Python, NM & Data Science

SharafatKarim

Keywords python, jupyter-notebook, numerical-methods, data-analysis, data-visualization, data-science

Published Nov 19, 2025

Contents

1. Numerical Methods

1.a. Linear Equations

A linear equation is a mathematical statement that represents a straight line when graphed.

Then let's take input from user, in the following format,

```
Enter number of unknowns: 3
Enter row 1 in the format (ax + by + cz = d)
```

```python
# User input (auto)
# N = int(input("Enter number of unknowns: "))

# arr = []
# for i in range(N):
#     arr.append(list(map(int, input().split())))

# User input (manual)
N = 3
arr = [[2, 1, -1, 8], [-3, -1, 2, -11], [-2, 1, 2, -3]]
```

```python
# User input's output
print(N)

def print_arr(arr):
  for i in range(N):
    print(arr[i])

print_arr(arr)
```

```
3
[2, 1, -1, 8]
[-3, -1, 2, -11]
[-2, 1, 2, -3]
```

1.a.i. *Gaussian Elimination:*

```python
import copy

def gaussian_elimination(N, arr):
  arr = copy.deepcopy(arr)
  for i in range(N):
    for j in range(N, -1, -1):
      arr[i][j] /= arr[i][0+i]

    for j in range(i+1, N):
      for k in range(N, -1, -1):
        arr[j][k] -= arr[i][k] * arr[j][0+i]

    print_arr(arr)
    print()

  solve = [0 for j in range(N)]
  for i in range(N-1, -1, -1):
    for j in range(N):
      solve[i] = arr[i][N]
      for k in range(i+1, N):
        solve[i] -= arr[i][k] * solve[k]
    solve[i] /= arr[i][i]

  print(solve)

gaussian_elimination(N, arr)
```

```
[1.0, 0.5, -0.5, 4.0]
[0.0, 0.5, 0.5, 1.0]
[0.0, 2.0, 1.0, 5.0]

[1.0, 0.5, -0.5, 4.0]
[0.0, 1.0, 1.0, 2.0]
[0.0, 0.0, -1.0, 1.0]

[1.0, 0.5, -0.5, 4.0]
[0.0, 1.0, 1.0, 2.0]
[0.0, 0.0, 1.0, -1.0]

[2.0, 3.0, -1.0]
```

1.a.ii. *Gauss Jordan:*

```
import copy
arr = copy.deepcopy(arr)

def gauss_jordan(arr, N):
    for i in range(N):
        for j in range(N):
            if i != j:
                p = arr[j][i] / arr[i][i]
                for k in range(N+1):
                    arr[j][k] -= arr[i][k] * p
        print_arr(arr)
        print()

    for i in range(N):
        print(arr[i][3] / arr[i][i])

gauss_jordan(arr, N)
```

```
[2, 1, -1, 8]
[0.0, 0.5, 0.5, 1.0]
[0.0, 2.0, 1.0, 5.0]

[2.0, 0.0, -2.0, 6.0]
[0.0, 0.5, 0.5, 1.0]
[0.0, 0.0, -1.0, 1.0]

[2.0, 0.0, 0.0, 4.0]
```

```
[0.0, 0.5, 0.0, 1.5]
[0.0, 0.0, -1.0, 1.0]

2.0
3.0
-1.0
```

1.a.iii. *Cramers' rule:*

Another one easiest way is to use cramer's rule. It's not so dynamic but it works for small systems of equations. The idea is to express the solution in terms of determinants.

```
import numpy as np

main_array = np.array(arr)
D = np.array(main_array[:,:-1])

last_col  = np.array(main_array[:,-1])

D1 = np.array([last_col, main_array[:,1], main_array[:,2]])
D2 = np.array([main_array[:,0], last_col, main_array[:,2]])
D3 = np.array([main_array[:,0], main_array[:,1], last_col])

# We can also define our own det function for n specific variable easily
like,
# def det(arr):
#     a =   arr[0][0] * ( arr[1][1] * arr[2][2] - arr[1][2] * arr[2]
[1] )
#     b = - arr[1][0] * ( arr[0][1] * arr[2][2] - arr[0][2] * arr[2]
[1] )
#     c =   arr[2][0] * ( arr[0][1] * arr[1][2] - arr[0][2] * arr[1]
[1] )
#     return a + b + c

D_det = np.linalg.det(D)
D1_det = np.linalg.det(D1)
D2_det = np.linalg.det(D2)
D3_det = np.linalg.det(D3)

print("x = ", D1_det / D_det)
print("y = ", D2_det / D_det)
print("z = ", D3_det / D_det)
```

```
x =  2.0
y =  2.9999999999999996
z =  -0.9999999999999998
```

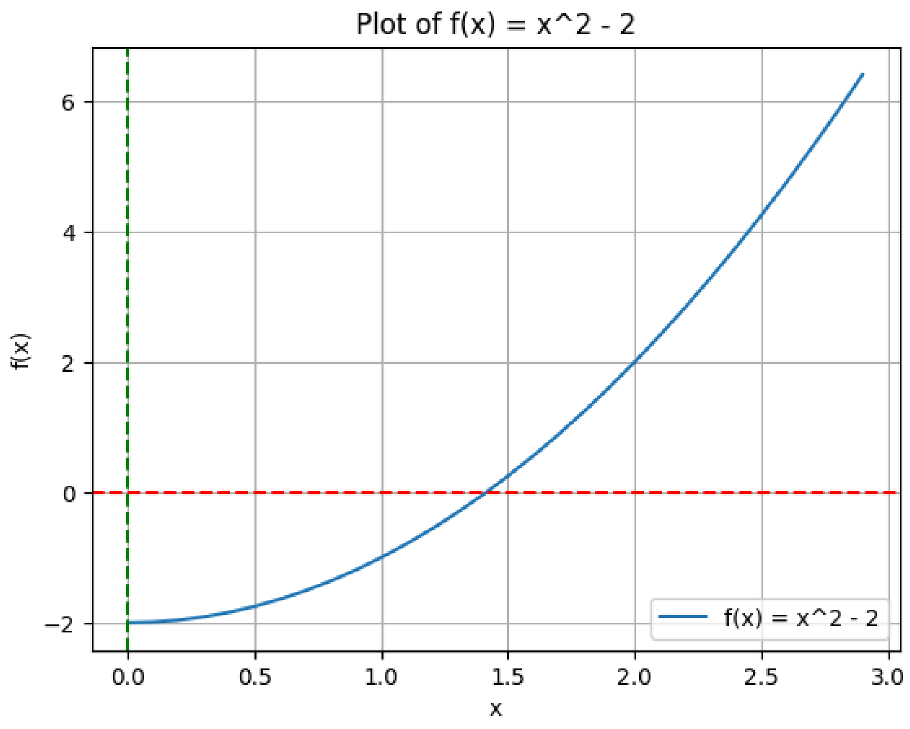
1.b. Root Finding

Root finding refers to the process of finding solutions to equations of the form f(x) = 0. This is a fundamental problem in numerical analysis and has various applications in science and engineering.

```python
# First let's import necessary libs
import matplotlib.pyplot as plt
import numpy as np
import math
```

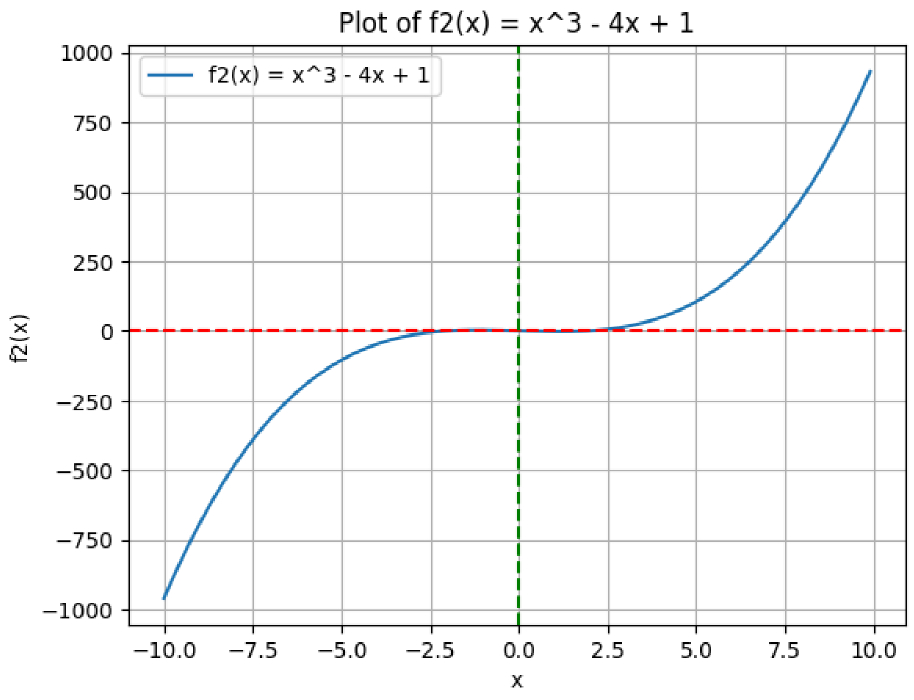
```python
# Our first function
def f(x):
  return x**2 - 2

# Let's plot the function
x = np.arange(0, 3, 0.1)
plt.plot(x, f(x), label='f(x) = x^2 - 2')

plt.axhline(0, color='red', linestyle='--')
plt.axvline(0, color='green', linestyle='--')

plt.xlabel('x')
plt.ylabel('f(x)')

plt.title('Plot of f(x) = x^2 - 2')
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef91f83b60>
```



```
# Our second function
def f2(x):
  return x**3 - 4 * x + 1

# Let's plot the function
x = np.arange(-10, 10, 0.1)
plt.plot(x, f2(x), label='f2(x) = x^3 - 4x + 1')

plt.axhline(0, color='red', linestyle='--')
plt.axvline(0, color='green', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')

plt.title('Plot of f2(x) = x^3 - 4x + 1')
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fc59090>
```



1.b.i. *Bisection Method:*

Bisection method finds the root of a function *f* in the interval [a, b].

