Import packages

Note that for this script to work, you need to have the following packages installed:

- sympy with its dependencies - for symbolic computation (specifying functions and computing derivatives)
- matplotlib - graphical visualization of convergence of methods

In [ ]:
```
import sympy as sp # for symbolic math, to calculate derivatives and define functions
import warnings
import matplotlib.pyplot as plt
import math
```

Specify precission for numerical operations

In [ ]:
```
EPS = 1e-8 # eplison value, precision of numerical convergence
```

Define functions: Newton's, secant and bisection methods

In [ ]:
```
def newton(function, x0, keep_data = False):

    xn = x0 # init start value
    i = 0 # init counter

    if keep_data:
        data = []

    while True:

        if keep_data:
            data.append((i, xn))

        i = i + 1 # increment counter
        x_prev = xn # store previous value
        xn = xn - function.subs(x, xn) / sp.diff(function, x).subs(x, xn) # new estimate

        if abs(xn - x_prev) < EPS: # if close enough, stop
            if keep_data:
```

```
                return xn.evalf(), i, data
            return xn.evalf(), i # return value as float and number of iterations

        if i > 1000: # prevent infinite loop
            warnings.warn("\nNewton's method did not converge! Returning last estimate.\n")
            if keep_data:
                return xn.evalf(), i, data
            return xn.evalf(), i # return value as float and number of iterations



def secant(function, x0, x1, keep_data = False):


    xn = x1 # init start value x n
    xn_1 = x0 # init start value x n-1
    i = 0 # init counter

    if keep_data:
        data = []

    while True:

        if keep_data:
            data.append((i, xn))

        i = i + 1 # increment counter
        x_prev = xn # store previous value
        xn = xn - function.subs(x, xn) / (function.subs(x, xn) - function.subs(x, xn_1)) * (xn - xn_1) # new estimate
        xn_1 = x_prev # store previous value

        if abs(xn - x_prev) < EPS: # if close enough, stop
            if keep_data:
                return xn.evalf(), i, data
            return xn.evalf(), i # return value as float and number of iterations
        if i > 1000: # prevent infinite loop
            warnings.warn("\nSecant method did not converge! Returning last estimate.\n")
            if keep_data:
                return xn.evalf(), i, data
            return xn.evalf(), i # return value as float and number of iterations



def bisection(function, a, b, keep_data = False):
```

```
    i = 0 # init counter

    if keep_data:
        data = []

    while True:

        c = (a + b) / 2 # midpoint

        if keep_data:
            data.append((i, c))

        i = i + 1 # increment counter
        if abs(function.subs(x, c)) < EPS: # if root found, stop
            if keep_data:
                return c, i, data
            return c, i # return value as float and number of iterations

        elif function.subs(x, a) * function.subs(x, b) > 0:
            return None, i # no root or multiple roots in interval

        else:
            if function.subs(x, a) * function.subs(x, c) < 0:
                b = c
            elif function.subs(x, b) * function.subs(x, c) < 0:
                a = c

        if i > 1000: # prevent infinite loop
            warnings.warn("\nBisection method did not converge! Returning last estimate.\n")
            if keep_data:
                return c, i, data
            return c, i # return value as float and number of iterations
```

Testing script

Specify function to be solved:

In [ ]:

```
x = sp.symbols('x') # define x as a symbol

function = sp.exp(x)-2 # define function, use sympy packages for symbolic math
```

Specify initial guesses for different methods:

In [ ]:
```
newton_x0 = 1.8
secant_x0, secant_x1 = 0.1, 1.8
bisection_a, bisection_b = 0.1, 1.8
```

Solve:

In [ ]:
```
if __name__ == "__main__": # run script

    try:
        newton_ans, newton_iter  = newton(function, newton_x0)
        secant_ans, secant_iter = secant(function, secant_x0, secant_x1)
        bisection_ans, bisection_iter = bisection(function, bisection_a, bisection_b)

        # format output information
        result_string = f"""
            Function: \tf(x) = {function}
            Precision: \t{EPS}

            Method: \tInit values: \t\tResult: \tIterations:
            Newton's: \tx0={newton_x0:.1f} \t\t\t{newton_ans:.9f}\t{newton_iter}
            Secant: \tx0={secant_x0:.1f}, x1={secant_x1:.1f} \t\t{secant_ans:.9f} \t{secant_iter}
            Bisection:\ta={bisection_a:.1f}, b={bisection_b:.1f} \t\t{bisection_ans:.9f} \t{bisection_iter}
            """

        print(result_string)


    except:
        print("Error occured while solving for given function and init values. Please check your input.")
```

```
            Function:   f(x) = exp(x) - 2
            Precision:  1e-08

            Method:     Init values:            Result:         Iterations:
```

```
Newton's:   x0=1.8                  0.693147181     6
Secant:     x0=0.1, x1=1.8          0.693147181     7
Bisection:  a=0.1, b=1.8            0.693147184     26
```

