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

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. Use the one-dimensional methods of exhaustive search, dichotomy and golden section search to find an approximate (with precision ε=0.001) solution x: f(x)→min for the following functions and domains:
   1. ;
   2. ;
   3. ;

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.

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 exhaustive search, Gauss and Nelder-Mead. 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. Analyze the results obtained (in terms of number of iterations, precision, number of function evaluations, etc.).

# 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

Dichotomy method

Let f(x): [a0, b0] → R and be convex. Approximately solve the optimization problem f(x)→min by finding x\* with error ε > 0

Calculate:

then f(x1) and f(x2). Reduce the indeterminacy segment down to the segment [a1, b1] as follows:

* if then a1=a0 and b1=x2
* a1 = x1 and b1 = b0 otherwise

Search is stopped if at the current k-th iteration it holds that

Golden section method

Calculate:

then f(x1) and f(x2). Reduce the indeterminacy segment down to the segment [a1, b1] as follows:

* if then a1=a0, b1=x2 and x2 = x1
* a1 = x1, b1 = b0 and x1 = x2 otherwise

Search is stopped if at the current k-th iteration it holds that

Gauss method

Let be initial approximation. In the first iteration, find the minimum point of f as a function of the first variable, while others are fixed to get a new point

Furthermore, using x1, find the minimum point by varying only the second variable and get a new point . Start searching again by the first variable, etc.

The search is stopper under one of following criteria:

Nelder-Mead method

Parameters: -- reflection coefficient, shrinking coefficient , dilatation coefficient .

1. Preparation. Choose three point for the initial simplex. Calculate f1 = f(x1), f2 = f(x2) and f3 = f(x3)
2. Sorting. Choose three points from the simplex vertices as follows: xh with the largest value of fh , xg with the second-large value of fg and xi with the smallest value of fi. The goal of the forthcoming manipulation is decreasing of fh at least.
3. Gravity center. Find gravity center for all points except xh: .
4. Reflection. Reflect the point xh with respect to xc with the coefficient : . Calculate f­r = f(xr)
5. Decision.
   1. If then the direction is right, and we can dilate: calculate and fe = f(xe)
   2. If the simplex can be extended: set xh=xe and go to step 7.
   3. If then we moved too far: set xh = xr and go to step 7.
   4. If then the choice of new point is good: set xh = xr and go to step 7
   5. If then the exchange xr and xh and fr and fh. After this, go to step 6

As result

1. Shrinking. Calculate and fs = f(xs)
   1. If then set xh = xs and go to step 7
   2. If then the initial point is the best

Shrink the simplex globally as follows:

1. Convergence check. Check the mutual closeness of the simplex vertices. If required precision is not achieved, go to 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. To find an approximate solution x with exhaustive search and precision ε=0.001 such that f(x)→min, it uses brute\_force(test\_function, a, b, eps). See code in Appendix 2. The results obtained are shown in table 1.

Table 1 – Result obtained by brute force

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Function elevation | Quantity of iteration | X |
|  | 1000 | 1000 | 0 |
|  | 1000 | 1000 | 0 |
|  | 990 | 990 | -0.21722461258083 |

1. The same task has been performed with dichotomy method. See code in Appendix 3. The results are presented in the table 2.

Table 2 – Result obtained with dichotomy method

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Function elevation | Quantity of iteration | X |
|  | 16 | 8 | 8.772721048e-08 |
|  | 16 | 8 | 0.001220703125 |
|  | 16 | 8 | -0.03855764695551 |

1. The same task has been performed with golden section method. See code in Appendix 4. The obtained results are presented in the table 3.

Table 3 – Result obtained with golden section method

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Function elevation | Quantity of iteration | X |
|  | 12 | 10 | 6.718631248194e-08 |
|  | 12 | 10 | 0.0024391856267351 |
|  | 12 | 10 | -0.217156635949684 |

The results obtained for each one-dimensional direct method summarize in the pivot table 4.

Table 4 – Pivot table for one-dimensional direct methods

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Method | Function elevation | Quantity of iteration | x |
| Brute force | 1000 | 1000 | 0 |
| Dichotomy method | 16 | 8 | 8.772721048e-08 |
| Golden section method | 12 | 10 | 6.718631248194e-08 |
|  | | | |
| Brute force | 1000 | 1000 | 0 |
| Dichotomy method | 16 | 8 | 0.001220703125 |
| Golden section method | 12 | 10 | 0.0024391856267351 |
|  | | | |
| Brute force | 990 | 990 | -0.21722461258083 |
| Dichotomy method | 16 | 8 | -0.03855764695551 |
| Golden section method | 12 | 10 | -0.217156635949684 |

In summary, brute force has achieved fairly good results for first and second functions. However, that took it a lot more numbers of f-calculating. This is because method has calculated all value of test function and then determine minimum value among these.