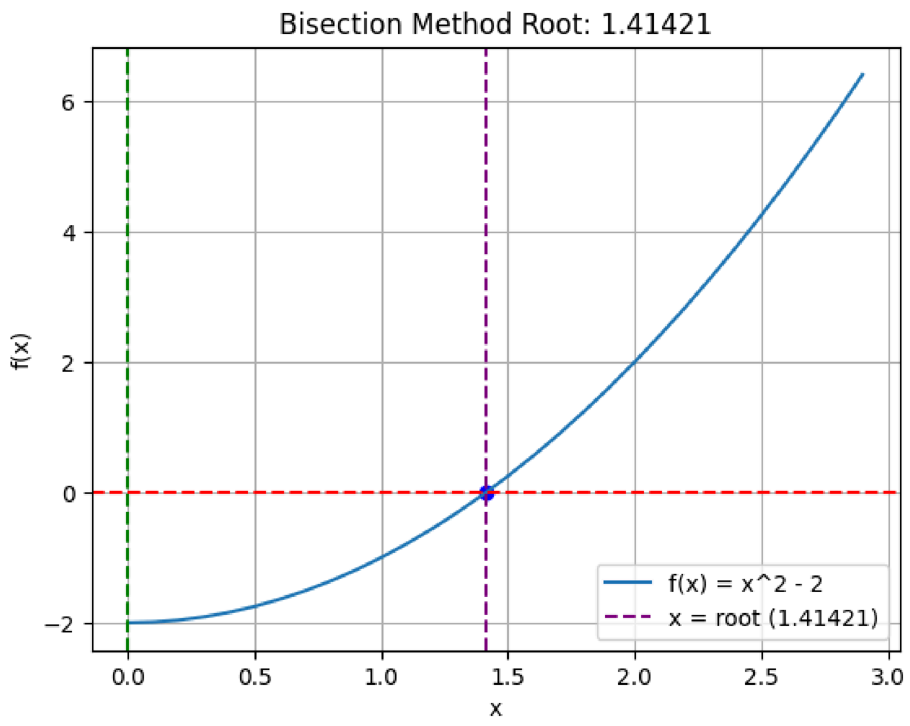
```
Parameters:
- f : function
  The function for which we want to find the root.
- a : float
  The start of the interval.
- b : float
  The end of the interval.
```

```
- tol : float
  The tolerance for convergence.

Returns:
- float
  The approximate root of the function.
```

```python
def bisection_method(f, a, b, tol=1e-5):
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) and f(b) must have opposite signs.")

    mid = (a + b) / 2.0

    if abs(f(mid)) < tol:
        return mid
    elif f(a) * f(mid) < 0:
        return bisection_method(f, a, mid, tol)
    else:
        return bisection_method(f, mid, b, tol)
```

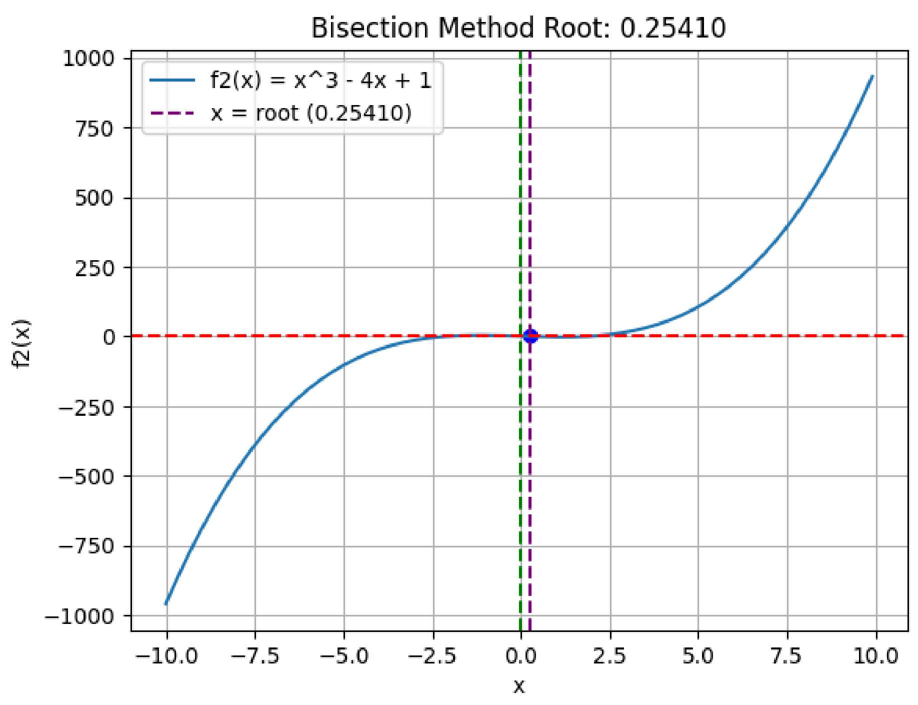
```python
root = bisection_method(f, 0, 10)

# Let's plot the result
x = np.arange(0, 3, 0.1)

plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title(f"Bisection Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fd27d90>
```



```
root = bisection_method(f2, -1, 1)

# Let's plot the result
x = np.arange(-10, 10, 0.1)

plt.plot(x, f2(x), label='f2(x) = x^3 - 4x + 1')
plt.scatter(root, f2(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')
plt.title(f"Bisection Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

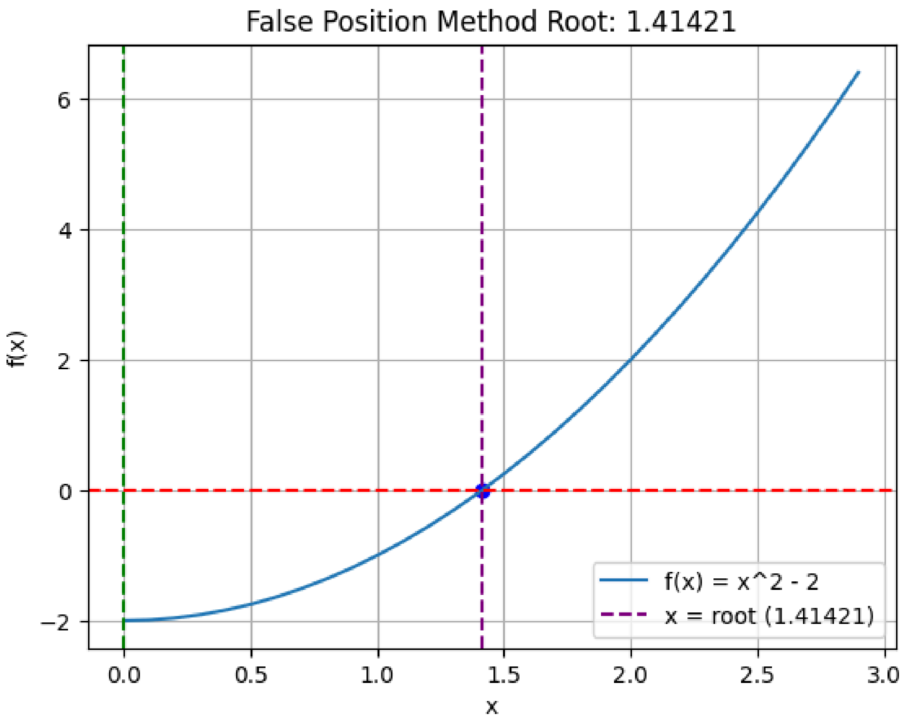
```
<matplotlib.legend.Legend at 0x7fef8fdb3750>
```



1.b.ii. *False Position Method:*

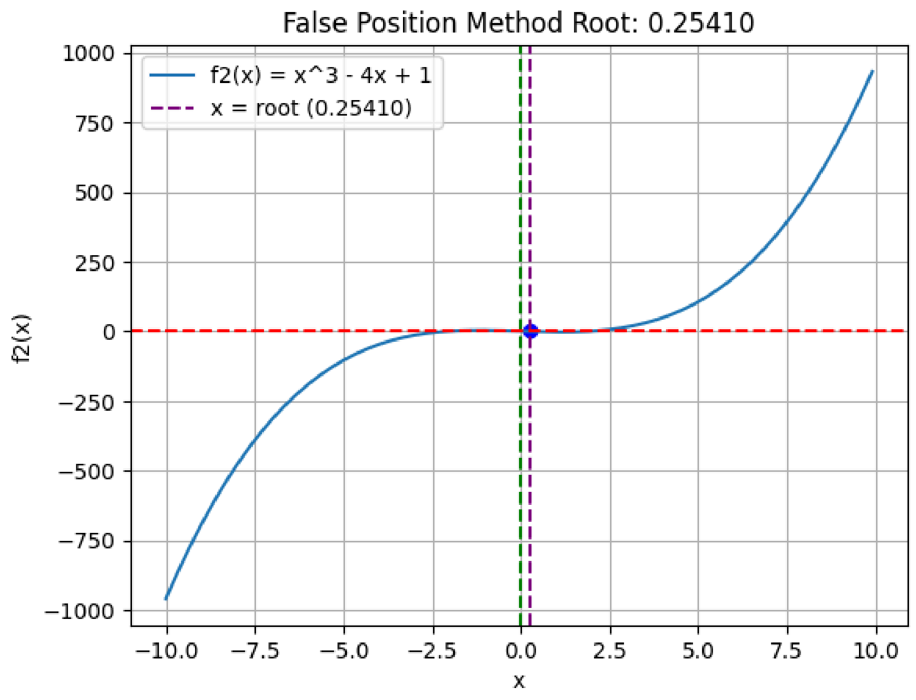
```
def false_position_method(f, a, b, tol=1e-5):
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) and f(b) must have opposite signs.")

    c = a - (f(a) * (b - a)) / (f(b) - f(a))

    if abs(f(c)) < tol:
        return c
    elif f(a) * f(c) < 0:
        return false_position_method(f, a, c, tol)
    else:
        return false_position_method(f, c, b, tol)
root = false_position_method(f, 0, 10)

# Let's plot the result
x = np.arange(0, 3, 0.1)

plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title(f"False Position Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fa49d10>
```



```
root = false_position_method(f2, -1, 1)

# Let's plot the result
x = np.arange(-10, 10, 0.1)

plt.plot(x, f2(x), label='f2(x) = x^3 - 4x + 1')
plt.scatter(root, f2(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')
plt.title(f"False Position Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8f8dd590>
```



1.b.iii. *Iteration Method:*

The iteration method is a common approach to find roots of equations. It involves rearranging the equation into the form x = g(x) and then iteratively applying g to an initial guess until convergence.

Let's consider the equation $x^3 - 4x + 1 = 0$. We can rearrange it to $x = \frac{x^3+1}{4}$.

```python
def f2(x):
    return x**3 - 4*x + 1

def g2(x):
    return (x**3 + 1) / 4
```

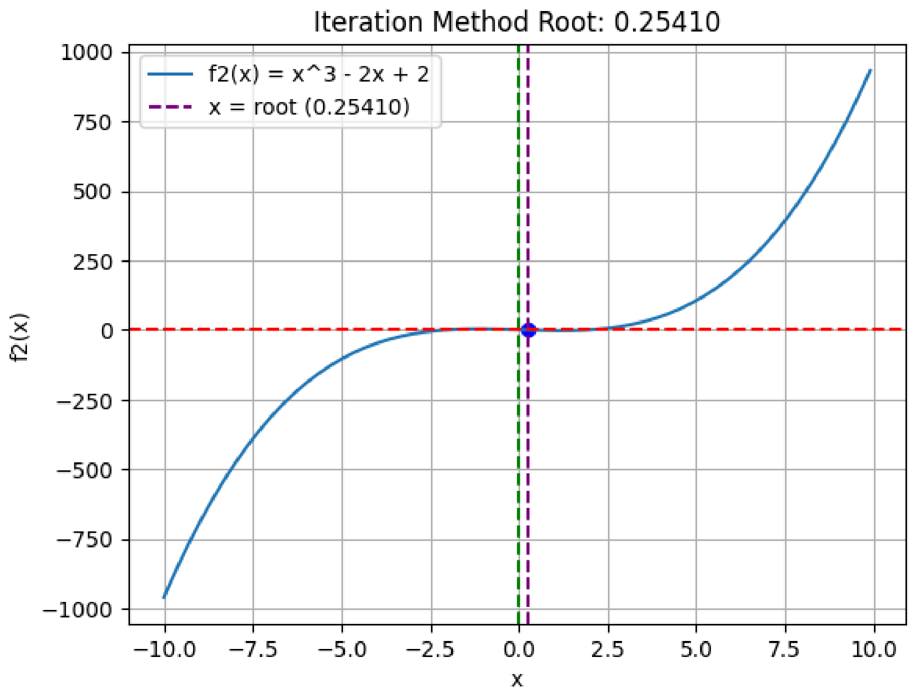
Let's implement the iteration methdod for this case,

