Task 2

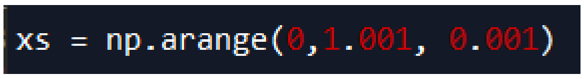
Egor Turukhanov

Group C4134

09.10.19

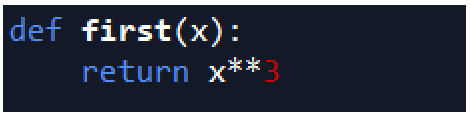
First

Generation of X’s (boundaries)



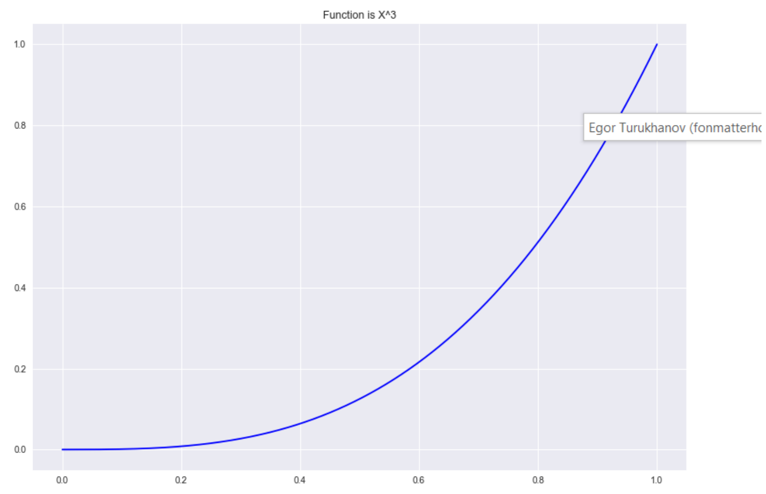
```
xs = np.arange(0,1.001, 0.001)
```

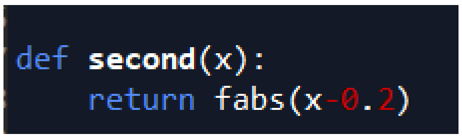
First



```
def first(x):
    return x**3
```

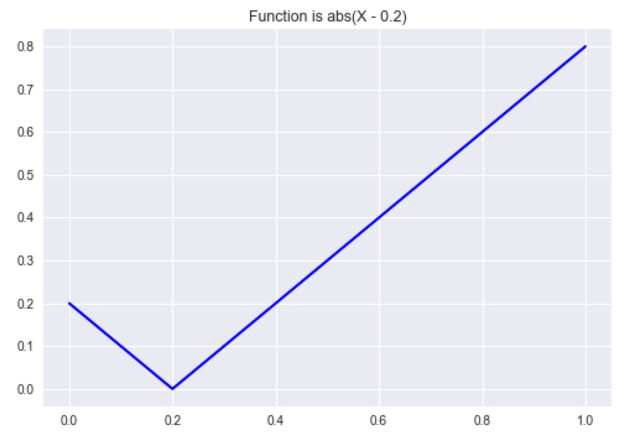
Graph of First Function

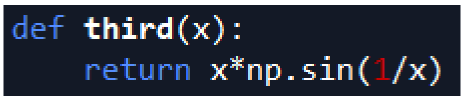




```
def second(x):
    return fabs(x-0.2)
```

