FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report

on the practical task No. 2

“Algorithms for unconstrained nonlinear optimization. Direct methods”

Performed by
Zakhar Pinaev
J4132c
Accepted by
Dr Petr Chunaev

St. Petersburg
2021

## Goal

The use of direct methods (one-dimensional methods of exhaustive search, dichotomy, golden section search; multidimensional methods of exhaustive search, Gauss (coordinate descent), Nelder-Mead) in the tasks of unconstrained nonlinear optimization.

## Problems and methods

**I.** Use the one-dimensional methods of exhaustive search, dichotomy and golden section search to find an approximate (with precision $\varepsilon = 0.001$) solution $x: f(x) \to min$ for the set of functions and domains. Calculate the number of *f*-calculations and the number of iterations performed in each method and analyze the results. Explain differences (if any) in the results obtained.

**II.** Approximate the data by linear and rational functions by means of least squares through the numerical minimization of the given function. To solve the minimization problem, use the methods of exhaustive, Gauss and Nelder-Mead search. 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.).

## 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.

Unconstrained optimization methods are divided into one-dimensional and multivariate optimization methods. All methods of one-dimensional optimization are based on the assumption that the studied objective function in the feasible region at least has the property of unimodality, since for the unimodal function W (x), the comparison of the values of W (t) at two different points of the search interval makes it possible to determine at which of the given the two indicated subinterval points do not have the optimum point.

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.

## Results

**I.** In the first task, the search for the minimum was carried out for the following functions and domains:

1. $f_1(x) = x^3, x \in [0,1]$;
2. $f_2(x) = |x - 0.2|, x \in [0,1]$;
3. $f_3(x) = xsin\frac{1}{x}, x \in [0.01,1]$.

The results for each of the investigated functions and methods are shown in Table 1.

Table 1 – results for the minimum search of functions $f_1$, $f_2$, $f_3$ using exhaustive, dichotomy and golden section search

| function | method | result, *x* | number of iterations | *f*-calculations |
|---|---|---|---|---|
| $f_1(x)$ | exhaustive | 0.0000 | 1001 | 1001 |
| | dichotomy | 0.0004 | 10 | 21 |
| | golden section | 0.0003 | 15 | 18 |
| $f_2(x)$ | exhaustive | 0.2000 | 1001 | 1001 |
| | dichotomy | 0.2000 | 10 | 21 |
| | golden section | 0.2000 | 15 | 18 |
| $f_3(x)$ | exhaustive | 0.2229 | 991 | 991 |
| | dichotomy | 0.2224 | 10 | 21 |
| | golden section | 0.2227 | 15 | 18 |

As can be seen from Table 1, for each of the functions, all methods found the same solution with a given accuracy ($\varepsilon$ = 0.001). Differences in solutions for each function began after the sign determined by the accuracy, except for the function $f_2$, where even after the required accuracy, the values of each method turned out to be the same. It is expected that for each of the considered functions, the exhaustive search method required the maximum number of iterations and the corresponding number of calculations of the function. This is since in it it was necessary to go over the grid of all possible values and calculate the value of the function at each point.

However, the dichotomy and gold section methods required much fewer iterations and function computation. This is explained by the fact that each of these does not pass through the entire grid of possible values, but uses the function's unimodality property, excluding certain intervals at each iteration. In addition, it can be seen that the golden section method required fewer function evaluations, which is explained by the fact that at each iteration (except for the first) the method requires only one function evaluation, in contrast to the dichotomy method.

**II.** In the second 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 exhaustive, Gauss and Nelder-Mead search.

For the corresponding generated data, Figures 1-2 show each of the approximating functions (linear and rational) using exhaustive, Gauss and Nelder-Mead methods.

