FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report

on the practical task No. 3

"***Algorithms for unconstrained nonlinear optimization. First- and second-order methods***"

Performed by
Abizer Safdari
J4134c
Accepted by
Dr Petr Chunaev

St. Petersburg
2020

Goal

The use of first- and second-order methods (Gradient Descent, Conjugate Gradient Descent, Newton's method and Levenberg-Marquardt algorithm) in the tasks of unconstrained nonlinear optimization

Formulation of the problem

Generate random numbers $\alpha \in (0,1)$ *and* $\beta \in (0,1)$. *Furthermore, generate the noisy data* $\{x_k, y_k\}$, *where* $k = 0, \dots, 100$, *according to the following rule:*

$$y_k = \alpha x_k + \beta + \delta_k, \quad x_k = \frac{k}{100},$$

where $\delta_k \sim N(0,1)$ *are values of a random variable with standard normal distribution. Approximate the data by the following linear and rational functions:*

1. $F(x, a, b) = ax + b$ *(linear approximant),*
2. $F(x, a, b) = \frac{a}{1+bx}$ *(rational approximant),*

by means of least squares through the numerical minimization (with precision $\varepsilon = 0.001$*) of the following function:*

$$D(a, b) = \sum_{k=0}^{100} (F(x_k, a, b) - y_k)^2.$$

To solve the minimization problem, use the methods of Gradient Descent, Conjugate Gradient Descent, Newton's method and Levenberg-Marquardt algorithm. If necessary, set the initial approximations and other parameters of the methods. Visualize the data and the approximants obtained ***separately for each type of approximant****. Analyze the results obtained (in terms of number of iterations, precision, number of function evaluations, etc.) and compare them with those from Task 2 for the same dataset*

Brief theoretical part

- **Gradient descent** is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. To find a local minimum of a function using gradient descent, we take steps proportional to the *negative* of the gradient (or approximate gradient) of the function at the current point. But if we instead take steps proportional to the *positive* of the gradient, we approach a local maximum of that function; the procedure is then known as **gradient ascent**.
- The **conjugate gradient method** is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite. The conjugate gradient method is often implemented as an iterative algorithm, applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition. Large sparse systems often arise when numerically solving partial differential equations or optimization problems.
- The gradient descent method may not be efficient because it could get into the zigzag pattern and repeat the same search directions many times.
- **Newton's method** is an iterative method for finding the roots of a differentiable function *F*, which are solutions to the equation $F(x) = 0$. In optimization, Newton's method is applied to the derivative f' of a twice-differentiable function f to find the roots of the derivative (solutions to $f'(x) = 0$), also known as the stationary points of f. These solutions may be minima, maxima, or saddle points.
- The Levenberg-Marquardt algorithm, also known as the damped least-squares method, has been designed to work specifically with loss functions, which take the form of a sum of squared errors. It works without computing the exact Hessian matrix. Instead, it works with the gradient vector and the Jacobian matrix.

Results

Optimization terminated successfully.
Current function value: 115.091197
Iterations: 2
Function evaluations: 5
Gradient evaluations: 5
Optimization terminated successfully.
Current function value: 119.344376
Iterations: 9
Function evaluations: 35
Gradient evaluations: 35
Optimization terminated successfully.
Current function value: 115.091197
Iterations: 3
Function evaluations: 3
Gradient evaluations: 3
Hessian evaluations: 3
Optimization terminated successfully.
Current function value: 119.394586
Iterations: 8
Function evaluations: 28
Gradient evaluations: 28
Hessian evaluations: 8

Approximation	Method	Function	Least square error	Iteration	f-function
Linear	Gradient descent	$ax + b$	*115.091197*	2	5
	Conjugate gradient	$ax + b$	*115.091197*	2	5
	Newtons	$ax + b$	*115.091197*	3	3
	Levenberg-Marquardt	$ax + b$	*115.091197*	3	3
Rational	Gradient descent	$\frac{a}{1 + bx}$	*119.344376*	9	35
	Conjugate gradient	$\frac{a}{1 + bx}$	*119.344376*	9	35
	Newtons	$\frac{a}{1 + bx}$	*119.394586*	8	28
	Levenberg-Marquardt	$\frac{a}{1 + bx}$	*119.394586*	8	8

Table 1: Number of iteration and approximation functions using first and second order methods.