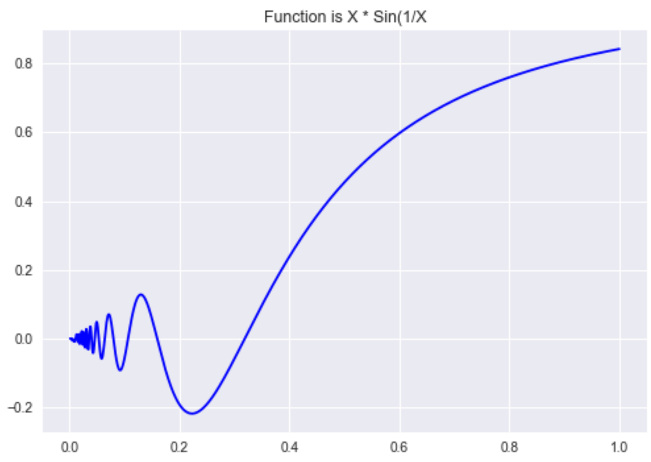
Graph of Second Function





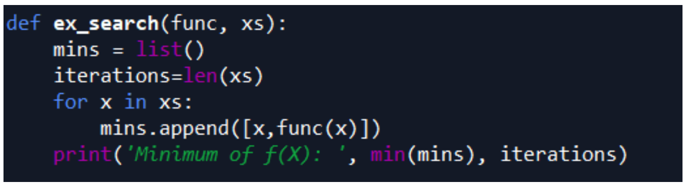
```
def third(x):
    return x*np.sin(1/x)
```

And graph



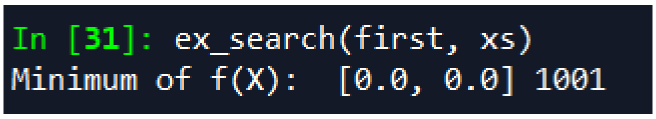
Exhaustive search method

First function



```
def ex_search(func, xs):
    mins = list()
    iterations=len(xs)
    for x in xs:
        mins.append([x,func(x)])
    print('Minimum of f(X): ', min(mins), iterations)
```

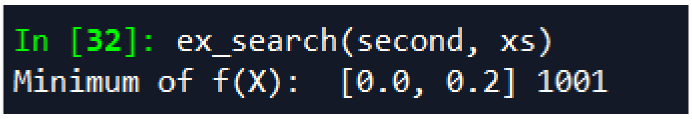
As we can see for the first function minimum is point 0, and value of function at this point is also 0. Exhaustive search method takes 1001 iteration to find minimum.



```
In [31]: ex_search(first, xs)
Minimum of f(X):  [0.0, 0.0] 1001
```

Second function

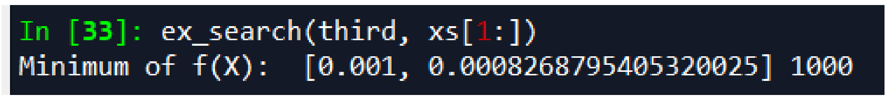
For second function minimum is also 0, and value of function at this point is 0.2 Exhaustive search method also takes 1001 iteration to find minimum for second function.



```
In [32]: ex_search(second, xs)
Minimum of f(X):  [0.0, 0.2] 1001
```

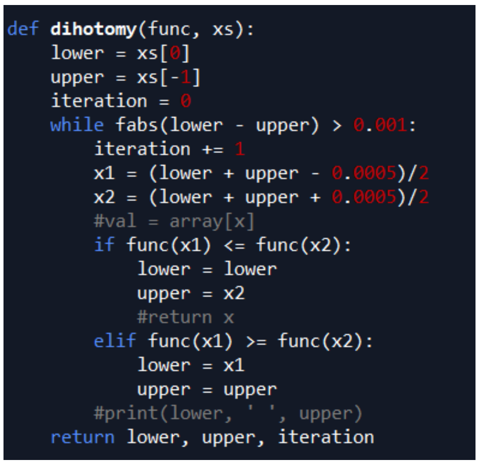
Third function

For third function minimum is 0.001, and value of function at this point is 0.00083 Exhaustive search method also takes 1000 iteration to find minimum for second function.



```
In [33]: ex_search(third, xs[1:])
Minimum of f(X):  [0.001, 0.0008268795405320025] 1000
```

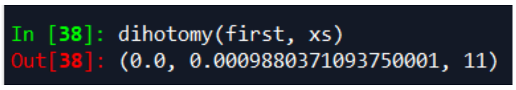
Dichotomy method



```
def dihotomy(func, xs):
    lower = xs[0]
    upper = xs[-1]
    iteration = 0
    while fabs(lower - upper) > 0.001:
        iteration += 1
        x1 = (lower + upper - 0.0005)/2
        x2 = (lower + upper + 0.0005)/2
        #val = array[x]
        if func(x1) <= func(x2):
            lower = lower
            upper = x2
            #return x
        elif func(x1) >= func(x2):
            lower = x1
            upper = upper
        #print(lower, ' ', upper)
    return lower, upper, iteration
```

