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
Zakhar Pinaev
J4132c
Accepted by
Dr Petr Chunaev

St. Petersburg
2021

Goal

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

Problems and methods

I. Approximate the data by linear and rational functions by means of least squares through the numerical minimization (with precision $\varepsilon = 0.001$) of the given 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 in a plot separately for each type of approximant so that one can compare the results for the numerical methods used. Analyze the results obtained (in terms of number of iterations, precision, number of function evaluations, etc.) and compare them with those from Task 2.

Brief theoretical part

When solving a specific optimization problem, the researcher must first choose a mathematical method that would lead to final results with the least computational costs or make it possible to obtain the largest amount of information about the desired solution. The choice of one method or another is largely determined by the formulation of the optimal problem, as well as the mathematical model of the optimization object used.

Common to nonlinear programming methods is that they are used to solve problems with nonlinear optimality criteria. All nonlinear programming methods are search-type numerical methods. Their essence lies in the definition of a set of independent variables that give the greatest increment of the optimized function.

Methods for the numerical solution of multidimensional unconstrained minimization problems are numerous and varied. They can be conventionally divided into three large classes, depending on the information used in the implementation of the method:

1. Methods of order zero, or direct search, the strategy of which is based on the use of information only about the properties of the objective function.

2. Methods of the first order, in which, when constructing an iterative procedure, along with information about the objective function, information about the values of the first derivatives of this function is used.

3. Second-order methods, in which, along with information about the values of the objective function and its first-order derivatives, information about the second derivatives of the function is used.

Results

In the task, it was necessary to generate data in a certain way and approximate it by linear and rational functions by means of least squares through the numerical minimization of the given function. To solve the minimization problem, were used the methods of Gradient Descent, Conjugate Gradient Descent, Newton's method and Levenberg-Marquardt algorithm.

For the corresponding generated data, Figures 1-2 show each of the approximating functions (linear and rational) using Gradient Descent, Conjugate Gradient Descent, Newton's method and Levenberg-Marquardt algorithm.

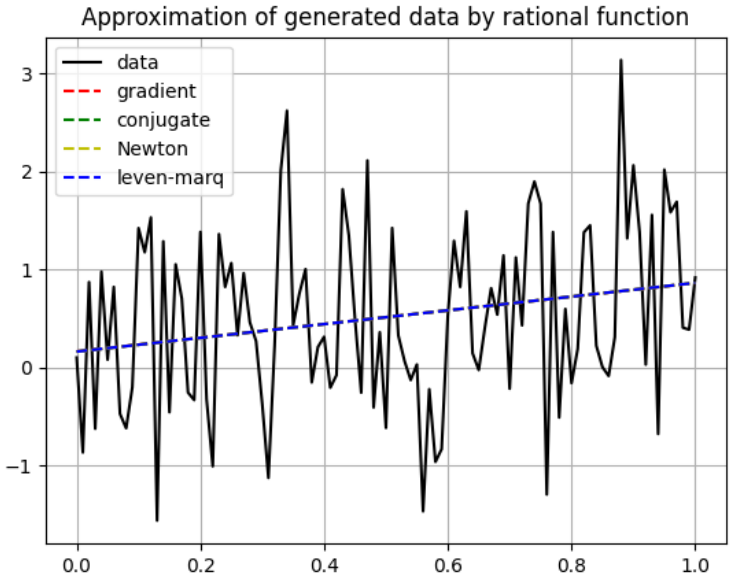

Figure 1 – linear approximating function using Gradient Descent (red dashed), Conjugate Gradient Descent (green dashed) Newton's method (yellow dashed) and Levenberg-Marquardt algorithm (blue dashed) methods (data is shown via black line)