```python
def iteration_method(g, x, tol=1e-5, max_iter=100):
    for i in range(max_iter):
        x_new = g(x)
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    raise ValueError(
        "Iteration did not converge within the maximum number of
iterations."
    )
```

And finally plotting time,

```
root = iteration_method(g2, 1)

# Let's plot the result
x = np.arange(-10, 10, 0.1)

plt.plot(x, f2(x), label='f2(x) = x^3 - 2x + 2')
plt.scatter(root, f2(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')
plt.title(f"Iteration Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8f957b10>
```



1.b.iv. *Newton-Raphson Method:*

The Newton-Raphson method is an efficient root-finding algorithm that uses the derivative of a function to find its roots. The method starts with an initial guess and iteratively refines it using the formula:

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

where:

- x_n - is the current guess,
- $f(x_n)$ - is the value of the function at x_n,
- $f'(x_n)$ - is the value of the derivative of the function at x_n.

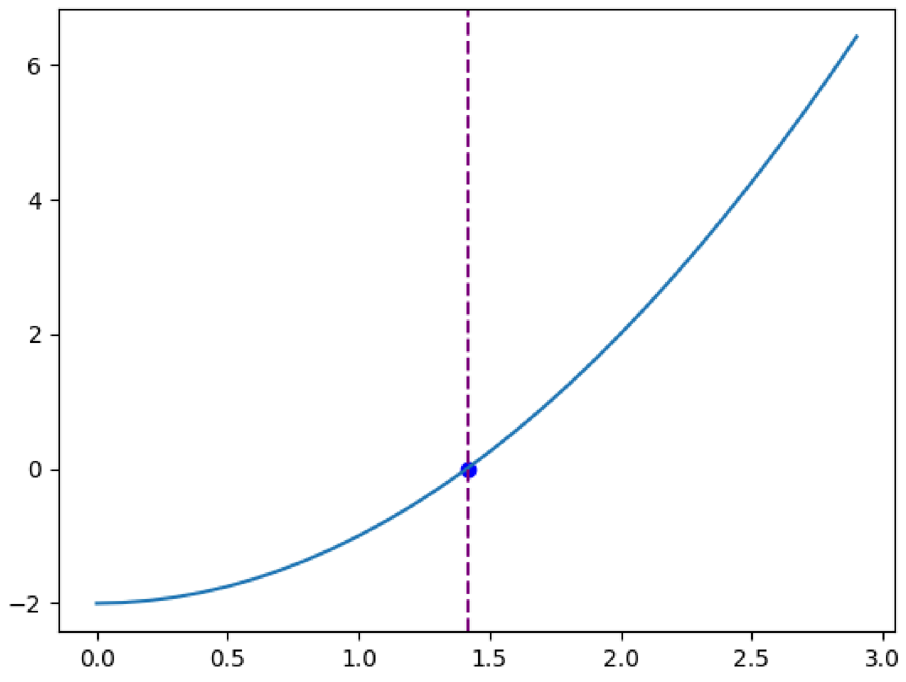
```
# Our function
def f(x):
    return x**2 - 2

def df(x):
    return 2*x

# Newton-Raphson Method
def newton_raphson_method(f, df, x0, tol=1e-5, max_iter=100):
    x = x0
    for i in range(max_iter):
        x_new = x - f(x) / df(x)
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    raise ValueError(
        "Newton-Raphson method did not converge within the maximum
number of iterations."
    )

root = newton_raphson_method(f, df, 1)
print(root)
```

```
1.4142135623746899
```

Let’s plot it…

```
root = newton_raphson_method(f, df, 10)

# Let's plot the result
x = np.arange(0, 3, 0.1)
plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')
```

```
<matplotlib.lines.Line2D at 0x7fef8f83dd10>
```



1.b.v. *Secant Method:*

The secant method is a root-finding algorithm that uses two initial approximations to find the root of a function. Unlike the Newton-Raphson method, it does not require the computation of the derivative. Instead, it approximates the derivative using the two initial points.

The secant method uses the following formula to iteratively refine the root approximation:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{2}$$

where:

- x_n is the current approximation,
- x_{n-1} is the previous approximation,
- $f(x_n)$ is the value of the function at x_n,
- $f(x_{n-1})$ is the value of the function at x_{n-1}.

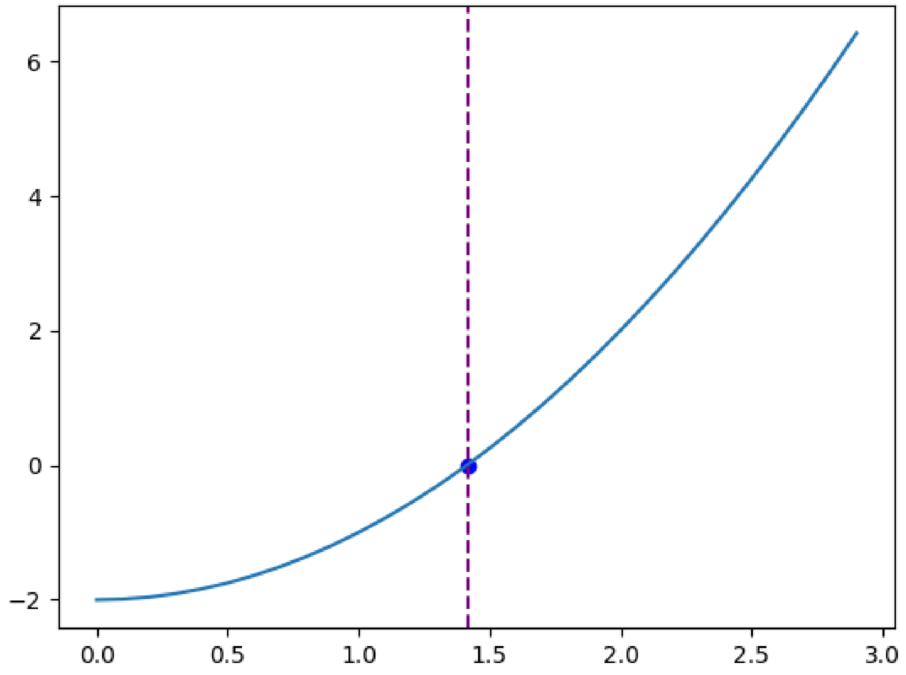
```python
def secant_method(f, x0, x1, tol=1e-7, max_iter=100):
    for i in range(max_iter):
        if abs(f(x1)) < tol:
            return x1
        if f(x1) == f(x0):  # Prevent division by zero
            print("Division by zero encountered in secant method.")
            return None
        # Secant method formula
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        x0, x1 = x1, x2
    print("Maximum iterations reached without convergence.")
    return None

print(secant_method(f, 1, 2))
```

```
1.4142135620573204
```

Let’s plot it…

```
root = secant_method(f, 1, 2)

# Let's plot the result
x = np.arange(0, 3, 0.1)
plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')
```

```
<matplotlib.lines.Line2D at 0x7fef8f89bed0>
```



1.c. Linear Regression

In linear regression, we model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. The goal is to find the best-fitting line (or hyperplane in higher dimensions) that minimizes the difference between the predicted values and the actual values.

Let's import the necessary libraries and create some sample data for linear regression.

```
import numpy as np
from sklearn.linear_model import LinearRegression
from matplotlib import pyplot as plt
```

Let's create some sample data for linear regression. Let's say we want to predict y from x using a linear relationship!

```
np.random.seed(0)
x = 2 * np.random.rand(100, 1)
y = 4 + 3 * x + np.random.randn(100, 1)
```

Create a linear regression model

```
model = LinearRegression()
model.fit(x, y)
```

```
LinearRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org. [x]LinearRegression ?Documentation for LinearRegressioniFitted

Make predictions

```
x_new = np.array([[0], [2]])
y_predict = model.predict(x_new)
```

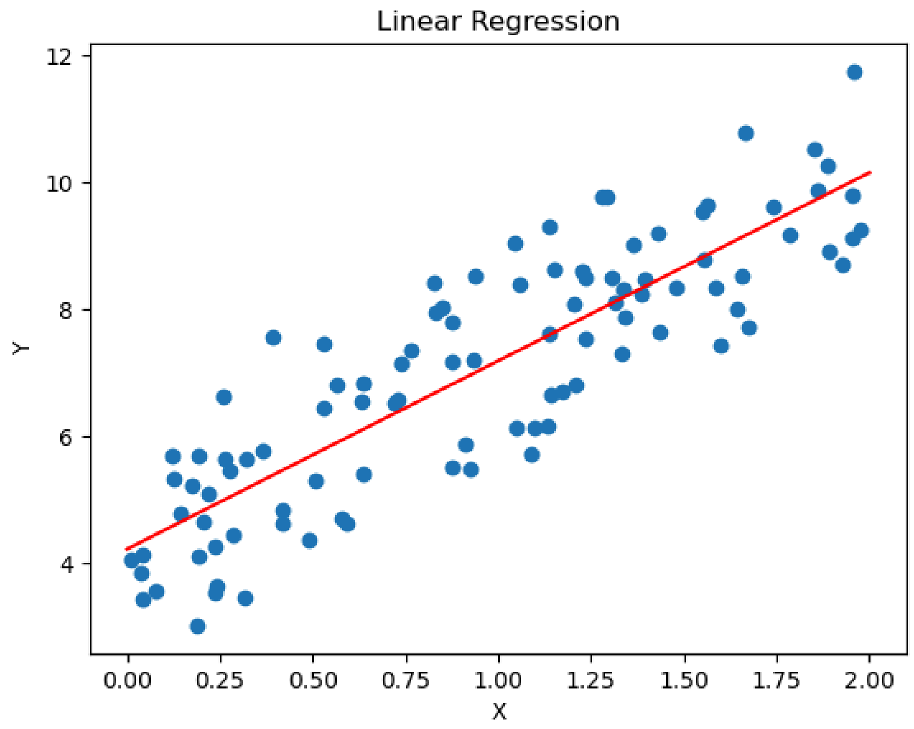
Print the coefficients

```
print("Intercept:", model.intercept_)
print("Coefficient:", model.coef_)
```

```
Intercept: [4.22215108]
Coefficient: [[2.96846751]]
```

Plot the results

```
plt.scatter(x, y)
plt.plot(x_new, y_predict, color='red')

plt.xlabel('X')
plt.ylabel('Y')
plt.title('Linear Regression')
plt.show()
```



