retselis_set1

December 30, 2020

1 Problem Set #1

Computational Mathematics (ΥΦΥ101)

Implemented by: **Anastasios-Faidon Retselis (AEM: 4394)**

2 Exercise 1

2.1 Problem Statement:

Find the root to the following equations:

$$(a)\ \ e^x - 2x\cos(x) - 3 = 0,\ x \in (0, 2)$$
$$(b)\ \ x^2 + \sin(x) + e^x - 2 = 0,\ x \in (0, 1)$$

using the methods of bisection and Newton-Raphson. Compare the methods by computing the number of iterations needed to achieve accuracy of 5 significant digits.

Let us first import the necessary packages to solve this exercise using Python:

2.2 Solution for (a)

$$(a)\ \ e^x - 2x\cos(x) - 3 = 0,\ x \in (0, 2)$$

Let's first examine the bi-section method. We are interested in also documenting the number of iterations required to achieve an accuracy of 5 significant digits. We will therefore use the **Scarborough criterion**, computing it in the beginning of our code and then calculating the approximate error of each iteration. We will meet the criterion once the approximater error is below the error given by the Scarborough formula.

[1]:
```
import math
import matplotlib.pyplot as plt

# Bi-section method

x_lower = 0
x_upper = 2
n = 5 # Number of significant digits to be computed
```

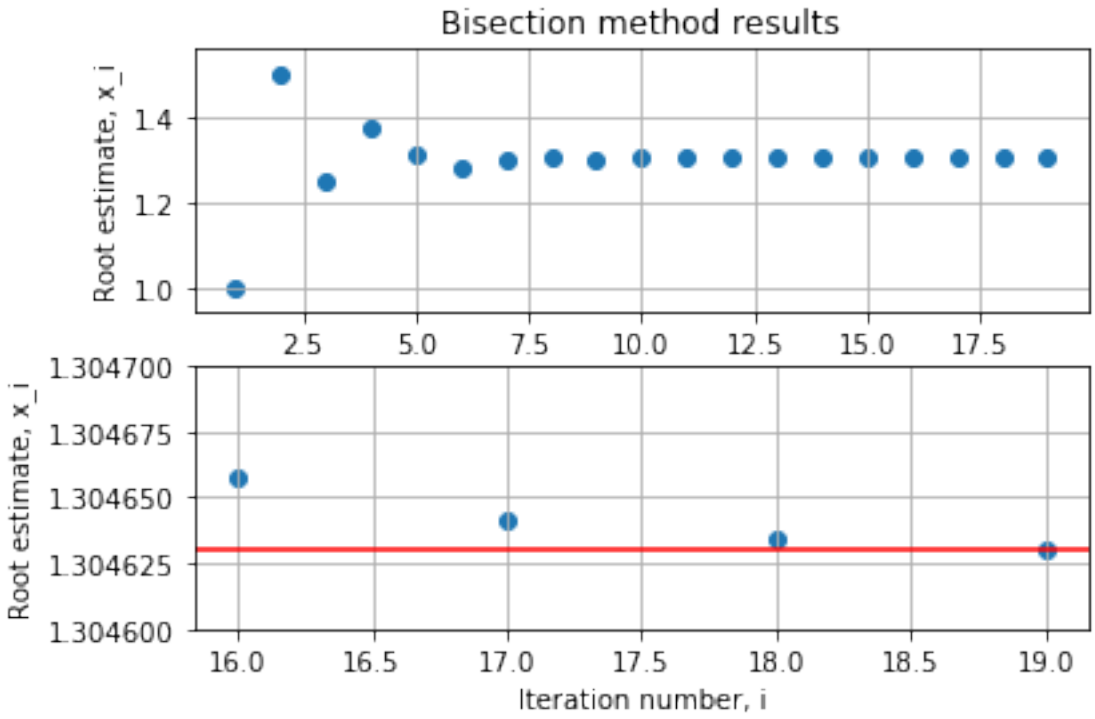
```
es = 0.5*pow(10, (2-n))
ea = 100
repetitions = 0
value_prev = 100

bisec_iter_list = []
bisec_value_list = []

# Define function and begin computation

f = lambda x : math.exp(x)-(2*x*math.cos(x))-3

while ea>es:
    xr = (x_lower+x_upper)/2
    repetitions = repetitions + 1
    if f(x_lower)*f(xr)<0:
        x_upper = xr
        value = xr
    elif f(x_upper)*f(xr)<0:
        x_lower = xr
        value = xr
    ea = math.fabs((value-value_prev)*100/value)
    value_prev = xr
    bisec_iter_list.append(repetitions)
    bisec_value_list.append(xr)

print('Root found (after %d iterations)! x_root = %.5f' % (repetitions, xr))
```

```
Root found (after 19 iterations)! x_root = 1.30463
```

We can also plot the results to see how the method approaches the desired accuracy:

[2]:
```
# Plot bisection method results

plt.subplot(2,1,1)
plt.title('Bisection method results')
plt.scatter(bisec_iter_list,bisec_value_list)
axes = plt.gca()
axes.set_ylabel('Root estimate, x_i')
plt.grid()
plt.subplot(2,1,2)
plt.scatter(bisec_iter_list[-4:],bisec_value_list[-4:])
plt.axhline(y=1.30463,color='r')
axes = plt.gca()
axes.set_ylim([1.3046,1.3047])
axes.set_ylabel('Root estimate, x_i')
axes.set_xlabel('Iteration number, i')
axes.ticklabel_format(useOffset=False)
```

```
plt.grid()
```



