Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Solution 1

In the entire question report we shall use j to represent the imaginary number, i.e. $\sqrt{-1}$. This is to ensure consistency with the Python programming language, which shall be used extensively in this question.

Before we can start to calculate the roots using Muller's method, we shall require three sets of co-ordinates, i.e. $]\omega_0, f(\omega_0))$, $(\omega_1, f(\omega_1))$ and $(\omega_2, f(\omega_2))$. We have been asked to use $\omega_0 = 0, \omega_1 = 1 + 0.5j, \omega_2 = 2 + j$.

The formulae to calculate the coefficients are as follows:

$$a = \frac{(\omega_1 - \omega_2)\left(f(\omega_0) - f(\omega_2)\right) - (\omega_0 - \omega_2)\left(f(\omega_1) - f(\omega_2)\right)}{(\omega_0 - \omega_2)(\omega_1 - \omega_2)(\omega_0 - \omega_1)}$$
$$b = \frac{(\omega_0 - \omega_2)^2\left(f(\omega_1) - f(\omega_2)\right) - (\omega_1 - \omega_2)^2\left(f(\omega_0) - f(\omega_2)\right)}{(\omega_0 - \omega_2)(\omega_1 - \omega_2)(\omega_0 - \omega_1)}$$
$$c = f(\omega_2)$$

The formula to calculate the next iteration of ω is as follows:

$$\omega_3 = \omega_2 - \frac{2c}{b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}}$$

The $\mathrm{sgn}(b)$ is used to so that the magnitude of the denominator is maximised. However, because the $\mathrm{sgn}(b)$ is not defined for complex number, we will have to use a more straightforward approach to maximise the denominator:

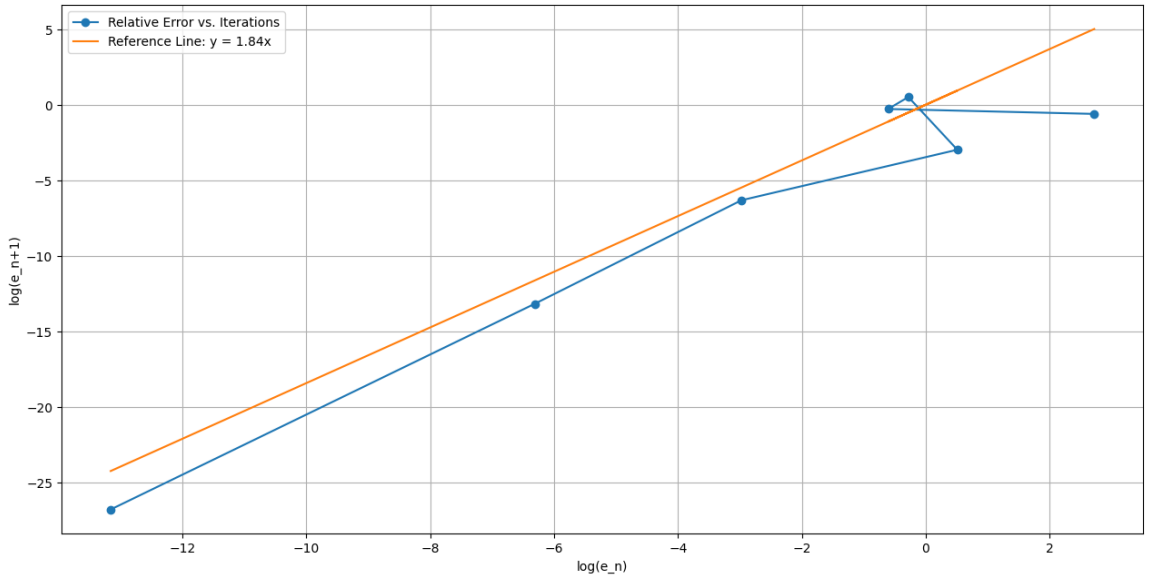
$$denominator = \max(b \pm \sqrt{b^2 - 4ac})$$

This can be done using a simple `if-else` block.

