Imports

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

ODE

Taylor Series

Taylor series are expansions of a function $f(x)$ by some finite distance dx to $f(x+dx)$.

- In essence, the Taylor series provides a means to predict a function value at one point in terms of the function value and its derivatives at another point.
- In particular, the theorem states that any smooth function can be approximated as a polynomial:

$$f(x_{i+1}) = f(x_i) + f^{'}(x_i)h + f^{''}(x_i)\frac{h^2}{2!} + \ldots + f^n(x_i)\frac{h^n}{n!}$$

We can use this to get the value of a function at a given given point given a deffrential equation.

Example

Let the ODE be:

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

Given initial conditions $y(0) = 1$ find $y(0.1)$ using the Taylor series expansion.

The general solution of the above DE is

$$y(x) = x - 1 + 2e^{-x}$$

In [20]:
```
def f(x,y):
    return x-y

def sol(x):
    return x-1+2*np.exp(-x)
```

In [21]:
```
sol(0.1)
```

Out[21]: 0.909674836071919

In [26]:
```
def error(x, y_calc):
    y_true = sol(x)
    ae = np.abs(y_true-y_calc)
    re = np.abs(y_true-y_calc)/np.abs(y_true)
    print(f"Absolute error: {ae}")
    print(f"Relative error: {re}")
```

Picard Method

The aim is to solve the ODE using the integration. Given:

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

and boundary conditions $y(x_0) = y_0$ we have to find the value of $y(x)$. In Picard method, this is done by:

$$\begin{aligned} y^{(1)} &= y_0 + \int_{x_0}^{x} f(x, y_0)dx \\ y^{(2)} &= y^{(1)} + \int_{x_0}^{x} f(x, y^{(1)})dx \\ y^{(3)} &= y^{(2)} + \int_{x_0}^{x} f(x, y^{(2)})dx \\ &\vdots \end{aligned}$$

Implementation

In [3]:
```
def integrate(f, a, b, y=0, n=1000):
    h = (b-a)/n
    sum = 0
    sum+=f(a, y)
    sum+=f(b, y)
    for i in range(1, n, 2):
        sum += 4*f(a+i*h, y)
    for i in range(2, n, 2):
        sum += 2*f(a+i*h, y)
    y_true = sum*h/3
    return y_true
```

In [9]:
```
def picard(f, x0, y0, x, n):
    y_prev = y0
    for i in range(n):
        y_prev = y_prev + integrate(f=f, a=x0, b=x, y=y_prev)
    return y_prev
```

In [23]:
```
integrate(f, 0, 1, 0.1)
```

Out[23]: 0.39999999999999986

In [24]:
```
picard(f, 0, 1, 0.1, 10)
```

Out[24]: 0.38124451809499993

Euler Method

Also known as *First Order Runge-Kutta Method*, the Euler method is a numerical method for solving a differential equation using stepwise approximation. The formula is:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

Implementation

In [27]:
```
def euler(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        y_prev = y_prev + f(x0+i*h, y_prev)*h
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [28]:
```
euler(f, 0, 1, 0.1, 10)
```

```
Absolute error: 0.0009106860543101059
Relative error: 0.0010011116260427225
```

Out[28]: 0.9087641500176089

Euler's Modified Method

Here, we use a two steo predictor corrector approximation.

$$y^*_{n+1} = y_n + hf(x_n, y_n)$$
$$y_{n+1} = y_n + \frac{h}{2}[f(x_n, y_n) + f(x_n + h, y^*_{n+1})]$$

In [29]:
```
def euler_modified(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
       y_next = y_prev + h*f(x0+i*h, y_prev)
       y_prev = y_prev + h*(f(x0+i*h, y_prev)+f(x0+(i+1)*h, y_next))/2
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [30]:
```
euler_modified(f, 0, 1, 0.1, 10)
```

```
Absolute error: 3.0388386941249124e-06
Relative error: 3.3405768452900914e-06
```

Out[30]: 0.9096778749106131

So, Euler's modified method is far more accurate than the Euler method.

Runge-Kutta Method

In the Runge-Kutta method, the next value of y is calculated using an increment function ϕ. We get:

$$y_{n+1} = y_n + \phi(x_i, y_i, h) \times h$$
$$\phi = a_1k_1 + a_2k_2 + a_3k_3 + \cdots + a_nk_n$$

k's are called the increment slope factor and are given by:

$$\begin{gathered} k_1 = f(x_i, y_i) \\ k_2 = f(x_i + p_1h, y_i + q_{11}k_1h) \\ k_3 = f(x_i + p_2h, y_i + q_{21}k_1h + q_{22}k_2h) \\ \vdots \\ k_n = f(x_i + p_{n-1}h, y_i + q_{n-1,1}k_1h + q_{n-1,2}k_2h + \cdots + q_{n-1,n-1}k_{n-1}h) \end{gathered}$$

Second Order Runge-Kutta Method

The second order R-K method is:

$$\begin{gathered} y_{i+1} = y_i + (a_1k_1 + a_2k_2)h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + p_1h, y_i + q_{11}k_1h) \end{gathered}$$

Where the constants are related by:

$$\begin{gathered} a_1 + a_2 = 1 \\ a_2p_1 = \frac{1}{2} \\ a_2q_{11} = \frac{1}{2} \end{gathered}$$

Heun Method

Using $a_2 = 1/2$ gives the Heun method. The constants becomes:

$$\begin{gathered} a_1 = a_2 = 1/2 \\ p_1 = q_{11} = 1 \end{gathered}$$

And hence, we get:

$$\begin{gathered} y_{i+1} = y_i + (\frac{1}{2}k_1 + \frac{1}{2}a_2k_2)h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + h, y_i + k_1h) \end{gathered}$$