Dichotomy method was bit faster than golden section: 8 iterations versus 10, but it needed more numbers f-calculation. Absence of big difference in numbers of iteration can be explain by math-simplicity of function. Golden section method will be faster when function is more difficult thanks system features of this method.

1. For part 2 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 5.

1. To find an coefficients a and b of linear approximate function x with exhaustive search and precision ε=0.001, it uses brute\_force\_mult\_dim(test\_function, print\_report = False). See code in appendix 6. The results obtained are shown in figure 1 for linear approximation and figure 2 for rational approximation.

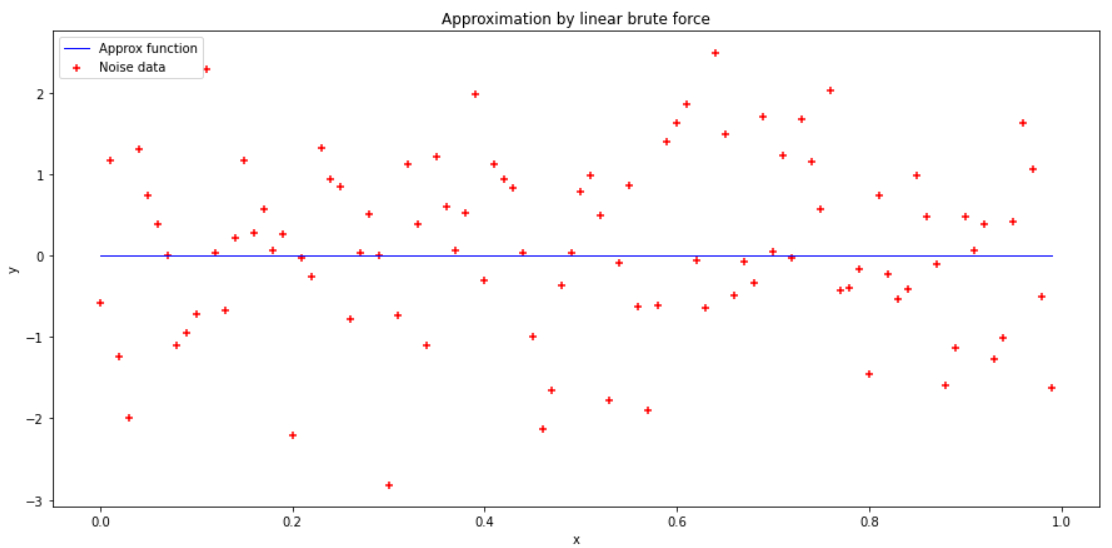


Figure Linear approximation by brute force

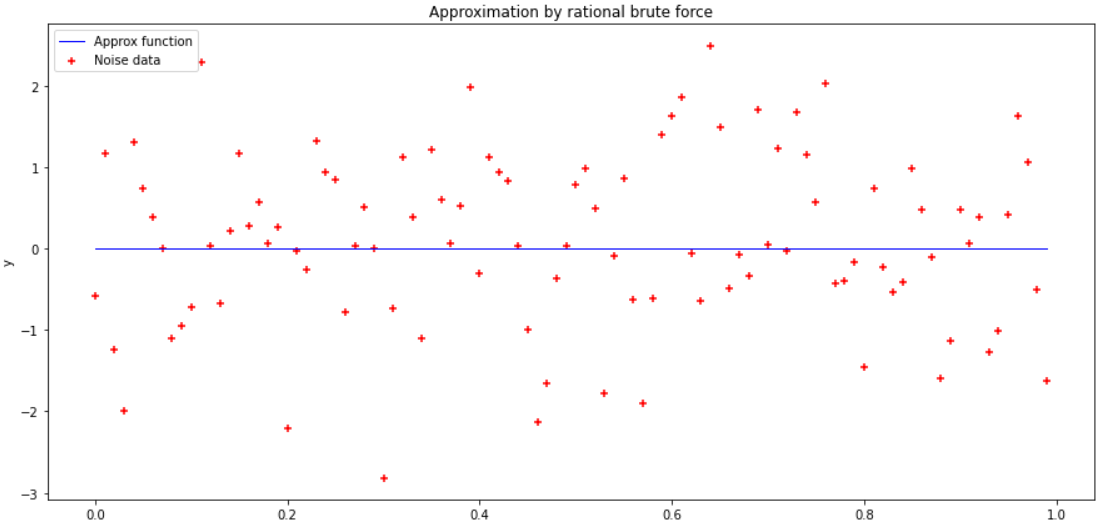


Figure Rational approximation by brute force

1. To find an coefficients a and b of linear approximate function x with gauss method and precision ε=0.001, it uses gauss\_method(test\_function, init\_approx, eps, max\_iteration). See code in appendix 7. The results obtained are shown in figure 3 for linear approximation and figure 4 for rational approximation.