First

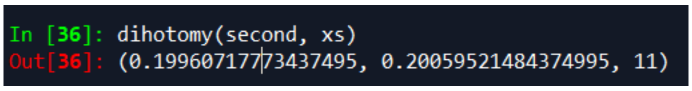
As we can see for the first function minimum is point 0, and value of function at this point is 0.00099. Dichotomy method takes only 11 iteration to find minimum, comparing to 1000 iterations for exhaustive search. The same for second and third function.



```
In [38]: dihotomy(first, xs)
Out[38]: (0.0, 0.0009880371093750001, 11)
```

Second

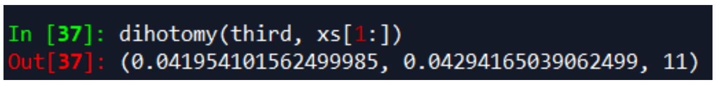
Minimum = 0.1996, function value = 0.2.



```
In [36]: dihotomy(second, xs)
Out[36]: (0.19960717773437495, 0.20059521484374995, 11)
```

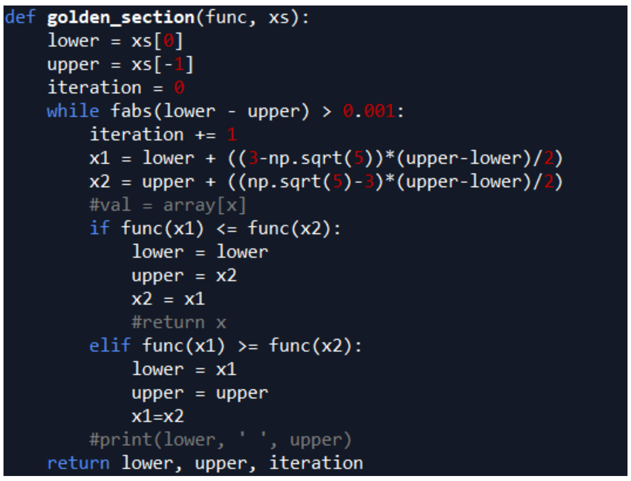
Third

Minimum = 0.042, function value = 0.043.



```
In [37]: dihotomy(third, xs[1:])
Out[37]: (0.041954101562499985, 0.04294165039062499, 11)
```

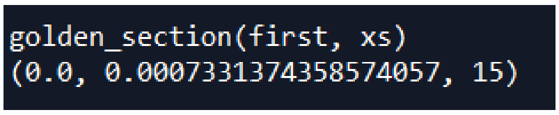
Golden section method



```
def golden_section(func, xs):
    lower = xs[0]
    upper = xs[-1]
    iteration = 0
    while fabs(lower - upper) > 0.001:
        iteration += 1
        x1 = lower + ((3-np.sqrt(5))*(upper-lower)/2)
        x2 = upper + ((np.sqrt(5)-3)*(upper-lower)/2)
        #val = array[x]
        if func(x1) <= func(x2):
            lower = lower
            upper = x2
            x2 = x1
            #return x
        elif func(x1) >= func(x2):
            lower = x1
            upper = upper
            x1=x2
        #print(lower, ' ', upper)
    return lower, upper, iteration
```

First

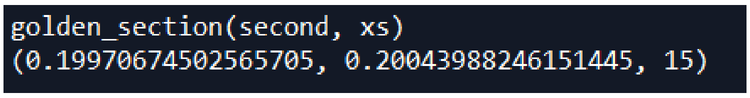
As we can see for the first function minimum is point 0, and value of function at this point is 0.0073. Dichotomy method takes 15 iteration to find minimum, comparing to 1000 iterations for exhaustive search and 11 iterations for Dichotomy method. The same for second and third function.



```
golden_section(first, xs)
(0.0, 0.0007331374358574057, 15)
```

Second

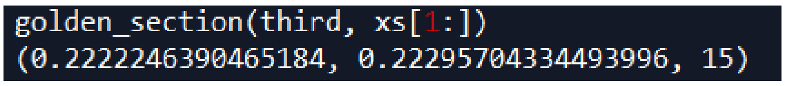
Minimum = 0.1997, function value = 0.2004



```
golden_section(second, xs)
(0.19970674502565705, 0.20043988246151445, 15)
```

Third

Minimum = 0.222, function value = 0.223.