The Python code required to total of 8 iterations to narrow down the relative error between consecutive frequencies (i.e. values of ω) to the order of 10^{-6}. The final approximated root is $-0.069965 - 0.109208j$. The function value at root is $(-9.540979 \times 10^{-18} + 0j)$.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

The convergence plot is as follows:



The above plot demonstrates that the Mullers method converges in $O(n^{1.84})$ time.

However, to test this, I decided to implement Newton's method for this question. However, before we test using Newton's method, we shall require the first derivative of the function, i.e. $f'(\omega)$. Let us calculate that:

$$f(\omega) = \omega^3 + (2 - 0.5j)\omega^2 + (1.5 + j)\omega + 0.2j$$
$$f'(\omega) = 3\omega^2 + 2(2 - 0.5j)\omega + (1.5 + j)$$
$$f'(\omega) = 3\omega^2 + (4 - j)\omega + (1.5 + j)$$

The formula to calculate the next iteration of ω using Newton's method is:

$$\omega_{n+1} = \omega_n - \frac{f(\omega)}{f'(\omega)}$$

The iteration converged in 7 iterations, and displayed the root to be $-0.069964 - 0.109208j$. This actually matches the answer given by Muller's method. Hence, we can say that Newton's method converges to the correct answer for this particular question. However, the Newton's method does not provide any guarantee for equations involving complex numbers and complex roots.

In other words, although Newton's method can sometimes find complex roots, its behaviour in the complex plane is unpredictable if the initial guess is not close enough to the root. Muller's method is relatively more robust as it doesn't require a derivative and its convergence (order 1.84) is nearly as fast as Newton's (order 2).

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Algorithm

Because ω is not a valid variable name in Python, we will be using x to represent ω in both the Python code and in the below algorithm.

The program steps are as follows:

1. The initial guesses, number of iterations and tolerance, i.e. ϵ are all initialised.
2. After this, the iterations are started.
   (a) We first calculate the functional values of the three points that we have, i.e.$f(x_0), f(x_1)$ and $f(x_2)$.
   (b) In each iteration, we shall calculate the values of a, b and c based on the formulae above.
   (c) We calculate the denominator using the maximisation as mentioned above.
   (d) The next value of x is calculated, again using the above formula.
   (e) The relative error is calculated and be added to an array of error, which will be used to plot the convergence graph later.
   (f) If the relative error satisfies our tolerance requirement, we simply break the loop, and jump to the final output.
3. Once the iterations are completed, the data is processed and the graph is plotted.
4. Then, the program tries to re-calculate the solution using Newton's method. The program, in this case, ends up calculating the same root as Muller's method.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Solution 2

A divided difference table is used to find the coefficients of an interpolating polynomial. It systematically organizes the process of calculating divided differences, which are approximations of derivatives. The divided difference table, as calculated using Python is as follows:

x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$
0.0	150	30.0	-16.667	10.417
0.5	165	10.0	4.167	-
1.2	172	16.25	-	-
2.0	185	-	-	-

Once the divided difference table has been calculated, the interpolating polynomial can be calculated as follows:

$$P_n(x) = f[x_0] + (x - x_0) f[x_0, x_1] + \cdots + (x - x_0)(x - x_1) \ldots (x - x_{n-1}) f[x_0, x_1, \ldots, x_n]$$

Hence, the interpolating polynomial calculated using this method was: $T(x) = 150 + 44.583x - 34.375x^2 + 10.417x^3$.

Using the above polynomial, we can find the value of wall temperature when $x = 1.0$m. The calculation is:

$$f(1.0) \approx T(1.0) = 170.63^{\cdot}\ \text{C}.$$

Recalculating using system of linear equations

There is a rule with regards to interpolating polynomials. This rule states that "The interpolating polynomial should perfectly pass through the data points. Mathematically, this can be written as $P(x_i) = f(x_i)$.

Let us assume that the interpolating polynomial can be written as $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$.

