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

Victor Voronin

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2020

# Goal

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

# Formulation of the problem

1. 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:

(linear approximant),

(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

To solve the task, it is supposed to use the following standard libraries:

* library NumPy to generate values of a random variable with standard normal distribution
* matplotlib.pyplot to create graphs
* library math to calculate sin, square root etc.
* library SciPy to apply Nelder-Mead method
* autograd to calculate Jacobean

Gradient descent

1. Sets the initial approximation x0 and accuracy ε
2. Calculate
3. Increment j=j+1 and go to step 2
4. Search is stopped under one of the following criteria:

Conjugate gradient descent

1. Sets the initial approximation x = a0 and one starts in the steepest descent direction:
2. Find the step length and the next point .
3. Calculate the steepest direction
4. Compute
5. Update the conjugate direction
6. Find
7. Update the position and go to step 2
8. Search is stopped under one of the following criteria:

Newton’s method

1. Sets the initial approximation x0
2. Calculate
3. Search is stopped under one of the following criteria:

Levenberg-Marquardt algorithm (LMA)

1. Initialize values for the parameters, x, the Levenberg-Marquardt parameter λ, as well as λup and λdown to be used to adjust the damping term. Evaluate the residuals r and the Jacobean J at the initial parameter guess.
2. Calculate the metric, , and the cost gradient,
3. Evaluate the new residuals rnew­ at the point given by , and calculate the cost at the new point,
4. If , accept the step, , and set and . Otherwise, reject the step, keep the old parameter guess x and the old residuals r, and adjust
5. Check for convergence. If method has converged, return x as the best-fit parameters. If the method has not yet converged but the step aws accepted, evaluate the Jacobean J at the new parameter values. Go to the step 2.

# Results

1. Plotting graphs was carried out using method get\_plot(title\_plot, x, y, x1 = [ ], y1 = [ ]. For more details see Appendix 1.
2. For perform of the task, some functions were added:
   1. get\_sample() return lists of 100 value of x and y as function of x.
   2. D\_linear(a, b) return the sum of squares of the difference between the values of the linear approximating function and function y(x) obtained with get\_sample(). Changing the coefficients, a and b, you should minimize the return value.
   3. D\_rational(a, b) the same as function above, but approximate function is rational.
   4. linear\_approximation(a, b) and rational\_approximation(a, b) is needed to plot graph. These functions receive coefficient a, b corresponding minimum sum square of deviation and return 100 pair of x and y which are the basis for plotting approximation function. The first method for line approximation, the second – for rational approximation.

For more details see appendix 2.

Code of methods presents in Appendix from 3 to 6.

The results obtained for each methods summarize in the pivot table 1.

Table 1 – Pivot table for one-dimensional direct methods

|  |  |  |  |
| --- | --- | --- | --- |
| Linear approximation | | | |
| Method | Function elevation | Quantity of iteration | D(a, b)min |
| Gradient descent | 6 | 6 | 115.1728613 |
| Conjugate descent | 20 | 2 | 115.1726538 |
| Newton’s method | 4 | 3 | 115.1726538 |
| Levenberg-Marquardt | 7 | not provided | 115.1726538 |
| Rational approximation | | | |
| Gradient descent | 8 | 8 | 115.2525143 |
| Conjugate descent | 44 | 6 | 115.2176497 |
| Newton’s method | 5 | 3 | 115.2178727 |
| Levenberg-Marquardt | 18 | not provided | 115.2176494 |

Visualization of pivot table 1 presents on figure 1 for linear approximation and figure 2 for rational approximation.

