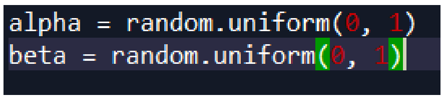
Task 3. Algorithms for unconstrained nonlinear optimization. First- and second-order methods

Egor Turukhanov

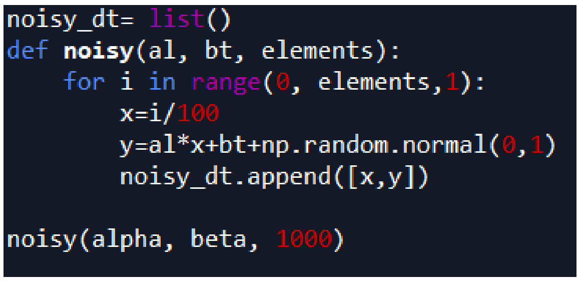
Group C4134

22.10.19

Data generation



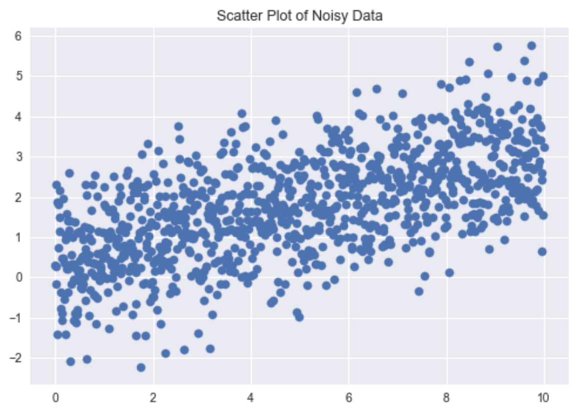
```
alpha = random.uniform(0, 1)
beta = random.uniform(0, 1)
```



```
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, 1000)
```

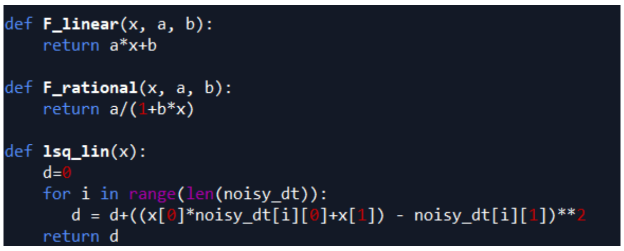
In our case α= 0.26479 and β= 0.45638.

Below you can scatter plot of our noisy data.



Linear Function

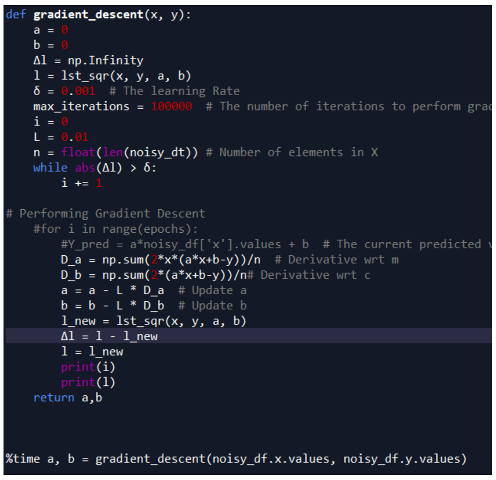
Below you can see code for functions minimization



```
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
```

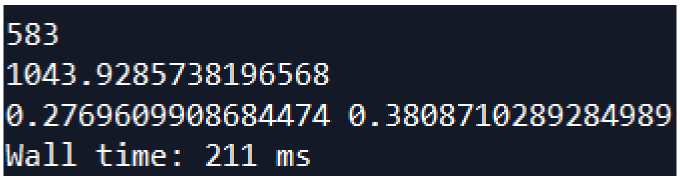
Gradient Descent

Now we applied Gradient Descent method to minimize our linear function.



```
def gradient_descent(x, y):
    a = 0
    b = 0
    Δl = np.Infinity
    l = lst_sqr(x, y, a, b)
    δ = 0.001  # The learning Rate
    max_iterations = 100000  # The number of iterations to perform grac
    i = 0
    L = 0.01
    n = float(len(noisy_dt)) # Number of elements in X
    while abs(Δl) > δ:
        i += 1

# Performing Gradient Descent
    #for i in range(epochs):
        #Y_pred = a*noisy_df['x'].values + b  # The current predicted v
        D_a = np.sum(2*x*(a*x+b-y))/n  # Derivative wrt m
        D_b = np.sum(2*(a*x+b-y))/n# Derivative wrt c
        a = a - L * D_a  # Update a
        b = b - L * D_b  # Update b
        l_new = lst_sqr(x, y, a, b)
        Δl = l - l_new
        l = l_new
        print(i)
        print(l)
    return a,b



%time a, b = gradient_descent(noisy_df.x.values, noisy_df.y.values)
```

