations for **Jacobi** and **Gauss-Siedel** methods based on the input argument acting as a flag. The method uses simples loops to evaluate the required quantities. The implementations for both the methods are almost identical, just differ at a few places while computing the values. Finally, I am implementing the `jSolve` and `gsSolve` methods which returns the results of the helper method `_iterativeMethod` and passes arguments to it appropriately.

Q3.

I am implementing a method `visualizeRateOfConvergence` to visualize the rate of convergence of Jacobi and Gauss-Siedel method of a linear system with a diagonally dominant by row (DDR) square symmetric matrix. For this, I am first generating a DDR square symmetric matrix and then getting the error values for the Jacobi and Gauss-Siedel method using the `jSolve` and `gsSolve` methods and then plotting the curve between the number of iterations and the error in each iteration.

Here is an example curve I obtained.



Q4.

I am creating a `getSuperScript` function that converts an input string of integers to its superscript form. It is just a helper function for pretty-printing.

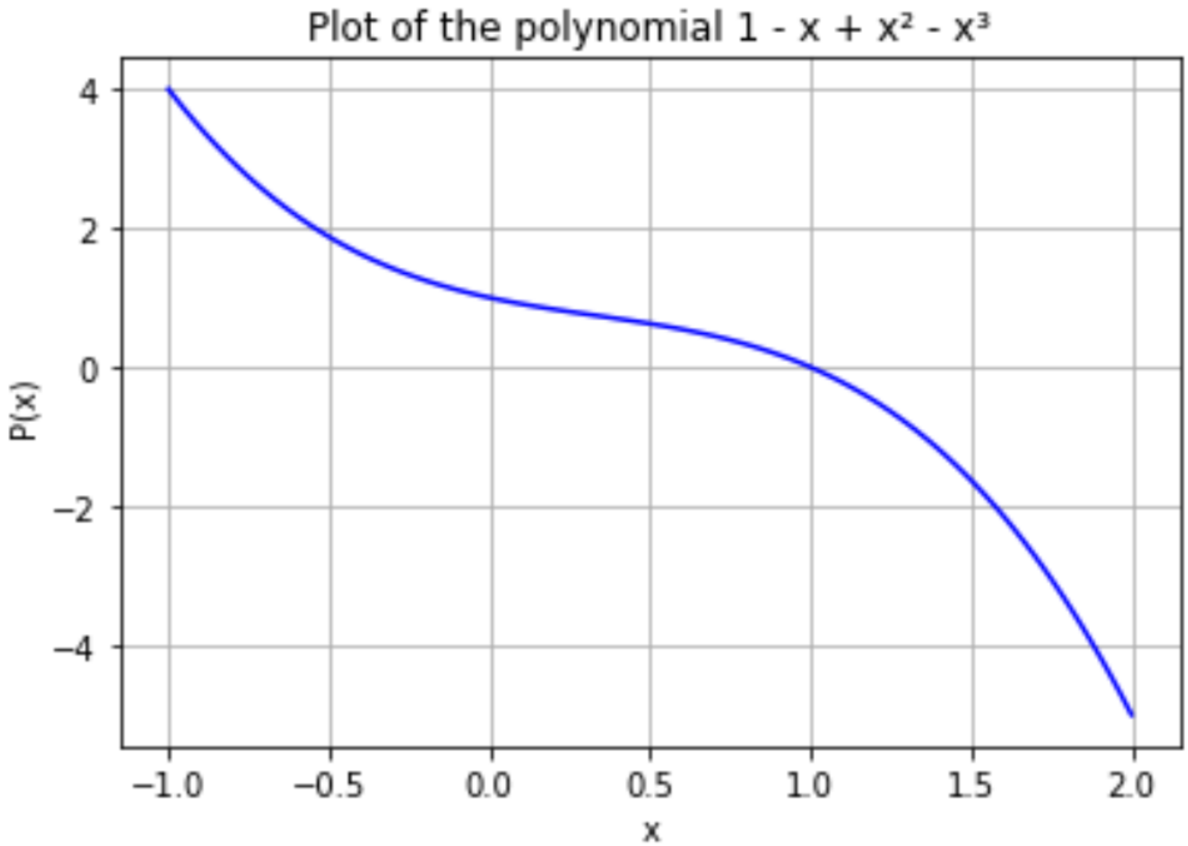
Next, I am creating the `Polynomial` class to represent an algebraic polynomial.
I am creating the `_validateCoefficients` method to validate the input coefficients received as a list of float/integers.
Then I am implementing the `__init__` constructor method to initialize the polynomial.
Next, I am implementing the `__str__` dunder method so that we could print a polynomial using the `print` function.

