(Github link: `https://github.com/RONYkGT/interpolation-methods`)

Contents

1 Introduction

In this paper we will talk about implementing interpolation methods in Python, like Newton Interpolation, Cubic Splines Interpolation, and multiple least squares models interpolation.
The above was done using Object Oriented concepts, and package management enhancing the modularity and scalability of the project.

2 Instructions

2.1 Download

Head to the Github repository and click on "code" green button at the top right and click download zip, or you can clone the repository to your local machine using `git clone https://github.com/RONYkGT/interpolation-methods` (you need git installed)

2.2 Dependencies

The project requires LaTeX to be installed on your computer to run. You can do that on Windows by installing MiKTeX, run it and run the `main.py` file from this project, and it will prompt to automatically install required packages. When it does that, just click agree and wait - be patient.

2.2.1 Python Packages

```
matplotlib
numpy
sympy
```

You can automatically install required packages by running "`pip install requirements.txt`" after opening the project directory in the terminal or command prompt.

3 Package Structure

```
README.md
requirements.txt
src
    cubic_splines.py
    least_squares.py
    main.py
    newton_interpolation.py
    utils
        divided_differences.py
        matrix_operations.py
        plotting.py"
```

3.1 \src

The "src" file contains classes that are used for interpolation, and the `main.py` file that is executable and runs an interactive menu to input x and y points and choose the interpolation method you want.

3.2 \src\utils

This file contains functions that does useful math operations and graphing operations that are shared among all interpolation classes

4 Functionality

The main idea behind this package structure is modularity and trying to group common steps into single functions, that all the methods can use.
And about the classes, they are all supposed to take ONLY x and y points in their constructors, nothing else, as these are the only things needed for doing interpolation calculations. However for method specific inputs like number of degrees for polynomial least squares interpolation, would go into their specific methods that execute the interpolation computation and not their constructors that are supposed to only initialize the class.

4.1 divided_differences.py

```
def calculate_divided_differences(x_points, y_points, n=0):
```

Takes in x and y inputs, and returns a 2D array in the form of:

$$\begin{bmatrix} [x_0, \ldots, x_n], \\ [f[x_0], \ldots, f[x_n]], \\ [f[x_0\ x_1], \ldots, f[x_{n-1}\ x_n]], \\ \vdots \\ [f[x_0\ \ldots\ x_n]] \end{bmatrix}$$

Where the divided differences are calculated as the following code:

```
(table[row-1][col+1]-table[row-1][col])/(table[0][col+row-1]-table[0][col])
```

Where the divided differences count on the ones calculated in the rows before them, which means it's recursive.
So you can say for example:

$$f[x_i \ldots\ x_j] = \frac{f_{i+1} - f_i}{x_j - x_i}$$

where f_{i+1} is the divided difference retrieved in the row before but 1 column in advance and f_i is retrieved in the row before but same column.

Example form: let's take the inputs (x_i, y_i): $(0, 1), (2, 2), (3, 4)$
The array would be:

$$\begin{bmatrix} 0 & 2 & 3 \\ 1 & 2 & 4 \\ 0.5 & 2 & \\ 0.5 & & \end{bmatrix}$$

The first two rows are reserved for x and y values.

```
def beautify_difftable(data):
```

Would take a divided difference table in the format mention above and turns it into a horizontal table with a triangular forms and spaced terms for clarity and design. For Example:

$$\begin{bmatrix} 0 & 2 & 3 \\ 1 & 2 & 4 \\ 0.5 & 2 & \\ 0.5 & & \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 4 & 〃 & 〃 \\ 〃 & 〃 & 2 & 〃 \\ 2 & 2 & 〃 & 0.5 \\ 〃 & 〃 & 0.5 & 〃 \\ 0 & 1 & 〃 & 〃 \end{bmatrix}$$

4.2 matrix_operations.py

```
def generate_vandermonde_matrix(x_values, degree):
```

```
    """
    Generate a Vandermonde matrix with the given x_values.
    Args:
    x_values (list): A list of x values.
    degree (int): The degree of the polynomial.
    Returns:
    A Vandermonde matrix (ndarray).
    """
    n = len(x_values)
    A = ones((n,degree+1))
    for row in range(0,n):
        x = x_values[row]
        for col in range(1,degree+1):
            A[row][col] = x**col
    return A
```