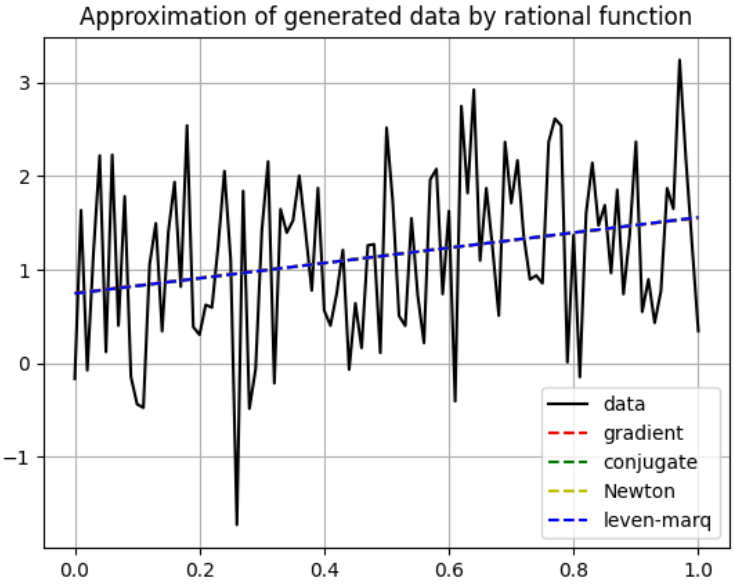

Figure 2 – rational approximating function using Gradient Descent (red dashed), Conjugate Gradient Descent (green dashed) Newton's method (yellow dashed) and Levenberg-Marquardt algorithm (blue dashed) methods (data is shown via black line)

As can be seen from Figures 1-2, all methods for each of the approximation functions gave extremely close results, therefore, for a more detailed comparison, table 2 show the detailed results for calculating the approximation functions by different methods.

Table 2 – Detailed results of calculating approximation functions (linear and rational)

function	method	result (minimization function)	number of iterations	*f*-calculations
linear	Gradient Descent	91.133	408	408
	Conjugate Gradient Descent	91.132	43	129
	Newton's method	91.131	124	248
	Levenberg-Marquardt algorithm	91.132	3	6
rational	Gradient Descent	103.805	936	936
	Conjugate Gradient Descent	103.805	20	60
	Newton's method	103.806	218	436
	Levenberg-Marquardt algorithm	103.806	3	6

As can be seen from Table 2, all methods in each of the approximation functions came to practically the same value of the minimization function. A slight difference in values corresponds to the value of the specified precision. As expected, the gradient descent method required the largest number of iterations and the calculation of the function, since at each iteration it needs to calculate the direction of motion and the values of the function. In turn, Newton's method requires fewer iterations than gradient descent, however, it requires additional information about the second derivatives of the original function. As a compromise for these two functions, can be used the conjugate gradient method, which proved to be better both in the number of iterations and in function calculations among the three mentioned methods, and does not require information about the second derivatives of the original function. And, finally, the Levenberg-Marquardt method became the most efficient in terms of iterations and calculations of the function. For almost every dataset, it took 3 iterations to achieve the specified accuracy. This may be partly since this method was taken from the *scipy* library, so it is optimized.

Further, a comparison was made between the first and second order methods implemented in this work and the direct methods from the previous work. For this, the data were re-generated, which were further approximated by a linear function for example, and the minimization problem was solved by direct methods and methods of the first and second order. The comparison results of the algorithms are shown in Table 3.

Table 3 – Detailed results of calculating linear approximation function using different methods

method	method	result (minimization function)	number of iterations	*f*-calculations
direct	Exhaustive	98.980	4000000	4000000
	Gauss	98.982	15	158
	Nelder-Mead	98.980	21	320
1-st order	Gradient Descent	98.981	741	741
	Conjugate Gradient Descent	98.981	30	90
2-nd order	Newton's method	98.982	243	486
	Levenberg-Marquardt algorithm	98.981	3	7

Analysis of the results described in Table 3 showed that to solve the minimization problem, it is necessary to choose a method depending on a specific situation, for example, on the available data. For example, if, along with information about the values of the objective function and its first-order derivatives, there is information about the second derivatives of the function, then it is preferable to use the Levenberg-Marquardt algorithm, because it allows to get a solution most efficiently and quickly. Further, if, in addition to information about the objective function, there is information only about the values of the first derivatives of this function, then it will be preferable to use the conjugate gradient descent method. This method requires a relatively small (though not the least) number of iterations and, most importantly – it requires the least amount of function calculations under the conditions described above. And finally, if information is available in the task only about the values of the objective function, then it is preferable to use the direct Gauss method.