Let's now examine the Newton-Raphson Method. We will again use the **Scarborough criterion** as the stopping condition, computing it in the beginning of our code and then calculating the approximate error of each iteration. We will stop the calculation once the approximate error is below the error given by the Scarborough formula. For the Newton-Raphson Method, we also have to calculate the derivative of the function. Let's assume:

$$f(x) = e^x - 2x\cos(x) - 3$$

the derivative of $f(x)$ is:

$$f'(x) = e^x - 2x\cos(x) + 2x\sin(x)$$

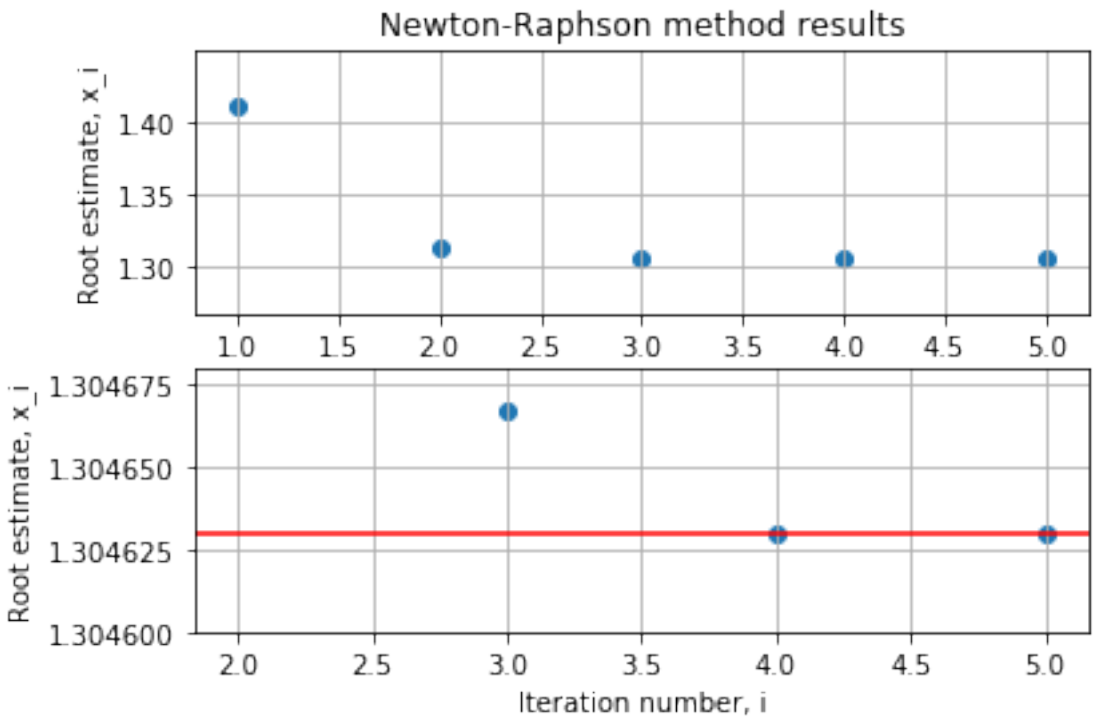
[3]:
```
import math
import matplotlib.pyplot as plt

# Newton-Raphson method

x0 = 1
```

```
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))
ea = 100
repetitions = 0
x_prev = x0

newton_iter_list = []
newton_value_list = []

# Define function, derivative and begin computation

f = lambda x : math.exp(x)-(2*x*math.cos(x))-3
fdot = lambda x : math.exp(x)+2*(x*math.sin(x)-math.cos(x))

while ea>es:
    x_next = x_prev - (f(x_prev)/fdot(x_prev))
    ea = math.fabs((x_next-x_prev)*100/x_next)
    repetitions = repetitions + 1
    x_prev = x_next
    newton_iter_list.append(repetitions)
    newton_value_list.append(x_next)

print('Root found (after %d iterations)! x_root = %.5f' % (repetitions, x_next))
```

```
Root found (after 5 iterations)! x_root = 1.30463
```

Let's again plot the results to see how the method approaches the desired accuracy:

[4]:
```
# Plot Newton-Raphson method results

plt.subplot(2,1,1)
plt.title('Newton-Raphson method results')
plt.scatter(newton_iter_list,newton_value_list)
axes = plt.gca()
axes.set_ylabel('Root estimate, x_i')
plt.grid()
plt.subplot(2,1,2)
plt.scatter(newton_iter_list[-4:],newton_value_list[-4:])
plt.axhline(y=1.30463,color='r')
axes = plt.gca()
axes.set_ylim([1.3046,1.30468])
axes.set_ylabel('Root estimate, x_i')
axes.set_xlabel('Iteration number, i')
axes.ticklabel_format(useOffset=False)
plt.grid()
```



Looking at the results we can assume that the Newton-Raphson Method performs better the bisection method used above. However, let's not forget that we used assumed a value of $x_0 = 1$ for the Newton-Raphson method, while we used the entire interval values $x_{lower} = 0$ and $x_{upper} = 2$ for the bisection method. It would be interesting to see if there are worse case scenarios for the Newton-Raphson method, for which we wouuld have to perform more iterations.

We can re-run the Newton-Raphson method starting from $x_0 = 1$ moving up towards to $x_0 = 2$ with a step of $step = 0.1$. For each step we will ask for the number of iterations to achieve the requested accuracy, in order to ultimately obtain the maximum possible number of iterations for this scenario. We will also provide the minimum amount of iterations needed (although this is not a "fair" comparison).