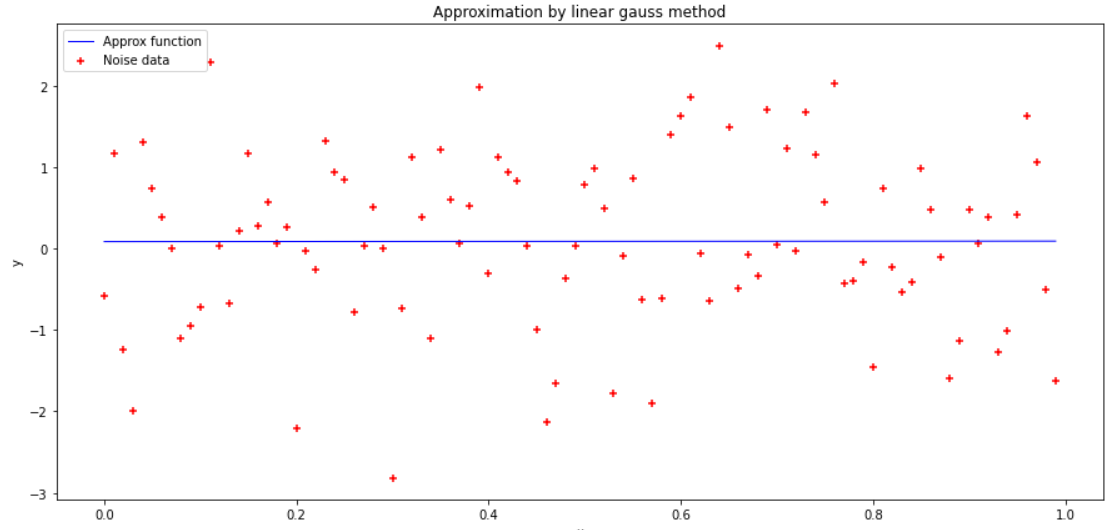


Figure Linear approximation by gauss method

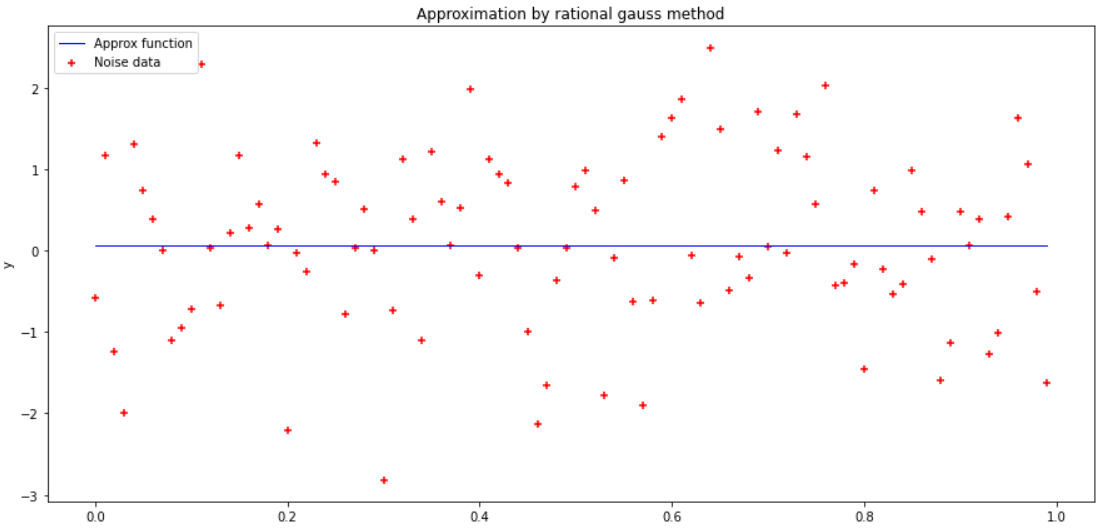


Figure Rational approximation by gauss method

1. To find an coefficients a and b of linear approximate function x with Nelder-Mead method and precision ε=0.001, it uses standard function scipy.optimize.minimize() from SciPy library. See code in appendix 8. The results obtained are shown in figure 5 for linear approximation and figure 6 for rational approximation.