Next, I am implementing the `_addOrSub` helper method which helps in the addition/subtraction of two polynomials and performs the operation based on the input flag, and returns the resultant polynomial.
To overload the `+` and the `-` operator, I am implementing the `__add__` and the `__sub__` dunder methods which calls the `_addOrSub` helper method to perform the operation and return the resultant polynomial.

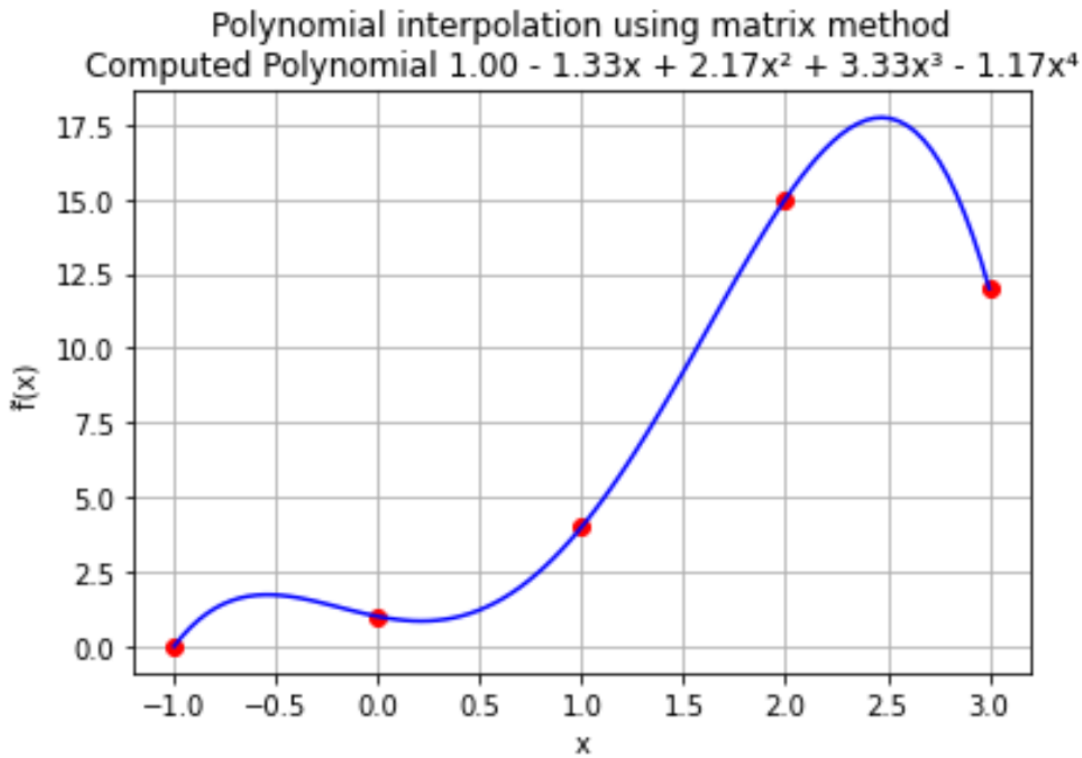
Next, I am implementing the `__mul__` dunder method to overload the `*` operator to multiply two polynomials. Using simple for loops I am multiplying every coefficient of the first polynomial with every coefficient of the second polynomial and then finally returning the resultant polynomial.
To overload the `*` operator to pre-multiply a polynomial with a scalar, I am implementing the `__rmul__` dunder method and simply multiplying each coefficient with that scalar value.