The function above would generate what so called "Vandermonde" by taking an array of x values, and the max degree. The array would be in the following form:

$$A = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^n \end{pmatrix}, \text{ where } n \text{ is the "degree" parameter}$$

```
def solve_inconsistent_system(A_matrix, b_vector) -> ndarray:
    """
    Solve an inconsistent system of linear equations.
    Args:
    A_matrix (ndarray): A matrix of coefficients.
    b_vector (ndarray): A column vector of constants.
    Returns:
    A solution vector (ndarray).
    """
    # Create the A matrix and b vector
    A_norm = dot(A_matrix.T, A_matrix) #A^T . A
    b_norm = dot(A_matrix.T, b_vector) #A^T . b
    # Solve the normal equation
    solution = solve(A_norm, b_norm)
    return solution, A_norm, b_norm
```

This function takes in an inconsistent system (A system with no solution) and returns the closest solution along with the normalized matrices versions of A and b passed to the function (A^TA, A^Tb). It applies the following formula:

$$A^TAx = A^Tb$$

We will see examples in the least squares sections.

4.3 newton_interpolation.py

```
class NewtonInterpolation:
    def __init__(self, x_points, y_points):
        """
        Initialize Newton Interpolation with data points, and
            automatically stores the result of
        the divided differences calculation of x and y

        Args:
            x_points (list/array): x coordinates
            y_points (list/array): y coordinates
        """
        self.difftable = calculate_divided_differences(x_points,y_points)
        #x_values = self.difftable[0]
        #y_values = self.difftable[1]

        self.polynomial = None
        self.latex_polynomial = None
```

The class takes in only x and y inputs in it's constructor as usual and generates the divided difference table in its `self.difftable` array using the function `calculate_divided_differences` explained above.
The remaining two attributes `self.polynomial` and `self.latex_polynomial` will respectively store the mathematical polynomial result and it's latex factorized form.

```
    def interpolate(self):
        """Calculate Newton interpolation polynomial."""
        ...
        #Skipped a few lines from the code about initialization
        ...
        for col in range(1,len(self.difftable)): # from y column until
            the last divided difference column

            coefficient = self.difftable[col][0]

            if col != 1:
                xProduct *= (x - self.difftable[0][col-2])
                latex_xProduct += f"(x - {self.difftable[0][col-2]})"

            poly += coefficient * xProduct # a0*(x-x0)(x-x1)...
```

```
        if col != 1:
            latex_poly += f"+ {coefficient} * " + latex_xProduct
        else:
            latex_poly += f"{coefficient}"
```

Will iterate through the "difftable" attribute array starting from the second row which contains the values of y until the last row that contains the last divided difference.
The coefficient variable will store the first column of each row iterated on, for example:

$$\begin{bmatrix} 0 & 2 & 3 \\ \boxed{1} & 2 & 4 \\ \boxed{0.5} & 2 & \\ \boxed{0.5} & & \end{bmatrix}$$

and `xProduct` will iteratively aggregate the following form $(x - x_{i-2})$ starting from `i = 2` until `i = len(self.difftable)`. For example using the matrix given above:
i = 2: `xProduct = (x - 0)`
i = 3: `xProduct = (x - 0)(x - 2)`

4.3.1 Full Example:

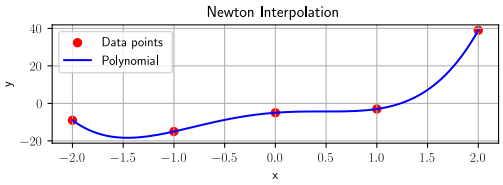
Let's take the following inputs:

```
x_values = [0, 1, -1, 2, -2]
y_values = [-5, -3, -15, 39, -9]
```

and pass them into the constructor. The first thing it would do is generate the divided differences table.

```
self.difftable =
[[0, 1, -1, 2, -2],
[-5, -3, -15, 39, -9],
[2.0, 6.0, 18.0, 12.0],
[-4.0, 12.0, 6.0],
[8.0, 2.0],
[3.0]]
```

and generate the following graphical output in matplotlib:



Deduced Polynomial from table:

$$P(x) = -5 + 2.0 * (x - 0) + -4.0 * (x - 0)(x - 1) + 8.0 * (x - 0)(x - 1)(x - -1) + 3.0 * (x - 0)(x - 1)(x - -1)(x - 2)$$

Expanded Form:

$$P(x) = 3.0x^4 + 2.0x^3 - 7.0x^2 + 4.0x - 5$$

Figure 1: Newton Interpolation Graph and it's deduced polynomial

x	y				
0	-5				
		2.0			
1	-3		-4.0		
		6.0		8.0	
-1	-15		12.0		3.0
		18.0		2.0	
2	39		6.0		
		12.0			
-2	-9				

Neuville's table.

4.4 cubic_splines.py

```
class CubicSplineInterpolation:

    def __init__(self, x_points, y_points,w0 = 0,wn = 0):
```

The cubic spline interpolation class takes in x and y points as usual, but can also take optional weights w_0 and w_n.

```
        # Store the divded differences with row0 and row1 as x and y
            values
        self.difftable =
            calculate_divided_differences(x_points,y_points,3)

        # Calculate hi values where hi = x_i+1 - xi
        self.h_values = [x2 - x1 for x1, x2 in zip(x_points[:-1],
            x_points[1:])]
```

Then it generates two attributes, `self.difftable` and `self.h_values`. `self.difftable` is like the one in the Newton interpolation, but this time only with 3 iterations until $[x_{i-1}, x_i, x_{i+1}]$.
And `self.h_values` will contain an array of the values $[x_{i+1} - x_i]$
The following interpolation class attributes:

```
self.A_matrix = None
self.b_vector = None
self.w_values = None
self.piecewise = None
```

Where A is the following form:

$$A = \begin{bmatrix} \frac{h_0+h_1}{3} & \frac{h_1}{6} & 0 & \cdots & 0 \\ \frac{h_1}{6} & \frac{h_1+h_2}{3} & \frac{h_2}{6} & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \frac{h_{n-1}}{6} & \frac{h_{n-1}+h_n}{3} & \end{bmatrix}$$

And is calculated using the following python formula:

```
n = len(self.h_values) - 1 # Length of matrix diagonal
matrix = sp.zeros(n) # Fill matrix with 0's initially

# Fill the diagonal with hi+hi1/3
for i in range(n):
    matrix[i, i] = (self.h_values[i] + self.h_values[i+1])/3

# Fill the upper and lower diagonal with hi+1/6
for i in range(n - 1):
    matrix[i, i+1] = self.h_values[i+1]/6
    matrix[i+1, i] = self.h_values[i+1]/6
```

Vector b has the following form:

$$b_i = (h_i + h_{i+1})[x_i, x_{i+1}, x_{i+2}]$$

And is generated using the following Python calculation:

```
n = len(self.h_values) - 1
vector = sp.zeros(n,1)
for i in range(n):
    vector[i] = (self.h_values[i] +
        self.h_values[i+1])*self.difftable[3][i]
```

`self.w_values` is the solution vector of the linear equation system: $Aw = b$, and w_0, w_n are inserted respectively to the start and the end of the solution vector.

Using the arrays generated, we can now calculate the piecewise function made out of the splines calculated using the following formula for each spline:

$$S_i(x) = \frac{(x_{i+1} - x)^3 w_i + (x - x_i)^3 w_{i+1}}{6h_i} + \frac{(x_{i+1} - x)y_i + (x - x_i)y_{i+1}}{h_i} - \frac{h_i}{6}[(x_{i+1} - x)w_i + (x - x_i)w_{i+1}] \quad (1)$$

And the following conditions for each one: $x >= x_i$ & $x <= x_{i+1}$
Then the spline and its condition are stored in an array that takes them as pairs:

```
splines.append((func.expand(),condition))
```

And that array is then converted to a sympy.Piecewise object that takes these pairs and stored inside `self.piecewise`

4.4.1 Example

Let's take the following inputs:

```
x_values = [-1, 0, 1]
y_values = [1, 2, -1]
```

