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

*Mariia Koroleva*

*J42322c*

Accepted by

Dr Petr Chunaev

St. Petersburg

2021

**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 and . Furthermore, generate the noisy data , where , according to the following rule:*

*,*

*where are values of a random variable with standard normal distribution. Approximate the data by the following linear and rational functions:*

1. *(linear approximant),*
2. *(rational approximant),*

*by means of least squares through the numerical minimization (with precision ) of the following function:*

*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**

In this paper, we consider such minimization methods as Gradient Descent, Conjugate Gradient Descent, Newton’s method and ﻿Levenberg-Marquardt algorithm. Here is their description:

*Gradient descent* is based on the observation that if f(x) is defined and differentiable in a neighborhood of a point a, then f(x) decreases fastest in a neighborhood of a in the direction of −∇af. One obtains the following formula:

an+1 = an − γ∇an f

for γ ∈ R+ small enough, then f(an) ≥ f(an+1). With this observation in mind, one starts with a guess a0 for a local minimum of f and considers the sequence {an} such that an+1 = an − γn∇an f, n ≥ 0. Here the value of the step size γn may be non-fixed and changed at every iteration (many possible ways to choose).

*Conjugate Gradient Descent.*

Given a function f(x), x ∈ R n and an initial approximation a0, one starts in the steepest descent direction: ∆a0 = −∇a0 f.

Find the step length α0:= arg minα f (a0 + α∆a0) and the next point a1 = a0 + α0∆a0. After this iteration, the following steps constitute one iteration of moving along a subsequent conjugate direction sn, where s0 = ∆a0:

* Calculate the steepest direction ∆an = −∇an f.
* Compute βn according to certain formulas.
* Update the conjugate direction sn = ∆an + βnsn−1.
* Find αn = arg minα f(an + αsn).
* Update the position: an+1 = an + αnsn

The choice of βn due to Fletcher-Reeves:

Изображение выглядит как текст

Автоматически созданное описание

The choice of βn due to Polak-Ribiere:

Изображение выглядит как текст

Автоматически созданное описание

*Newton’s method.*

Let f : Rn → R be convex and Hxf is invertible for x ∈ Rn . The one-dimensional scheme can be generalized to several dimensions by replacing the derivative with the gradient, ∇f, and the reciprocal of the second derivative with the inverse of the Hessian matrix, Hf: an+1 = an − [Han f] −1∇an f, n ≥ 0.

﻿*Levenberg-Marquardt algorithm*

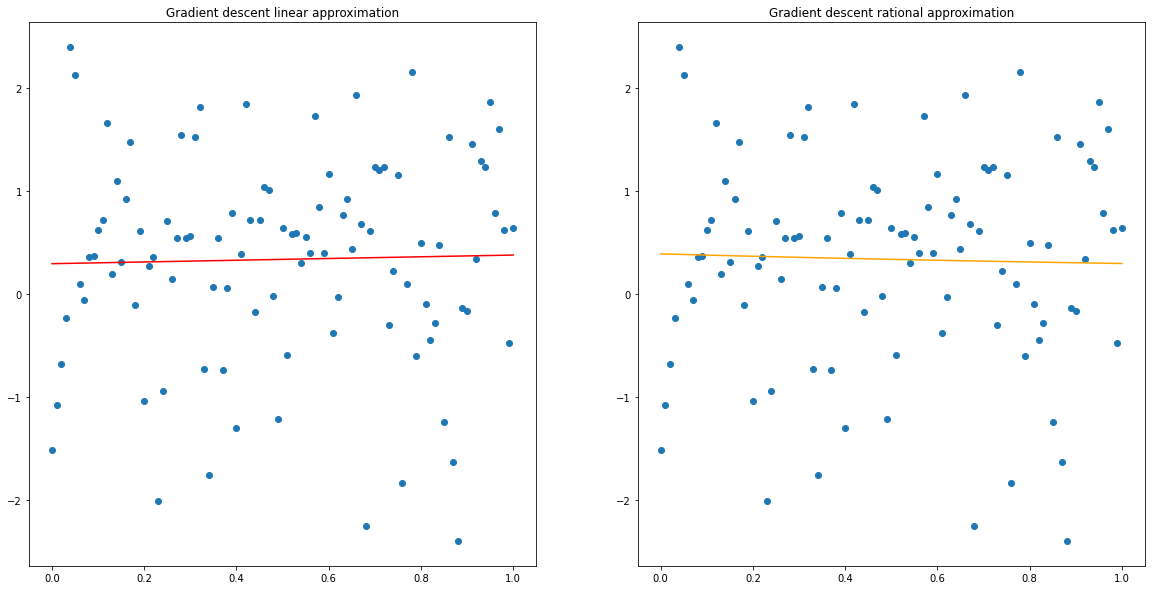
The application of LMA is the least-squares curve fitting problem: given a set (xi, yi) m i=1, find the parameters β (column vector) of the model curve f(x, β) so that the sum of the squares of the deviations S(β) is minimized: Xm i=1.