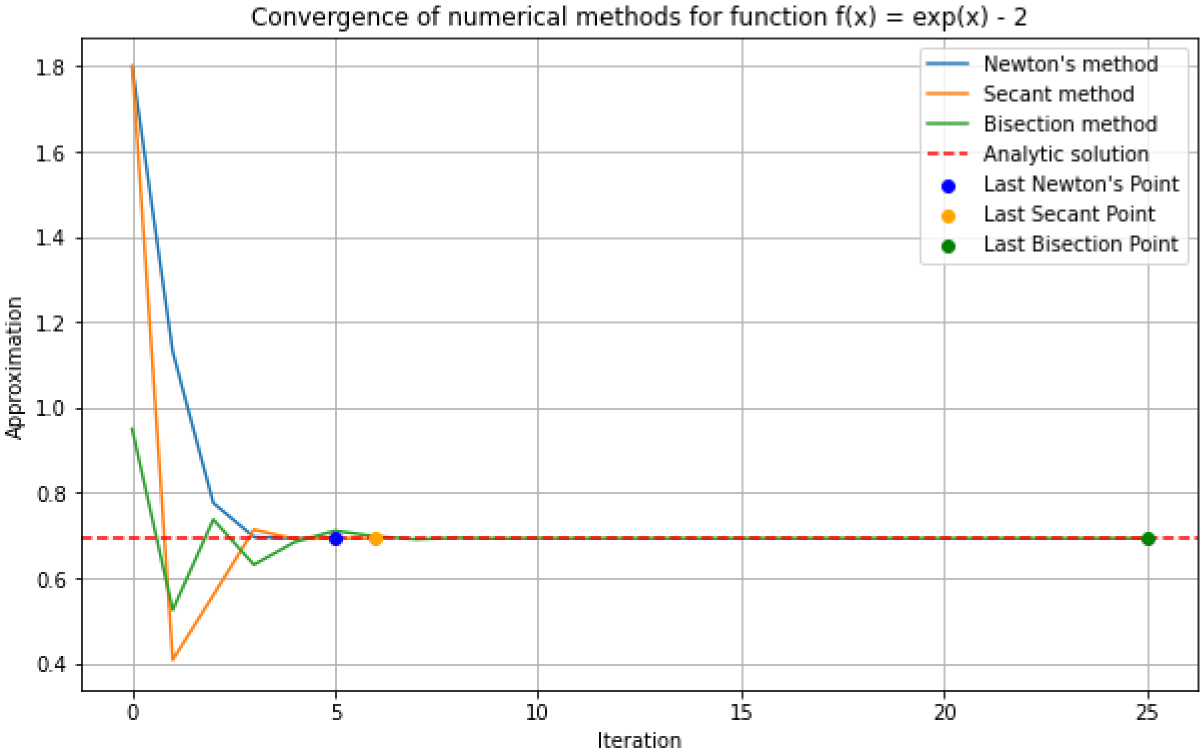
Plotting rate of convergence for different methods:

In [ ]:

```
_, __, nd = newton(function, newton_x0, keep_data=True) # only data is needed in this case
_, __, sd = secant(function, secant_x0, secant_x1, keep_data=True) # only data is needed in this case
_, __, bd = bisection(function, bisection_a, bisection_b, keep_data=True) # only data is needed in this case
```

In [ ]:

```
sol = sp.solve(function, x)[0] # analytic solution for given function, usefull to visualize convergence

plt.figure(figsize=(10, 6))

# data
plt.plot([i for i, _ in nd], [x for _, x in nd], label="Newton's method")
plt.plot([i for i, _ in sd], [x for _, x in sd], label="Secant method")
plt.plot([i for i, _ in bd], [x for _, x in bd], label="Bisection method")

# markers for last point
plt.scatter([nd[-1][0]], [nd[-1][1]], color='blue', marker='o', label="Last Newton's Point", zorder=5)
plt.scatter([sd[-1][0]], [sd[-1][1]], color='orange', marker='o', label='Last Secant Point', zorder=5)
plt.scatter([bd[-1][0]], [bd[-1][1]], color='green', marker='o', label='Last Bisection Point', zorder=5)



# adjust plot
plt.grid()
plt.xlabel("Iteration")
plt.ylabel("Approximation")
plt.title(f"Convergence of numerical methods for function f(x) = {function}")
plt.axhline(sol, color='r', linestyle='--', label="Analytic solution")
plt.legend()
plt.show()
```


Convergence of numerical methods for function f(x) = exp(x) - 2
Newton's method
Secant method
Bisection method
Analytic solution
Last Newton's Point
Last Secant Point
Last Bisection Point
Approximation
Iteration
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
5
10
15
20
25

```
In [ ]:
```