The gradient descent method is faster method, because it solve equations in less amount of time. Conjugate gradient method is slower, but more productive,

because, it converges after less iterations. So, we can see, that one method can be used, when we want to find solution very fast and another can be converge to maximum in less iteration.

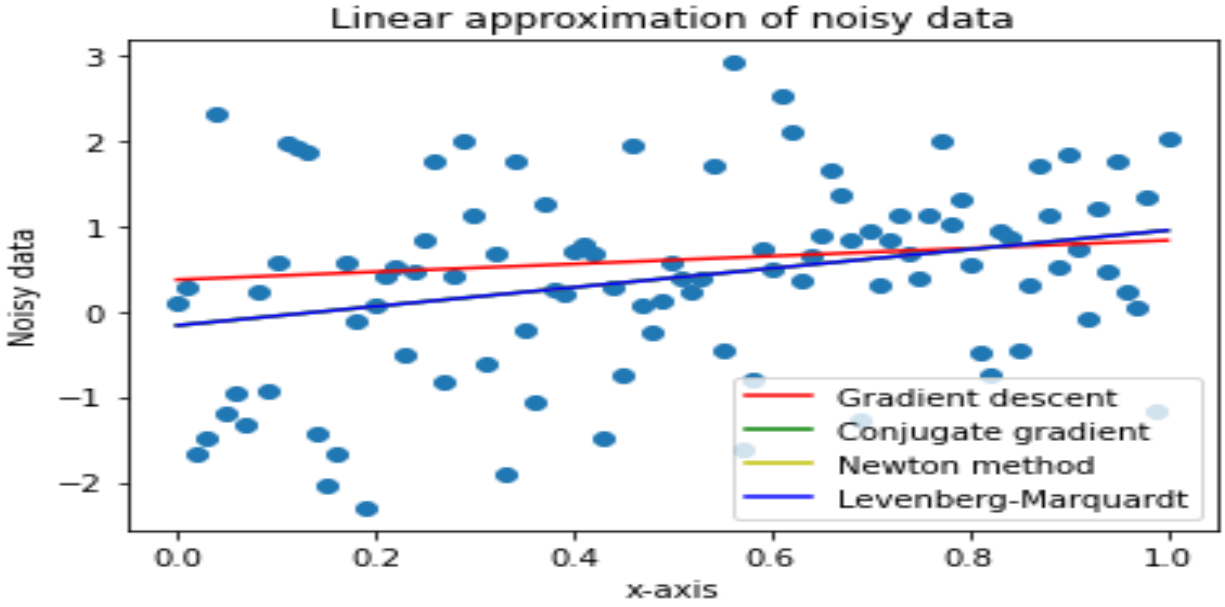

Figure 1: The linear approximation of the noisy data using Gradient descent, Conjugate gradient descent, Newton's method and Levenberg-Marquardt algorithm.