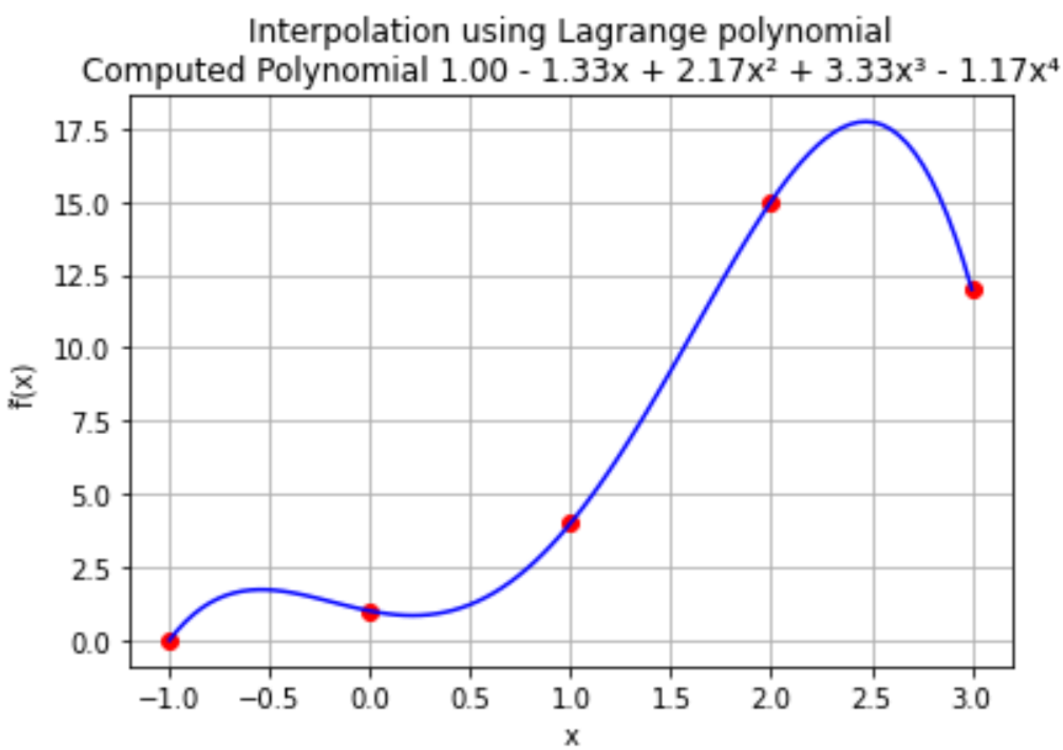
To evaluate the polynomial at any real value using *`[]`*, I am implementing the `__getitem__` dunder method and evaluating the result of the polynomial evaluation.

Next, I am creating a `_getPolyString` helper method which pretty-prints the polynomial in mathematical form and returns the final string representation.
Next, I am implementing the `_plotPolynomial` helper method which only plots the polynomial curve in the given interval.
I am creating the `show` method to visualize the polynomial in the given interval which makes use of the `_plotPolynomial` method to plot the curve. Here is an example plot I obtained.