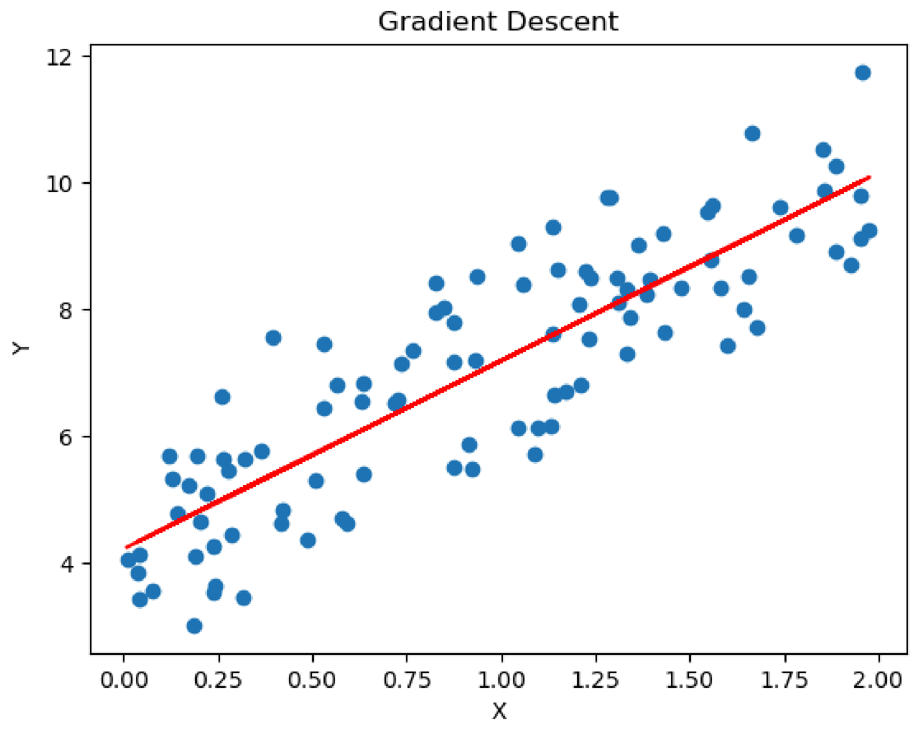
1.c.i. *Gradient Descent:*

Gradient descent is an optimization algorithm used to minimize a function by iteratively moving towards the steepest descent, which is determined by the negative of the gradient. In the context of linear regression, gradient descent is used to find the optimal coefficients that minimize the cost function (usually the mean squared error).

```
def gradient_descent(x, y, m = 0, b = 0, learning_rate = 0.01, epochs =
10000):
    n = len(y)
    for _ in range(epochs):
        y_pred = m * x + b
        dm = (-2/n) * sum(x * (y - y_pred))
        db = (-2/n) * sum(y - y_pred)
        m -= learning_rate * dm
        b -= learning_rate * db
    return m, b
```

Let's plot…

```
plt.scatter(x, y)

m, b = gradient_descent(x, y)
plt.plot(x, m*x + b, color='red')

plt.xlabel('X')
plt.ylabel('Y')

plt.title('Gradient Descent')
plt.show()
```



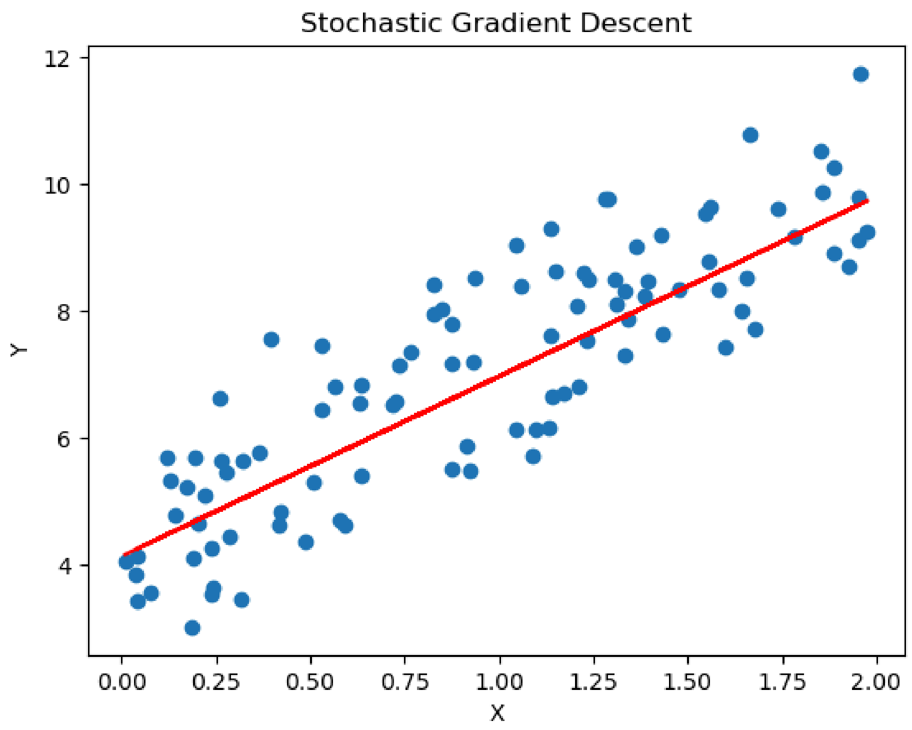
1.c.ii. *Stochastic Gradient Descent:*

Stochastic Gradient Descent (SGD) is a variant of the gradient descent algorithm that updates the model parameters using only a single or a few training examples at each iteration, rather than the entire dataset. This makes SGD more efficient for large datasets and can help the model converge faster.

```
def stochastic_gradient_descent(x, y, m=0, b=0, learning_rate=0.01,
epochs=10000):
    n = len(y)
    for _ in range(epochs):
        for i in range(n):
            xi = x[i:i+1]
            yi = y[i:i+1]
            y_pred = m * xi + b
            dm = -2 * xi * (yi - y_pred)
            db = -2 * (yi - y_pred)
            m -= learning_rate * dm
            b -= learning_rate * db
    return m, b
```

Let’s plot it…

```
plt.scatter(x, y)

m, b = stochastic_gradient_descent(x, y)
plt.plot(x, m*x + b, color='red')

plt.xlabel('X')
plt.ylabel('Y')

plt.title('Stochastic Gradient Descent')
plt.show()
```



1.c.iii. *Logistic Regression (Bonus):*

Logistic regression is a statistical method used for binary classification problems, where the goal is to predict the probability of an instance belonging to one of two classes. Unlike linear regression, which predicts continuous values, logistic regression predicts probabilities that are then mapped to discrete classes (0 or 1).

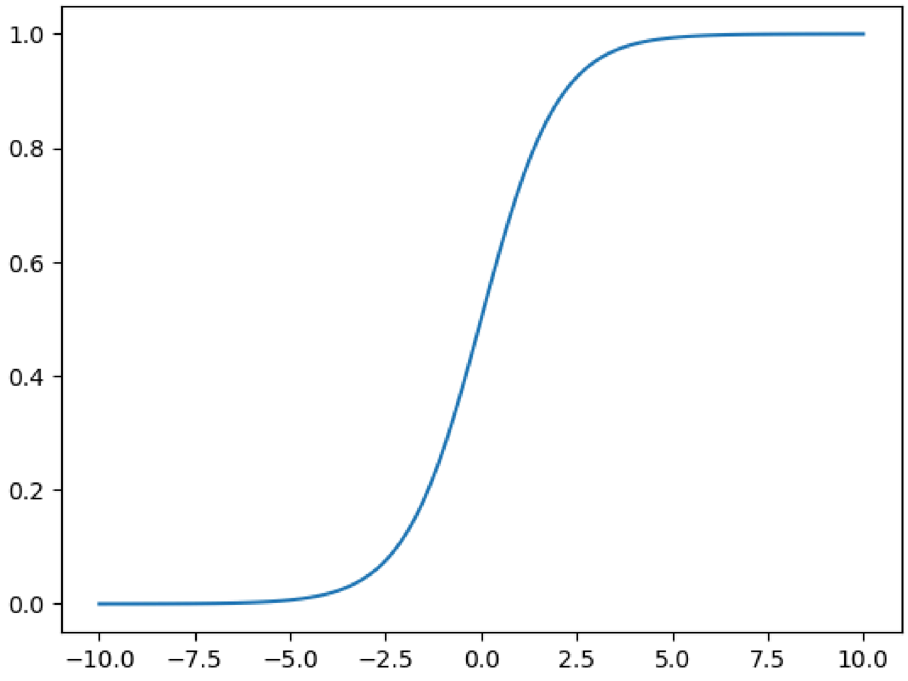
```
def logistic_regression(x):
    return 1 / (1 + np.exp(-x))

def predict_logistic(x, m, b):
    linear_combination = m * x + b
    return logistic_regression(linear_combination)

# An example usage
predict_logistic(0, 1, 0), predict_logistic(1, 1, 0),
predict_logistic(-1, 1, 0)
```

```
(np.float64(0.5),
 np.float64(0.7310585786300049),
 np.float64(0.2689414213699951))
```

Let's plot the curve...

```
x = np.linspace(-10, 10, 100)
y = logistic_regression(x)
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7feae2157ed0>]
```



1.d. *Polynomial Regression*

1.d.i. *Sample funtion:*

First, let's import `matplotlib.pyplot` and `numpy`, one for plotting and the other for numerical operations.

```
import matplotlib.pyplot as plt
import numpy as np
```

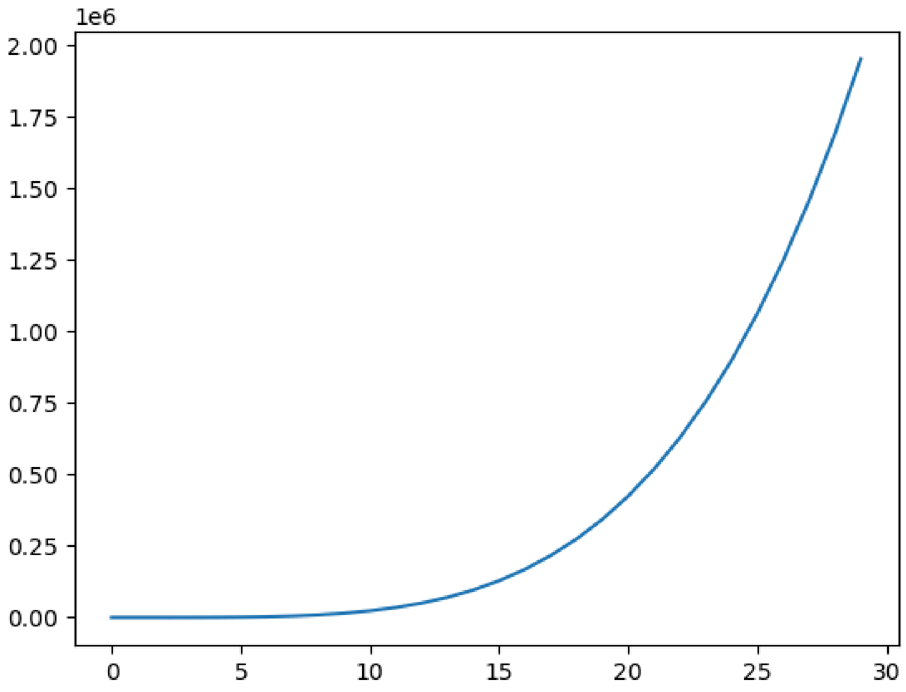
A basic function,

```
def fun(x):
    return 3 * x**4 - 7 * x**3 + 2 * x**2 + 11
```

Let's plot it!

```
x = np.arange(0, 30)
plt.plot(x, fun(x))
```

```
[<matplotlib.lines.Line2D at 0x7f51737a6710>]
```