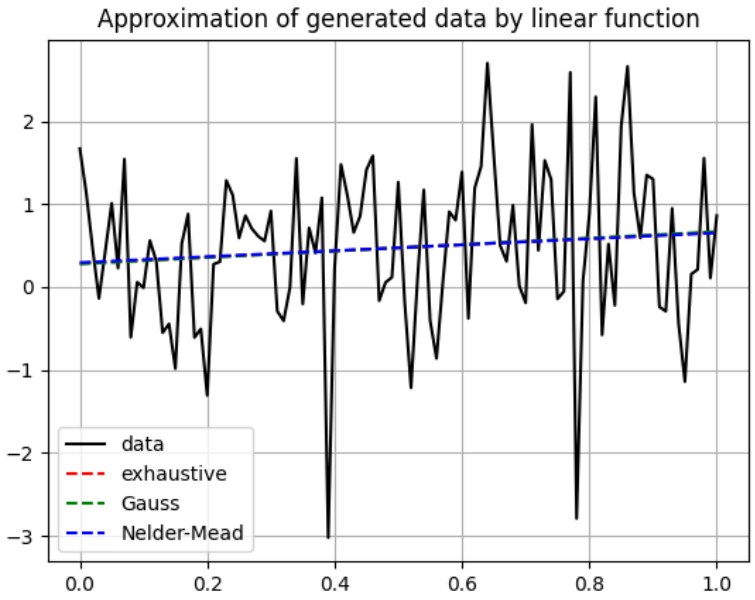


Figure 1 – linear approximating function using exhaustive (red dashed), Gauss (green dashed) and Nelder-Mead (blue dashed) methods (data is shown via black line)

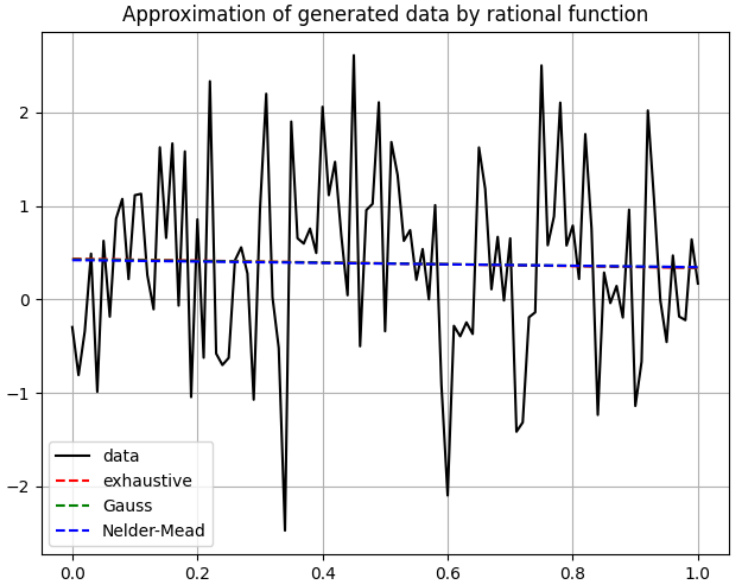


Figure 2 – rational approximating function using exhaustive (red dashed), Gauss (green dashed) and Nelder-Mead (blue dashed) methods (data is shown via black line)

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 | exhaustive | 93.599 | 4000000 | 4000000 |
| | Gauss | 93.601 | 13 | 478 |
| | Nelder-Mead | 93.602 | 16 | 240 |
| rational | exhaustive | 98.207 | 3610000 | 3610000 |
| | Gauss | 98.209 | 14 | 530 |
| | Nelder-Mead | 98.213 | 20 | 308 |

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. As expected, the exhaustive method required most of the iterations and calculation of the function, due to the fact that it needed to iterate over all possible values. In turn, for each approximation function, the Gauss method required fewer iterations than the Nelder-Mead method. However, the Nelder-Mead method required fewer function evaluations.

## Conclusion

As a result of the work, several methods of unconstrained nonlinear optimization were implemented in two particular problems. One-dimensional optimization was considered in the problem of finding the minimum of a function, for which three methods were implemented (brute force, dichotomies, Nelder-Mead). The last two methods have shown themselves in the best way, each of which is good in its own way either in the minimum number of iterations, or in the minimum number of function calculations.

For multidimensional (in particular, two-dimensional) optimization, a function approximation problem was considered, for which three types of minimization were implemented for two approximation functions. As a result, as expected, the enumeration method became the most resource-intensive, and the Gauss and Nelder-Mead methods turned out to be good in their own way, since the first required fewer iterations, and the second required fewer function evaluations.