Midpoint Method

Here, we use $a_2 = 1$ and hence the R-K equation becomes:

$$\begin{gathered} y_{i+1} = y_i + k_2h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + h/2, y_i + k_1h/2) \end{gathered}$$

Ralston's Method

Here, we use $a_2 = 2/3$ and hence the R-K equation becomes:

$$\begin{gathered} y_{i+1} = y_i + (\frac{1}{3}k_1 + \frac{2}{3}a_2k_2)h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1h) \end{gathered}$$

Implementations

In [31]:
```
#Heun method
def heun(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        k1 = f(x0+i*h, y_prev)
        k2 = f(x0+(i+1)*h, y_prev+h*k1)
        y_prev = y_prev + (k1+k2)*h/2
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [32]:
```
heun(f, 0, 1, 0.1, 10)
```

```
Absolute error: 3.0388386941249124e-06
Relative error: 3.3405768452900914e-06
```

Out[32]: 0.9096778749106131

In [33]:
```
# Midpoint method
def midpoint(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        k1 = f(x0+i*h, y_prev)
        k2 = f(x0+(i+0.5)*h, y_prev+h*k1*0.5)
        y_prev = y_prev + k2*h
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [34]:
```
midpoint(f, 0, 1, 0.1, 10)
```

```
Absolute error: 3.0388386941249124e-06
Relative error: 3.3405768452900914e-06
```

Out[34]: 0.9096778749106131

In [35]:
```
# Ralston method
def ralston(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        k1 = f(x0+i*h, y_prev)
        k2 = f(x0+(i+0.75)*h, y_prev+h*k1*0.75)
        y_prev = y_prev + (k1+2*k2)*h/3
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [36]:
```
ralston(f, 0, 1, 0.1, 10)
```

```
Absolute error: 3.0388386941249124e-06
Relative error: 3.3405768452900914e-06
```

Out[36]: 0.9096778749106131

Third Order Runge-Kutta Method

Here, we use:

$$\begin{gathered} y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h) \\ k_3 = f(x_i + h, y_i - k_1h + 2k_2h) \end{gathered}$$

Implementations

In [39]:
```
def rk3(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        k1 = f(x0+i*h, y_prev)
        k2 = f(x0+(i+0.5)*h, y_prev+h*k1*0.5)
        k3 = f(x0+(i+1)*h, y_prev-h*k1+2*k2*h)
        y_prev = y_prev + (k1+4*k2+k3)*h/6
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [40]:
```
rk3(f, 0, 1, 0.1, 10)
```

```
Absolute error: 7.600886253733563e-09
Relative error: 8.355607907716859e-09
```

Out[40]: 0.9096748284710328

Fourth Order Runge-Kutta Method

Here, we use:

$$\begin{gathered} y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h) \\ k_3 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h) \\ k_4 = f(x_i + h, y_i + k_3h) \end{gathered}$$

Implementation

In [41]:
```
def rk4(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        k1 = f(x0+i*h, y_prev)
        k2 = f(x0+(i+0.5)*h, y_prev+h*k1*0.5)
        k3 = f(x0+(i+0.5)*h, y_prev+h*k2*0.5)
        k4 = f(x0+(i+1)*h, y_prev+h*k3)
        y_prev = y_prev + (k1+2*k2+2*k3+k4)*h/6
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [42]:
```
rk4(f, 0, 1, 0.1, 10)
```

```
Absolute error: 1.5207057835198157e-11
Relative error: 1.6717025943977946e-11
```

Out[42]: 0.9096748360871261

Fifth Order Runge-Kutta Method (Buther's Method)

Here, we use:

$$\begin{gathered} y_{i+1} = y_i + \frac{1}{90}(7k_1 + 23k_3 + 12k_4 + 32k_5 + 7k_6)h \\ k_1 = f(x_i, y_i) \\ k_2 = f(x_i + \frac{1}{4}h, y_i + \frac{1}{4}k_1h) \\ k_3 = f(x_i + \frac{1}{4}h, y_i + \frac{1}{8}(k_1 + k_2)h) \\ k_4 = f(x_i + \frac{1}{2}h, y_i + (\frac{-k_2}{2} + k_3)h) \\ k_5 = f(x_i + \frac{3}{4}h, y_i + \frac{3}{16}(k_1 + 3k_4)h) \\ k_6 = f(x_i + h, y_i + \frac{1}{7}(-3k_1 + 2k_2 + 12k_3 - 12k_4 + 8k_5)h) \end{gathered}$$

Implementation

In [48]:
```
def rk5(f, x0, y0, x, n, report_error=True):
    h = (x-x0)/n
    y_prev = y0
    for i in range(n):
        k1 = f(x0+i*h, y_prev)
        k2 = f(x0+(i+0.25)*h, y_prev+h*k1*0.25)
        k3 = f(x0+(i+0.25)*h, y_prev+(k1+k2)*h/8)
        k4 = f(x0+(i+0.5)*h, y_prev+(-0.5*k2+k3)*h)
        k5 = f(x0+(i+0.75)*h, y_prev+3*(k1+3*k4)*h/16)
        k6 = f(x0+(i+1)*h, y_prev+(-3*k1+2*k2+12*k3-12*k4+8*k5)*h/7)
        y_prev = y_prev + (7*k1+32*k3+12*k4+32*k5+7*k6)*h/90
    if report_error:
        error(x, y_prev)
    return y_prev
```

In [49]:
```
rk5(f, 0, 1, 0.1, 10)
```

```
Absolute error: 3.3306690738754696e-15
Relative error: 3.661384202137198e-15
```

Out[49]: 0.9096748360719223