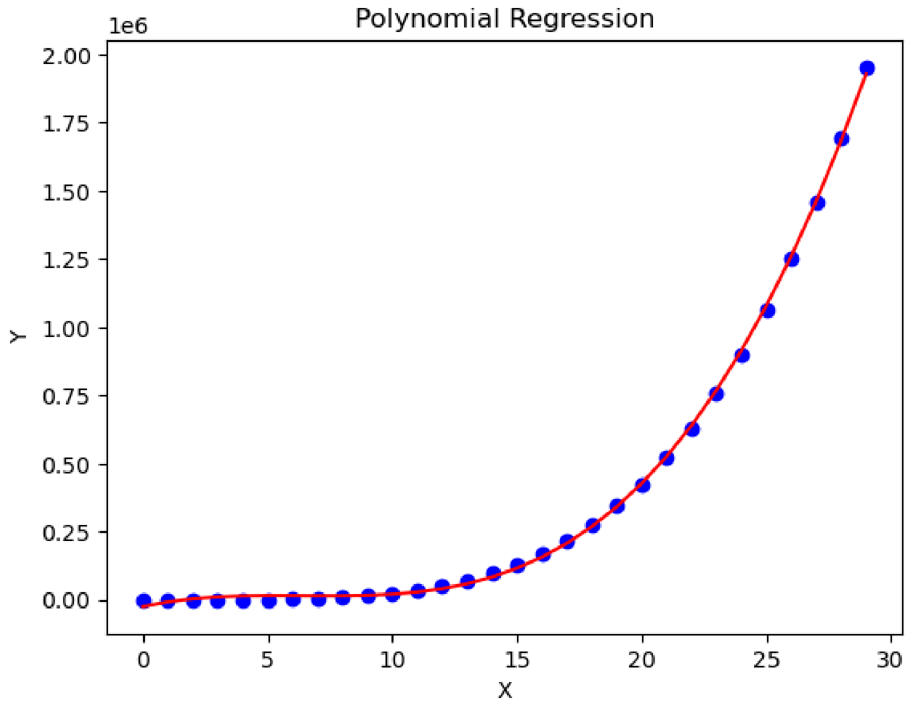
1.d.ii. *Polynomial Regression:*

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

def polynomial_regression(x, y, degree=3):
  poly = PolynomialFeatures(degree)
  x_poly = poly.fit_transform(x.reshape(-1, 1))

  model = LinearRegression()
  model.fit(x_poly, y)
  return model.predict(x_poly)
```

Let’s visualize our result,

```
y_pred = polynomial_regression(x, fun(x))
plt.title('Polynomial Regression')
plt.plot(x, y_pred, color='red')
plt.scatter(x, fun(x), color='blue')
plt.xlabel('X')
plt.ylabel('Y')
```

```
Text(0, 0.5, 'Y')
```

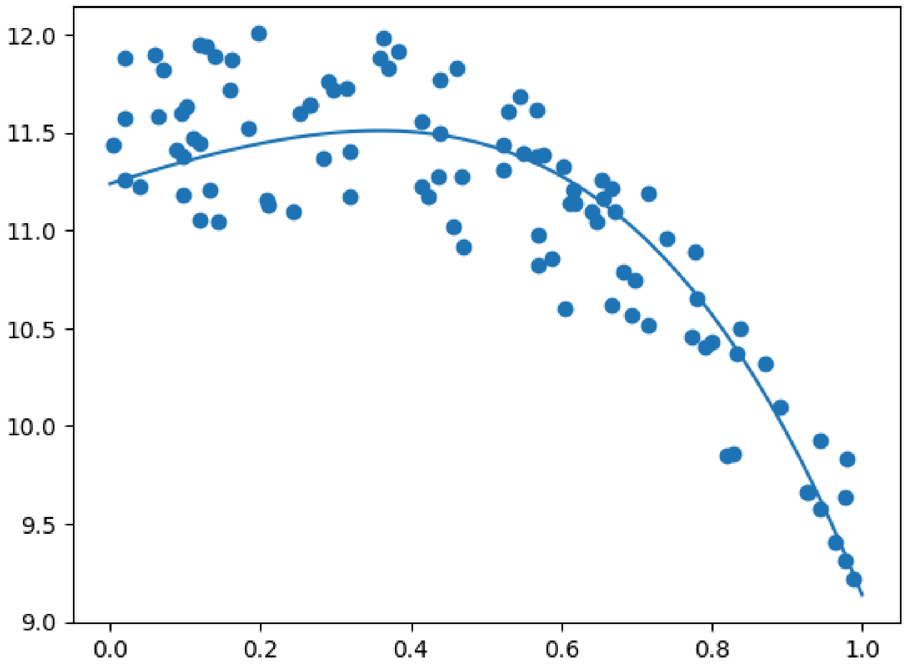


1.d.iii. *Gradient Descent:*

We can implement gradient descent to minimize the cost function for polynomial regression.

```
np.random.seed(0)
x = np.random.rand(100,1)
y = 3 * x**4 - 7 * x**3 + 2 * x**2 + 11 + np.random.rand(100,1)

# m0 x4 + m1 x3 + m2 x2 + m3 x + m4
def gd(x, y, m0=0, m1=0, m2=0, m3=0, m4=0, epoch=10000, learn=0.001):
    n = len(x)
    for i in range(epoch):
        y_n = m0 * x**4 + m1 * x**3 + m2 * x**2 + m3 * x + m4
        m4_l = - 2 / n * np.sum(y - y_n)
        m3_l = - 2 / n * np.sum((y - y_n) * x)
        m2_l = - 2 / n * np.sum((y - y_n) * x**2)
        m1_l = - 2 / n * np.sum((y - y_n) * x**3)
        m0_l = - 2 / n * np.sum((y - y_n) * x**4)

        m4 = m4 - learn * m4_l
        m3 = m3 - learn * m3_l
        m2 = m2 - learn * m2_l
        m1 = m1 - learn * m1_l
        m0 = m0 - learn * m0_l
    return m0, m1, m2, m3, m4


m0, m1, m2, m3, m4 = gd(x, y)
print(gd(x, y))
x_p = np.linspace(0, 1)

plt.plot(x_p, m0 * x_p**4 + m1 * x_p**3 + m2 * x_p**2 + m3 * x_p + m4)
plt.scatter(x,y)
plt.show()
```

```
(np.float64(-1.4008617749474424), np.float64(-1.237676294880945),
np.float64(-0.689765797956417), np.float64(1.228140471950126),
np.float64(11.237160966215388))
```



1.e. *Logistic Regression*

Logistic regression is used for binary classification tasks. It models the probability that a given input belongs to a particular class using the logistic function.

1.e.i. *From Linear:*

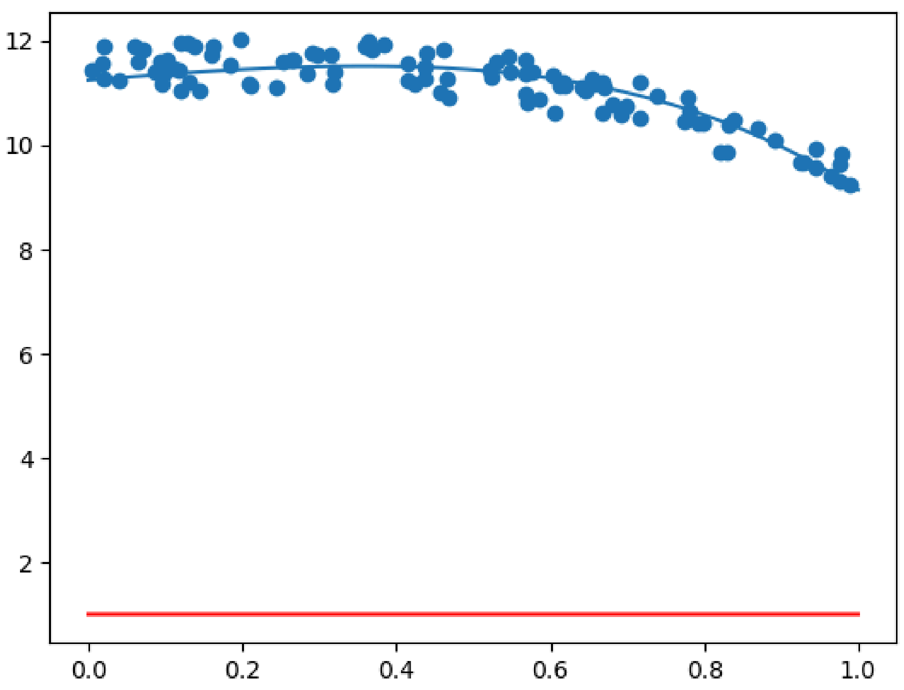
Check the ***Linear Regression*** section's last part.

1.e.ii. *From Polynomial:*

Let's take the function from our previous polynomial regression example and convert it into a binary classification problem. We'll classify points as belonging to class 1 if the output is greater than a certain threshold, and class 0 otherwise.

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
x = np.random.rand(100,1)
y = 3 * x**4 - 7 * x**3 + 2 * x**2 + 11 + np.random.rand(100,1)

# m0 x4 + m1 x3 + m2 x2 + m3 x + m4
def gd(x, y, m0=0, m1=0, m2=0, m3=0, m4=0, epoch=10000, learn=0.001):
    n = len(x)
    for i in range(epoch):
        y_n = m0 * x**4 + m1 * x**3 + m2 * x**2 + m3 * x + m4
        m4_l = - 2 / n * np.sum(y - y_n)
        m3_l = - 2 / n * np.sum((y - y_n) * x)
        m2_l = - 2 / n * np.sum((y - y_n) * x**2)
        m1_l = - 2 / n * np.sum((y - y_n) * x**3)
        m0_l = - 2 / n * np.sum((y - y_n) * x**4)

        m4 = m4 - learn * m4_l
        m3 = m3 - learn * m3_l
        m2 = m2 - learn * m2_l
        m1 = m1 - learn * m1_l
        m0 = m0 - learn * m0_l
    return m0, m1, m2, m3, m4

m0, m1, m2, m3, m4 = gd(x, y)
print(gd(x, y))
x_p = np.linspace(0, 1)

plt.plot(x_p, m0 * x_p**4 + m1 * x_p**3 + m2 * x_p**2 + m3 * x_p + m4)
plt.scatter(x,y)
```

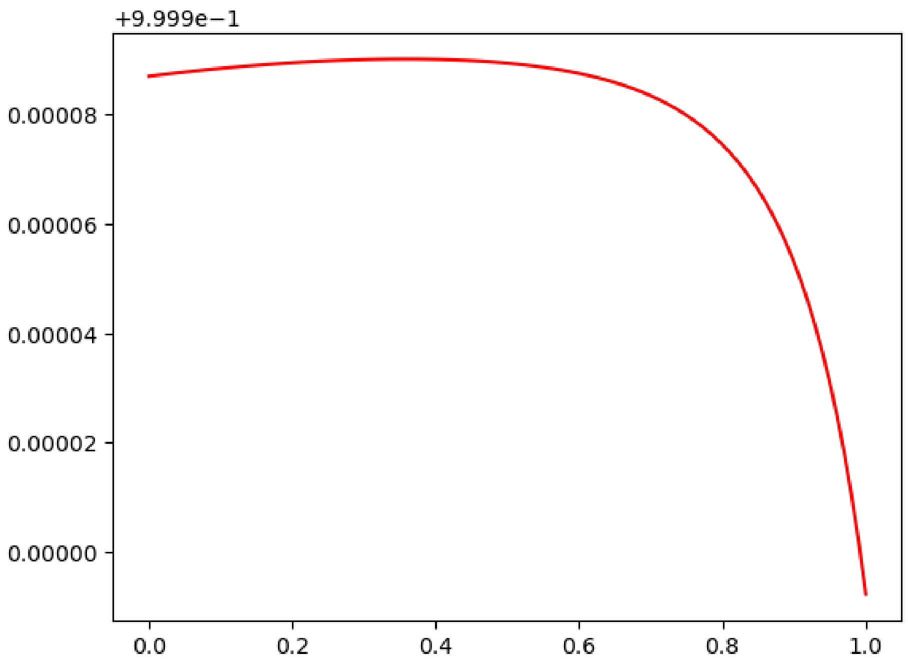
```
(np.float64(-1.4008617749474424), np.float64(-1.237676294880945),
np.float64(-0.689765797956417), np.float64(1.228140471950126),
np.float64(11.237160966215388))
```



Let’s implement logistic based on the above function.

```
def logistic(z):
    return 1 / (1 + np.exp(-z))

def predict(x, m0, m1, m2, m3, m4, threshold=0.5):
    z = m0 * x**4 + m1 * x**3 + m2 * x**2 + m3 * x + m4
    return logistic(z)

x = np.linspace(0, 1, 100).reshape(-1, 1)
plt.plot(x, predict(x, m0, m1, m2, m3, m4), color='red')
```

```
[<matplotlib.lines.Line2D at 0x7fc171cdd6d0>]
```


+9.999e−1
0.00008
0.00006
0.00004
0.00002
0.00000
0.0
0.2
0.4
0.6
0.8
1.0

1.f. *Polynomial Interpolation*

Polynomial interpolation is a method of estimating values between known data points using polynomials. Given a set of data points, the goal is to find a polynomial that passes through all of these points.