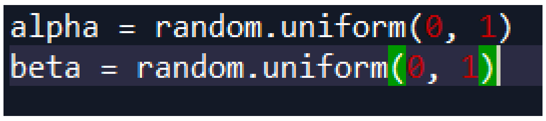


```
golden_section(third, xs[1:])
(0.2222246390465184, 0.22295704334493996, 15)
```

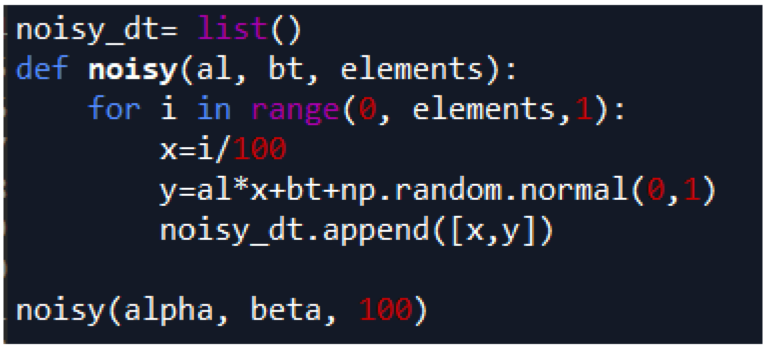
As can be seen from results above Exhaustive Search is the least efficient algorithms comparing to Golden Search and Dichotomy methods. However, Dichotomy method of function optimization takes less iteration than Golden Search method, but for Dichotomy search number of iterations can be increased of decreased by changing δ parameter. Also, these algorithms got us a bit different optimum.

Second

Data generation



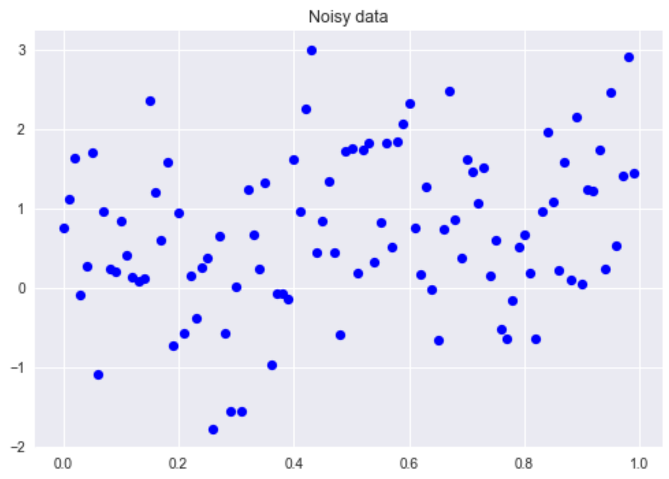
```
alpha = random.uniform(0, 1)
beta = random.uniform(0, 1)
```



```
noisy_dt= list()
def noisy(al, bt, elements):
    for i in range(0, elements,1):
        x=i/100
        y=al*x+bt+np.random.normal(0,1)
        noisy_dt.append([x,y])

noisy(alpha, beta, 100)
```

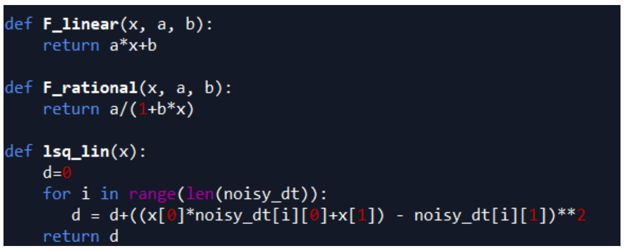
In our case α= 0.84809 and β= 0.33604.