Conclusion

As a result of the work, several algorithms for unconstrained nonlinear optimization of the first and second order were considered. As a result of comparison in the framework of the first and second order algorithms, the Levenberg-Marquardt algorithm performed best. However, after considering the results of several types of methods, it was concluded that the choice of methods for solving a specific problem should be based on the information available within the framework of this problem.

Appendix

```
import math
import numpy as np
import random
import matplotlib.pyplot as plt
import sys
from scipy.optimize import least_squares


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

```
def aprox_ratio(x, a, b):
    #if b*x == -1:
    #    b+=0.0001
    return a / (1 + b * x)


def num_minimize(func, x, y, a, b):
    D = 0
    for k in range(101):
        D += (func(x[k], a, b) - y[k])**2
    return D


def minim(x0, func, x, y):
    a, b = x0
    D = 0
    y_new = np.array([y])
    x_new = np.array([x])
    #for k in range(len(y)):
    #    D += func(x[k], a, b) - y[k]
    res = a * x_new + b - y_new

    return sum(res)


def num_minimize_vec(point, func, x, y):
    a, b = point
    D = 0
    for k in range(len(y)):
        D += (func(x[k], a, b) - y[k])**2
    return D


def gradient_function(xk, x, y):
    y_pred = [xk[0] * xx + xk[1] for xx in x]
    a = b = 0
    for i in range(len(x)):
        a += -(2) * (x[i] * (y[i] - y_pred[i]))
        b += -(2) * (y[i] - y_pred[i])
    gk = np.array([a, b])
    return gk


def hesse(xk, x, y):
    a1 = a2 = a3 = a4 = 0
    for i in range(len(x)):
        a1 += 2
        a2 += 2 * x[i]
        a3 += 2 * x[i]
        a4 += 2 * (x[i]**2)

    gk = np.array([ [a1, a2], [a3, a4] ])
    return gk


def jacob(x0, func, x, y):
    a1 = a2 = a3 = a4 = 0
    for i in range(len(x)):
        a1 += 2
        a2 += 2 * x[i]
        a3 += 2 * x[i]
        a4 += 2 * (x[i]**2)
```

```
    J = np.array([ [a1, a2], [a3, a4] ])
    return J


def wolfe_powell(xk, sk, func, x, y):
    alpha = 1.0
    a = 0.0
    b = -sys.maxsize
    c_1 = 0.1
    c_2 = 0.5
    k = 0
    func_calcs = 0
    while k < 100:
        k += 1
        func_calcs += 3
        if num_minimize_vec(xk, func, x, y) - num_minimize_vec(xk + alpha *
sk, func, x, y) >= -c_1 * alpha * np.dot(gradient_function(xk, x, y), sk):
            # print ('Выполнить условие 1')
            func_calcs += 2
            if np.dot(gradient_function(xk + alpha * sk, x, y), sk) >= c_2 *
np.dot(gradient_function(xk, x, y), sk):
                # print ('Условие 2 выполнено')
                return alpha, func_calcs
            else:
                a = alpha
                alpha = min(2 * alpha, (alpha + b) / 2)

        else:
            b = alpha
            alpha = 0.5 * (alpha + a)

    return alpha, func_calcs


def gradient_descent(x, y, func, stopping_threshold=1e-3):
    # Initializing weight, bias, learning rate and iterations
    current_weight = 0.1 # a
    current_bias = 0.01 # b