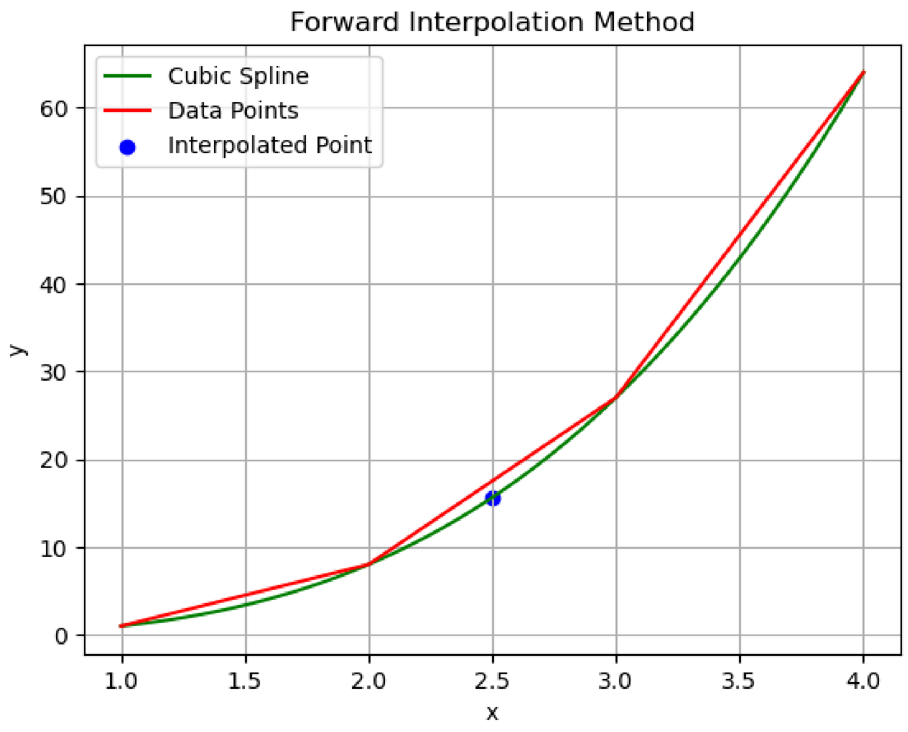
```
# some libs
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
import numpy as np
```

1.f.i. *Newton's Forward Interpolation:*

Let's define for the first 3 differences,

```
def forward(x, y, x_n):
    arr = [y]
    for i in range(3):
        temp = []
        for j in range(0, len(arr[-1])-1):
            temp.append(arr[-1][j+1] - arr[-1][j])
        arr.append(temp)
    print(arr)

    h = (x_n - x[0]) / (x[1] - x[0])
    a = arr[0][0]
    b = h * arr[1][0]
    c = h * (h-1) / 2 * arr[2][0]
    d = h * (h-1) * (h-2) / 6 * arr[3][0]

    return a + b + c + d
x = [1, 2,  3,  4]
y = [1, 8, 27, 64]
x_n = 2.5
y_n = forward(x, y, x_n)

x, y = np.array(x), np.array(y)
fn = interp1d(x, y, kind='cubic')
x_new = np.linspace(x.min(), x.max(), 100)
y_new = fn(x_new)

plt.plot(x_new, y_new, label='Cubic Spline', color='green')
plt.plot(x, y, color='red', label='Data Points')
plt.scatter(x_n, y_n, color='blue', label='Interpolated Point')
plt.title('Forward Interpolation Method')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
plt.show()
```

```
[[1, 8, 27, 64], [7, 19, 37], [12, 18], [6]]
```



1.f.ii. *Newton's Backward Interpolation:*

Same as before, but using backward differences. **Try it yourself!**

1.f.iii. *Lagrange Interpolation:*

Lagrange interpolation is another method for polynomial interpolation that constructs a polynomial that passes through a given set of points.

```python
def lagrange(x, y, x_n):
    n = len(x)
    res = 0
    for i in range(n):
        temp = y[i]
        for j in range(n):
            if i != j:
                temp *= x_n - x[j]
                temp /= x[i] - x[j]
        res += temp
    return res

y_n_lagrange = lagrange(x, y, x_n)
print(f"Lagrange Interpolated value at x = {x_n} is y = {y_n_lagrange}")
```

```
Lagrange Interpolated value at x = 2.5 is y = 15.625
```

1.g. *Differential Equations*

A differential equation is a mathematical equation that relates a function with its derivatives. In simpler terms, it describes how a quantity changes in relation to another quantity. Differential equations are fundamental in various fields such as physics, engineering, and economics, as they model dynamic systems and processes.

1.g.i. *Euler's Method:*

Euler's method is a simple and widely used numerical technique for solving ordinary differential equations (ODEs) with a given initial value. It is particularly useful for approximating solutions to first-order ODEs of the form:

$$\frac{dy}{dx} = f(x, y) \tag{3}$$

with an initial condition $y(x_0) = y_0$.

Let's consider an example,

$$\frac{dy}{dx} = x + y \tag{4}$$

$$y(0) = 1 \tag{5}$$

Let's import libraries first,

```
import numpy as np
import matplotlib.pyplot as plt
```

Our differential equation,

$$\frac{dy}{dx} = x + y \tag{6}$$

```
def f(x, y):
    return x + y
```

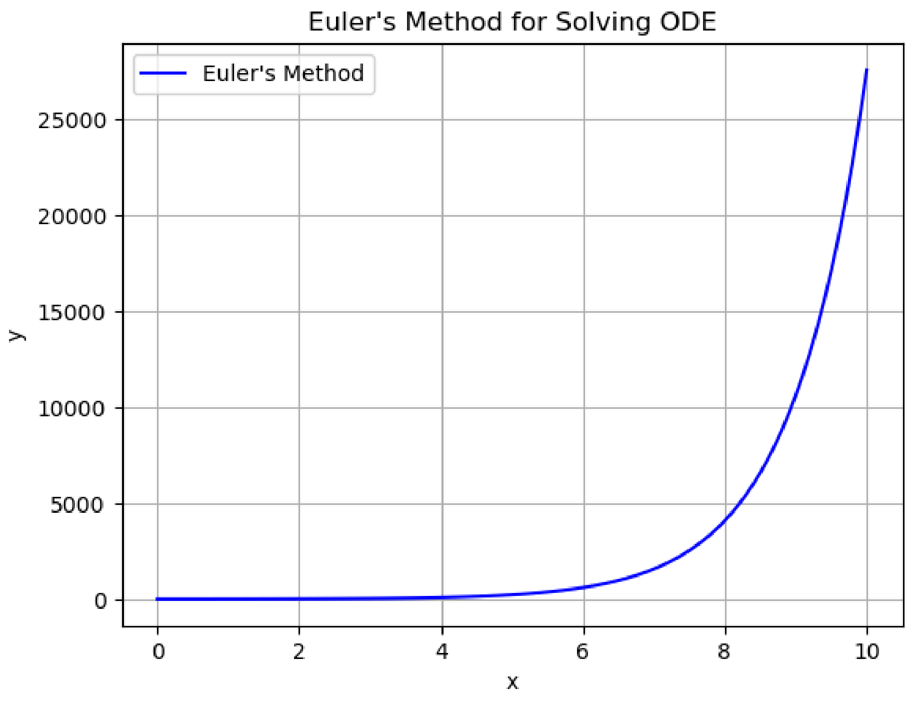
And our euler function,

```
def euler_method(f, x0, y0, h=0.1, n=100):
    x_values = [x0]
    y_values = [y0]

    for i in range(n):
        y0 = y0 + h * f(x0, y0)
        x0 = x0 + h
        x_values.append(x0)
        y_values.append(y0)

    return np.array(x_values), np.array(y_values)
```

Let's plot and visualize the results,

```
# Initial conditions and parameters
x0 = 0
```

```
y0 = 1
h = 0.1  # Step size
n = 100  # Number of steps
x_values, y_values = euler_method(f, x0, y0, h, n)

# Plotting the results
plt.plot(x_values, y_values, label="Euler's Method", color='blue')
plt.title("Euler's Method for Solving ODE")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
```



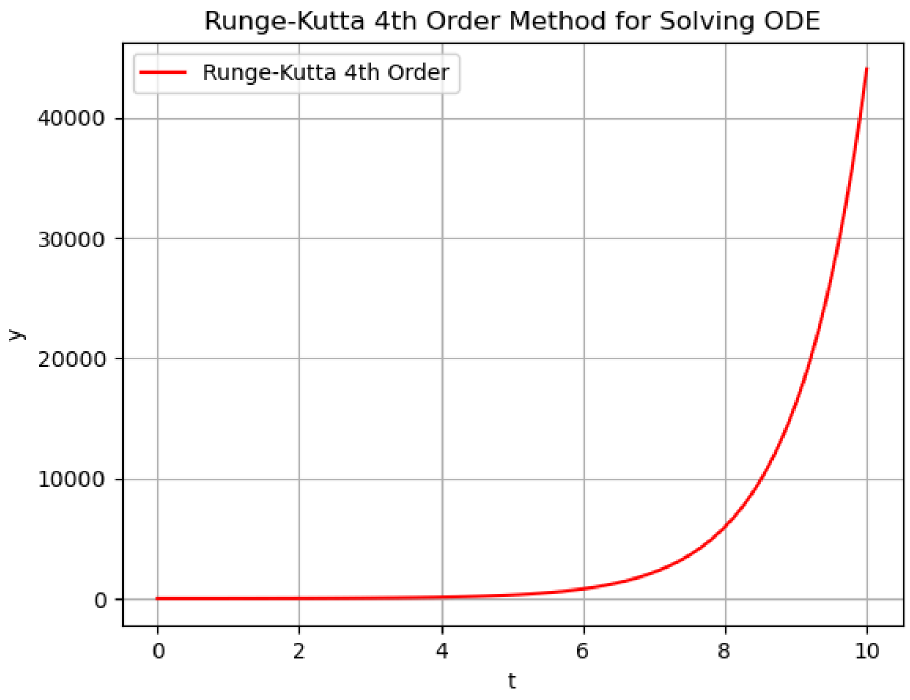
1.g.ii. *Runge-Kutta Method:*

The Runge-Kutta methods are a family of iterative methods used to solve ordinary differential equations (ODEs). The most commonly used version is the fourth-order Runge-Kutta method (RK4), which provides a good balance between accuracy and computational efficiency.

```
def runge_kutta_4th_order(f, y0, x0, x1, h):
    """
    y0: Initial value of y at x0
    x0: Initial value of x
    x1: Final value of x
    h: Step size
    f: Function that returns dy/dt given t and y
    """
    n = int((x1 - x0) / h)
    x_values = np.linspace(x0, x1, n + 1)
    y_values = np.zeros(n + 1)
    y_values[0] = y0

    for i in range(n):
        x = x_values[i]
        y = y_values[i]

        k1 = h * f(x, y)
        k2 = h * f(x + h / 2, y + k1 / 2)
        k3 = h * f(x + h / 2, y + k2 / 2)
        k4 = h * f(x + h, y + k3)

        y_values[i + 1] = y + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    return x_values, y_values
```

Let’s plot it,

```
# Initial conditions and parameters for RK4
y0 = 1
x0 = 0
x1 = 10
h = 0.1
x_values, y_values_rk4 = runge_kutta_4th_order(f, y0, x0, x1, h)

# Plotting the results of RK4
plt.plot(x_values, y_values_rk4, label="Runge-Kutta 4th Order",
color='red')
plt.title("Runge-Kutta 4th Order Method for Solving ODE")
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.grid()
```



1.g.iii. *Picard's Method:*

Picard's method is an iterative technique used to approximate the solutions of ordinary differential equations (ODEs). It is particularly useful for solving initial value problems of the form:

$$\frac{dy}{dx} = f(x, y) \tag{7}$$

with an initial condition $y(x_0) = y_0$.