[5]:
```
import math
import numpy as np

# Newton-Raphson method (step increasing - max/min_iterations finder)

x_lower = 0
x_upper = 2
step_size = 0.1
max_iterations = 0
min_iterations = 100
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))
```

```
# Define function, derivative and begin computation

f = lambda x : math.exp(x)-(2*x*math.cos(x))-3
fdot = lambda x : math.exp(x)+2*(x*math.sin(x)-math.cos(x))

for i in np.arange(x_lower, x_upper, step_size):
#Looking for max/min iterations in interval with given step size
    repetitions = 0
    x_prev = i
    ea = 100
    while ea>es:
        repetitions = repetitions + 1
        x_next = x_prev - (f(x_prev)/fdot(x_prev))
        ea = math.fabs((x_next-x_prev)*100/x_next)
        x_prev = x_next
    if repetitions > max_iterations:
        max_iterations = repetitions
        max_iterations_initial = i
    elif repetitions < min_iterations:
        min_iterations = repetitions
        min_iterations_initial = i

print('Max iterations needed to find root is for initial value x_0 = %.2f (%d␣
 ↪iterations)!' % (max_iterations_initial, max_iterations))
print('Min iterations needed to find root is for initial value x_0 = %.2f (%d␣
 ↪iterations)!' % (min_iterations_initial, min_iterations))
```

```
Max iterations needed to find root is for initial value x_0 = 0.00 (14
iterations)!
Min iterations needed to find root is for initial value x_0 = 1.30 (3
iterations)!
```

Based on this result it is evident that for this specific example, the **Newton-Raphson** method outperforms the bi-section method, with the worst case scenario requiring 4 less iterations than the bisection method to achieve the requested accuracy.

2.3 Solution for (b)

$$(b)\ \ x^2 + \sin(x) + e^x - 2 = 0,\ x \in (0, 1)$$

In a similar manner, let's begin with the bisection method and plot the results:

[6]:
```
import math
import numpy as np
import matplotlib.pyplot as plt

# Bi-section method
```

```
x_lower = 0
x_upper = 1
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))
ea = 100
repetitions = 0
value_prev = 100

# Define function and begin computation

f = lambda x : pow(x,2)+math.sin(x)+math.exp(x)-2

while ea>es:
    xr = (x_lower+x_upper)/2
    repetitions = repetitions + 1
    if f(x_lower)*f(xr)<0:
        x_upper = xr
        value = xr
    elif f(x_upper)*f(xr)<0:
        x_lower = xr
        value = xr
    ea = math.fabs((value-value_prev)*100/value)
    value_prev = xr

print('Root found (after %d iterations)! x_root = %.5f' % (repetitions, xr))
```

```
Root found (after 19 iterations)! x_root = 0.38708
```

To use the Newton-Raphson method, we have again to calculate the derivative. Let's assume:

$$f(x) = x^2 + \sin(x) + e^x - 2$$

then the derivative of $f(x)$ is:

$$f'(x) = 2x + \cos(x) + e^x$$

And let's also assume that we start from the middle of the interval $(0, 1)$, and therefore we have $x_0 = 0.5$. To make the comparison "fair" again, we will also determine maximum possible amount of iterations needed if we start from any value inside $(0, 1)$ with a $step = 0.1$.

[7]:
```
import math

# Newton-Raphson method
```

```
x0 = 0.5
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))
ea = 100
repetitions = 0
x_prev = x0

# Define function, derivative and begin computation

f = lambda x : pow(x,2)+math.sin(x)+math.exp(x)-2
fdot = lambda x : (2*x)+math.cos(x)+math.exp(x)

while ea>es:
    x_next = x_prev - (f(x_prev)/fdot(x_prev))
    ea = math.fabs((x_next-x_prev)*100/x_next)
    repetitions = repetitions + 1
    x_prev = x_next

print('Root found (after %d iterations)! x_root = %.5f' % (repetitions, x_next))
```

```
Root found (after 4 iterations)! x_root = 0.38708
```

[8]:
```
import math
import numpy as np

# Newton-Raphson method (step increasing - max/min_iterations finder)

x_lower = 0
x_upper = 2
step_size = 0.1
max_iterations = 0
min_iterations = 100
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))

# Define function, derivative and begin computation

f = lambda x : pow(x,2)+math.sin(x)+math.exp(x)-2
fdot = lambda x : (2*x)+math.cos(x)+math.exp(x)

for i in np.arange(x_lower, x_upper, step_size):
#Looking for max/min iterations in interval with given step size
    repetitions = 0
    x_prev = i
    ea = 100
    while ea>es:
        repetitions = repetitions + 1
```

```
        x_next = x_prev - (f(x_prev)/fdot(x_prev))
        ea = math.fabs((x_next-x_prev)*100/x_next)
        x_prev = x_next
    if repetitions > max_iterations:
        max_iterations = repetitions
        max_iterations_initial = i
    elif repetitions < min_iterations:
        min_iterations = repetitions
        min_iterations_initial = i

print('Max iterations needed to find root is for initial value x_0 = %.2f (%d␣
 ↪iterations)!' % (max_iterations_initial, max_iterations))
print('Min iterations needed to find root is for initial value x_0 = %.2f (%d␣
 ↪iterations)!' % (min_iterations_initial, min_iterations))
```

```
Max iterations needed to find root is for initial value x_0 = 1.70 (6
iterations)!
Min iterations needed to find root is for initial value x_0 = 0.40 (3
iterations)!
```

2.4 Conclusion

For both functions used in (a) and (b), we can clearly see that the Newton-Raphson method outperforms the bisection method.