Hence,

$$\begin{aligned} P(x_0) &= a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n = f(x_0) \\ P(x_1) &= a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n = f(x_1) \\ &\vdots \\ P(x_n) &= a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_n x_n^n = f(x_n) \end{aligned}$$

The system of equations shall look as follows:

$$\begin{bmatrix} 1 & x_0 & \ldots & x_0^n \\ 1 & x_1 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & \ldots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_2) \end{bmatrix}$$

The same question, when solved by calculating solutions to linear equations, was as: $T_p(x) = 150 + 44.583x - 34.375x^2 + 10.417x^3$.

Hence, it is proved that the interpolating polynomial is unique for the given data points.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Algorithm

The program steps are as follows:

1. The given data points are initialised.
2. The divided difference table is constructed, and printed as output.
3. The polynomial coefficients are then calculated, such that the interpolating polynomial can be output in the format $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Technically calculating the standard version of the polynomial is not necessary. We can perform interpolation using just the divided difference table. However, There are two advantages of calculating the polynomial like this:
   (a) The polynomial output is "clean" and easier to interpret.
   (b) The degree of the polynomial is clearly visible. In this case, for example, the degree of the polynomial is expected to be 3 (i.e. a cubic polynomial) because there are four data points.
   (c) The comparison with the polynomial generated using linear systems of equations will be easier.
   (d) The interpolation results will be easier to calculate. If we do not do this, $O(n^2)$ time will be required. Now, it can be done in $O(n)$ time. This is very helpful if multiple interpolations need to be made.
4. After the coefficients are calculated, they are neatly output to the terminal.
5. The function value at $x = 1.0$ is then calculated and output to the terminal.
6. Finally, we try to re-calculate the polynomial by solving a system of linear equations.
7. For this, I implemented a new function which is called solve_linear_equation(). This method is basically the np.linalg.solve() method, but a custom implementation.
8. Lastly, the polynomial generated by the linear system of equations is also output. In this case, both equations are exactly the same.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Solution 3

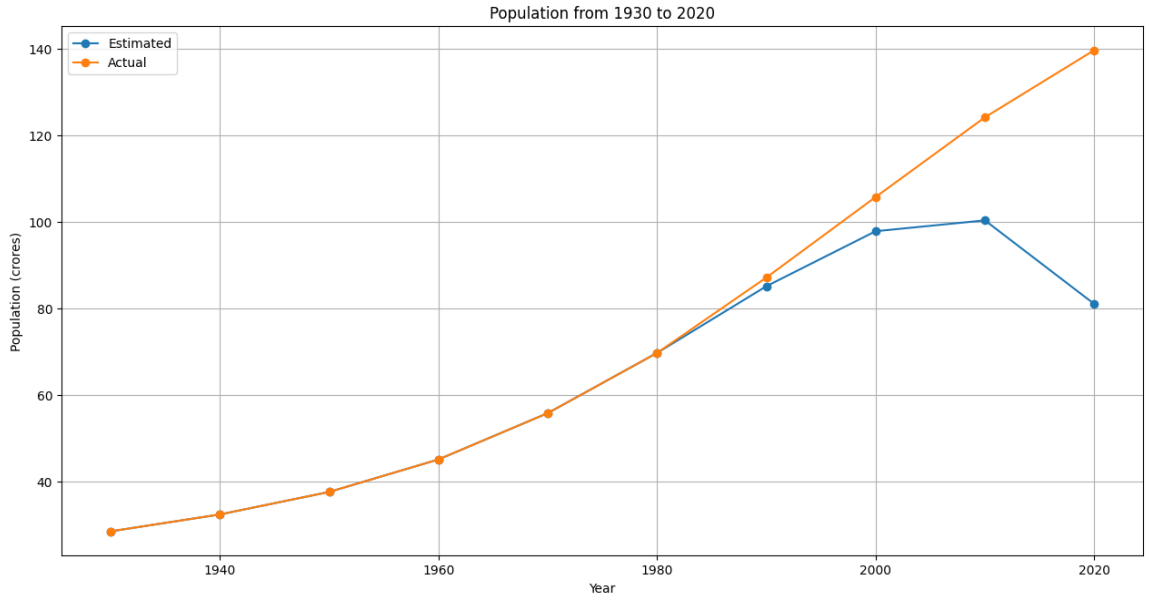
To calculate the Lagrange interpolating polynomial, we have to first calculate the Lagrange coefficients, i.e. $l_i(x)$. The formula for $l_i(x)$ is:

$$l_i(x) = \prod_{j=0}^{j=n, j\neq i} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \times \ldots \times \frac{x - x_{i-1}}{x_i - x_{i-1}} \times \frac{x - x_{i+1}}{x_i - x_i + 1} \times \ldots \times \frac{x - x_n}{x_i - x_n}$$

