MA 3525 ELEMENTARY NUMERICAL METHODS
PROJECT 1

Veljko Kovac
55173123

香港城市大學
City University of Hong Kong

Remark about project:
Codes that were written for the solutions and implementation of the algorithms and exercises were written in **Python** using Jupyter Notebook and PyCharm**.**

Question 1:

For the first question of the project we basically had to implement Newton's and Secant method on specific function on the interval [0,2] with error less than 10e-10.
The true solution of this function is: 1.1391941468830,
The equation was the following:

$$x^3 - 9x^2 + 23x - 16 = 0$$

Having as reference the pseudocodes from the book (Introduction to Scientific Computing and Data Analysis) which is the following: (On the left we have the pseudocode for secant method and on the right we have my Python implementation.

Secant Method:

pick: x and X
$tol > 0$
$M > 0$
$I > 0$
let: $err = 3 * tol$
$i = 0$
loop: while $err > tol$
$z = f(x)(x - X)/(f(x) - f(X))$
$err = abs(z)$
$X = x$
$x = x - z$
$i = i + 1$
if $abs(x) > M$ or $I < i$, then stop
end

```
def f(x):
    return x**3 - 9*x**2 + 23*x - 16

def der_f(x):
    return 3*x**2 - 18*x + 23

tol = 10e-10

import numpy as np
def secant(f, x, x2, tol):
    err = 3*tol
    err_arr = []
    err_arr.append(err)
    old_err = abs(x2-x)
    err_arr.append(old_err)
    x_arr = []
    x_arr.append(x)
    x_arr.append(x2)
    gamma_arr = []
    gamma_arr.append(np.nan)
    gamma_arr.append(np.log(abs(x2-x))/np.log(err))
    i = 0


    while err>tol:
        z = (f(x)*(x-x2))/((f(x)-f(x2)))
        err = abs(z)
        err_arr.append(err)
        gamma_arr.append(np.log(err)/np.log(old_err))
        x2 = x
        x = x - z
        x_arr.append(x)
        i = i + 1
        old_err = err
    return x, x_arr,err_arr,gamma_arr
```

For the sake of this exercise I kept an array with errors, which are xi - x exact, and I also kept an array with the Gamma Rate. Inputting, the x values, the errors and the gamma rates as columns in a Pandas Dataframe we got the following table.

	X Values	Xi - x_exact	Gamma
0	0.500000	6.391941e-01	NaN
1	0.700000	4.391941e-01	0.082011
2	0.998495	1.406990e-01	0.432549
3	1.088377	5.081697e-02	3.460773
4	1.133680	5.514370e-03	1.284376
5	1.138958	2.356561e-04	1.694703
6	1.139193	1.131129e-06	1.593790
7	1.139194	2.276159e-10	1.638260
8	1.139194	5.296430e-12	1.619892

For my initial guesses x0,x1 I chose the values 0.5 and 0.7. As you can see from the table that the specific root finding method converge to the solution after 8 iterations with an error of 5.296430e-12. However, it is worth mentioning that after only 4 iterations the value of X is already 1.133680 which is already pretty close to the exact solution.
Worth mentioning that **Gamma** rate reaches close to its golden ratio which equals approximately to 1.6180.
Just for clarification, the first 2 errors and gamma rates are not so instructive because we use as x values the ones that we guessed by ourselves. First gamma had to be Nan because gamma rate is defined like:

$$\gamma \approx \frac{\ln e_{i+1}}{\ln e_i}.$$

Newton's Method:

Having as reference the pseudocodes from the book (Introduction to Scientific Computing and Data Analysis): (On the left we have the pseudocode for Newton's method and on the right we have my Python implementation.

pick: x
$tol > 0$
let: $err = 3 * tol$
loop: while $err > tol$
$z = f(x)/f'(x)$
$err = abs(z)$
$x = x - z$
end

```
def newton(f, df, tol):

    x = 1
    old_err = 3*tol
    err_arr = []
    err_arr.append(old_err)
    x_arr = []
    gamma_arr = []
    x_arr.append(x)
    it_count = 0
    gamma_arr.append(np.nan)

    while old_err > tol:
        z = f(x)/df(x)
        err = abs(z)
        err_arr.append(err)

        gamma_arr.append(np.log(err)/np.log(old_err))
        x = x - z
        x_arr.append(x)
        it_count += 1
        old_err = err
        print(x)

    return x, x_arr, err_arr,gamma_arr
```

Again, as we did for the secant method I kept record the x values, errors and the gamma rates. What we got is recorded, on the following table.