Below you can scatter plot of our noisy data.



Linear Function

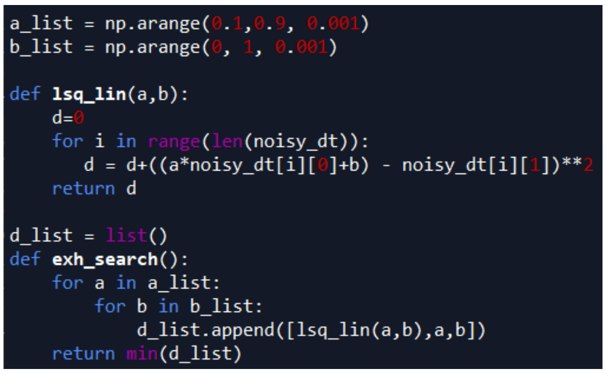
Below you can see code for functions minimization



```
def F_linear(x, a, b):
    return a*x+b

def F_rational(x, a, b):
    return a/(1+b*x)

def lsq_lin(x):
    d=0
    for i in range(len(noisy_dt)):
       d = d+((x[0]*noisy_dt[i][0]+x[1]) - noisy_dt[i][1])**2
    return d
```

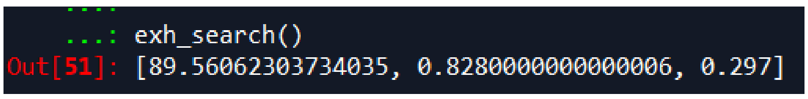
Exhaustive search method

Now we have applied Exhaustive search method to minimize of function.



```
a_list = np.arange(0.1,0.9, 0.001)
b_list = np.arange(0, 1, 0.001)

def lsq_lin(a,b):
    d=0
    for i in range(len(noisy_dt)):
       d = d+((a*noisy_dt[i][0]+b) - noisy_dt[i][1])**2
    return d

d_list = list()
def exh_search():
    for a in a_list:
        for b in b_list:
            d_list.append([lsq_lin(a,b),a,b])
    return min(d_list)
```

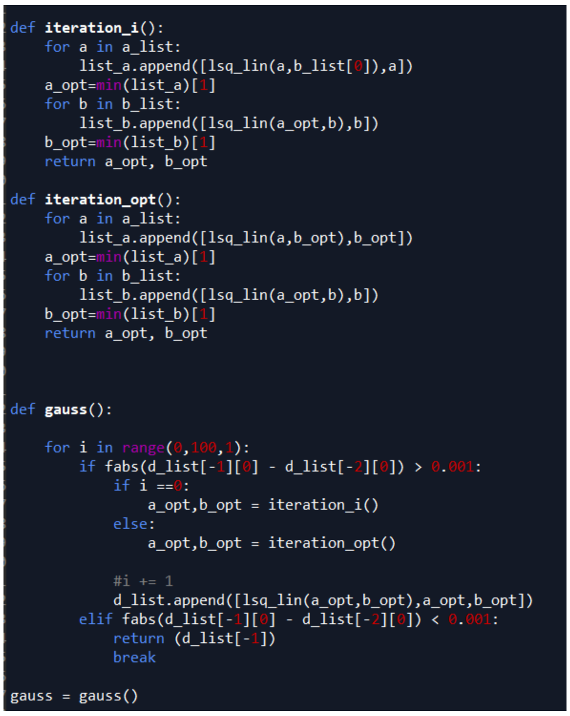
For Exhaustive search method function minimum is 89.56 and a=0.82, b=0.297



```
     ...: exh_search()
Out[51]: [89.56062303734035, 0.8280000000000006, 0.297]
```

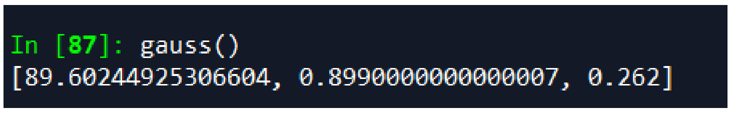
Next, we have applied Gauss method for function minimization.