The Lagrange interpolating polynomial, as calculated by Python is:

$$P(x) = (-6.666667 \times 10^{-8})x^5 + 0.000649583x^4 - 2.53156x^3 + 4932.63x^2 - (4.80517 \times 10^6)x + (1.87227 \times 10^9)$$

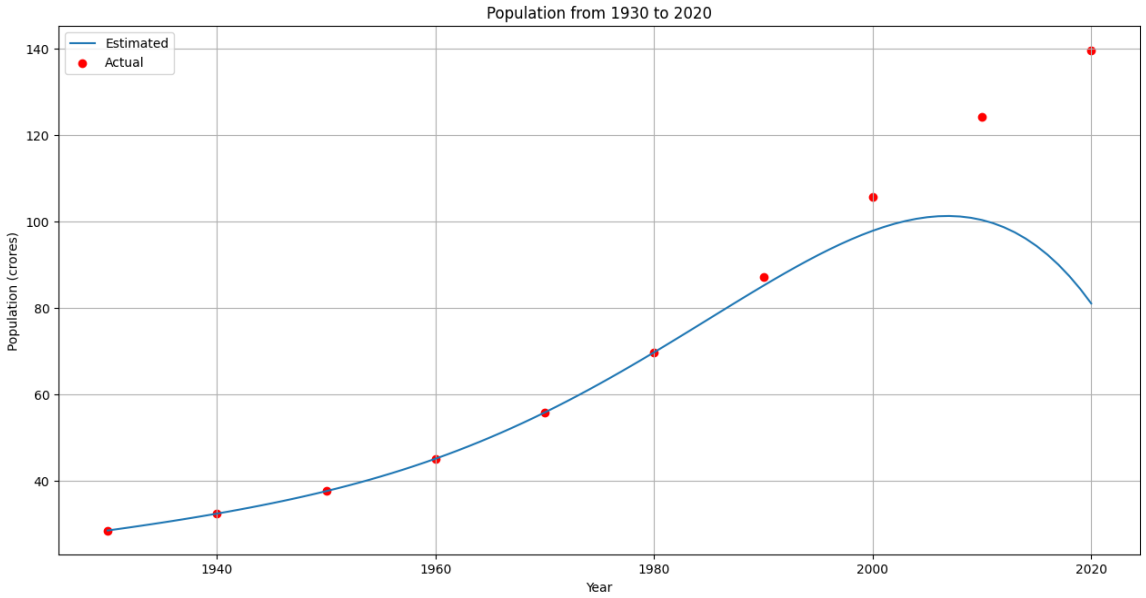
When this polynomial was plotted for against the years, the graph is as follows:



As can be seen in the graph, the graph is extremely accurate from the year 1930 until the year 1980. This is because we have considered only the points from 1930 to 1980. After this range, the polynomial starts giving wrong answers. In other words, the polynomial outputs values which are not equal to actual values.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

This can be better visualised when the Lagrange polynomial is plotted as a smooth curve. This smooth curve was obtained by estimating the population at every year from 1930 to 2020. The graph is as follows:



One of the noticeable things in the graph is that the graph at least predicts an increase in population for 1990, 2000 and 2010. However, it predicts a sudden decrease in population in 2020.

The estimates generated by the polynomial are as follows:

Year	Estimated Population (crores)	Actual Population(crores)
1990	85.099	87.1
2000	97.799	105.7
2010	100.299	124.1
2020	81.000	139.6

We use the following formula to calculate the mean square error:

$$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i')^2}{n}$$

The mean square error of these estimates comes out to 1016.756 crores. Hence, we can confidently state that this method of using Lagrange interpolation to predict the future is not reasonable. This is because Lagrange interpolation is mostly supposed to interpolate the data point.