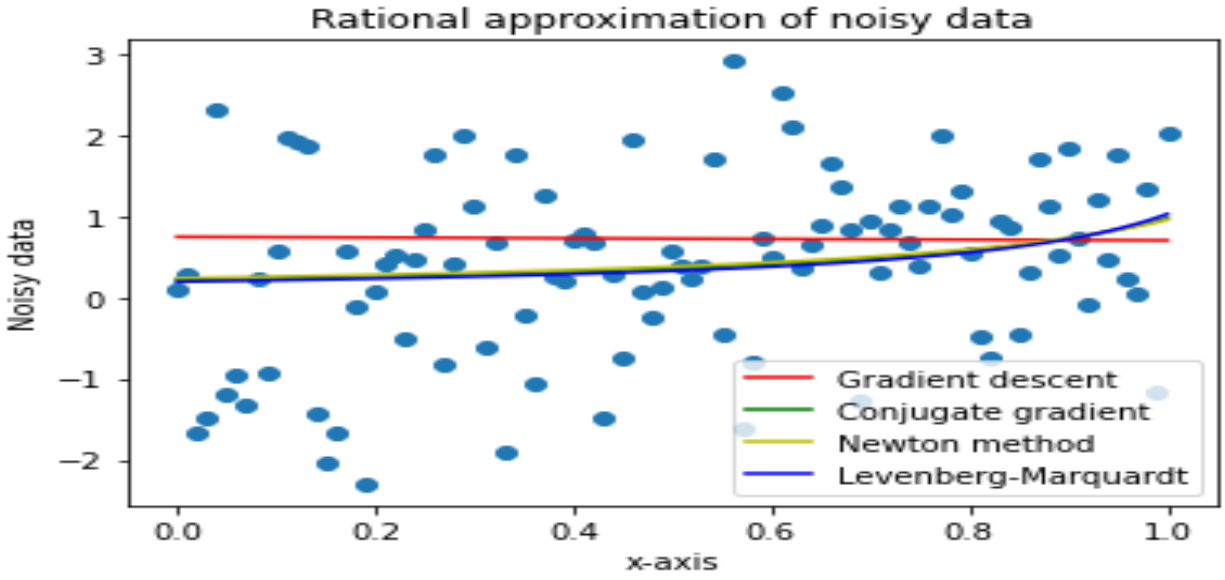

Figure 2: The rational approximation of the noisy data using Gradient descent, Conjugate gradient descent, Newton's method and Levenberg-Marquardt algorithm.

Conclusions

- If we many thousands of parameters, we can use gradient descent or conjugate gradient, to save memory.
- If we have just a few thousands of instances and a few hundreds of parameters, the best choice might be the Levenberg-Marquardt algorithm.
- as the speed of the algorithm increases, the complexity and the space required increases due to huge computations.