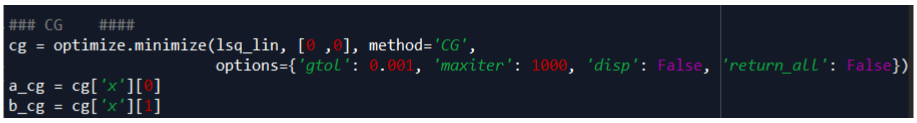
For Gradient Descent method a=0.27696, b=0.38087. Iterations = 583 and time = 211ms.



```
583
1043.9285738196568
0.2769609908684474 0.3808710289284989
Wall time: 211 ms
```

Next, we have applied Conjugate Gradient Descent method for linear function minimization.

Conjugate Gradient Descent method

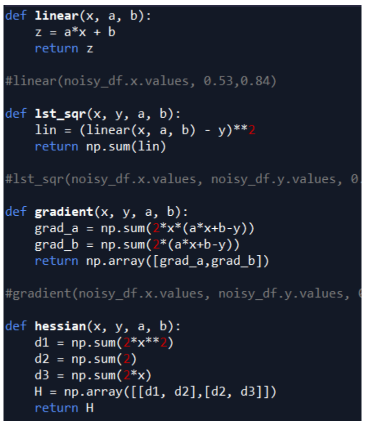


```
### CG    ####
cg = optimize.minimize(lsq_lin, [0 ,0], method='CG',
                       options={'gtol': 0.001, 'maxiter': 1000, 'disp': False, 'return_all': False})
a_cg = cg['x'][0]
b_cg = cg['x'][1]
```

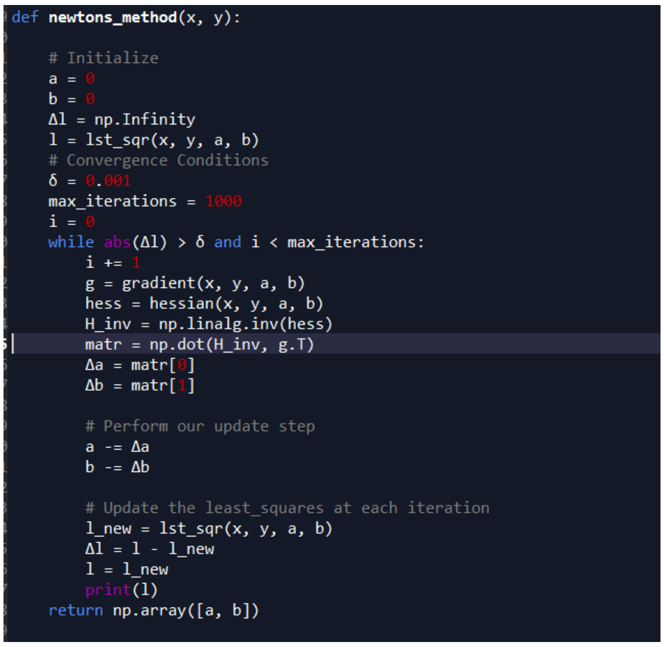
For Conjugate Gradient Descent method, a=0.2739, b=0.40. Iterations = 42. Time is 67ms.

Newton Method.

Code for elements of Newton method and Newton Method itself is below.



```python
def linear(x, a, b):
    z = a*x + b
    return z

#linear(noisy_df.x.values, 0.53,0.84)

def lst_sqr(x, y, a, b):
    lin = (linear(x, a, b) - y)**2
    return np.sum(lin)

#lst_sqr(noisy_df.x.values, noisy_df.y.values, 0

def gradient(x, y, a, b):
    grad_a = np.sum(2*x*(a*x+b-y))
    grad_b = np.sum(2*(a*x+b-y))
    return np.array([grad_a,grad_b])

#gradient(noisy_df.x.values, noisy_df.y.values,

def hessian(x, y, a, b):
    d1 = np.sum(2*x**2)
    d2 = np.sum(2)
    d3 = np.sum(2*x)
    H = np.array([[d1, d2],[d2, d3]])
    return H
```