	X Values	Xi-x_exact	Gamma
0	1.000000	1.391941e-01	NaN
1	1.125000	1.419415e-02	0.105961
2	1.139021	1.726672e-04	2.052073
3	1.139194	2.604344e-08	2.030457
4	1.139194	5.296430e-12	2.015547

For my initial guess I chose the value 1, which is clearly the midpoint of the given interval [0,2]. We can see from the table that Newton's method within 4 iteration achieves the asked error of 5.296430e-12. We must also note that the gamma rate is approaching 2, which is the actual convergence gamma rate for Newton's Method.

Conclusions:

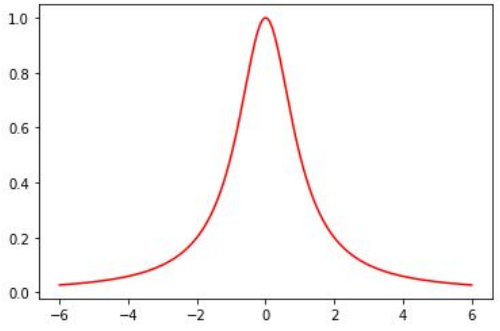
We can clearly see that the Newton's method converged much faster than the Secant Method. And this statement is also supported by the fact that gamma rate is approximately 2 for the Newton's Method while is only 1.6 for the Secant.

Question 2:

In this question we were asked to perform interpolation with two different techniques, lagrange and chebyshev. Later, we should compare these 2 interpolation methods to see how they perform in relation to the actual function polynomial.

This is our original function and has the shape that you see next to it.

$$f(x) = \frac{1}{x^2+1},$$



Now, we have to perform interpolation on the interval [-6,6] with 21 equally distributed nodes. Knowing that lagrange interpolation polynomial is the following:

$$p_n(x) = \sum_{i=1}^{n+1} y_i \ell_i(x),$$

with li(x) equal to

$$\prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{x - x_j}{x_i - x_j}.$$

And that the formula for Chebyshev Points is:

$$x_i = \frac{1}{2}[a + b + (b - a)z_i], \text{ for } i = 1, 2, \cdots, n+1,$$

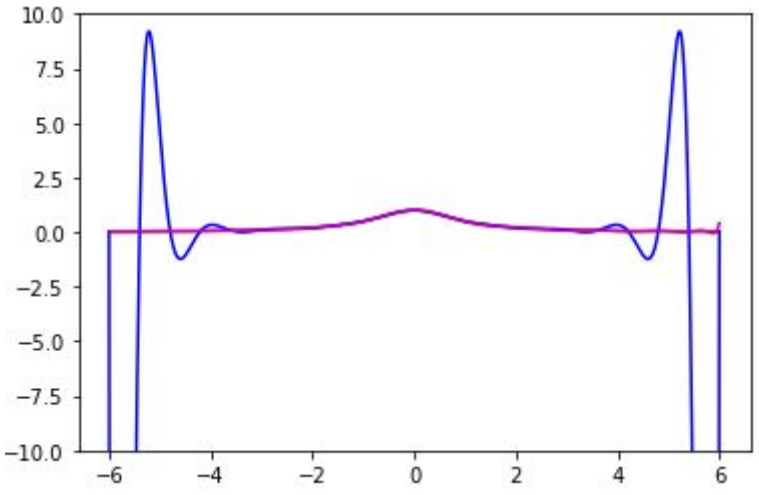
$$z_i = \cos\left(\frac{2i-1}{2(n+1)}\pi\right).$$

Applying these two formulas in Python with the following way we get from the lagrange function the formula for the li(x) and later in the interpolation function we evaluate for different Xs.

```
def lagrange(points, i):
    s = ''
    for j in range(len(points)):
        if j == i:
            continue
        s += ('('+'x'+'-'+ str(points[j])+')'+'/'+'('+str(points[i]) +'-'+ str(points[j])+')*')


    return s[:-1]

def interpolate(points, f):
    s = ''
    for i, x in enumerate(points):
        s += str(f(x)) + '*' + lagrange(points, i) +'+'

    return lambda x : eval(s[:-1])
```

```
def f(x):
    return 1/(x**2 + 1)


f_lagrange = interpolate(np.linspace(a, b, n), f)

z_values = []
for i in range(1,n+2):
    z_values.append(np.cos((2*i/1)/(2*n+2)*np.pi))
#     z values

x_cheb = []
for i in range(0,n+1):
    x_cheb.append(0.5*(a+b+(b-a)*z_values[i]))
    #Chebyshev points

f_chebyshev = interpolate(x_cheb, f)
```

Plotting now, the actual function with comparison to the other two interpolation techniques we get the following graph. The real line is plotted with red color, the lagrange interpolation is plotted with blue while the chebyshev interpolation was plotted with magenta.