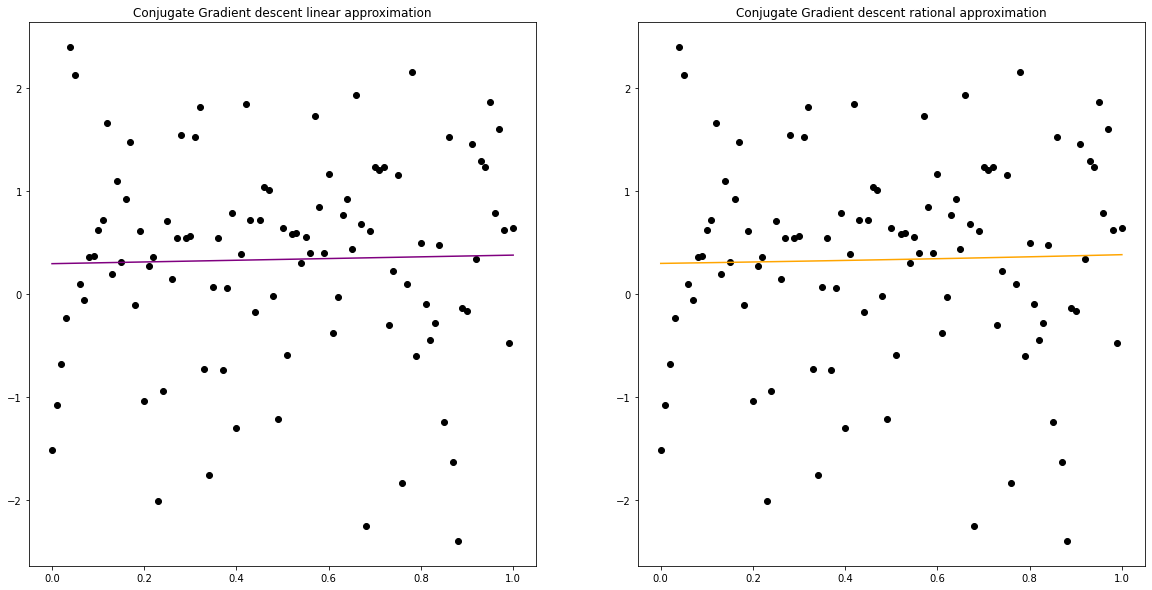
arg min β S(β) ≡ arg min β

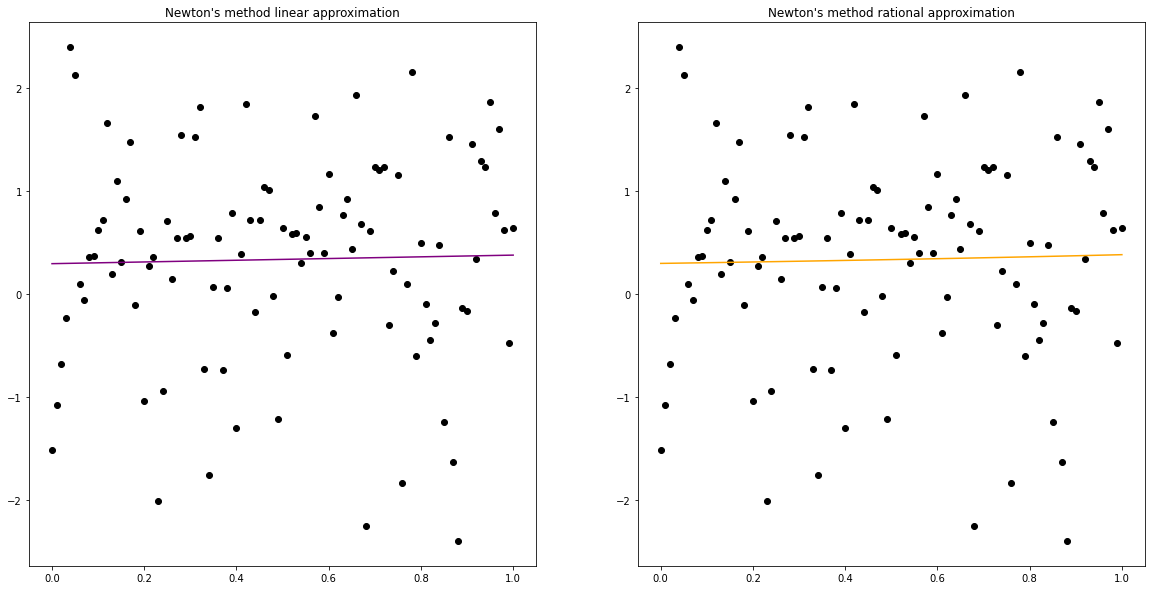
**Results**

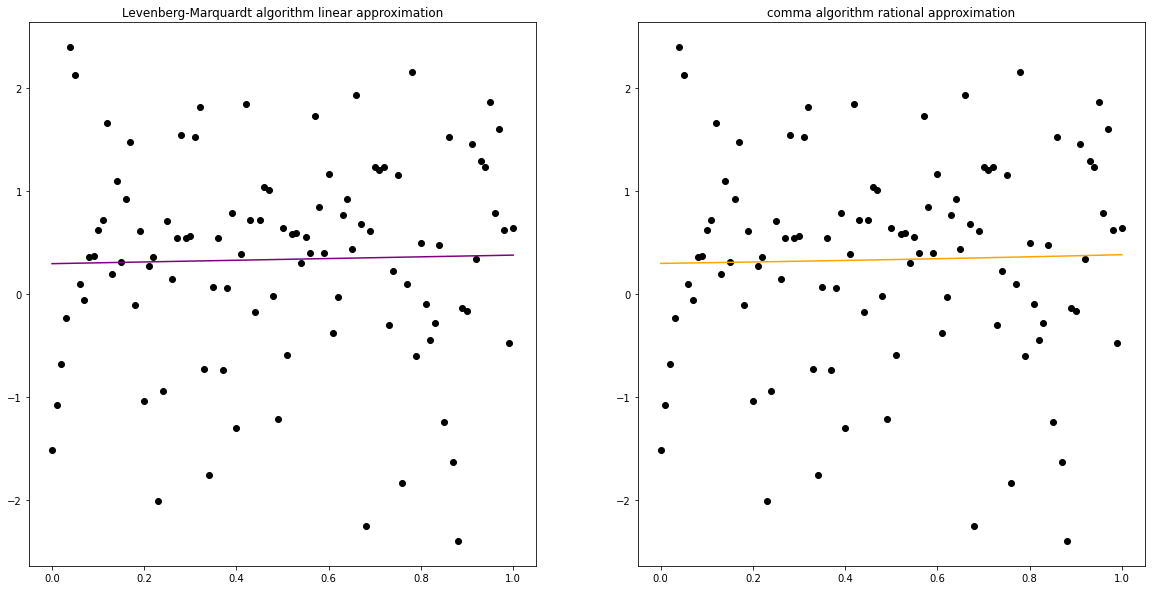
|  |  |  |  |
| --- | --- | --- | --- |
| **Results of Linear Approximation** | | | |
| **Algorithm name** | **Number of iterations** | **Function min** | **Number of function calculations** |
| **Gradient descent** | 8 | 101,405 | 6 |
| **Conjugate gradient descent** | 2 | 101,404 | 24 |
| **Newton's method** | 3 | 101,404 | 46 |
| **Levenberg-Marquardt algorithm** | - | 101,404 | 7 |
| **\*Exhaustive search** | 4004002 | 101,404 | 4004001 |
| **\*Gauss method** | 30 | 101,405 | 30 |
| **\*Nelder-Mead** | 24 | 101,404 | 47 |