Appendix

```
"""
Created on Fri Oct 14 17:33:46 2020

@author: abizer
"""


import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt
# generating noisy data
alpha=np.random.rand()
beta=np.random.rand()
delta=[]
for i in range(101):
    delta.append(np.random.normal(0,1))
xk=np.arange(0, 1.01, 0.01)
yk=alpha*xk+beta+delta

# defining square error for linear approximation
def DL(x):
    err=0
    a=x[0]
    b=x[1]
    #xk, yk=args
    for i in range(101):
        err+=((a*xk[i]+b) -yk[i])**2
    return err

# defining square error for rational approximation
def DR(x):
    err = 0
    a=x[0]
    b=x[1]
    for i in range(101):
        err += ((a/ (xk[i] * b+1) )- yk[i]) ** 2
    return err

# gradient for linear square errors function
def gradDL(x):
    a=x[0]
    b=x[1]
    y=a*xk+b
    ga=2*sum(xk*(y-yk))
```

```
    gb=2*sum(y-yk)
    return np.asarray((ga, gb))

# gradient for rational square error function
def gradDR(x):
    a=x[0]
    b=x[1]
    y=a/(1+b*xk)
    ga=2*sum((y/a)*(y-yk))
    gb=2*sum((yk-y)*y*(xk*y/a))
    return np.asarray((ga, gb))

# hesian matrix for linear square errors function
def Hessian_L(x):
    a=x[0]
    b=x[1]
    y=a*xk+b
    gaa=2*sum(xk**2)
    gab=2*sum(xk)
    gba=2*sum(xk)
    gbb=2*101
    return np.asarray([[gaa, gab], [gba, gbb]])

# hesian matrix for rational square errors function
def Hessian_R(x):
    a=x[0]
    b=x[1]
    y=a/(xk*b+1)
    gaa=2*sum((y/a)**2)
    gab=2*sum((xk)*(-2*a+yk+b*xk*yk)/(1+b*xk)**3)
    gba=2*sum(((xk)*(b*xk*yk+yk-2*a))/(a+b*xk)**3)
    gbb=2*sum((2*a*(xk**2))*(-2*b*xk*yk-2*yk+3*a)/(1+b*xk)**4)
    return np.asarray([[gaa, gab], [gba, gbb]])

#defining linear approximation function
def FL(x):
    return x[0]*xk+x[1]

#defining rational approximation function
def FR(x):
    return x[0]/(1+x[1]*xk)

#defining residuals for linear approximation function
def residuel_L(x):
    return FL(x)-yk
```

```
#defining residuals for rational approximation function
def residuel_R(x):
    return FR(x)-yk

#defining jacobian matrix for linear approximation function
def jacobian_L(x):
    j=np.zeros((xk.size, x.size))
    j[:,0]=xk
    j[:,1]=1
    return j

#defining jacobian matrix for rational approximation function
def jacobian_R(x):
    j=np.empty((xk.size, x.size))
    j[:,0]=1/(1+x[1]*xk)
    j[:,1]=-FR(x)*xk/x[0]
    return j

# linear approximation using gradient descent
al,bl=0.5,0.5
gamma=0.00001
err=0.0001
pal, pbl=1,1
i=0
while pal>err and pbl>err:
    y=al*xk+bl
    a0l=al
    b0l=bl
    gal= gradDL([al,bl])[0]
    gbl= gradDL([al,bl])[1]
    al=al-gamma*gal
    bl=bl-gamma*gbl
    # precision=np.sqrt((al-a0)**2+(bl-b0)**2)
    pal= abs(al-a0l)
    pbl=abs(bl-b0l)
    i+=1
yl_g=FL([al,bl])

# Rational approximation using gradient descent
ar, br=1,0
gamma=0.00001
err=0.0001
# precision=1
par, pbr=1,1
```

```
j=0
while par>err and pbr> err:
    y=ar/(1+br*xk)
    a0r=ar
    b0r=br
    ga=gradDR([ar,br])[0]
    gb=gradDR([ar,br])[1]
    ar=ar-gamma*ga
    br=br-gamma*gb
    # precision=np.sqrt((ar-a0)**2+(br-b0)**2)
    par= abs(ar-a0r)
    pbr=abs(br-b0r)
    j+=1
yr_g=ar/(1+xk*br)

#initial approximation
x0=np.array([0.5,0.5])
#caluclating the linear approximation function using conjugate method
l_conjugate_gradient=optimize.fmin_cg(DL, x0, fprime=gradDL, gtol=0.001 )
yl_cg=FL(l_conjugate_gradient)
#caluclating the rational approximation function using conjugate gradient method
r_conjugate_gradient=optimize.fmin_cg(DR, x0, fprime=gradDR, gtol=0.001)
yr_cg=FR(r_conjugate_gradient)
#caluclating the linear approximation function using Newton's method
l_newton= optimize.minimize(DL, x0, jac=gradDL, method='Newton-CG', \
 hess=Hessian_L, tol=0.001,options={'disp':True})
yl_n=FL(l_newton.x)
#caluclating the rational approximation function using Newton's method
r_newton= optimize.minimize(DR, x0, jac=gradDR, method='Newton-CG',\
 hess=Hessian_R, tol=0.001,options={'disp':True})
yr_n=FR(r_newton.x)
#caluclating the linear approximation function using Levenberg-Marquardt
res_l=optimize.least_squares(residuel_L, x0, jac=jacobian_L,\
 method='lm',ftol=0.001 )
yl_lm=FL(res_l.x)
#caluclating the Rational approximation function using Levenberg-Marquardt
res_R=optimize.least_squares(residuel_R, x0, jac=jacobian_R,\
 method='lm',ftol=0.001 )
yr_lm=FR(res_R.x)
# plotting linear approximation of noisy data
plt.scatter(xk, yk)
plt.plot(xk, yl_g, '-r', label='Gradient descent')
plt.plot(xk, yl_cg, '-g', label='Conjugate gradient')
plt.plot(xk, yl_n, '-y', label='Newton method')
plt.plot(xk, yl_lm, '-b', label='Levenberg-Marquardt')
```

```
plt.xlabel('x-axis')
plt.ylabel('Noisy data')
plt.title('Linear approximation of noisy data')
plt.legend()
plt.show()
# plotting rational approximation of noisy data
plt.scatter(xk, yk)
plt.plot(xk, yr_g, '-r', label='Gradient descent')
plt.plot(xk, yr_cg, '-g', label='Conjugate gradient')
plt.plot(xk, yr_n, '-y', label='Newton method')
plt.plot(xk, yr_lm, '-b', label='Levenberg-Marquardt')
plt.xlabel('x-axis')
plt.ylabel('Noisy data')
plt.title('Rational approximation of noisy data')
plt.legend()
plt.show().
```