In other words, Lagrange interpolation can find the data which is within the boundary of the provided data points. For example, in this case, Lagrange interpolation may work fairly accurately if we have to estimate the data of somewhere around 1955 or so.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Algorithm

The program steps are as follows:

1. The given data points are initialised.
2. The Lagrange coefficients are calculated.
3. Just like before, the polynomial coefficients are then calculated, such that the interpolating polynomial can be output in the format $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Technically calculating the standard version of the polynomial is not necessary. We can perform interpolation using just the Lagrange coefficients. However, There are two advantages of calculating the polynomial like this:
   (a) The polynomial output is "clean" and easier to interpret.
   (b) The degree of the polynomial is clearly visible. In this case, for example, the degree of the polynomial is expected to be 5 (i.e. a quintic polynomial) because there are six data points.
   (c) The interpolation results will be easier to calculate. If we do not do this, $O(n^2)$ time will be required. Now, it can be done in $O(n)$ time. This is very helpful if multiple interpolations need to be made.
4. After the coefficients are calculated, they are neatly output to the terminal.
5. The function values are calculated at $x = 1930, 1940, \cdots, 2020$ and output to the terminal.
6. Then, the actual data and the calculated data are plotted on the same graph. It can be seen that the interpolating polynomial shall match the input data exactly, i.e. with zero error. However, the error tends to be large as we foray outside the range of the input data.
7. Lastly, the same Lagrange polynomial is evaluated from 1930 to 2020 in order to plot a smooth curve, and compare it with the actual data.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Solution 4

Part (a)

To obtain the Hermitian interpolating polynomial, we first have to calculate the divided difference table. The divided difference table is as follows:

z	x	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$	$j=8$	$j=9$
0	0.0000	–	–	–	–	–	–	–	–	–
1	0.0000	75.0000	–	–	–	–	–	–	–	–
2	225.0000	75.0000	0.0000	–	–	–	–	–	–	–
3	225.0000	77.0000	0.6667	0.2222	–	–	–	–	–	–
4	383.0000	79.0000	1.0000	0.0667	−0.0311	–	–	–	–	–
5	383.0000	80.0000	0.5000	−0.2500	−0.0633	−0.0064	–	–	–	–
6	623.0000	80.0000	0.0000	−0.1000	0.0300	0.0117	0.0023	–	–	–
7	623.0000	74.0000	−2.0000	−0.6667	−0.1133	−0.0287	−0.0050	−0.0009	–	–
8	993.0000	74.0000	0.0000	0.2500	0.1146	0.0228	0.0051	0.0008	0.0001	–
9	993.0000	72.0000	−0.4000	−0.0800	−0.0413	−0.0195	−0.0042	−0.0009	−0.0001	−0.0000

In the above table, the value of j represents the order of difference. For example, $j=1$ represents the first order divided difference, i.e. $f[x_i, x_{i+1}]$, and so on.

Using this table and the formula for obtains the Hermitian polynomial, we can calculate the Hermitian polynomial. The formula used is as follows:

$$H(x) = f[z_0] + (z - z_0) f[z_0, z_1] + \cdots + (z - z_0)(z - z_1) \ldots (z - z_{2n+1}) f[z_0, z_1, \ldots, z_{2n+1}]$$

The Hermitian polynomial obtained using this formula is:

$$H(x) = 75x + 7.16191x^2 - 10.0953x^3 + 5.50812x^4 - 1.5383x^5$$
$$+0.243041x^6 - 0.0218757x^7 + 0.00104059x^8 - (2.02236 \times 10^{-5})x^9$$

Substituting $x = 10$ seconds into the above polynomial, we get $f(x) = 742.502$ feet.