## Appendix

```
import math
import numpy as np
import random
import matplotlib.pyplot as plt


def f1(x):
    return pow(x, 3) # [0, 1]


def f2(x):
    return abs(x - 0.2) # [0, 1]


def f3(x):
    return x * math.sin(1/x) # [0.01, 1]
```

```python
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(len(y)):
        D += (func(x[k], a, b) - y[k])**2
    return D


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 min_exhaustive(func, start, end, precision):
    iterations = 0
    func_calc = 0
    min_cur = float(func(start))
    x_min = float(start)

    for x in np.arange(start, end + precision * 0.001, precision):
        iterations += 1
        temp = func(x)
        func_calc += 1
        if temp < min_cur:
            min_cur = temp
            x_min = x

    return min_cur, x_min, iterations, func_calc


def min_dichotomy(func, start, end, precision):
    iterations = 0
    func_calc = 0
    b = end
    a = start
    while (b - a) / 2 >= precision:
        iterations += 1
        m = (a + b) / 2
        c = m - (precision / 2)
        d = m + (precision / 2)

        if func(c) < func(d):
            b = d
        else:
            a = c
        func_calc += 2
    min_cur = func((b+a) / 2)
    func_calc += 1
    return min_cur, (b+a) / 2, iterations, func_calc
```

```
def min_golden(func, start, end, precision):
    iterations = 0
    func_calc = 0
    # phi = (1 + math.sqrt(5)) / 2.0
    b = end
    a = start

    c = a + ((3 - math.sqrt(5)) / 2) * (b - a)
    d = b + ((math.sqrt(5) - 3) / 2) * (b - a)

    fc = func(c)
    fd = func(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)
            fc = func(c)
            func_calc += 1
        else:
            a = c
            c = d
            fc = fd
            d = b + ((math.sqrt(5) - 3) / 2) * (b - a)
            fd = func(d)
            func_calc += 1

    min_cur = func((a + b) / 2)
    func_calc += 1
    return min_cur, (a + b) / 2, iterations, func_calc


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_exhaustive(func, y, x, start, end, prec):
    D = num_minimize(func, x, y, 0, 0)
    best_a = 0
    best_b = 0
    iterations = 0
    func_calcs = 0
    # step = 0.1
    for a in np.arange(start, end, prec):
        for b in np.arange(start, end, prec):
            iterations += 1
            func_calcs += 1
            temp = num_minimize(func, x, y, a, b)
            if temp < D: ############################ need D(i+1) - D() <
precision
                D = temp
                best_a = a
                best_b = b
    return D, best_a, best_b, iterations, func_calcs


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
        #    d = temp_d
        #    break
```

```
        #d = temp_d

    return d, best_a, best_b, iterations, func_calcs

#print(min_exhaustive(f3, 0.01, 1, 0.001))
#print(min_dichotomy(f3, 0.01, 1, 0.001))
#print(min_golden(f3, 0.01, 1, 0.001))

#print(min_exhaustive(f2, 0, 1, 0.001))
#print(min_dichotomy(f2, 0, 1, 0.001))
#print(min_golden(f2, 0, 1, 0.001))


################################ part 2 ################################

precision = 0.001

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]

d_1, a_1, b_1, iterations_1, func_calcs_1 = min_2_exhaustive(aprox_linear, y,
x, -0.9, 1, precision) # 0.5069999999996138 0.586999999999605
123.65071503845361

d_2, a_2, b_2, iterations_while, iterations_golden, func_calcs_2 =
min_2_hessian(aprox_linear, y, x, -0.9, 1, precision)

d_3, a_3, b_3, iterations_3, func_calcs_3 = min_2_nelder_mead(aprox_linear,
y, x, precision)

print(d_1, a_1, b_1, iterations_1, func_calcs_1)
print(d_2, a_2, b_2, iterations_while, iterations_golden, func_calcs_2)
print(d_3, a_3, b_3, iterations_3, func_calcs_3)

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]

plt.plot(x, y, 'k-', x, y_aprox_1, 'r--', x, y_aprox_2, 'g--', x, y_aprox_3,
'b--')
plt.title('Approximation of generated data by linear function')
plt.legend(['data', 'exhaustive', 'Gauss', 'Nelder-Mead'], loc='best')
plt.grid(True)
plt.show()
```