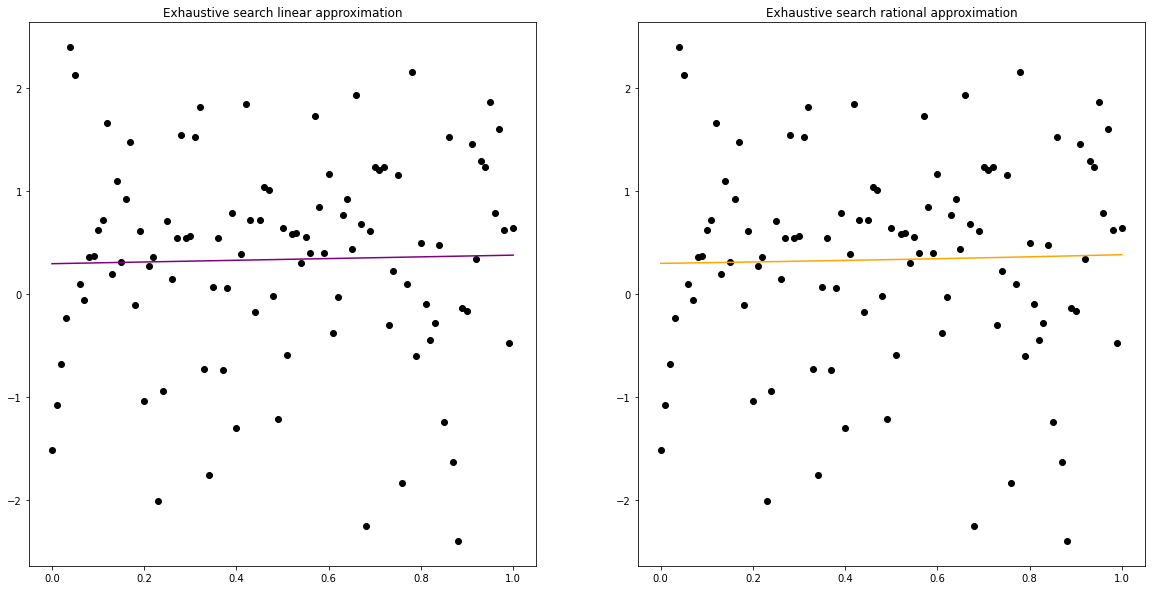
|  |  |  |  |
| --- | --- | --- | --- |
| **Results of Rational Approximation** | | | |
| **Algorithm name** | **Number of iterations** | **Function min** | **Number of function calculations** |
| **Gradient descent** | 10 | 101,677 | 8 |
| **Conjugate gradient descent** | 8 | 101,404 | 72 |
| **Newton's method** | 8 | 101,404 | 13 |
| **Levenberg-Marquardt algorithm** | - | 118,747 | 20 |
| **\*Exhaustive search** | 4004002 | 101,404 | 4004001 |
| **\*Gauss method** | 253 | 115,752 | 253 |
| **\*Nelder-Mead** | 53 | 101,404 | 102 |