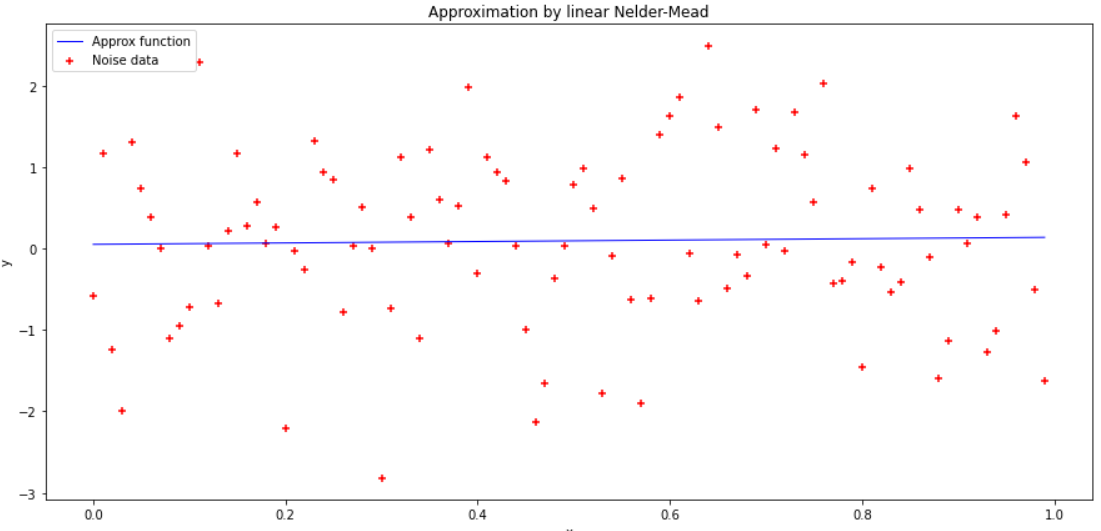


Figure Linear approximation by Nelder-Mead method

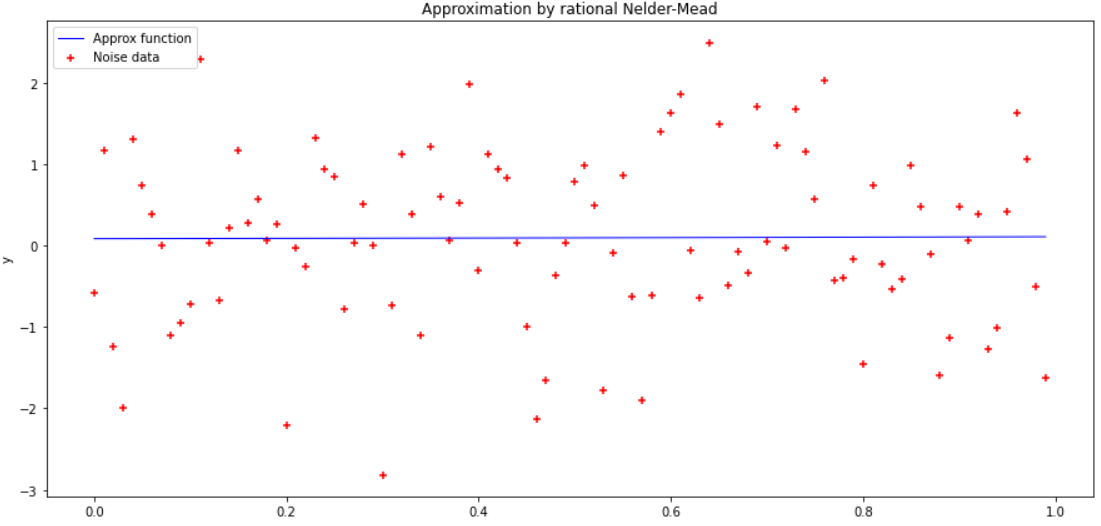


Figure Rational approximation by Nelder-Mead method

1. The results obtained for each multidimensional direct method summarize in the pivot table 5

Table 5 – Pivot table for multidimensional direct methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Linear approximate | | Rational approximate | |
| D(a,b) | Quantity of iteration | D(a,b) | Quantity of iteration |
| Brute force | 116.10068 | 10000 | 116.10068 | 10000 |
| Gauss | 115.23034 | 20180 | 115.32256 | 30127 |
| Nelder-Mead | 115.17265 | 51 | 115.21764 | 62 |

The fastest was the Nelder-Mead method. Gauss method gave almost as well results as Nelder-Mead method did, but it was doing this slower. The most non-performance of all these methods was brute force. It is also less accurate.