It will generate the following Neuville table:

x	y	h_{i+1}	$[x_i, x_{i+1}]$	$[x_{i-1}, x_i, x_{i+1}]$
-1	1			
		1	1.0	
0	2			-2.0
		1	-3.0	
1	-1			

and the following steps:

Deduced Piecewise from table:

$$P(x) = \begin{cases} -1.0x^3 - 3.0x^2 - 1.0x + 2.0 & \text{for } x \geq -1 \wedge x < 0 \\ 1.0x^3 - 3.0x^2 - 1.0x + 2.0 & \text{for } x \geq 0 \wedge x \leq 1 \end{cases}$$

Matrix: $A = \begin{bmatrix} 0.6667 \end{bmatrix}$ Vectors b and w: $b = \begin{bmatrix} -4.0 \end{bmatrix}, \omega = \begin{bmatrix} 0 \\ -6.0 \\ 0 \end{bmatrix}$

And will output the following graph:

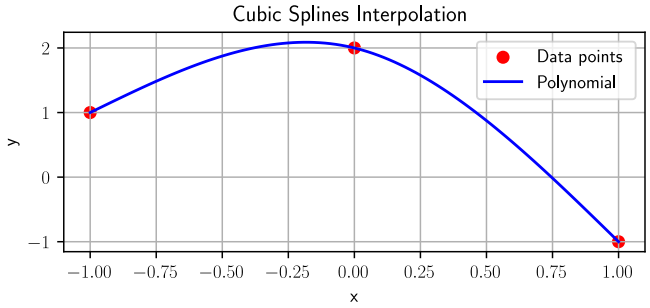


Figure 2: Cubic Splines Interpolation

4.5 least_squares.py

The least squares class can do interpolation for the following models:

- Polynomial
- Periodic
- Power Law
- Exponential
- Drug concentration

Each in its own method in the class.

They all have in common the attributes of the class:

```
self.A_matrix = None
self.b_vector = None
self.A_norm = None # A^TA
self.b_norm = None # A^TB
self.c_vector = None

self.function = None
```

`self.A_matrix` contains the vandermonde matrix which is generated from the function mentioned above, with each model having its own passed degree paramter, or modified x values.
`self.b_vector` will contain the values of y or its modified form depending on the model, but transposed to a column vector.
`self.A_norm` will contain the matrix A^TA which is the normalized form of A
`self.b_norm` will contain the matrix A^Tb which is the normalized form of b
`self.c_vector` will contain the result coefficients of the model initial formula by calculating $A^TAc = A^Tb$

`self.function` will contain the result function after replaced the model formula with the coefficients.

4.5.1 Polynomial Model

This model is the easiest, and its general form is: $f(x) = c_0+c_1x+c_2x+\cdots+c_nx^n$ where n is the degree passed to the method.

```
def polynomial_interpolation(self, degree = 1):
```

It would then generate a normal Vandermonde matrix with no modification to the x values passed to the method, with the degree n originally received from the parameter.

```
self.A_matrix = mat.generate_vandermonde_matrix(self.x_values,
    degree)
```

and would transpose the y values to the column vector `self.b_vector` Then the inconsistent system is solved using the following:

```
self.c_vector, self.A_norm, self.b_norm =
    mat.solve_inconsistent_system(self.A_matrix, self.b_vector)
```

The c vector be in the following:

$$c = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}$$

And then we iterate through the matrix, assigning each coefficiant to its correct place in the function $f(x) = c_0 + c_1x + c_2x + \cdots + c_nx^n$

4.5.2 Polynomial Model Example:

```
x_values = [0, 1, 2, 3, 4, 5, 6 ,7 ,8, 9, 10]
y_values = [0, 1, 2, 2.5, 3, 3.1, 3.1, 3.1, 3.25, 4]
degree = 3
```