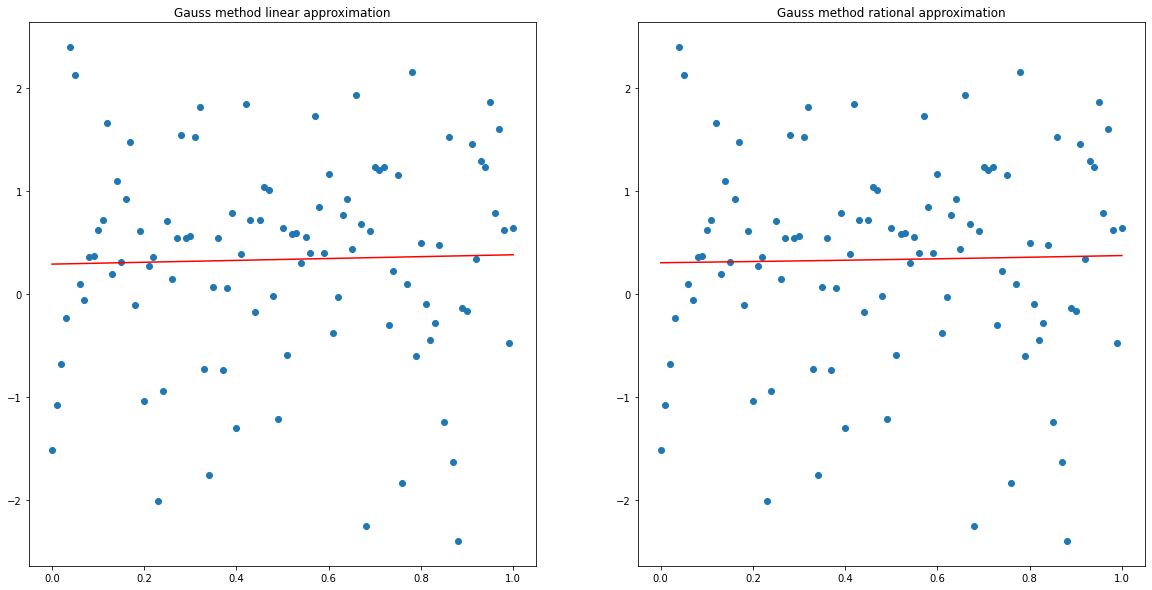


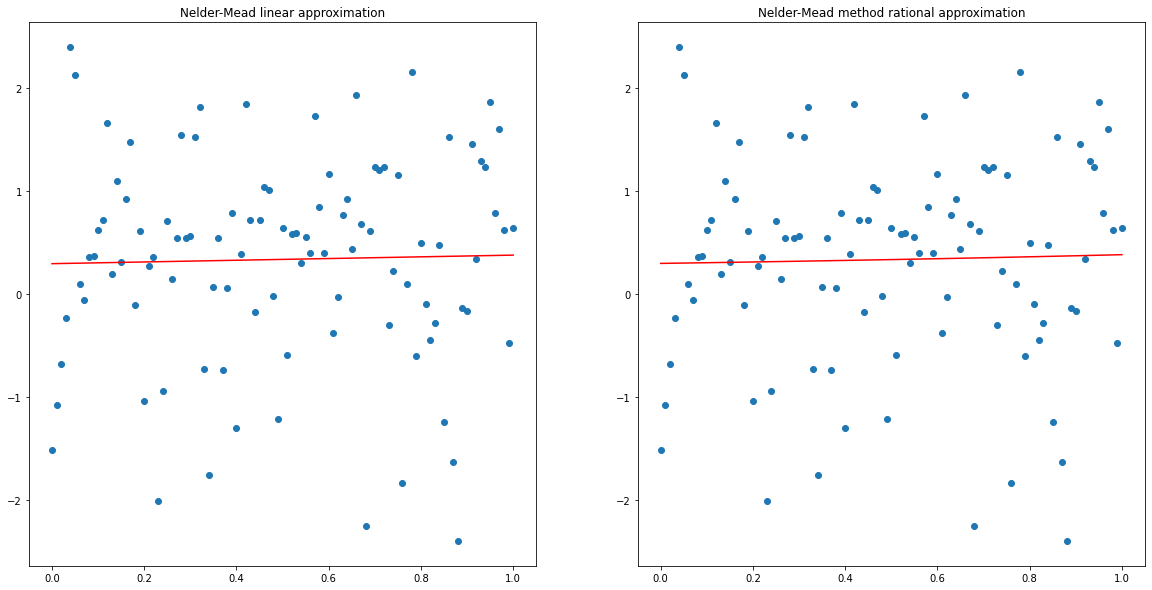












**Conclusions**

• Due to the significant disparity of the generated data, there are no significant graphical differences between the approximations.

• In linear approximation, the accuracy of all algorithms is almost the same.

• Exhaustive search turned out to be the most inefficient in terms of the number of iterations and time spent in both linear and non-linear approximations.

• Judging by such an indicator as the number of iterations, the most efficient algorithms are Conjugate gradient descent and Newton’s method

• Judging by such an indicator as the number of function evaluations, the most efficient algorithm is Gradient descent.

• In the non-linear approximation, the Levenberg-Marquardt algorithm showed the lowest accuracy.

**Appendix**

import numpy as np

import itertools

import matplotlib.pyplot as plt

import random

from scipy import optimize

from autograd import jacobian, grad

import pandas as pd

eps = 0.001

delta = np.random.randint(1, 100) / 100

alpha = np.random.randint(1, 100) / 100

betta = np.random.randint(1, 100) / 100

k = np.array(list(range(0, 101)))

X = np.array(list(map(lambda x: x/100, k)))

Y= np.array([alpha\*x + betta + np.random.normal(0, 1)

               for i, x in enumerate(X)])

f\_lin = lambda x, point: point[0] \* x + point[1]

f\_rat = lambda x, point: point[0] / (1 + point[1] \* x)

D = lambda args: sum([(args[0](x, args[1]) - y) \*\* 2 for x, y in zip(X, Y)])

D\_fun\_linear   = lambda args: D((f\_lin, args))

D\_fun\_rational = lambda args: D((f\_rat, args))

results\_lin = pd.DataFrame()

results\_lin['Algorithm name'] = ['Gradient descent', 'Conjugate gradient descent', 'Newton\'s method', 'Levenberg-Marquardt algorithm', '\*Exhaustive search', '\*Gauss method', '\*Nelder-Mead']

results\_lin.set\_index('Algorithm name', inplace=True)

results\_lin.style.set\_caption("Results of linear approximation")

header\_list = ['Number of iters', 'Function min', 'Number of func calculations']

results\_lin = results\_lin.reindex(columns = header\_list)

results\_rat = pd.DataFrame()

algs = ['Gradient descent', 'Conjugate gradient descent', 'Newton\'s method', 'Levenberg-Marquardt algorithm', '\*Exhaustive search', '\*Gauss method', '\*Nelder-Mead']

results\_rat['Algorithm name'] = algs

results\_rat.set\_index('Algorithm name', inplace=True)

results\_rat.style.set\_caption("Results of rational approximation")

header\_list = ['Number of iters', 'Function min', 'Number of func calculations']

results\_rat = results\_lin.reindex(columns = header\_list)