Mathematically, Picard's method constructs a sequence of functions that converge to the solution of the differential equation. The iterative formula is given by:

$$y_{n+1}(x) = y_0 + \int_{x_0}^{x} f(t, y_n(t))\, dt \tag{8}$$

where $y_n(x)$ is the nth approximation of the solution.

So, first apporximation is,

$$y_1(x) = y_0 + \int_{x_0}^{x} f(t, y_0)\, dt \tag{9}$$

The second approximation is,

$$y_2(x) = y_0 + \int_{x_0}^{x} f(t, y_1(t))\, dt \tag{10}$$

And so on...

For example, consider the initial value problem:

$$\frac{dy}{dx} = 1 + xy \tag{11}$$

$$y(0) = 1 \tag{12}$$

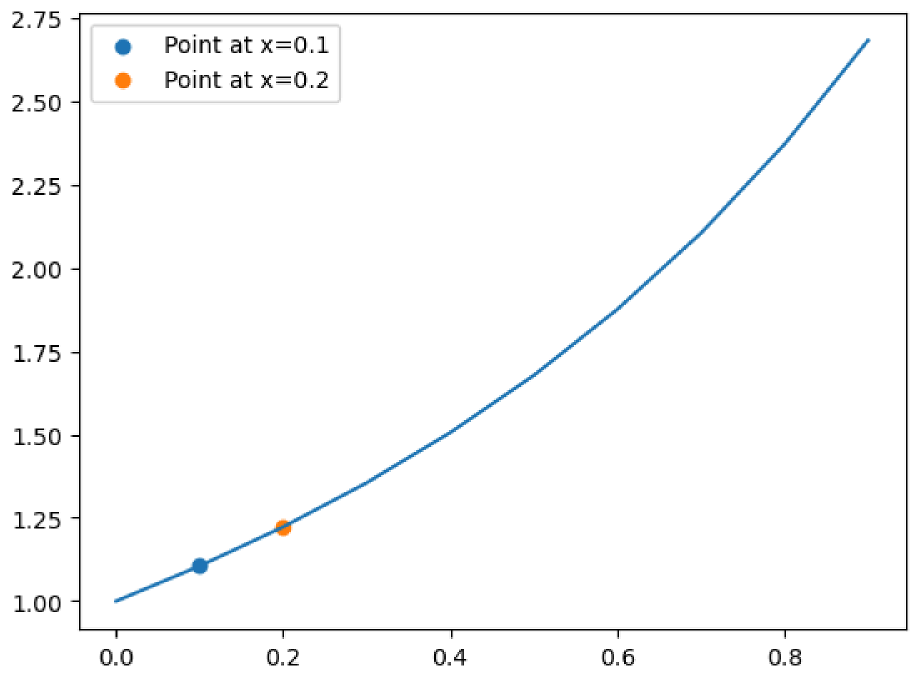
```python
# Apporoximations
def Y1(x):
    return 1 + (x) + pow(x, 2) / 2


def Y2(x):
    return 1 + (x) + pow(x, 2) / 2 + pow(x, 3) / 3 + pow(x, 4) / 8


def Y3(x):
    return (
        1
        + (x)
        + pow(x, 2) / 2
        + pow(x, 3) / 3
        + pow(x, 4) / 8
        + pow(x, 5) / 15
        + pow(x, 6) / 48
    )


def picard_method(f, x0, h=0.1, n=10, iterations=3):
    x_values = np.arange(x0, x0 + n * h, h)
    y_values = np.array([f(i) for i in x_values])

    return x_values, y_values

picard_method(f=Y3, x0=0, h=0.1)
```

```
(array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]),
 array([1.        , 1.10534652, 1.22288933, 1.35518969, 1.50530133,
        1.67688802, 1.874356  , 2.10300152, 2.36917333, 2.68045019]))
```

Let’s plot it…

```python
x_values, y_values = picard_method(f=Y3, x0=0, h=0.1)

plt.plot(x_values, y_values)
plt.scatter(x_values[1], y_values[1], label=f'Point at
x={x_values[1]:.1f}')
plt.scatter(x_values[2], y_values[2], label=f'Point at
x={x_values[2]:.1f}')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f7e20086350>
```



1.g.iv. *Milne's Predictor-Corrector Method:*

Milne's predictor-corrector method is a numerical technique used to solve ordinary differential equations (ODEs). It is a multi-step method that combines both prediction and correction steps to improve the accuracy of the solution.

The formula is,

$$y_{n+1,p} = y_{n-3} + \frac{4h}{3}(2f_n - f_{n-1} + 2f_{n-2}) \tag{13}$$

$$y_{n+1} = y_{n-1} + \frac{h}{3}(f_{n+1,p} + 4f_n + f_{n-1}) \tag{14}$$

Where $y_{n+1,p}$ is the predicted value and y_{n+1} is the corrected value.

If the value of n = 3, then our formula becomes,

$$y_{4,p} = y_0 + \frac{4h}{3}(2f_3 - f_2 + 2f_1) \tag{15}$$

$$y_4 = y_2 + \frac{h}{3}(f_{4,p} + 4f_3 + f_2) \tag{16}$$

```python
# Let's get x0, y0, x1, y1, x2, y2, x3, y3 from the picard method
x0, y0 = x_values[0], y_values[0]
x1, y1 = x_values[1], y_values[1]
x2, y2 = x_values[2], y_values[2]
x3, y3 = x_values[3], y_values[3]

print(f"x0: {x0}, y0: {y0}")
print(f"x1: {x1}, y1: {y1}")
print(f"x2: {x2}, y2: {y2}")
print(f"x3: {x3}, y3: {y3}")

# Let's get y4 from milne's method
def milne_method(x0, y0, x1, y1, x2, y2, x3, y3, x4, h=x1-x0):
    # Predictor
    y4_pred = y0 + (4 * h / 3) * (2 * f(x3, y3) - f(x2, y2) + 2 * f(x1,
y1))
    # Corrector
    y4_corr = y2 + (h / 3) * (f(x2, y2) + 4 * f(x3, y3) + f(x4,
y4_pred))
    return y4_corr

y4_milne = milne_method(x0, y0, x1, y1, x2, y2, x3, y3, x4=x3 + (x1 -
x0))
print(f"y4 from Milne's method: {y4_milne}")
```

```
x0: 0.0, y0: 1.0
x1: 0.1, y1: 1.1053465208333333
x2: 0.2, y2: 1.2228893333333333
x3: 0.30000000000000004, y3: 1.3551896875
y4 from Milne's method: 1.5567806387037035
```

1.h. Integrals

Let's approximate the integral of a function using numerical methods.

1.h.i. *Trapezoidal Rule:*

At first our necessary libs,

```
import numpy as np
import matplotlib.pyplot as plt
```

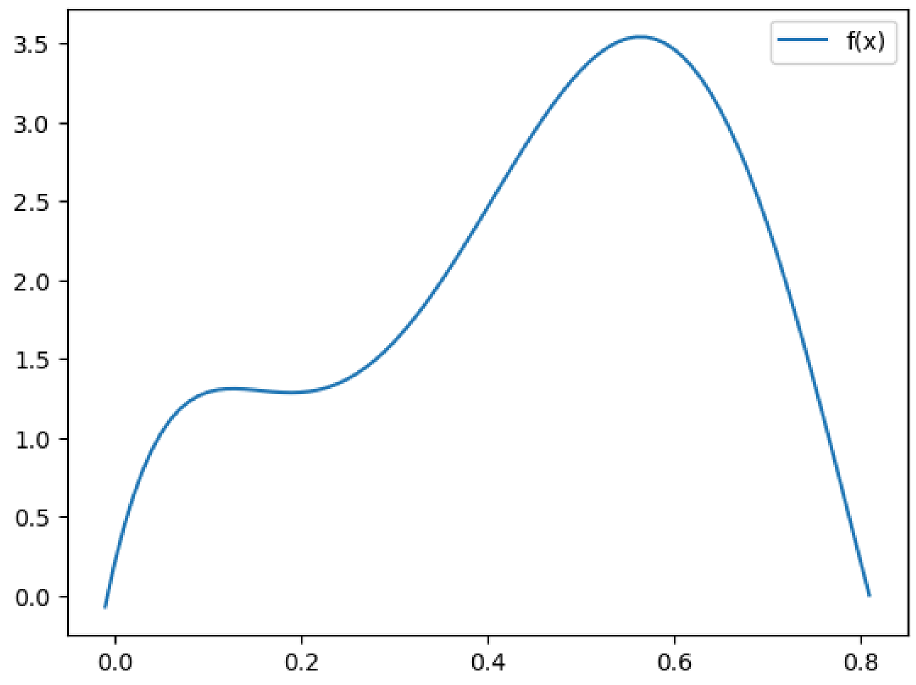
Let's define a sample function to integrate.

```
def fun(x):
  return 0.2 + 25 * x - 200 * x**2 + 675 * x**3 - 900 * x**4 + 400 *
x**5
```

Let's plot it!

```
array = np.arange(-0.01, 0.82, 0.01)
plt.plot(array, fun(array), label='f(x)')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fd35fb6ee90>
```



Let's define our Trapezoidal Rule!