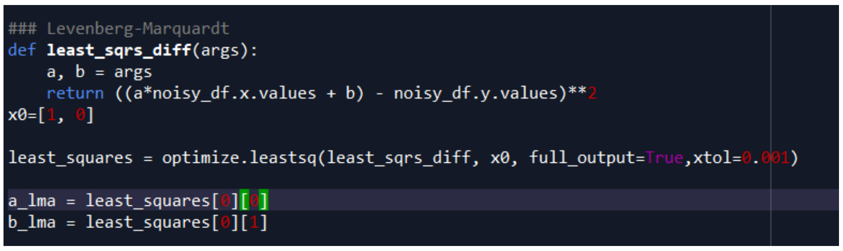


```python
def newtons_method(x, y):

    # Initialize
    a = 0
    b = 0
    Δl = np.Infinity
    l = lst_sqr(x, y, a, b)
    # Convergence Conditions
    δ = 0.001
    max_iterations = 1000
    i = 0
    while abs(Δl) > δ and i < max_iterations:
        i += 1
        g = gradient(x, y, a, b)
        hess = hessian(x, y, a, b)
        H_inv = np.linalg.inv(hess)
        matr = np.dot(H_inv, g.T)
        Δa = matr[0]
        Δb = matr[1]

        # Perform our update step
        a -= Δa
        b -= Δb

        # Update the least_squares at each iteration
        l_new = lst_sqr(x, y, a, b)
        Δl = l - l_new
        l = l_new
        print(l)
    return np.array([a, b])
```

For Newton Method a=0.2749 b=0.3946. Iterations = 62. Time is 33.9ms.

Levenberg-Marquardt method

For Levenberg-Marquardt method a=0.2749 b=0.315. Iterations = 34. Time is 2ms.

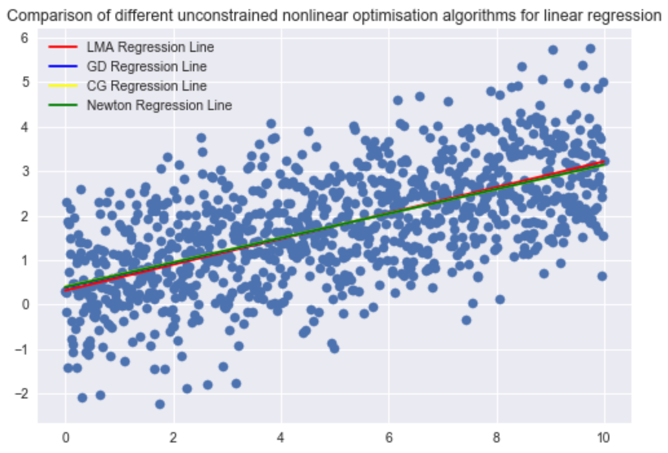


```python
### Levenberg-Marquardt
def least_sqrs_diff(args):
    a, b = args
    return ((a*noisy_df.x.values + b) - noisy_df.y.values)**2
x0=[1, 0]

least_squares = optimize.leastsq(least_sqrs_diff, x0, full_output=True,xtol=0.001)

a_lma = least_squares[0][0]
b_lma = least_squares[0][1]
```

Comparison of different optimization methods for linear function

As can be seen from graph below all optimization methods regression lines are close to each other.

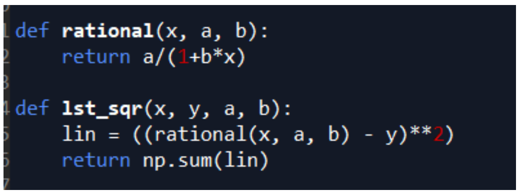
Method	Time, ms	Number of iterations
Gradient Descent	211	583
Conjugate Gradient Descent	67	42
Newton	33	62
Levenberg-Marquardt	2	34



Rational function

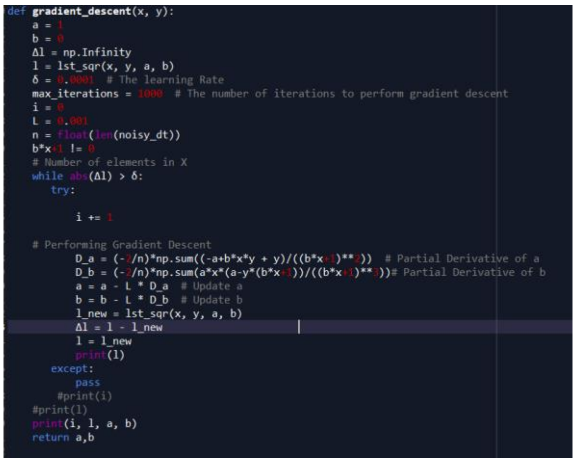
Below you can see code for functions minimization