    # iterations = iterations
    learning_rate = 0.0001
    n = float(len(x))

    previous_cost = None
    iterations = 0
    func_calc = 0

    # Estimation of optimal parameters
    while True:
        # Making predictions
        # y_predicted = (current_weight * x) + current_bias
        y_predicted = [current_weight * xx + current_bias for xx in x]

        current_cost = num_minimize(func, x, y, current_weight, current_bias)
        func_calc += 1

        if previous_cost and abs(previous_cost - current_cost) <=
stopping_threshold:
            break

        previous_cost = current_cost

        weight_derivative = 0
```

```
        bias_derivative = 0

        # Calculating the gradients
        for ii in range(len(x)):
            weight_derivative += -(2) * (x[ii] * (y[ii] - y_predicted[ii]))
            bias_derivative += -(2) * (y[ii] - y_predicted[ii])

        # Updating weights and bias
        current_weight = current_weight - (learning_rate * weight_derivative)
        current_bias = current_bias - (learning_rate * bias_derivative)

        iterations += 1

    return current_weight, current_bias, iterations, func_calc, current_cost,
previous_cost


def conjugate_gradient(x, y, func, prec=1e-3):
    xk = np.array([0.0, 0.0])
    gk = gradient_function(xk, x, y)

    #sigma = num_minimize_vec(gk, func, x, y)
    sigma = np.linalg.norm(gk)
    sk = -gk
    iterations = 0
    func_calcs = 0
    # w = np.zeros((2, 10 ** 3))  # Сохраняем итерацию и устанавливаем
переменную xk

    while sigma > prec:
        # w[:, step] = np.transpose(xk)
        iterations += 1
        alpha, calculate = wolfe_powell(xk, sk, func, x, y)
        xk = xk + alpha * sk
        g0 = gk
        gk = gradient_function(xk, x, y)
        miu = (np.linalg.norm(gk) / np.linalg.norm(g0)) ** 2
        sk = -1 * gk + miu * sk
        sigma = np.linalg.norm(gk)
        # sigma = np.linalg.norm(gk)
        func_calcs += + 3

    return xk[0], xk[1], iterations, func_calcs, sigma


def newton(x, y, func, prec=1e-3):
    xk = np.array([0.0, 0.0])
    iterations = func_calcs = 0
    gk = gradient_function(xk, x, y)
    hessen = hesse(xk, x, y)
    sigma = np.linalg.norm(gk)
    sk = -1 * np.dot(np.linalg.inv(hessen), gk)

    while sigma > prec:

        iterations += 1
        xk = xk + sk
        gk = gradient_function(xk, x, y) * 0.25 #0.283
        hessen = hesse(xk, x, y)
        func_calcs += 2
        sigma = np.linalg.norm(gk)
        sk = -1 * np.dot(np.linalg.inv(hessen), gk) # / 5
    return xk[0], xk[1], iterations, func_calcs, sigma
```

```
def leven_marq(x, y, func, prec=1e-3):
    x0 = np.array([0, 0])

    res = least_squares(minim, x0, args=(func, x, y), ftol=prec, method='lm')

    return res.x[0], res.x[1], res.nfev, res.nfev // 2, num_minimize(func, x,
y, res.x[0], res.x[1])


def min_golden_2(func, x, y, start, end, arg_fix, arg_ind, precision):
    # func - approximation func (linear or rational)
    # start, end - first & second element for searching best a (or b)
    # arg_fix, arg_ind - if arg_ind = 1, we calc 'a' and arg_fix is  fixed
'b' (and contrary)
    iterations = 0
    func_calc = 0
    b = end
    a = start
    c = a + ((3 - math.sqrt(5)) / 2) * (b - a)
    d = b + ((math.sqrt(5) - 3) / 2) * (b - a)