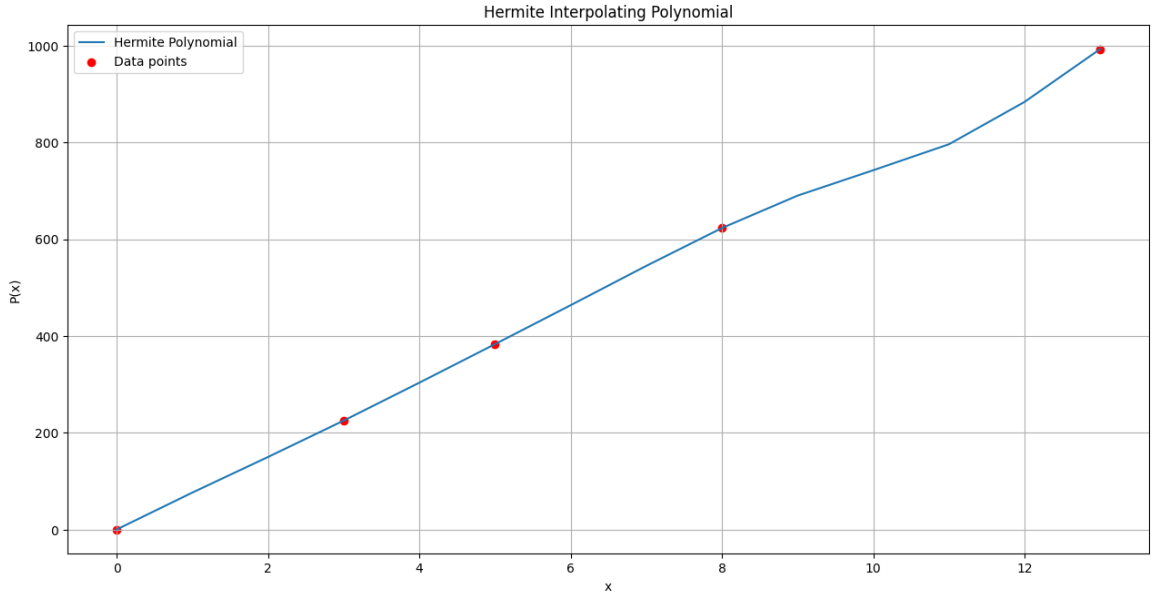
Now, we have to calculate the approximate speed at $x = 10$ seconds. For this, we can use the fundamental definition of derivative. This definition is:

$$f'(\alpha) = \lim_{h \to 0} \left(\frac{f(\alpha + h) - f(\alpha)}{h} \right)$$

Using this formula, we get that $f'(10) \approx H'(10) = 48.382$ feet/second.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

Lastly, we have to plot the graph of $H(x)$ from $x = 0$ seconds to $x = 13$ seconds. The graph is as follows:



As can be seen in the graph, the Hermitian polynomial is actually an interpolating polynomial as it perfectly passes through the original data points, which are marked in red.

Algorithm

The program steps are as follows:

1. Create three array with provided input data.
2. Get the divided difference table
   (a) The table is populated using the rules provided in the lecture notes.
   (b) For each original index i assign $z_{2i} = z_{2i+1} = x_i$.
   (c) Set the zeroth column $Q[2i, 0] = Q[2i + 1, 0] = f_i$.
   (d) For the repeated node pairs insert derivative information as $Q[2i + 1, 1] = f'(x_i)$.
   (e) For the other first divided differences compute $Q[2i, 1] = \dfrac{Q[2i, 0] - Q[2i - 1, 0]}{z_{2i} - z_{2i-1}}$ when applicable.
3. Use the divided difference table to obtain the Newton coefficients. These Newton coefficients can be used to complete the polynomial using this formula:

$$H(x) = f[z_0] + (z - z_0) f[z_0, z_1] + \cdots + (z - z_0)(z - z_1) \ldots (z - z_{2n+1}) f[z_0, z_1, \ldots, z_{2n+1}]$$

4. Use these Newton coefficients to obtain the Hermite polynomial in standard form, i.e. $H(x) = a + bx^1 + cx^2 + \cdots$.
5. We then evaluate this polynomial at $x = 10$ seconds, as mentioned in the question.