Gradient Descent



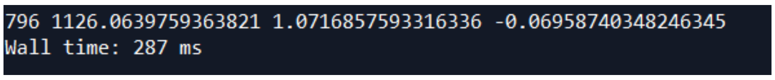
```
def rational(x, a, b):
    return a/(1+b*x)

def lst_sqr(x, y, a, b):
    lin = ((rational(x, a, b) - y)**2)
    return np.sum(lin)
```

Now we applied Gradient Descent method to minimize our linear function.



```
def gradient_descent(x, y):
    a = 1
    b = 0
    Δl = np.Infinity
    l = lst_sqr(x, y, a, b)
    δ = 0.0001  # The learning Rate
    max_iterations = 1000  # The number of iterations to perform gradient descent
    i = 0
    L = 0.001
    n = float(len(noisy_dt))
    b*x+1 != 0
    # Number of elements in X
    while abs(Δl) > δ:
       try:

           i += 1

    # Performing Gradient Descent
           D_a = (-2/n)*np.sum((-a+b*x*y + y)/((b*x+1)**2))  # Partial Derivative of a
           D_b = (-2/n)*np.sum(a*x*(a-y*(b*x+1))/((b*x+1)**3))# Partial Derivative of b
           a = a - L * D_a  # Update a
           b = b - L * D_b  # Update b
           l_new = lst_sqr(x, y, a, b)
           Δl = l - l_new
           l = l_new
           print(l)
       except:
           pass
        #print(i)
    #print(l)
    print(i, l, a, b)
    return a,b
```

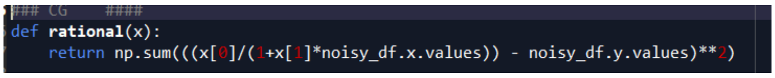
For Gradient Descent method a=1.0717, b=-0.0696. Iterations = 796 and time = 287ms.



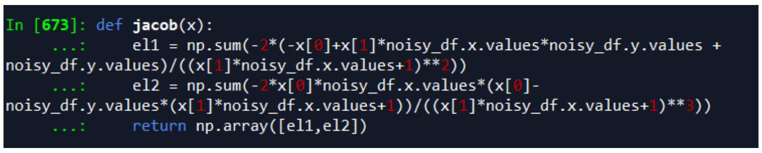
```
796 1126.0639759363821 1.0716857593316336 -0.06958740348246345
Wall time: 287 ms
```

Next, we have applied Conjugate Gradient Descent method for linear function minimization.

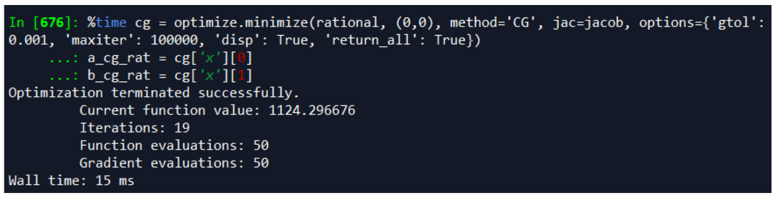
Conjugate Gradient Descent method



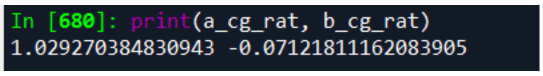
```
#### CG    ####
def rational(x):
    return np.sum(((x[0]/(1+x[1]*noisy_df.x.values)) - noisy_df.y.values)**2)
```



```
In [673]: def jacob(x):
     ...:     el1 = np.sum(-2*(-x[0]+x[1]*noisy_df.x.values*noisy_df.y.values +
noisy_df.y.values)/((x[1]*noisy_df.x.values+1)**2))
     ...:     el2 = np.sum(-2*x[0]*noisy_df.x.values*(x[0]-
noisy_df.y.values*(x[1]*noisy_df.x.values+1))/((x[1]*noisy_df.x.values+1)**3))
     ...:     return np.array([el1,el2])
```



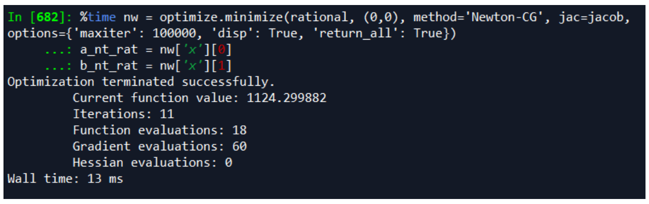
```
In [676]: %time cg = optimize.minimize(rational, (0,0), method='CG', jac=jacob, options={'gtol':
0.001, 'maxiter': 100000, 'disp': True, 'return_all': True})
     ...: a_cg_rat = cg['x'][0]
     ...: b_cg_rat = cg['x'][1]
Optimization terminated successfully.
         Current function value: 1124.296676
         Iterations: 19
         Function evaluations: 50
         Gradient evaluations: 50
Wall time: 15 ms
```