Gradient descent

def golden\_section(f, a, b, eps=eps):

    x1 = a + (3 - 5\*\*0.5)\*(b - a)/2

    x2 = b + (5\*\*0.5 - 3)\*(b - a)/2

    i = 0

    f1 = f(x1)

    f2 = f(x2)

    while np.abs(a - b) >= eps:

        if f(x1) <= f(x2):

            b = x2

            x2 = x1

            x1 = a + (3 - np.sqrt(5))/2\*(b - a)

            f2 = f1

            f1 = f(x1)

        else:

            a = x1

            x1 = x2

            x2 = b + (np.sqrt(5) - 3)/2\*(b - a)

            f1 = f2

            f2 = f(x2)

        i += 1

    return (a+b)/2, i + 2, i

def gradient\_descent(f, args):

    f\_min\_previous = f(args)

    args = np.random.randint(1, 100) / 100, np.random.randint(1, 100) / 100

    f\_min = f(args)

    i = 0

    while np.abs(f\_min - f\_min\_previous) > eps:

        f\_min\_previous = f\_min

        gradient = grad(f)

        def one\_dim\_step(h):

            return f(args - h \* np.array(gradient(args)))

        h = golden\_section(one\_dim\_step, 0, 1)[0]

        args = args - h \* np.array(gradient(args))

        f\_min = f(args - h \* np.array(gradient(args)))

        i += 1

    return args[0], args[1], f\_min, i+2, i

%%time

grad\_lin\_res = gradient\_descent(D\_fun\_linear, [0, 0])

grad\_lin\_res

num\_of\_iterations, func\_value, func\_count = grad\_lin\_res[3], round(grad\_lin\_res[2], 3), grad\_lin\_res[4]

results\_lin.loc[algs[0], :] = [num\_of\_iterations, func\_value, func\_count]

%%time

grad\_rat\_res = gradient\_descent(D\_fun\_rational, [0, 0])

grad\_rat\_res

num\_of\_iterations, func\_value, func\_count = grad\_rat\_res[3], round(grad\_rat\_res[2], 3), grad\_rat\_res[4]

results\_rat.loc[algs[0], :] = [num\_of\_iterations, func\_value, func\_count]

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, (grad\_lin\_res[0], grad\_lin\_res[1])), color='red')

axes.flat[0].scatter(X, Y)

axes.flat[0].set\_title('Gradient descent linear approximation')

axes.flat[1].plot(X, f\_rat(X, (grad\_rat\_res[0], grad\_rat\_res[1])), color='orange')

axes.flat[1].scatter(X, Y)

axes.flat[1].set\_title('Gradient descent rational approximation');

## Conjugate Gradient descent

cg\_lin\_res = optimize.minimize(D\_fun\_linear, np.random.rand(2), method='CG', tol=0.001)

cg\_lin\_res

num\_of\_iterations, func\_value, func\_count = cg\_lin\_res.nit, round(cg\_lin\_res.fun, 3), cg\_lin\_res.nfev

results\_lin.loc[algs[1], :] = [num\_of\_iterations, func\_value, func\_count]

cg\_rat\_res = optimize.minimize(D\_fun\_rational, np.random.rand(2), method='CG', tol=0.001)

cg\_rat\_res

num\_of\_iterations, func\_value, func\_count = cg\_rat\_res.nit, round(cg\_rat\_res.fun, 3), cg\_rat\_res.nfev

results\_rat.loc[algs[1], :] = [num\_of\_iterations, func\_value, func\_count]

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, cg\_lin\_res.x), color='purple')

axes.flat[0].scatter(X, Y, color='k')

axes.flat[0].set\_title('Conjugate Gradient descent linear approximation')

axes.flat[1].plot(X, f\_rat(X, cg\_rat\_res.x), color='orange')

axes.flat[1].scatter(X, Y, color='k')

axes.flat[1].set\_title('Conjugate Gradient descent rational approximation');

## Newton's method

ncg\_lin\_res = optimize.minimize(D\_fun\_linear, np.random.rand(2),jac=jacobian(D\_fun\_linear), method='Newton-CG', tol=0.001)

ncg\_lin\_res

num\_of\_iterations, func\_value, func\_count = ncg\_lin\_res.nit, round(ncg\_lin\_res.fun, 3), ncg\_lin\_res.nfev

results\_lin.loc[algs[2], :] = [num\_of\_iterations, func\_value, func\_count]

ncg\_rat\_res = optimize.minimize(D\_fun\_rational, np.random.rand(2),jac=jacobian(D\_fun\_rational),

                       method='Newton-CG', tol=0.001)

ncg\_rat\_res

num\_of\_iterations, func\_value, func\_count = ncg\_rat\_res.nit, round(ncg\_rat\_res.fun, 3), ncg\_rat\_res.nfev

results\_rat.loc[algs[2], :] = [num\_of\_iterations, func\_value, func\_count]