3 Exercise 2

3.1 Problem Statement

Find the root to the following equation, using the $x = g(x)$ method (carefully select the $x = g(x)$ format that converges):

$$2e^x - 3x^2 = 0,\ x \in [-1, 1]$$

Also use Newton-Raphson and compare the results of the two methods.

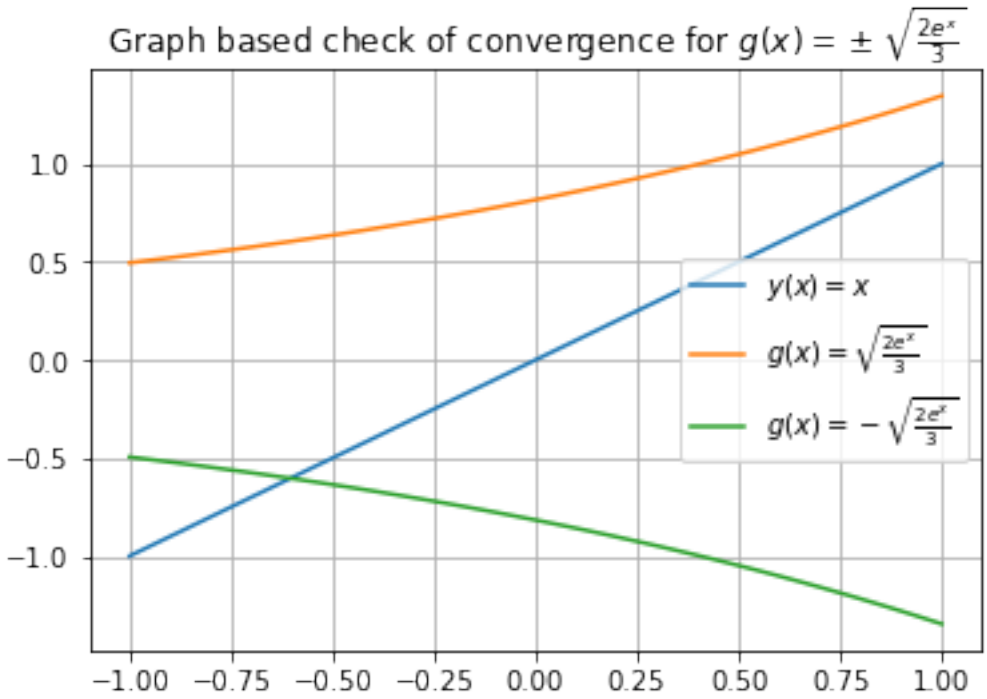
3.2 Solution using the $x = g(x)$ method

Let us first examine the simple fixed point iteration method or $x = g(x)$ method. We first have to bring the original $f(x) = 0$ equation in the $x = g(x)$ form. By rearranging the equation, we obtain

$$\begin{aligned} & 2e^x - 3x^2 = 0 \\ \Rightarrow & 2e^x = 3x^2 \\ \Rightarrow & \frac{2e^x}{3} = x^2 \\ \Rightarrow & x = \pm\sqrt{\frac{2e^x}{3}} \end{aligned}$$

which is in the desired $x = g(x)$ format. Before beginning the actual calculation, let's determine if the plus or minus sign will converge by plotting $y(x) = x$ and $g(x) = \pm\sqrt{\frac{2e^x}{3}}$, and also checking the derivative for the correct function.

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

# Plot the selected g(x) to see if it converges

x = np.linspace(-1,1,100)
y = x
g1 = np.sqrt((2*np.exp(x))/3)
g2 = -np.sqrt((2*np.exp(x))/3)

plt.figure()
plt.plot(x,y,label=r'$y(x)=x$')
plt.plot(x,g1,label=r'$g(x)=\sqrt{\frac{2e^{x}}{3}}$')
plt.plot(x,g2,label=r'$g(x)=-\sqrt{\frac{2e^{x}}{3}}$')
plt.title(r'Graph based check of convergence for␣
 ↪$g(x)=\pm\sqrt{\frac{2e^{x}}{3}}$')
plt.legend()
plt.grid()
```



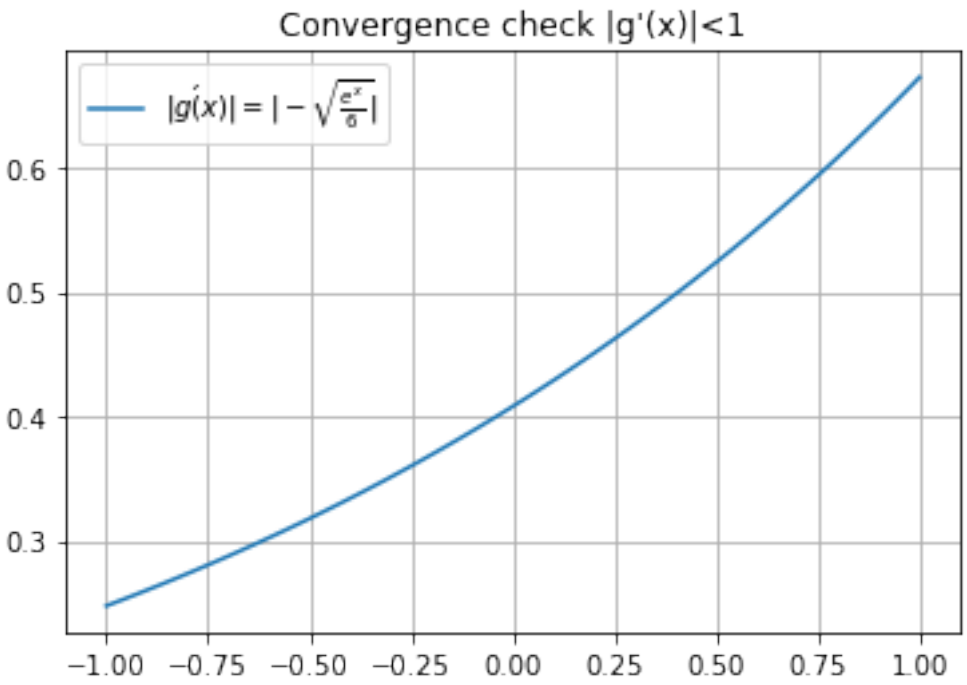
Based on the figure above, we can conclude that the solution exists for $g(x) = -\sqrt{\frac{2e^x}{3}}$. Let's also

prove the convergence by checking the derivative of $g(x)$:

$$
\begin{aligned}
g'(x) &= (-\sqrt{\frac{2e^x}{3}})' \\
\Rightarrow g'(x) &= -\sqrt{\frac{2}{3}}(\sqrt{e^x})' \\
\Rightarrow g'(x) &= -\sqrt{\frac{2}{3}}\frac{e^x}{2\sqrt{e^x}} \\
\Rightarrow g'(x) &= -\sqrt{\frac{e^x}{6}}
\end{aligned}
$$

Let's now plot $|g'(x)|$ to determine if the $x = g(x)$ method will converge:

[10]:
```
import numpy as np
import matplotlib.pyplot as plt