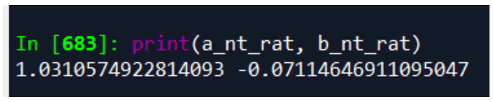
```
In [680]: print(a_cg_rat, b_cg_rat)
1.029270384830943 -0.07121811162083905
```

For Conjugate Gradient Descent method, a=1.02927, b=-0.0712. Function evaluations = 50. Time is 15ms.

Newton Method.



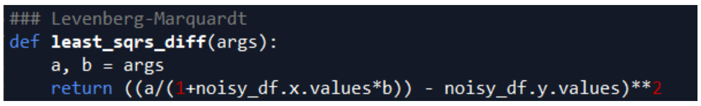
```
In [682]: %time nw = optimize.minimize(rational, (0,0), method='Newton-CG', jac=jacob,
options={'maxiter': 100000, 'disp': True, 'return_all': True})
     ...: a_nt_rat = nw['x'][0]
     ...: b_nt_rat = nw['x'][1]
Optimization terminated successfully.
         Current function value: 1124.299882
         Iterations: 11
         Function evaluations: 18
         Gradient evaluations: 60
         Hessian evaluations: 0
Wall time: 13 ms
```



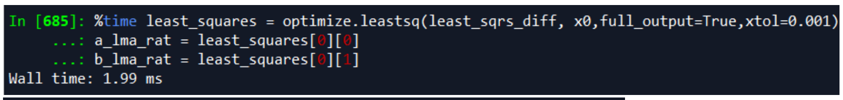
```
In [683]: print(a_nt_rat, b_nt_rat)
1.0310574922814093 -0.07114646911095047
```

For Newton Method a=1.0311 b=-0.07114. Iterations = 11. Function eval = 18. Time is 13ms.

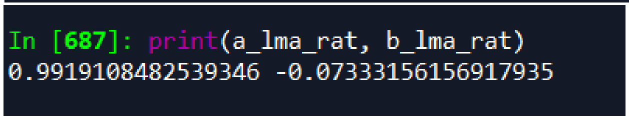
Levenberg-Marquardt method



```
### Levenberg-Marquardt
def least_sqrs_diff(args):
    a, b = args
    return ((a/(1+noisy_df.x.values*b)) - noisy_df.y.values)**2
```



```
In [685]: %time least_squares = optimize.leastsq(least_sqrs_diff, x0,full_output=True,xtol=0.001)
     ...: a_lma_rat = least_squares[0][0]
     ...: b_lma_rat = least_squares[0][1]
Wall time: 1.99 ms
```



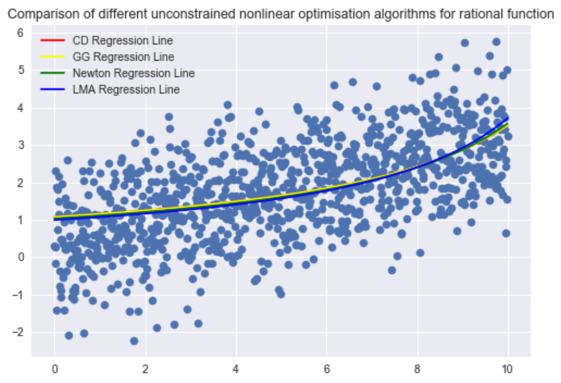
```
In [687]: print(a_lma_rat, b_lma_rat)
0.9919108482539346 -0.07333156156917935
```

For Levenberg-Marquardt method a=0.991 b=-0.0733. Iterations = 48. Time is 1.99ms.

Comparison of different optimization methods for rational function

As can be seen from graph below all optimization methods regression lines are close to each other. Gradient Descent is the most time-consuming algorithm in our case. Levenberg-Marquardt is the fastest optimization algorithms as well as for linear function.

Method	Time, ms	Number of iterations
Gradient Descent	287	796
Conjugate Gradient Descent	12	19
Newton	13	11
Levenberg-Marquardt	1.99	48



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 close as for linear function as for rational. However, most important difference between all these methods are time and resources consuming. For example, Gradient Descent method consumes huge amount of time and resources. That is why I recommend this method of function minimization in case when it's variations (SGD, CGD) can be applied. For least squares minimization problem Levenberg-Marquardt method shows the best performance in terms of time, resources and minimization results.