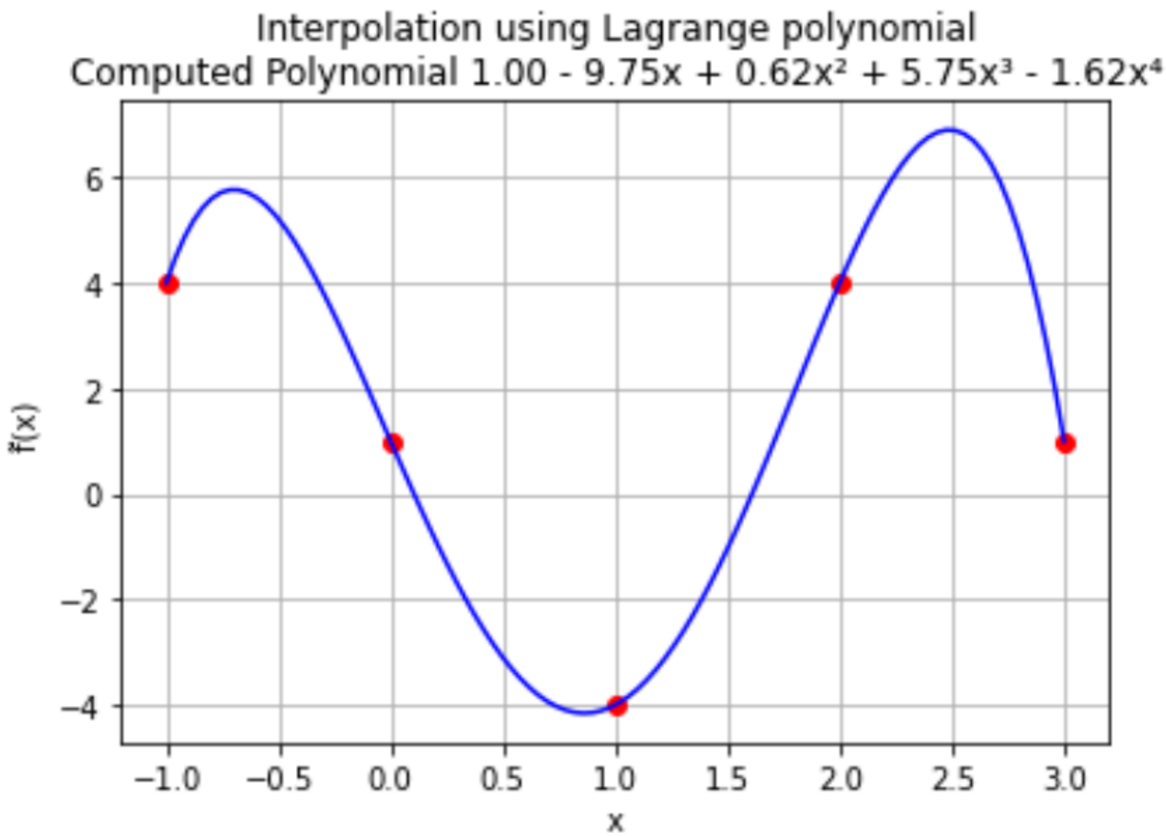


Next, I am implementing the `fitViaMatrixMethod` method which uses the idea of linear systems and fits a polynomial to input points and displays the final plot obtained.
I am simply creating the matrices *A* and *b* for the equation `Ax = b` and then evaluating the matrix *x* using the `numpy.linalg.solve` method and then displays the curve obtained.
Here is an example curve I obtained and its comparison with the Lagrangian interpolation I implemented as the next method.


Polynomial interpolation using matrix method
Computed Polynomial 1.00 - 1.33x + 2.17x² + 3.33x³ - 1.17x⁴
f(x)
x


Interpolation using Lagrange polynomial
Computed Polynomial 1.00 - 1.33x + 2.17x² + 3.33x³ - 1.17x⁴
f(x)
x

Finally, I am implementing the fitViaLagrangePoly method to compute the Lagrange polynomial for the input points and display the curve obtained. I am using simple for loops to evaluate each Ψ other required quantities to evaluate the polynomial and using the overloaded $*$ and $+$ methods to compute the final result.
Here is an example curve I obtained.



Q5.

I am creating a function `animation` that animates different interpolations of the given function as an input argument. I am plotting the actual function for reference and then trying animations of different interpolation methods like **CubicSpline**, **Akima1DInterpolator,** and **BarycentricInterpolator** from the `scipy.interpolate` module.

I am using the function `FunAnimation` from python's `matplotlib.animation` module which according to its documentation requires an update function that will update the curves which in my case is the `animate` function created as a nested function that uses the *scipy's* functions to update the different curves for different samples.

I observed that for large samples, the curve almost becomes identical to the actual curve for different interpolations.

I have attached the gif of the animation I got.