Would give the following outputs: $y = c_0 + c_1x^1 + c_2x^2 + c_3x^3$

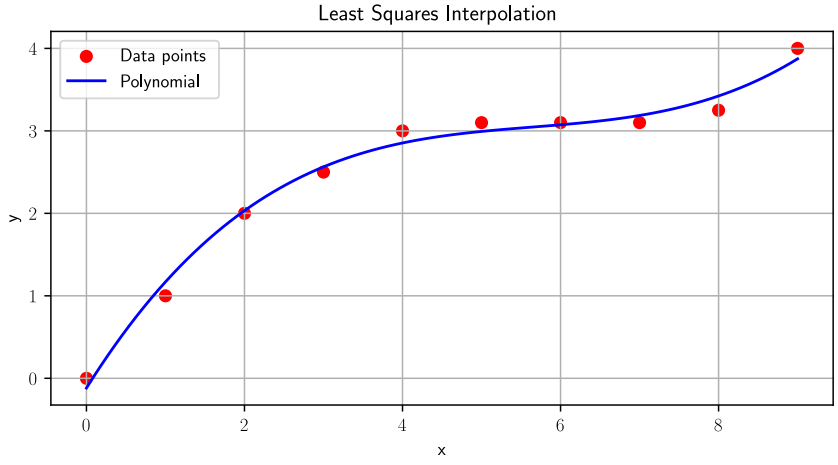
$$A = \begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 2.0 & 4.0 & 8.0 \\ 1.0 & 3.0 & 9.0 & 27.0 \\ 1.0 & 4.0 & 16.0 & 64.0 \\ 1.0 & 5.0 & 25.0 & 125.0 \\ 1.0 & 6.0 & 36.0 & 216.0 \\ 1.0 & 7.0 & 49.0 & 343.0 \\ 1.0 & 8.0 & 64.0 & 512.0 \\ 1.0 & 9.0 & 81.0 & 729.0 \end{bmatrix}, C = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 \\ 1.0 \\ 2.0 \\ 2.5 \\ 3.0 \\ 3.1 \\ 3.1 \\ 3.1 \\ 3.25 \\ 4.0 \end{bmatrix}$$

$$A^T AC = A^T B \rightarrow \begin{bmatrix} 10.0 & 45.0 & 285.0 & 2025.0 \\ 45.0 & 285.0 & 2025.0 & 15333.0 \\ 285.0 & 2025.0 & 15333.0 & 120825.0 \\ 2025.0 & 15333.0 & 120825.0 & 978405.0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 25.05 \\ 142.3 \\ 952.5 \\ 6976.9 \end{bmatrix}$$

$$C = \begin{bmatrix} -0.118251748251662 \\ 1.52921522921509 \\ -0.257342657342617 \\ 0.0151903651903623 \end{bmatrix}$$

$f(x) = 0.0151903651903623x^3 - 0.257342657342617x^2 + 1.52921522921509x - 0.118251748251662$

And the following graph:



4.5.3 Periodic Model

The general form of the periodic model is:

$$f(x) = c_0 + c_1 \sin 2\pi x + c_2 \cos 2\pi x$$

So there will be some modification to a copy of the values of x before getting passed in the Vandermonde matrix generator, which will have the form:

$$A = \begin{bmatrix} 1 & \sin 2\pi x_0 & \cos 2\pi x_0 \\ & \vdots & \\ 1 & \sin 2\pi x_n & \cos 2\pi x_n \end{bmatrix}, \text{ where n is the size of x values - 1}$$

4.5.4 Periodic Model Example

Let's take the following inputs:

```
x_values = [0, 0.25, 0.5, 0.75]
y_values = [1,3,2,0]
```

Will give out the following output:
$y = c_0 + c_1 \cos(2\pi x) + c_2 \cos(2\pi x)$

and $C = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$

$$A = \begin{bmatrix} 1.0 & 1.0 & 0 \\ 1.0 & 0 & 1.0 \\ 1.0 & -1.0 & 0 \\ 1.0 & 0 & -1.0 \end{bmatrix},$$

$$C = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 0 \end{bmatrix}$$

$$A^T AC = A^T B \rightarrow \begin{bmatrix} 4.0 & 0 & 0 \\ 0 & 2.0 & 0 \\ 0 & 0 & 2.0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 6.0 \\ -1.0 \\ 3.0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1.5 \\ -0.5 \\ 1.5 \end{bmatrix}$$