6. We then also find the approximate speed of the car at $x = 10$ seconds, using the fundamental definition of derivative. This definition is:

$$f'(\alpha) = \lim_{h \to 0} \left(\frac{f(\alpha + h) - f(\alpha)}{h} \right)$$

7. We then plot the Hermite polynomial.

Part (b)

To get the maximum value of speed, which is represented by $f'(x)$, we need to find the second derivate of $f(x)$, i.e. $f''(x)$. This is actually the function to approximate the acceleration of the car.

After we find the expression for $f''(x)$, we need to find its roots. Since this will be a polynomial of degree 8, we cannot use the traditional methods. Instead, we shall use one of the root finding methods from the previous assignment.

The first derivative of $f(x)$, i.e. $f'(x)$ is:

$$f'(x) \approx H^{'}(x) = 75 + 14.3238x - 30.2859x^2 + 22.0325x^3 - 7.69148x^4 + 1.45825x^5 - 0.15313x^6 + 0.00832472x^7 - 0.000182013x^8$$

This can also be considered as the speed function.

The second derivative of $f(x)$, i.e. $f''(x)$ is:

$$f''(x) \approx H^{''}(x) = 14.3238 - 60.5719x + 66.0974x^2 - 30.7659x^3 + 7.29124x^4 - 0.918778x^5 + 0.0582731x^6 - 0.0014561x^7$$

For finding the roots, we shall use the Newton-Raphson method. The Newton-Raphson method has the following iterative formula:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

This method, however, requires an initial guess. Another requirement of this method is that we require the first derivative of the function. However, in this case, because we are finding the roots of the second derivative of $f(x)$, we shall require the third derivative of $f(x)$, i.e. $f^{(3)}(x)$.

$$f^{(3)}(x) \approx H^{(3)}(x) = -60.5719 + 132.195x - 92.2977x^2 + 29.1649x^3 - 4.59389x^4 + 0.349638x^5 - 0.0101927x^6$$

For our initial guess, we shall take $x = 14$ seconds. Before we start the iterations, we also need to decide an error margin, i.e. ϵ. In this case, let us take the value of ϵ to be 10^{-6}. Also, we shall be using absolute error in this case.

The Python code took a total of 7 iterations to calculate the value of x as $x = 12.371$ seconds. The following values follow:

$x = 12.371$	Time
$f(x) \approx H(x) = 926.978$	Distance (function value)
$f'(x) \approx H'(x) = 119.417$	Speed (first derivative)
$f''(x) \approx H''(x) = -1.455 \times 10^{-11} < \epsilon$	Acceleration (second derivative)
$f^{(3)}(x) \approx H^{(3)}(X) = -158.36 < 0$	Jerk (third derivative)

To prove that the point is actually the maximum, we can use the second derivate test. From the above list of values, we can see that $f^{(3)}(x) < 0$. This states that the value is actually a maximum.

Hence, the maximum speed of the car is 119.417 feet/second.

Algorithm

From part (a) we have the Hermite polynomial. We can use it to continue to part (b). From this line onwards, the program steps are as follows:

1. We calculate the first, second and third derivatives of $H(x)$, which is take to be approximately equal to the first, second and third derivatives of $f(x)$.
2. To obtain the maximum speed, we take the second derivative and equate it to 0. We then have to find the roots of the equation.
   (a) To find the roots, we shall use the Newton-Raphson method.
   (b) The initial guess is 14 seconds, and the error tolerance is 10^{-6}.
3. Once the root of $f''(x) \approx H''(x) = 0$ is obtained, we print out the following values.
   - x
   - $f(x) \approx H(x)$
   - $f'(x) \approx H'(x)$
   - $f''(x) \approx H''(x)$
   - $f^{(3)}(x) \approx H^{(3)}(X)$
4. We can verify that the speed is actually maximum because $f^{(3)}(x) \approx H^{(3)}(X) < 0$.

Name:	Suhas Kamath	**Assignment No:**	Assignment 2
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS288/UMC202
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	September 10, 2025	**Term:**	AUG 2025

References

[1] Faires, J. Douglas, and Richard L. Burden. Numerical methods, 4th. Cengage Learning, 2012.

[2] Lecture Notes