# Plot g'(x) to determine if |g'x|<1

x = np.linspace(-1,1,100)
y = x
gdot = np.abs(-np.sqrt(np.exp(x)/6))

plt.figure()
plt.plot(x,gdot,label=r'$|g\'(x)|=|-\sqrt{\frac{e^{x}}{6}}|$')
plt.title('Convergence check |g\'(x)|<1')
plt.legend()
plt.grid()
```



We can indeed observe that for the entire interval $x \in [-1, 1]$ the inequality $|g'(x)| < 1$ holds true. We can proceed with finding the root using the simple fixed-point iteration method, computing the approximate error in each step and comparing it to a predetermined Scarborough criterion in order to use it as a stopping condition.

[11]:
```
import math
import numpy as np

# Simple fixed-point iteration or x=g(x) method

x0 = -0.5
max_iterations = 200
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))
ea = 100
g = lambda x : -np.sqrt(2*np.exp(x)/3)

xr = x0
iterations = 0

while ea>=es:
    xr_old = xr
    xr = g(xr_old)
    iterations += 1
    if xr != 0:
```

```
        ea = math.fabs((xr-xr_old)*100/xr)
    if iterations >= max_iterations:
        break
    #print('xr=%.5f xr_old=%.5f ea=%.5f es=%.5f' % (xr,xr_old,ea,es))

print('x=g(x) method:')
if iterations < max_iterations:
    print('Root found (after %d iterations)! x_root = %.5f' % (iterations, xr))
else:
    print('Max iterations reached without solution!')
```

```
x=g(x) method:
Root found (after 10 iterations)! x_root = -0.60374
```

3.3 Solution using the Newton-Raphson Method

Let us now compare the $x = g(x)$ method to the Newton-Raphson Method. To solve $2e^x - 3x^2 = 0$, let's assume:

$$f(x) = 2e^x - 3x^2, \; x \in [-1, 1]$$
$$f'(x) = 2e^x - 6x$$

[12]:
```
import math
import numpy as np

# Newton-Raphson method

f = lambda x : 2*np.exp(x)-3*pow(x,2)
fdot = lambda x : 2*np.exp(x)-6*x

x0 = -0.5
max_iterations = 200
n = 5 # Number of significant digits to be computed
es = 0.5*pow(10, (2-n))
ea = 100

xr = x0
iterations = 0

while ea>es:
    xr_old = xr
    xr = xr_old - (f(xr_old)/fdot(xr_old))
    iterations += 1
    if xr != 0:
        ea = math.fabs((xr-xr_old)*100/xr)
    if iterations >= max_iterations:
        break
```

```
print('Newton-Raphson Method:')
if iterations < max_iterations:
    print('Root found (after %d iterations)! x_root = %.5f' % (iterations, xr))
else:
    print('Max iterations reached without solution!')
```