Gauss method



```
def iteration_i():
    for a in a_list:
        list_a.append([lsq_lin(a,b_list[0]),a])
    a_opt=min(list_a)[1]
    for b in b_list:
        list_b.append([lsq_lin(a_opt,b),b])
    b_opt=min(list_b)[1]
    return a_opt, b_opt

def iteration_opt():
    for a in a_list:
        list_a.append([lsq_lin(a,b_opt),b_opt])
    a_opt=min(list_a)[1]
    for b in b_list:
        list_b.append([lsq_lin(a_opt,b),b])
    b_opt=min(list_b)[1]
    return a_opt, b_opt



def gauss():

    for i in range(0,100,1):
        if fabs(d_list[-1][0] - d_list[-2][0]) > 0.001:
            if i ==0:
                a_opt,b_opt = iteration_i()
            else:
                a_opt,b_opt = iteration_opt()

            #i += 1
            d_list.append([lsq_lin(a_opt,b_opt),a_opt,b_opt])
        elif fabs(d_list[-1][0] - d_list[-2][0]) < 0.001:
            return (d_list[-1])
            break

gauss = gauss()
```

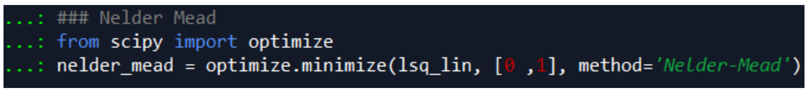
Results for gauss method is function minimum is 89.6 and a=0.89, b=0.262



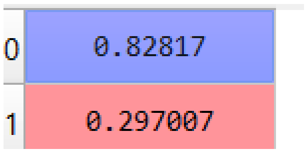
```
In [87]: gauss()
[89.60244925306604, 0.8990000000000007, 0.262]
```

Nelder-Mead

For Nelder-Mead results of optimization are following: function minimum is 89.54526 and a=0.82817, b=0.297007

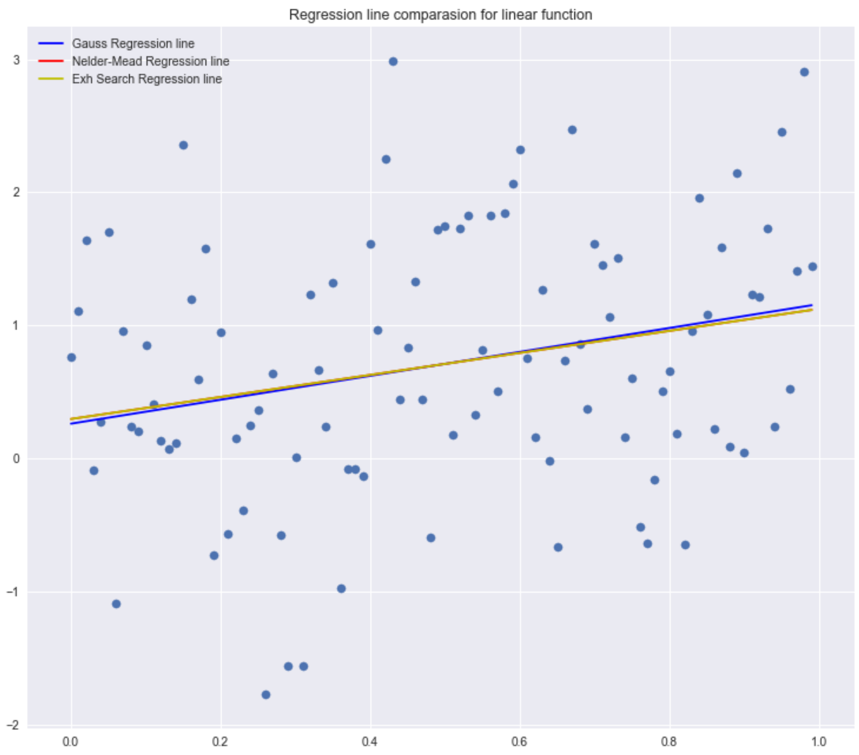


```
...: ### Nelder Mead
...: from scipy import optimize
...: nelder_mead = optimize.minimize(lsq_lin, [0 ,1], method='Nelder-Mead')
```