    if arg_ind:
        fc = num_minimize(func, x, y, c, arg_fix)  # if arg_ind = 1 -> we
calc 'a' param
        fd = num_minimize(func, x, y, d, arg_fix)
    else:
        fc = num_minimize(func, x, y, arg_fix, c)  # if arg_ind = 1 -> we
calc 'b' param
        fd = num_minimize(func, x, y, arg_fix, d)
    func_calc += 2
    while b - a >= precision:
        iterations += 1
        if fc < fd:
            b = d
            d = c
            fd = fc
            c = a + ((3 - math.sqrt(5)) / 2 ) * (b - a)
            if arg_ind:
                fc = num_minimize(func, x, y, c, arg_fix)
            else:
                fc = num_minimize(func, x, y, arg_fix, c)
            func_calc += 1
        else:
            a = c
            c = d
            fc = fd
            d = b + ((math.sqrt(5) - 3) / 2) * (b - a)
            if arg_ind:
                fd = num_minimize(func, x, y, d, arg_fix)
            else:
                fd = num_minimize(func, x, y, arg_fix, d)
            func_calc += 1
    if arg_ind:
        min_cur = num_minimize(func, x, y, (a + b) / 2.0, arg_fix)
    else:
        min_cur = num_minimize(func, x, y, arg_fix, (a + b) / 2.0)
    func_calc += 1
    # min_cur - minimum D with found a(or b) and fixed b(or a)
    # (a + b) / 2 - best found value of a(or b)
    return min_cur, (a + b) / 2, iterations, func_calc


def min_2_hessian(func, y, x, start, end, prec):
```

```
    d = num_minimize(func, x, y, 0, 0)
    iterations_while = 0
    iterations_golden = 0
    func_calcs = 0
    best_a = 0
    best_b = 0
    step = 0.1
    while True:
        iterations_while += 1
        d_cur, best_a, iter_cur, func_calc = min_golden_2(func, x, y, start,
end, best_b, 1, step)
        func_calcs += func_calc
        iterations_golden += iter_cur
        if round(abs(d - d_cur), 3) < prec:
            d = d_cur
            break
        d = d_cur
        d_cur, best_b, iter_cur, func_calc = min_golden_2(func, x, y, start,
end, best_a, 0, step)
        func_calcs += func_calc
        iterations_golden += iter_cur
        if abs(d - d_cur) < prec:
            d = d_cur
            break
        else:
            step = step / 2.0
            d = d_cur
    return d, best_a, best_b, iterations_while, iterations_golden, func_calcs
class Vector(object):
    def __init__(self, x, y):
        """ Create a vector, example: v = Vector(1,2) """
        self.x = x
        self.y = y
    def __repr__(self):
        return "({0}, {1})".format(self.x, self.y)
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)
    def __sub__(self, other):
        x = self.x - other.x
        y = self.y - other.y
        return Vector(x, y)
    def __rmul__(self, other):
        x = self.x * other
        y = self.y * other
        return Vector(x, y)
    def __truediv__(self, other):
        x = self.x / other
        y = self.y / other
        return Vector(x, y)
    def c(self):
        return (self.x, self.y)
def min_2_nelder_mead(func, y, x, prec):
    alpha = 4
    betta = 2
    gamma = 8
    func_calcs = 0
    # initialization
    v1 = Vector(0.5, 0.5)
    v2 = Vector(0.5, 0.0)
    v3 = Vector(0.0, 0.5)
    d = num_minimize_vec(v3.c(), func, x, y)
    func_calcs += 1
```

```
    best_a, best_b = v3.c()
    iterations = 0
    old_b = v3
    while True:
        iterations += 1
        adict = {v1:num_minimize_vec(v1.c(), func, x, y),
v2:num_minimize_vec(v2.c(), func, x, y), v3:num_minimize_vec(v3.c(), func, x,
y)}
        func_calcs += 3
        points = sorted(adict.items(), key=lambda xx: xx[1])
        b = points[0][0]
        g = points[1][0]
        w = points[2][0]
        mid = (g + b)/2
        a_old, b_old = old_b.c()
        a_new, b_new = b.c()