$f(x) = 1.5 \sin(2\pi x) - 0.5 \cos(2\pi x) + 1.5$

4.5.5 Power Law Model

The general form of the power law model is:

$$y = c_0 x^{c_1}$$
$$\ln y = \ln c_0 + c_1 \ln x$$
$$\ln y = k + c_1 \ln x. \text{ where } k = \ln x$$

$$b = \begin{bmatrix} \ln y_0 \\ \ln y_1 \\ \vdots \\ \ln y_n \end{bmatrix} \text{ and } C = \begin{bmatrix} k \\ c_1 \end{bmatrix}$$

There will be some modification to x values before being passed to the vandermond matrix generator, and A would have the form:

$$A = \begin{bmatrix} 1 & \ln x_0 \\ \vdots & \vdots \\ 1 & \ln x_n \end{bmatrix}$$

4.5.6 Power Law Model Example

Let's take as an example data from the CDC about height vs weight graph:

```
x_values = [0.912, 0.986, 1.06, 1.13, 1.19, 1.26, 1.32, 1.38, 1.41, 1.49]
y_values = [13.7, 15.9, 18.5, 21.3, 23.5, 27.2, 32.7, 36, 38.6, 43.7]
```

Will give out the following output:

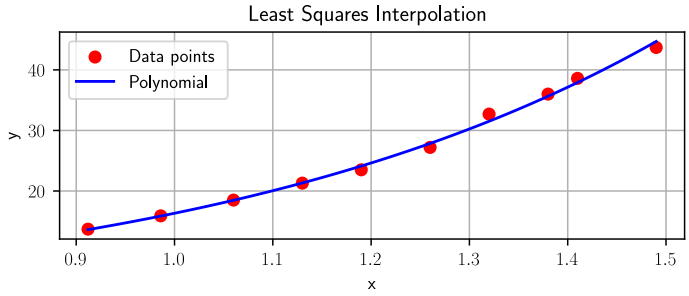
$$A = \begin{bmatrix} 1.0 & -0.092115 \\ 1.0 & -0.01409 \\ 1.0 & 0.05826 \\ 1.0 & 0.1222 \\ 1.0 & 0.17395 \\ 1.0 & 0.23111 \\ 1.0 & 0.27763 \\ 1.0 & 0.322083 \\ 1.0 & 0.343589 \\ 1.0 & 0.39877 \end{bmatrix}, C = \begin{bmatrix} k \\ c_1 \end{bmatrix} \text{ and } B = \begin{bmatrix} 2.617395 \\ 2.76631 \\ 2.917770 \\ 3.058707 \\ 3.15700 \\ 3.303216 \\ 3.487375 \\ 3.58351 \\ 3.65325 \\ 3.777348 \end{bmatrix}$$

$$A^T AC = A^T B \rightarrow \begin{bmatrix} 10.0 & 1.82141841576258 \\ 1.821418 & 0.568582 \end{bmatrix} \begin{bmatrix} k \\ c_1 \end{bmatrix} = \begin{bmatrix} 32.32190 \\ 6.46025 \end{bmatrix}$$

$$C = \begin{bmatrix} 2.7914328 \\ 2.419859 \end{bmatrix}$$

$$f(x) = 16.30436x^{2.419859}$$

And the following graph:



4.5.7 Exponential Model

The general form of the exponential law model is:

$$y = c_1 e^{c_2 t}$$
$$\ln y = \ln c_1 + c_2 t$$
$$\ln y = k + c_2 t \text{ where} k = \ln x$$
$$\mathrm{b} = \begin{bmatrix} \ln y_1 \\ \ln y_2 \\ \vdots \\ \ln y_n \end{bmatrix} \text{ and } \mathrm{C} = \begin{bmatrix} k \\ c_2 \end{bmatrix}$$

The matrix A would have the same form as a normal Vandermonde matrix with degree 1 using the unmodified x values.