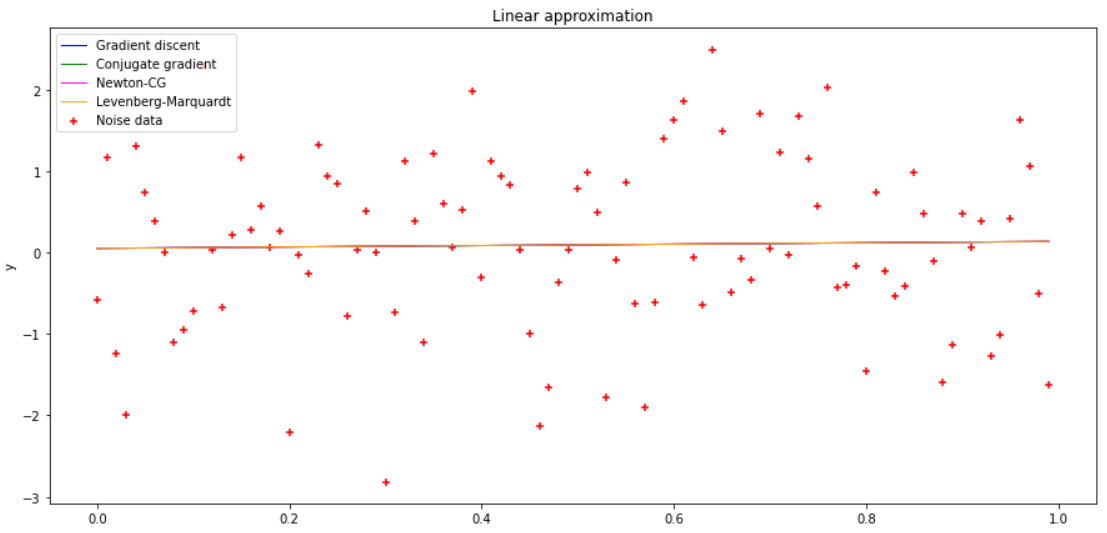


Figure 1 Linear approximation

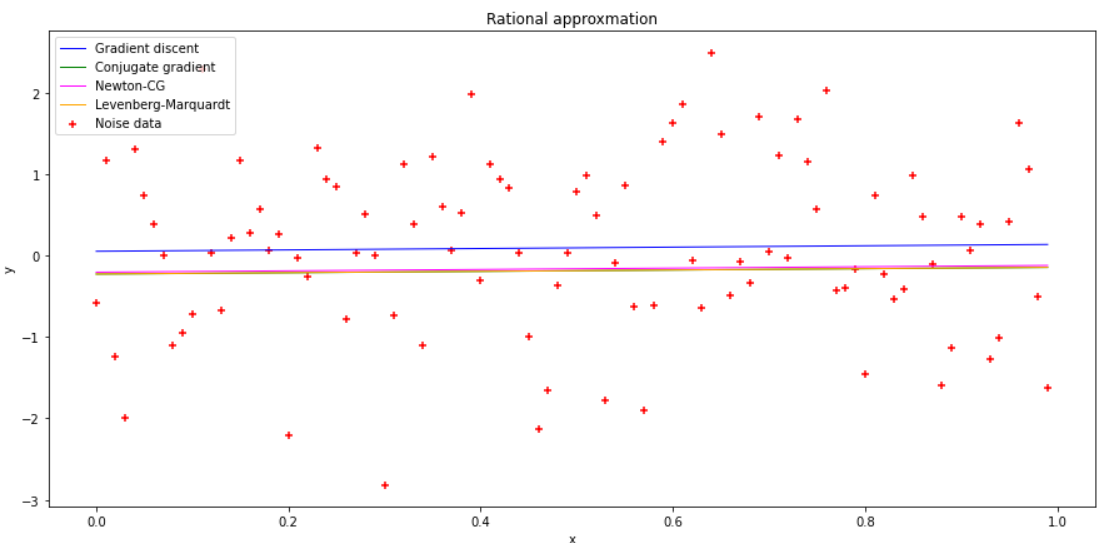


Figure 2 Rational approximation

All methods showed approximately about the same accuracy. The only difference is in the number of iterations. In the case of linear approximation, the coincidence of the results of the methods is extremely accurate. All 4 lines merged into one on the final chart. In the case of a rational approximation, the gradient descent method is out of the picture. Other methods, as in the linear case, give extremely close results. If look at pivot table again you can notice that gradient descent does not minimize target function fairly good. Its value of target function is more than other in 0,04.

# Conclusions

During the execution of the task, multi-dimensional method such as Gradient and conjugate descents, Newton’s method and Levenberg-Marquardt were applied to find coefficients linear and rational approximation functions and get corresponding graphs. The results obtained were analyzed.

Appendix 1

def get\_plot(title\_plot, x, y, x1 = [], y1 = [], x2 = [], y2 = [], x3 = [], y3 = [], x4 = [], y4 = []):

  plt.figure(figsize=(15,7)) #determed size of graph

  plt.title(title\_plot)   #give title to the graph

  plt.xlabel('x') #label of x axes

  plt.ylabel('y') #label of y axes

  plt.legend(labels=['Noise data', 'Approximity function'])

  plt.scatter(x, y, marker="+", label = "Noise data", color='red') #drew noise data

  plt.plot(x1, y1, linewidth = 1, label = "Gradient discent", color='blue') #drew approx function

  plt.plot(x2, y2, linewidth = 1, label = "Conjugate gradient", color='green') #drew approx function

  plt.plot(x3, y3, linewidth = 1, label = "Newton-CG", color='magenta') #drew approx function