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, ncg\_lin\_res.x), color='purple')

axes.flat[0].scatter(X, Y, color='k')

axes.flat[0].set\_title('Newton\'s method linear approximation')

axes.flat[1].plot(X, f\_rat(X, ncg\_rat\_res.x), color='orange')

axes.flat[1].scatter(X, Y, color='k')

axes.flat[1].set\_title('Newton\'s method rational approximation');

## Levenberg-Marquardt algorithm

f\_lin\_ = lambda args, x, y: args[0] \* x + args[1] - y

f\_rat\_ = lambda args, x, y: args[0] / (1 + args[1] \* x) - y

lm\_lin\_res = optimize.least\_squares(f\_lin\_, np.random.rand(2), args=(X, Y), method='lm')

num\_of\_iterations, func\_value, func\_count = '-', round(D\_fun\_linear(lm\_lin\_res.x), 3), lm\_lin\_res.nfev

results\_lin.loc[algs[3], :] = [num\_of\_iterations, func\_value, func\_count]

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, lm\_lin\_res.x), color='purple')

axes.flat[0].scatter(X, Y, color='k')

axes.flat[0].set\_title('Levenberg-Marquardt algorithm linear approximation')

axes.flat[1].plot(X, f\_rat(X, lm\_rat\_res.x), color='orange')

axes.flat[1].scatter(X, Y, color='k')

axes.flat[1].set\_title('comma algorithm rational approximation');

# **Lab 2 algorithms**

from itertools import product

f\_lin = lambda x, a, b: a \* x + b

f\_rat = lambda x, a, b: a / (1 + b \* x)

D = lambda f, X, Y, a, b: sum([(f(x, a, b) - y) \*\* 2 for x, y in zip(X, Y)])

## Exhaustive search

steps = np.arange(-1, 1.001, 0.001)

def exhaustive\_search\_multi(f, X, Y, alpha, betta):

    f\_min = D(f, X, Y, alpha, betta)

    alpha\_res = 0

    betta\_res = 0

    i = 0

    for alpha, betta in list(product(steps, steps)):

        curr\_f = D(f, X, Y, alpha, betta)

        if curr\_f < f\_min:

            f\_min = curr\_f

            alpha\_res = alpha

            betta\_res = betta

        i += 1

    return [alpha\_res, betta\_res, f\_min, i+1, i]

%%time

exh\_lin\_res = exhaustive\_search\_multi(f\_lin, X, Y, 0, 0)

exh\_lin\_res

num\_of\_iterations, func\_value, func\_count = exh\_lin\_res[3], round(exh\_lin\_res[2], 3), exh\_lin\_res[-1]

results\_lin.loc[algs[4], :] = [num\_of\_iterations, func\_value, func\_count]

%%time

exh\_rat\_res = exhaustive\_search\_multi(f\_rat, X, Y, 0, 0)

exh\_rat\_res

num\_of\_iterations, func\_value, func\_count = exh\_rat\_res[3], round(exh\_rat\_res[2], 3), exh\_rat\_res[-1]

results\_rat.loc[algs[4], :] = [num\_of\_iterations, func\_value, func\_count]

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, exh\_lin\_res[0], exh\_lin\_res[1]), color='purple')

axes.flat[0].scatter(X, Y, color='k')

axes.flat[0].set\_title('Exhaustive search linear approximation')

axes.flat[1].plot(X, f\_rat(X, exh\_rat\_res[0], exh\_rat\_res[1]), color='orange')

axes.flat[1].scatter(X, Y, color='k')

axes.flat[1].set\_title('Exhaustive search rational approximation');

## Gauss method

# Gauss

eps = 0.001

def dichotomy\_for\_D(f, X, Y, a, b, a\_opt, b\_opt, par = 'a'):

    i = 0

    delta = np.random.uniform(0, eps)

    while np.abs(a - b) > eps:

        x1 = (a + b - delta) / 2

        x2 = (a + b + delta) / 2

        f\_1 = D(f, X, Y, x1, b\_opt) if par == 'a' else D(f, X, Y, a\_opt, x1)

        f\_2 = D(f, X, Y, x2, b\_opt) if par == 'a' else D(f, X, Y, a\_opt, x2)

        if f\_1 <= f\_2:

            b = x2

            fmin = f\_1

        else:

            a = x1

            fmin = f\_2

        i += 1

    return x1, i\*2, i

def gauss(f, X, Y, z1 = 0, z2 = 0):

    f\_min\_prev = D(f, X, Y, z1, z2)