```
Newton-Raphson Method:
Root found (after 4 iterations)! x_root = -0.60374
```

3.4 Conclusion

Using exactly the same initial guess for both methods, we notice that the Newton-Raphson method outperforms the $x = g(x)$ method. This happens because the error of each iteration is approximately the square of the previous iteration error.

4 Exercise 3

4.1 Problem Statement

Find the Lagrange polynomials that have the same values as the given functions, at the points specified:

$$y(x) = 1 + sin(\pi x),\ for\ x = -1, 0\ and\ 1$$
$$f(x) = \frac{1}{1+x^2},\ for\ x = -5, -4, -3, ..., 3, 4, 5$$

Plot your results for comparison.

4.2 Solution for $y(x)$

For any given set of n data points, we can get a Langrange polynomial of n-1th order. Therefore, since we have 3 data sets based on $y(x)$ we can determine that our Lagrange polynomial will be a 2nd order (max) polynomial given by the formula:

$$P_n(x) = \sum_{i=0}^{n} f_i L_i(x)$$

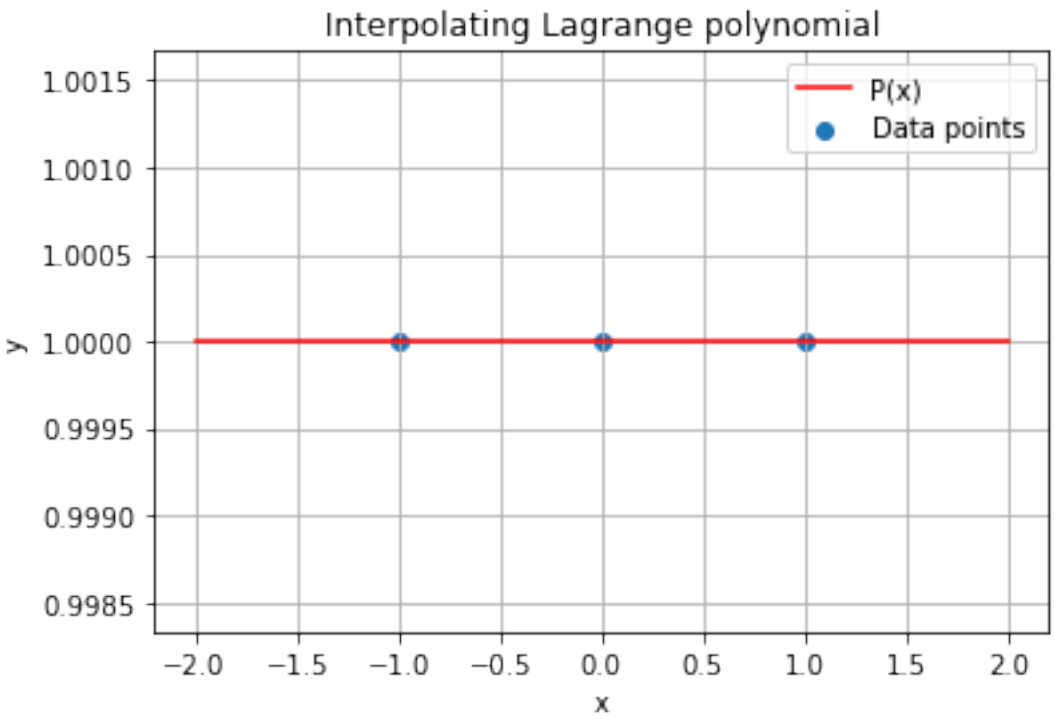
where L_i is:

$$L_i(x) = \prod_{j=0,\ j\neq 1}^{n} \frac{x - x_j}{x_i - x_j}$$

Let's now calculate it:

[13]:
```
import matplotlib.pyplot as plt
import sympy
import numpy as np

x = sympy.symbols('x')
# Insert data sets, data_y derived from function
data_x = np.array([-1,0,1])
data_y = 1+np.sin(data_x*np.pi)
P = 0

# Calculate the Lagrange interpolating polynomial
for i in range(len(data_x)):
    L = 1
    for j in range(len(data_x)):
        if i!=j:
            L = L * (x-data_x[j])/(data_x[i]-data_x[j])
    P += L*data_y[i]
print('The Lagrange interpolating polynomial is:')
print(P)
print('\nSimplifying the Lagrange interpolating polynomial...')
print(sympy.simplify(P))

# Plot with matplotlib instead of sympy.plot to also add scatter points
# Create a np.linspace and plot the function there
lam_P = sympy.lambdify(x,P,modules=['numpy'])
x_plot = np.linspace(-2,2,100)
y_plot = lam_P(x_plot)

plt.plot(x_plot,y_plot,color='red',label='P(x)')
plt.scatter(data_x,data_y,label='Data points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Interpolating Lagrange polynomial')
plt.legend()
plt.grid()
```

```
The Lagrange interpolating polynomial is:
1.0*x*(x/2 + 1/2) + 0.5*x*(x - 1) - 1.0*(x - 1)*(x + 1)

Simplifying the Lagrange interpolating polynomial…
1.66533453693773e-16*x + 1.0
```



We see that the terms cancel each other out and we therefore end up with a linear solution for the three datapoints.

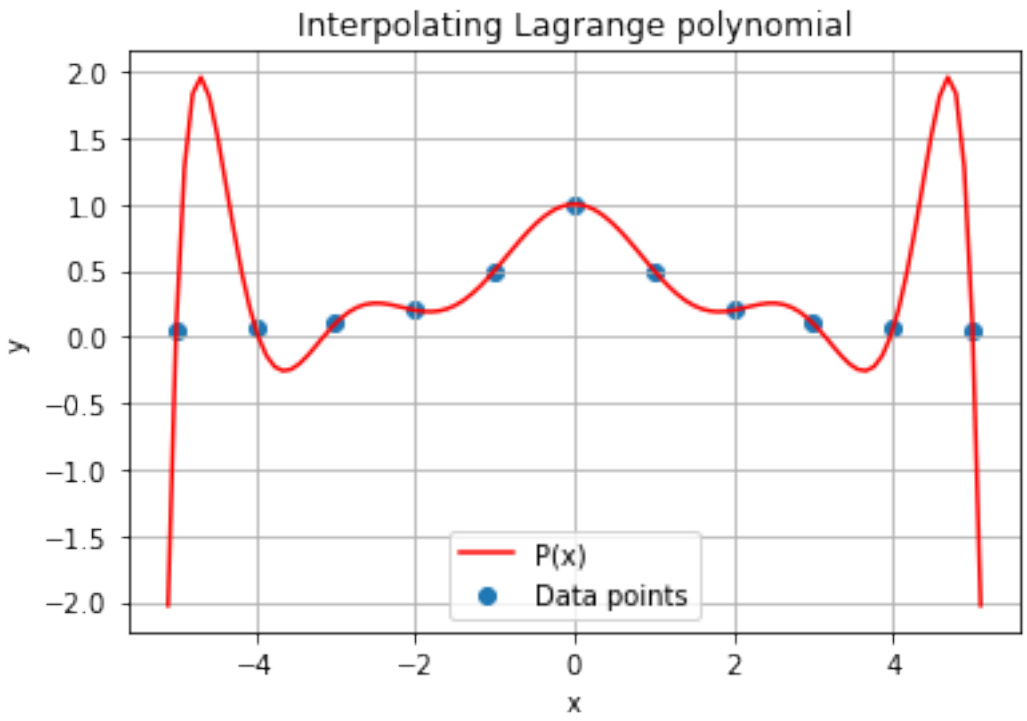
4.3 Solution for $f(x)$

For the given $f(x)$, since we have 11 data sets, we can determine that our Lagrange polynomial will be a 10th order polynomial. Let's now calculate it:

[14]:
```
import matplotlib.pyplot as plt
import sympy
import numpy as np