  plt.plot(x4, y4, linewidth = 1, label = "Levenberg-Marquardt", color='orange') #drew approx function

  plt.legend(loc='upper left') #show legend

  plt.show() # show plot

Appendix 2

def get\_sample():

  np.random.seed(seed=60) # fixed state of randgenerator

  alpha, beta = random.randrange(0, 1), random.randrange(0, 1)

  n = 100

  x = []

  y = []

  mu, sigma = 0, 1 # mean and standard deviation

  np.random.normal(mu, sigma)

  for k in range(n):

    x.append(k / n)

    y.append(alpha \* x[k] + beta + np.random.normal(mu, sigma))

  return x, y

def D1(x = []):

    a = x[0]

    b = x[1]

    x = get\_sample()[0]

    y = get\_sample()[1]

    D\_linear = 0

    for k in range(100):

      D\_linear += ((a \* x[k] + b) - y[k]) \*\* 2

    return D\_linear

def D2(x = []):

    a = x[0]

    b = x[1]

    x = get\_sample()[0]

    y = get\_sample()[1]

    D\_rational = 0

    for k in range(100):

      D\_rational += (a / (1 + b \* x[k]) - y[k]) \*\* 2

    return D\_rational

def linear\_approximation(a, b):

  x = []

  y = []

  for k in range(100):

    x.append(k / 100)

    y.append(a \* x[k] + b)

  return x, y

def rational\_approximation(a, b):

  x = []

  y = []

  for k in range(100):

    x.append(k / 100)

    y.append(a / (1 + b \* x[k]))

  return x, y

def get\_plot(title\_plot, x, y, x1 = [], y1 = []):

  plt.figure(figsize=(15,7)) #determed size of graph

  plt.title(title\_plot)   #give title to the graph

  plt.xlabel('x') #label of x axes

  plt.ylabel('y') #label of y axes

  plt.legend(labels=['Noise data', 'Approximity function'])

  plt.scatter(x, y, marker="+", label = "Noise data", color='red') #drew noise data

  plt.plot(x1, y1, linewidth = 1, label = "Approx function", color='blue') #drew approx function

  plt.legend(loc='upper left') #show legend

  plt.show() # show plot

Appendix 3

def gradient\_discent(test\_function, eps, max\_iters, x = []):

  a = x[0]

  b = x[1]

  a\_prev = x[0] - 1

  b\_prev = x[1] - 1

  rate = 0.01 # Learning rate

  iteration = 0

  if test\_function == D1:

    while abs(D1([a,b]) - D1([a\_prev, b\_prev])) > eps and iteration < max\_iters:

      a\_prev, b\_prev = a, b #Store current x value in prev\_x

      a = a - rate \* derivative\_D1\_a(a, b)

      b = b - rate \* derivative\_D1\_b(a, b)

      iteration += 1

    print("The local minimum occurs at [{0}, {1}]".format(a, b),

iteration, D1([a,b]), sep = "\n")

  if test\_function == D2:

    while abs(D2([a,b]) - D2([a\_prev, b\_prev])) > eps and iteration < max\_iters:

      a\_prev, b\_prev = a, b #Store current x value in prev\_x

      a = a - rate \* derivative\_D1\_a(a, b)

      b = b - rate \* derivative\_D1\_b(a, b)

      iteration += 1

    print("The local minimum occurs at [{0}, {1}]".format(a, b),

iteration, D2([a,b]), sep = "\n")

Appendix 4

print(scipy.optimize.minimize(D1, [0, 0], method="CG", tol = 0.001), "\n")

print(scipy.optimize.minimize(D2, [0, 0], method="CG", tol = 0.001))

Appendix 5

print(scipy.optimize.minimize(D1, [0, 0], method="Newton-CG",

jac = jacobian(D1), tol=0.001), "\n")

print(scipy.optimize.minimize(D2, [0, 0], method="Newton-CG",

jac = jacobian(D2), tol=0.001))

Appendix 6

x0=np.array([0,0])

n = 100

x = get\_sample()[0]

y = get\_sample()[1]

def f1\_to\_opt(z,\*arg):

  a, b =z

  return [(a\*x[i] + b -y[i]) for i in range(0,n)]

from scipy.optimize import least\_squares

lm1= least\_squares(f1\_to\_opt,x0,method='lm')

print(lm1)

def f2\_to\_opt(z,\*arg):

  a, b =z

  return [(a/(1+b\*x[i])-y[i]) for i in range(0,n)]

from scipy.optimize import least\_squares

lm2= least\_squares(f2\_to\_opt,x0,method='lm')

print(lm2)