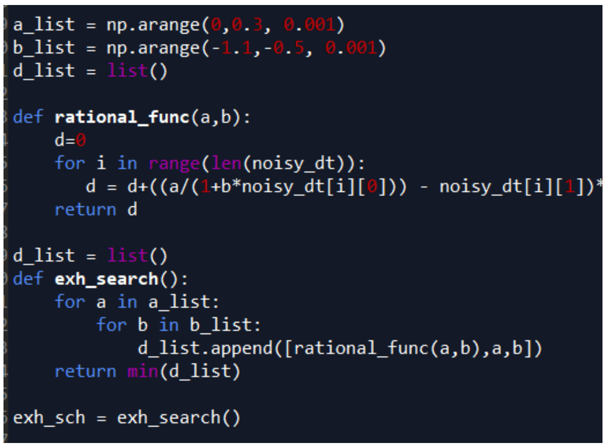


0	0.82817
1	0.297007

Comparison of different optimization methods for linear function



Rational function

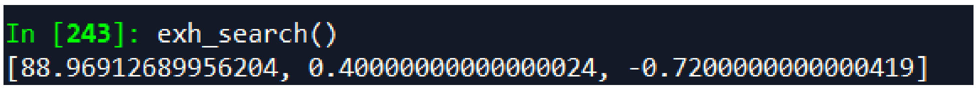


```
a_list = np.arange(0,0.3, 0.001)
b_list = np.arange(-1.1,-0.5, 0.001)
d_list = list()

def rational_func(a,b):
    d=0
    for i in range(len(noisy_dt)):
       d = d+((a/(1+b*noisy_dt[i][0])) - noisy_dt[i][1])*
    return d

d_list = list()
def exh_search():
    for a in a_list:
        for b in b_list:
            d_list.append([rational_func(a,b),a,b])
    return min(d_list)

exh_sch = exh_search()
```

Exhausting search

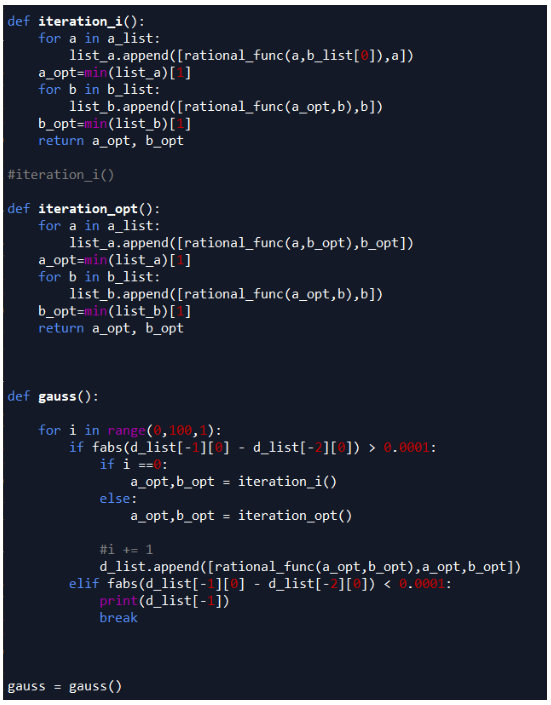
Now we have applied Exhaustive search method to minimize of rational function.

Results of minimization are function value = 88.9691, a = 0.4, b= -0.72



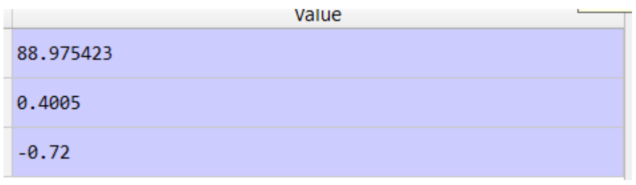
```
In [243]: exh_search()
[88.96912689956204, 0.40000000000000024, -0.7200000000000419]
```