x = sympy.symbols('x')
# Insert data sets, data_y derived from function
data_x = np.array([-5,-4,-3,-2,-1,0,1,2,3,4,5])
data_y = 1/(1+(data_x*data_x))
P = 0

# Calculate the Lagrange interpolating polynomial
for i in range(len(data_x)):
    L = 1
    for j in range(len(data_x)):
        if i!=j:
            L = L * (x-data_x[j])/(data_x[i]-data_x[j])
```

```
    P += L*data_y[i]
print('The Lagrange interpolating polynomial is:')
print(P)
print('\nSimplifying the Lagrange interpolating polynomial...')
print(sympy.simplify(P))


# Plot with matplotlib instead of sympy.plot to also add scatter points
# Create a np.linspace and plot the function there
lam_P = sympy.lambdify(x,P,modules=['numpy'])
x_plot = np.linspace(-5.1,5.1,100)
y_plot = lam_P(x_plot)

plt.plot(x_plot,y_plot,color='red',label='P(x)')
plt.scatter(data_x,data_y,label='Data points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Interpolating Lagrange polynomial')
plt.legend()
plt.grid()
```

```
The Lagrange interpolating polynomial is:
-1.05989689323023e-8*x*(-x - 4)*(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x - 1)*(x +
1)*(x + 2)*(x + 3) + 1.05989689323023e-7*x*(x/10 + 1/2)*(x - 4)*(x - 3)*(x -
2)*(x - 1)*(x + 1)*(x + 2)*(x + 3)*(x + 4) - 1.45891690009337e-6*x*(x/9 +
5/9)*(x - 5)*(x - 3)*(x - 2)*(x - 1)*(x + 1)*(x + 2)*(x + 3)*(x + 4) +
9.92063492063492e-6*x*(x/8 + 5/8)*(x - 5)*(x - 4)*(x - 2)*(x - 1)*(x + 1)*(x +
2)*(x + 3)*(x + 4) - 4.62962962962963e-5*x*(x/7 + 5/7)*(x - 5)*(x - 4)*(x -
3)*(x - 1)*(x + 1)*(x + 2)*(x + 3)*(x + 4) + 0.000173611111111111*x*(x/6 +
5/6)*(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x + 1)*(x + 2)*(x + 3)*(x + 4) +
0.000115740740740741*x*(x/4 + 5/4)*(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x - 1)*(x +
2)*(x + 3)*(x + 4) - 1.98412698412698e-5*x*(x/3 + 5/3)*(x - 5)*(x - 4)*(x -
3)*(x - 2)*(x - 1)*(x + 1)*(x + 3)*(x + 4) + 2.48015873015873e-6*x*(x/2 +
5/2)*(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x - 1)*(x + 1)*(x + 2)*(x + 4) -
1.62101877788152e-7*x*(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x - 1)*(x + 1)*(x + 2)*(x
+ 3)*(x + 5) - 0.000347222222222222*(x/5 + 1)*(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x
- 1)*(x + 1)*(x + 2)*(x + 3)*(x + 4)