<u>Conclusions:</u>

What this graph shows is that both interpolation performs so well that actually the red color (actual function) does not even show on the plot.
A crucial point to emphasize is that we can clearly see that the lagrange interpolation performs badly when X values goes closely to the endpoints of the interval. We can also support this fact by finding the maximum values of the difference between each interpolation and the actual function. The values that we got were: **128.08125956522707 for the Lagrange** and 0.**37630836740120804 for the Chebyshev**. It is clear that Lagrange maximum difference is huge which actually takes place at the endpoints.

Question 3:

For this question we had to compute the following integral with Trapezoidal and Simpson's composite rule for different alphas and different interval values.

$$n = 2^k + 1,\ k = 1, 2, 3, 4$$

$$\int_0^1 x^{\alpha+3/5} dx$$

Alpha value can take values 0,1,2. For this exercise I calculated by myself the exact values of the integrals.
For alpha = 0, exact value of the integral is: 0.625
For alpha = 1, exact value of the integral is: 0.3846153846153846
For alpha = 2, exact value of the integral is: 0.2777777777777778

For k = 1, n = 3
For k = 2, n = 5
For k = 3, n = 9
For k = 4, n = 17

Trapezoidal Rule:

```
def Trapezoidal(f, a, b):
    answ = []
    h_arr = []
    for k in range(1,5):
        n = 2**k + 1
        h = (b-a)/float(n)
        s = 0.5*(f(a) + f(b))
        for i in range(1,n,1):
            s = s + f(a + i*h)
        answ.append(h*s)
        h_arr.append(h)

    return answ, h_arr
```

This is the implementation of trapezoidal rule in Python.
We kept keeping record the x values that we get and the error from the actual x point.

For alpha = 0 we got the following:

	Values	Error
0	0.600445	0.024555
1	0.613704	0.011296
2	0.620426	0.004574
3	0.623297	0.001703

When we divide the error with h^2, we get a constant value of approximately : 0.22

For alpha = 1 we got the following:

	Values	Error
0	0.398376	0.013761
1	0.389669	0.005053
2	0.386201	0.001585
3	0.385065	0.000450

When we divide the error with h^2, we get a constant value of approximately : 0.12

For alpha = 2 we got the following:

	Values	Error
0	0.301981	0.024203
1	0.286466	0.008688
2	0.280455	0.002678
3	0.278528	0.000750

When we divide the error with h^2, we get a constant value of approximately :0.21

Conclusions for Trapezoidal:

As we can see, the trapezoidal works perfectly in our case. It does converge to the real solution. We must mention here that in both three cases, for different alphas, when we divide by h^2 we get an approximately constant number which proves the Theorem 6.2

Simpson's Rule:

```
def simpson(a, b, f):
    sum = 0
    ans = []
    for m in range(1,5):
        sum = 0
        n = 2**m
        inc = (b - a) / n
        for k in range(n+1):
            x = a + (k * inc)
            summand = f(x)
            if (k != 0) and (k != n):
                summand *= (2 + (2 * (k % 2)))
            sum += summand
        ans.append(((b - a) / (3 * n)) * sum)
    return ans
```

This is the implementation of simpson's composite rule in Python. We kept record the x values that we get and the error from the actual x point.

For alpha = 0 we got the following:

	Values	Error
0	0.606503	0.018497
1	0.618873	0.006127
2	0.622977	0.002023
3	0.624333	0.000667

For alpha = 1 we got the following:

	Values	Error
0	0.386585	0.001969
1	0.384952	0.000337
2	0.384672	0.000056
3	0.384625	0.000009

When we divide the error with h^4, we get a constant value of approximately : 0.5

For alpha = 2 we got the following:

	Values	Error
0	0.276626	1.152119e-03
1	0.277666	1.115189e-04
2	0.277768	1.026430e-05
3	0.277777	9.138607e-07

When we divide the error with h^4, we get a constant value of approximately :0.07

<u>Conclusions for Simpson's:</u>

As we can see, theSimpson'sl works perfectly. It does converge to the real solution with a high efficiency. We must mention here that , for different alphas, when we divide by h^4 we get an approximately constant number which proves the Theorem 6.3.