Gauss method



```
def iteration_i():
    for a in a_list:
        list_a.append([rational_func(a,b_list[0]),a])
    a_opt=min(list_a)[1]
    for b in b_list:
        list_b.append([rational_func(a_opt,b),b])
    b_opt=min(list_b)[1]
    return a_opt, b_opt

#iteration_i()

def iteration_opt():
    for a in a_list:
        list_a.append([rational_func(a,b_opt),b_opt])
    a_opt=min(list_a)[1]
    for b in b_list:
        list_b.append([rational_func(a_opt,b),b])
    b_opt=min(list_b)[1]
    return a_opt, b_opt



def gauss():

    for i in range(0,100,1):
        if fabs(d_list[-1][0] - d_list[-2][0]) > 0.0001:
            if i ==0:
                a_opt,b_opt = iteration_i()
            else:
                a_opt,b_opt = iteration_opt()

            #i += 1
            d_list.append([rational_func(a_opt,b_opt),a_opt,b_opt])
        elif fabs(d_list[-1][0] - d_list[-2][0]) < 0.0001:
            print(d_list[-1])
            break



gauss = gauss()
```

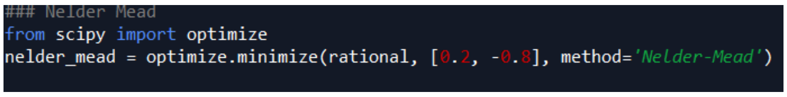
For gauss method function value is 88.975423 and a=0.4005, b=-0.72



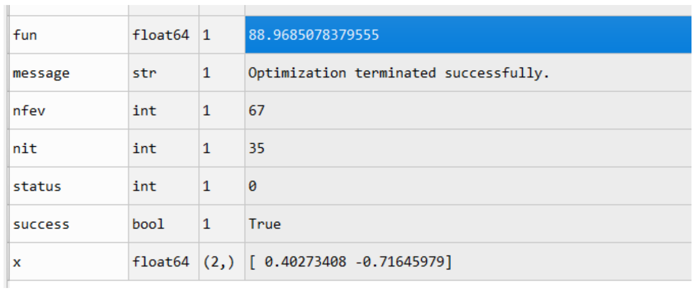
Value
88.975423
0.4005
-0.72

Nelder-Mead

And for Nelder-Mead function value is 88.9685 and a=0.40273408, b=0.71645979.

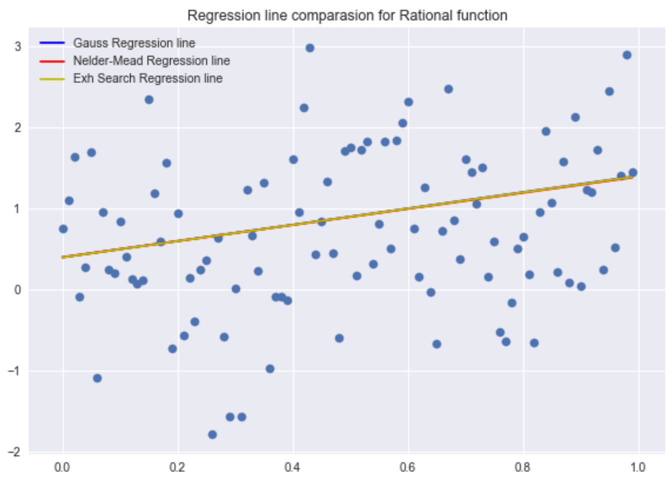


```
### Nelder Mead
from scipy import optimize
nelder_mead = optimize.minimize(rational, [0.2, -0.8], method='Nelder-Mead')
```



fun	float64	1	88.9685078379555
message	str	1	Optimization terminated successfully.
nfev	int	1	67
nit	int	1	35
status	int	1	0
success	bool	1	True
x	float64	(2,)	[0.40273408 -0.71645979]

Comparison of different optimization methods for rational function



Results and conclusion.

As can be seen from results above all method have good result in matter of function minimization, because value of function and A and B parameters are pretty close as for linear function as for rational. However, most important difference between all these methods are time and resources consuming. For example, Exhausting Search method consumes huge amount of time and resources. That is why I do not recommend to use this method of function minimization in case when different methods can be implied.