        if a_new != a_old and b_new != b_old:
            #print(old_b, b)
            d = num_minimize_vec(old_b.c(), func, x, y)
            new_d = num_minimize_vec(b.c(), func, x, y)
            if round(abs(d - new_d), 3) < prec:
                d = new_d
                best_a, best_b = b.c()
                break
        # reflection
        xr = mid + alpha * (mid - w)
        func_calcs += 2
        if num_minimize_vec(xr.c(), func, x, y) < num_minimize_vec(g.c(),
func, x, y):
            w = xr
        else:
            func_calcs += 2
            if num_minimize_vec(xr.c(), func, x, y) < num_minimize_vec(w.c(),
func, x, y):
                w = xr
            c = (w + mid)/2
            func_calcs += 2
            if num_minimize_vec(c.c(), func, x, y) < num_minimize_vec(w.c(),
func, x, y):
                w = c
        func_calcs += 2
        if num_minimize_vec(xr.c(), func, x, y) < num_minimize_vec(b.c(),
func, x, y):
            # expansion
            xe = mid + gamma * (xr - mid)
            func_calcs += 2
            if num_minimize_vec(xe.c(), func, x, y) <
num_minimize_vec(xr.c(), func, x, y):
                w = xe
            else:
                w = xr
        func_calcs += 2
        if num_minimize_vec(xr.c(), func, x, y) > num_minimize_vec(g.c(),
func, x, y):
            # contraction
            xc = mid + betta * (w - mid)
            func_calcs += 2
            if num_minimize_vec(xc.c(), func, x, y) < num_minimize_vec(w.c(),
func, x, y):
                w = xc
        # update points
        v1 = w
        v2 = g
```

```
        v3 = b
        old_b = b
        func_calcs += 1
        #temp_d = num_minimize_vec(v3.c(), func, x, y)
        #print(abs(d - temp_d))
        # best_a, best_b = v3.c()
        #if abs(d - temp_d) < prec:
        #    best_a, best_b = v3.c()
        #    d = temp_d
        #    break
        #d = temp_d

    return d, best_a, best_b, iterations, func_calcs

def main():
    precision = 0.001
    func_cur = aprox_linear

    a = random.uniform(0.0, 1.0)
    b = random.uniform(0.0, 1.0)

    k = range(0, 101)

    x = [0] * 101
    y = [0] * 101
    sigma = np.random.standard_normal(101)

    for kk in k:
        x[kk] = kk / 100
        y[kk] = a * x[kk] + b + sigma[kk]

    a_1, b_1, iter_1, func_calc_1, d_1, d_prev = gradient_descent(x, y,
func_cur, precision*0.0001)
    a_2, b_2, iter_2, func_calc_2, d_2 = conjugate_gradient(x, y,
aprox_linear, precision)
    a_3, b_3, iter_3, func_calc_3, d_3 = newton(x, y, func_cur, precision)
    a_4, b_4, func_calc_4, iter_4, d_4 = leven_marq(x, y, func_cur,
precision)

    d_5, a_5, b_5, iterations_while, iterations_golden, func_calcs_5 =
min_2_hessian(aprox_linear, y, x, -0.9, 1,

precision)

    d_6, a_6, b_6, iterations_6, func_calcs_6 =
min_2_nelder_mead(aprox_linear, y, x, precision)

    y_aprox_1 = [a_1 * xx + b_1 for xx in x]
    y_aprox_2 = [a_2 * xx + b_2 for xx in x]
    y_aprox_3 = [a_3 * xx + b_3 for xx in x]
    y_aprox_4 = [a_4 * xx + b_4 for xx in x]

    plt.plot(x, y, 'k-', x, y_aprox_1, 'r--', x, y_aprox_2, 'g--', x,
y_aprox_3, 'y--', x, y_aprox_4, 'b--')
    #plt.plot(x, y, 'k-', x, y_aprox_1, 'r--', x, y_aprox_2, 'g--')
    plt.title('Approximation of generated data by rational function')
    plt.legend(['data', 'gradient', 'conjugate', 'Newton', 'leven-marq'],
loc='best')
    plt.grid(True)
    plt.show()
if __name__ == "__main__":
    main()
```