4.5.8 Exponential Model Example

Let's take as an example the population growth of a city:

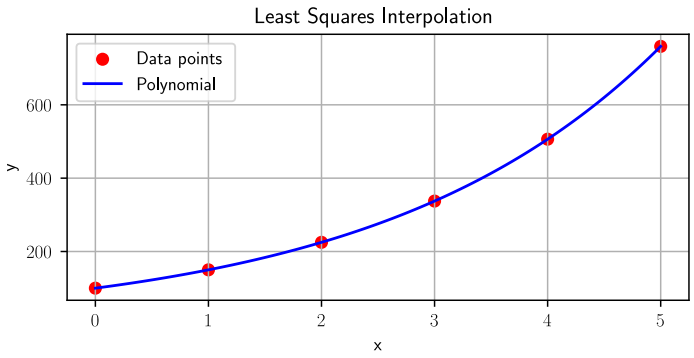
```
x_values = [0, 1, 2, 3, 4, 5]
y_values = [100, 150, 225, 337.5, 506.25, 759.375]
```

This is the output:

$$A = \begin{bmatrix} 1.0 & 0 \\ 1.0 & 1.0 \\ 1.0 & 2.0 \\ 1.0 & 3.0 \\ 1.0 & 4.0 \\ 1.0 & 5.0 \end{bmatrix}, C = \begin{bmatrix} k \\ c_2 \end{bmatrix} \text{ and } B = \begin{bmatrix} 4.6051 \\ 5.01063 \\ 5.41610 \\ 5.82156 \\ 6.2270 \\ 6.632495 \end{bmatrix}$$
$$A^T AC = A^T B \rightarrow \begin{bmatrix} 6.0 & 15.0 \\ 15.0 & 55.0 \end{bmatrix} \begin{bmatrix} k \\ c_2 \end{bmatrix} = \begin{bmatrix} 33.712997 \\ 91.37813 \end{bmatrix}$$

$$C = \begin{bmatrix} 4.60517 \\ 0.405465 \end{bmatrix}$$

$$f(x) = 100.0e^{0.4054x}$$



4.5.9 Drug Concentration Model

This model has the general form:

$$\mathrm{y} = \mathrm{c}_1 x e^{c_2 x}$$

$$\ln y = \ln c_1 + \ln t + c_2 t$$

$$\ln y - \ln t = \ln c_1 + c_2 t = k + c_2 t \text{ where } k = \ln x$$

$$\mathrm{b} = \begin{bmatrix} \ln y_1 - \ln x_1 \\ \ln y_2 - \ln x_2 \\ \vdots \\ \ln y_n - \ln x_n \end{bmatrix} \text{ and } \mathrm{C} = \begin{bmatrix} k \\ c_2 \end{bmatrix}$$

Matrix A will have the normal Vandermonde shape as the one of a first degree polynomial, same as the Exponential Model.

4.5.10 Drug Concentration Model Example

Let's the example of measured level of the drug norfluoxetine in a patient's bloodstream:

```
x_values = [1,2,3,4,5,6,7,8]
y_values = [8.0,12.3,15.5,16.8,17.1,15.8,15.2,14.0]
```

The output will be the following:

$$\mathrm{A} = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 2.0 \\ 1.0 & 3.0 \\ 1.0 & 4.0 \\ 1.0 & 5.0 \\ 1.0 & 6.0 \\ 1.0 & 7.0 \\ 1.0 & 8.0 \end{bmatrix}, C = \begin{bmatrix} k \\ c_2 \end{bmatrix} \text{ and } B = \begin{bmatrix} 2.07944154167984 \\ 1.81645208181843 \\ 1.64222773525709 \\ 1.43508452528932 \\ 1.22964055107451 \\ 0.968250470804866 \\ 0.775385278796917 \\ 0.559615787935423 \end{bmatrix}$$

$$\mathrm{A}^T AC = A^T B \rightarrow \begin{bmatrix} 8.0 & 36.0 \\ 36.0 & 204.0 \end{bmatrix} \begin{bmatrix} k \\ c_2 \end{bmatrix} = \begin{bmatrix} 10.5060979726564 \\ 38.2416958475088 \end{bmatrix}$$

$$\mathrm{C} = \begin{bmatrix} 2.28137778545115 \\ -0.215136786415356 \end{bmatrix}$$

$$\mathrm{f(x)} = 9.79015986132169 \text{ x } \mathrm{e}^{-0.215136786415356x}$$