    z1\_prev = z1

    z1 =  dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2)[0]

    f\_min = D(f, X, Y, z1, z2)

    i = 1 + dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2)[2]

    j = 2 + dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2)[1]

    l = 1

    while (np.abs(f\_min - f\_min\_prev) >= eps) and (np.abs(z1 - z1\_prev) >= eps):

      if l % 2 == 0:

          z1\_prev = z1

          z1 =  dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2)[0]

          f\_min\_prev = f\_min

          f\_min = D(f, X, Y, z1, z2)

          i += 1 + dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2)[2]

          j += 1 + dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2)[1]

      else:

          z2\_prev = z2

          z2 =  dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2, par = 'b')[0]

          f\_min\_prev = f\_min

          f\_min = D(f, X, Y, z1, z2)

          i += 1 + dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2, par = 'b')[2]

          j += 1 + dichotomy\_for\_D(f, X, Y, -1, 1 + eps, z1, z2, par = 'b')[1]

          if np.abs(z2 - z2\_prev) < eps:

              break

      l += 1

    return z1, z2, j, i

%%time

F\_linear\_opt\_2 = gauss(f\_lin, X, Y, alpha, betta)

num\_of\_iterations, func\_value, func\_count = F\_linear\_opt\_2[3], round(D(f\_lin, X, Y, F\_linear\_opt\_2[0], F\_linear\_opt\_2[1]), 3), F\_linear\_opt\_2[-1]

results\_lin.loc[algs[5], :] = [num\_of\_iterations, func\_value, func\_count]

%%time

F\_rational\_opt\_2 = gauss(f\_rat, X, Y, alpha, betta)

num\_of\_iterations, func\_value, func\_count = F\_rational\_opt\_2[3], round(D(f\_lin, X, Y, F\_rational\_opt\_2[0], F\_rational\_opt\_2[1]), 3), F\_rational\_opt\_2[-1]

results\_rat.loc[algs[5], :] = [num\_of\_iterations, func\_value, func\_count]

# Gauss method

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, F\_linear\_opt\_2[0], F\_linear\_opt\_2[1]), color = 'red')

axes.flat[0].scatter(X, Y)

axes.flat[0].set\_title('Gauss method linear approximation')

axes.flat[1].plot(X, f\_rat(X, F\_rational\_opt\_2[0], F\_rational\_opt\_2[1]), color = 'red')

axes.flat[1].scatter(X, Y)

axes.flat[1].set\_title('Gauss method rational approximation')

## Nelder-Mead

def F\_linear\_point(x, point):

    return point[0]\*x + point[1]

def F\_rational\_point(x, point):

    return point[0]/(1 + point[1]\*x)

def D\_point(point, f, X, Y):

    return sum([(f(x\_i, point) - y\_i)\*\*2 for x\_i, y\_i in zip(X, Y)])

def nelder\_mead(f, X, Y):

    result = optimize.minimize(D\_point, np.random.rand(2), args=(f, X, Y), method ='nelder-mead')

    return result

%%time

F\_linear\_opt\_3 = nelder\_mead(F\_linear\_point, X, Y)

num\_of\_iterations, func\_value, func\_count = F\_linear\_opt\_3.nit, round(F\_linear\_opt\_3.fun, 3), F\_linear\_opt\_3.nfev

results\_lin.loc[algs[6], :] = [num\_of\_iterations, func\_value, func\_count]

%%time

F\_rational\_opt\_3 = nelder\_mead(F\_rational\_point, X, Y)

num\_of\_iterations, func\_value, func\_count = F\_rational\_opt\_3.nit, round(F\_rational\_opt\_3.fun, 3), F\_rational\_opt\_3.nfev

results\_rat.loc[algs[6], :] = [num\_of\_iterations, func\_value, func\_count]

# Nelder-Mead

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes.flat[0].plot(X, f\_lin(X, F\_linear\_opt\_3.x[0], F\_linear\_opt\_3.x[1]), color = 'red')

axes.flat[0].scatter(X, Y)

axes.flat[0].set\_title('Nelder-Mead linear approximation')

axes.flat[1].plot(X, f\_rat(X, F\_rational\_opt\_3.x[0], F\_rational\_opt\_3.x[1]), color = 'red')

axes.flat[1].scatter(X, Y)

axes.flat[1].set\_title('Nelder-Mead method rational approximation')

!pip install openpyxl

writer = pd.ExcelWriter('results\_lin.xlsx')

results\_lin.to\_excel(writer)

writer.save()

writer\_ = pd.ExcelWriter('results\_rat.xlsx')

results\_rat.to\_excel(writer\_)

writer\_.save()