Simplifying the Lagrange interpolating polynomial…
-2.26244343891403e-5*x**10 - 1.6940658945086e-21*x**9 + 0.00126696832579185*x**8
+ 1.0842021724855e-19*x**7 - 0.0244117647058824*x**6 - 1.73472347597681e-18*x**5
+ 0.19737556561086*x**4 + 6.93889390390723e-18*x**3 - 0.67420814479638*x**2 -
6.93889390390723e-18*x + 1.0
```



4.4 Conclusion

The code used above can be used to calculate any Lagrange Interpolating Polynomial, given an initial set of data. We can also notice in the final case that some of the terms of the Lagrange polynomial are smaller than 10^{-17}, leading us to thinking that our computations are suffering from floating point arithmetic errors.

5 Exercise 4

5.1 Problem Statement

Use Newton-Gregory formula to calculate an interpolating polynomial going through the following equidistant points. Plot your result.

x	f(x)
0.2	0.185
0.3	0.106
0.4	0.093
0.5	0.24
0.6	0.579
0.7	0.561

5.2 Solution

For equally spaced points (equidistant), we can use the **Newton-Gregory** formula to calculate an interpolating polynomial through the points:

$$P_n(x) = f_0 + \binom{s}{1}\Delta f_0 + \binom{s}{2}\Delta^2 f_0 + ... + \binom{s}{n}\Delta^n f_0$$

Where Δf are the differences which are usually computed inside a difference table, and s is:

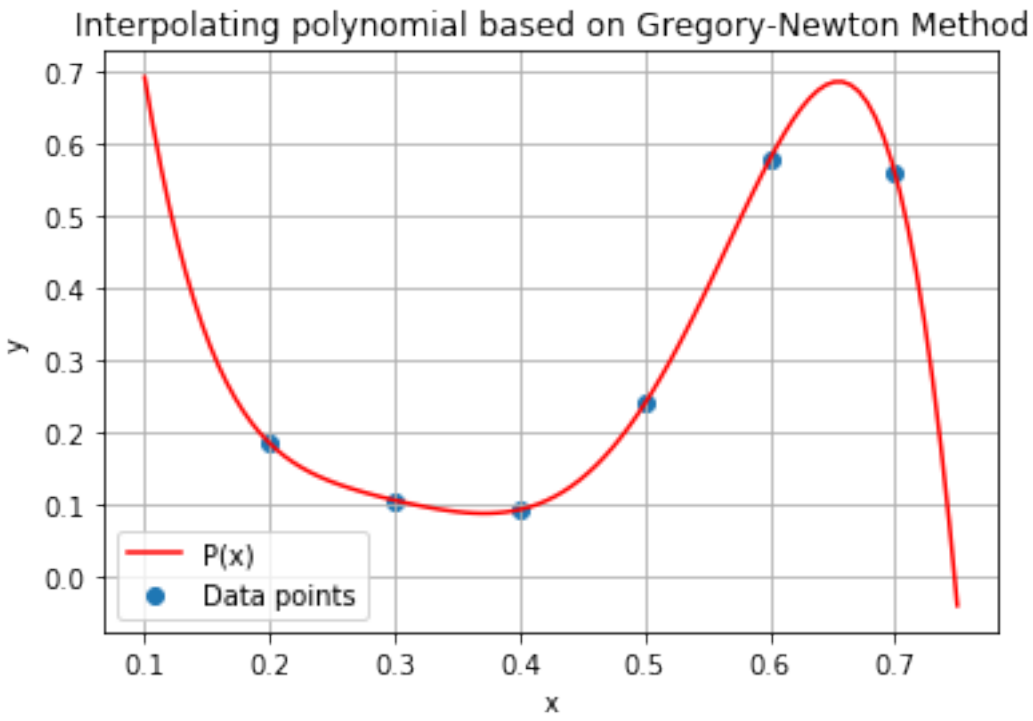
$$x = x_0 + sh \Rightarrow s = \frac{x - x_0}{h}$$

where h is the distance between the equally spaced points.

[15]:
```
import matplotlib.pyplot as plt
import sympy
import numpy as np
import math

x = sympy.symbols('x')

data_x = np.array([0.2,0.3,0.4,0.5,0.6,0.7])
data_y = np.array([0.185,0.106,0.093,0.24,0.579,0.561])
n = len(data_x)

# Equidistant check
for i in range(0,n-3):
    h = data_x[1] - data_x[0]
    if (h - data_x[i+2] - data_x[i+1] > pow(10,-9)):
        # Condition above is avoiding floating point arithmetic issues
        print('Warning: Input data are not equally spaced!')
        break
s = (x-data_x[0])/h

# Initialize dif_table and add f(x) to first column:
dif_table = np.zeros((n,n))
for i in range(n):
    dif_table[i,0] = data_y[i]

for i in range(1,n):
    for j in range(0,n-i):
        dif_table[j,i] = dif_table[j+1,i-1] - dif_table[j,i-1]

print('Using the Newton-Gregory formula to find polynomial...\n')
print('The difference table is:')
print(dif_table)
```

```python
# Compute Newton-Gregory polynomial

P = data_y[0]
temp_s = 1
for i in range(n-1):
    if i==0:
        P = data_y[0]
        temp_s = 1
        temp_s *= (s-i)
        P += dif_table[0,i+1]*temp_s/math.factorial(i)
    else:
        temp_s *= (s-i)
        P += dif_table[0,i+1]*temp_s/math.factorial(i+1)

print('\nThe interpolating polynomial is:')
print(sympy.simplify(P))


lam_P = sympy.lambdify(x,P,modules=['numpy'])
x_plot = np.linspace(0.1,0.75,100)
y_plot = lam_P(x_plot)

plt.plot(x_plot,y_plot,color='red',label='P(x)')
plt.scatter(data_x,data_y,label='Data points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Interpolating polynomial based on Gregory-Newton Method')
plt.legend()
plt.grid()
```

```
Using the Newton-Gregory formula to find polynomial…

The difference table is:
[[ 0.185 -0.079  0.066  0.094 -0.062 -0.519]
 [ 0.106 -0.013  0.16   0.032 -0.581  0.   ]
 [ 0.093  0.147  0.192 -0.549  0.     0.   ]
 [ 0.24   0.339 -0.357  0.     0.     0.   ]
 [ 0.579 -0.018  0.     0.     0.     0.   ]
 [ 0.561  0.     0.     0.     0.     0.   ]]

The interpolating polynomial is:
-432.5*x**5 + 839.166666666667*x**4 - 618.541666666667*x**3 +
221.708333333333*x**2 - 39.5413333333333*x + 2.969
```


Interpolating polynomial based on Gregory-Newton Method
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
y
P(x)
Data points
0.1
0.2
0.3
0.4
0.5
0.6
0.7
x