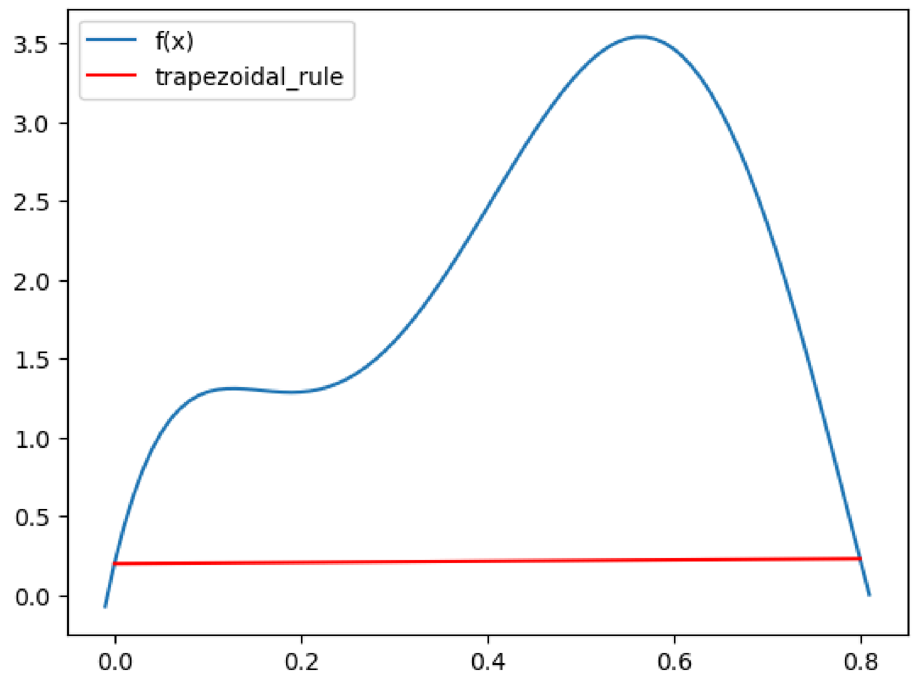
```
def trapezoidal_rule(fun, a, b):
  return (b - a) * (fun(a) + fun(b)) / 2
```

Our trapezoidal rule function is ready to use!

```
print(trapezoidal_rule(fun, 0, 0.8))
plt.plot(array, fun(array), label='f(x)')
plt.plot([0, 0.8], [fun(0), fun(0.8)], color='red',
label='trapezoidal_rule')
plt.legend()
```

```
0.1728000000000225
```

```
<matplotlib.legend.Legend at 0x7fd35fbde5d0>
```



1.h.ii. *Simspon's 1/3 Rule:*

Let's implement it,

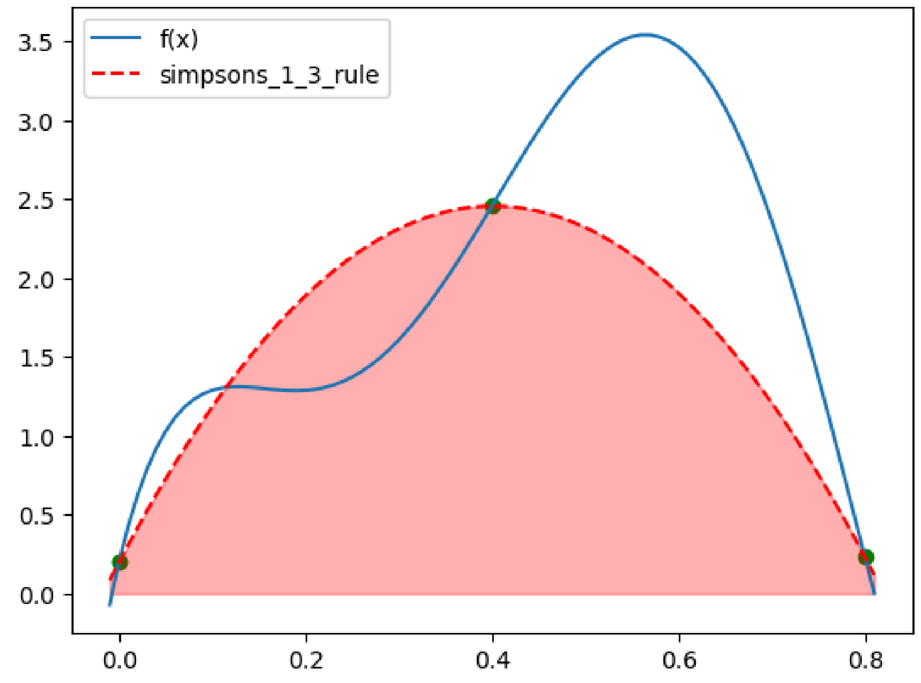
```
def simpsons_1_3_rule(fun, a, b):
    return (b - a) / 6 * (fun(a) + 4 * fun((a + b) / 2) + fun(b))
```

Let's plot it,

```
from scipy.interpolate import CubicSpline

print(simpsons_1_3_rule(fun, 0, 0.8))

plt.plot(array, fun(array), label="f(x)")
plt.scatter([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)], color='green')
plt.plot(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)])
(array), '--', color='red', label='simpsons_1_3_rule')
plt.fill_between(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4),
fun(0.8)])(array), color='red', alpha=0.3)
plt.legend()
```

```
1.3674666666666742
```

```
<matplotlib.legend.Legend at 0x7fd36173c7d0>
```



1.h.iii. *Simpson's 3/8 Rule:*

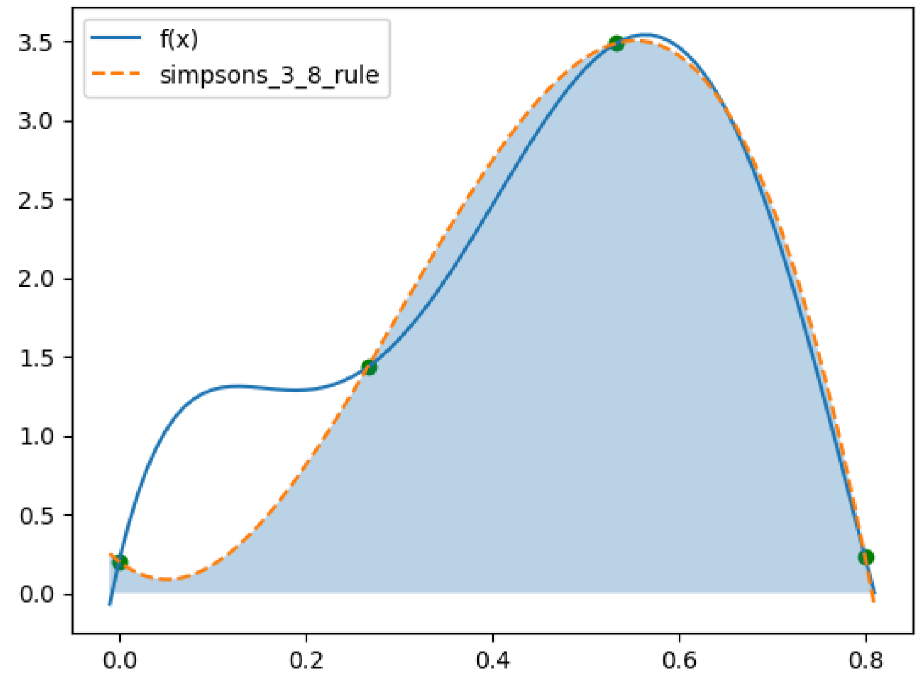
```
def simpsons_3_8_rule(fun, a, b):
    a, b = min(a, b), max(a, b)
    h = (b - a) / 3
    return (3 * h / 8) * (fun(a) + 3 * fun(a + h) + 3 * fun(a + 2 * h) +
fun(b))
```

Let's plot it,

```
from scipy.interpolate import CubicSpline

a, b, c, d = 0, 0.8/3, 0.8/3*2, 0.8
print(simpsons_3_8_rule(fun, a, d))

plt.plot(array, fun(array), label="f(x)")
plt.plot(array, CubicSpline([a, b, c, d], [fun(a), fun(b), fun(c),
fun(d)])(array), '--', label="simpsons_3_8_rule")
plt.scatter([a, b, c, d], [fun(a), fun(b), fun(c), fun(d)],
color='green')
plt.fill_between(array, CubicSpline([a, b, c, d], [fun(a), fun(b),
fun(c), fun(d)])(array), alpha=0.3)

plt.legend()
```

```
1.519170370370378
```

```
<matplotlib.legend.Legend at 0x7fd3615d9450>
```



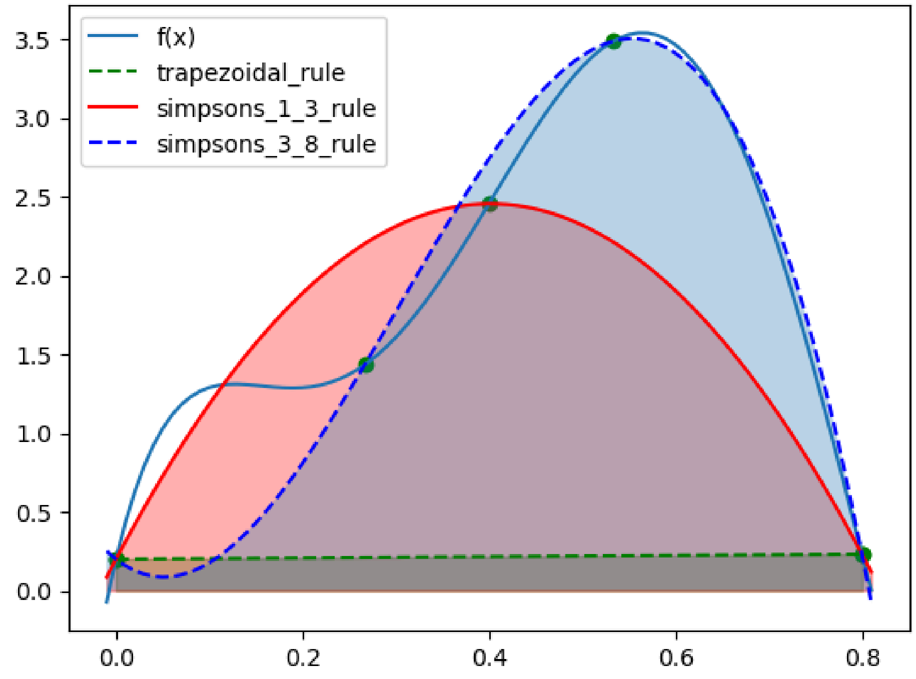
Let’s plot all of them altogether,

```
# Main function
plt.plot(array, fun(array), label="f(x)")

# Trapezoidal rule
print("Trapezoidal rule   -> ", trapezoidal_rule(fun, 0, 0.8))
plt.plot([0, 0.8], [fun(0), fun(0.8)], '--', color='green',
label='trapezoidal_rule')
plt.fill_between([0, 0.8], [fun(0), fun(0.8)], color='green', alpha=0.3)

# Simpson's 1/3 rule
print("Simpson's 1/3 rule -> ", simpsons_1_3_rule(fun, 0, 0.8))

plt.scatter([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)], color='green')
plt.plot(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)])
(array), color='red', label='simpsons_1_3_rule')
plt.fill_between(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4),
fun(0.8)])(array), color='red', alpha=0.3)

a, b, c, d = 0, 0.8/3, 0.8/3*2, 0.8

# Simpson's 3/8 rule
print("Simpson's 3/8 rule -> ", simpsons_3_8_rule(fun, a, d))

plt.scatter([a, b, c, d], [fun(a), fun(b), fun(c), fun(d)],
color='green')

plt.plot(array, CubicSpline([a, b, c, d], [fun(a), fun(b), fun(c),
fun(d)])(array), '--', color='blue', label="simpsons_3_8_rule")
```

```
plt.fill_between(array, CubicSpline([a, b, c, d], [fun(a), fun(b),
fun(c), fun(d)])(array), alpha=0.3)

plt.legend()
```

```
Trapezoidal rule   ->  0.1728000000000225
Simpson's 1/3 rule ->  1.3674666666666742
Simpson's 3/8 rule ->  1.519170370370378
```

```
<matplotlib.legend.Legend at 0x7fd35fc4f9d0>
```


f(x)
trapezoidal_rule
simpsons_1_3_rule
simpsons_3_8_rule