# Conclusions

During the execution of the task, approximate minimum of three one-dimensional functions was found with brute force, dichotomy and golden section methods. Multi-dimensional method such as Gauss method, Nelder-Mead and again brute force were applied to find coefficients linear and rational approximation functions and get corresponding graphs.

Appendix 1

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

  #sns.regplot(x, y)

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

  plt.show() # show plot

Appendix 2

def brute\_force(test\_function, a, b, eps):

  n = int((b - a) / eps)

  x = 0

  iteration = 0

  min\_test\_function = test\_function(a)

  for k in range(n):

    x = a + k \* (b - a) / n

    if test\_function(x) < min\_test\_function:

      min\_test\_function = test\_function(x)

    iteration += 1

  print("Minimum of {0} for x in range [{1}, {2}] is".format(

test\_function, a, b), min\_test\_function,

"\t Quantity of iterations is ", iteration)

Appendix 3

def dichotomy(test\_function, a, b, eps, max\_iteration):

  iteration = 0

  function\_elevation = 0

  delta = eps / 2

  while abs(a - b) >= eps and iteration != max\_iteration:

    x1 = (a + b - delta) / 2

    x2 = (a + b + delta) / 2

    if test\_function(x1) <= test\_function(x2):

      b = x2

    else:

      a = x1

    iteration += 1

    function\_elevation += 2

  min\_test\_fuction = test\_function((a + b ) / 2)

  print("Minimum of {0} for x in range [{1}, {2}] is ".format(test\_function, a, b), min\_test\_fuction, "\n",

        "Quantity of iterations is ", iteration, "\n",

        "Quantity of function elevating is ", function\_elevation, "\n",

sep="")

Appendix 4

def golden\_section(test\_function, a, b, eps, max\_iteration):

  iteration = 0

  dict\_value = {}

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

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

  dict\_value[x1] = test\_function(x1)

  dict\_value[x2] = test\_function(x2)

  while abs(a - b) >= eps and iteration != max\_iteration:

    if dict\_value.get(x1) <= dict\_value.get(x2):

      b = x2

      x2 = x1

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

      dict\_value[x1] = test\_function(x1)

    else:

      a = x1

      x1 = x2

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

      dict\_value[x2] = test\_function(x2)

    iteration += 1

  min\_test\_fuction = test\_function((a + b ) / 2)

  print("Minimum of {0} for x in range [{1}, {2}] is ".format(

test\_function, a, b), min\_test\_fuction, "\n",

        "Quantity of iterations is ", iteration, "\n",

        "Quantity of function elevating is ", len(dict\_value), "\n",

sep="")

Appendix 5

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 D\_linear(a, b):

  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 D\_rational(a, b):

  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

Appendix 6

def brute\_force\_mult\_dim(test\_function, print\_report = False):

  iteration = 0

  a\_min, b\_min = 0, 0

  min\_test\_function = test\_function(0, 0)

  for a in range(100):

    for b in range(100):

      iteration += 1

      if test\_function(a, b) < min\_test\_function:

        min\_test\_function = test\_function(a, b)

        a\_min, b\_min = a, b

  if print\_report == True:

    print("Minimum of {0} for x in range [{1}, {2}] is".format(

test\_function, a, b),

min\_test\_function, "\n"

        "Coefficients of approximity function: a = {0}, b = {1}".format(

a\_min, b\_min), "\n"

        "Quantily of iteration is {0}".format(iteration))

  return a\_min, b\_min

Appendix 7

def gauss\_method(test\_function, init\_approx, eps, max\_iteration):

  a = init\_approx[0]

  b = init\_approx[1]

  a\_min, b\_min, x, iteration = 0, 0, 0, 0

  min\_test\_function = test\_function([a, b])

  while iteration < max\_iteration:

    for b in np.arange(init\_approx[1] - 5, init\_approx[1] + 5, eps/2):

      if test\_function([a, b]) < min\_test\_function:

        if abs(test\_function([a, b]) - min\_test\_function) < eps:

          min\_test\_function = test\_function([a, b])

          b\_min = b

          break

        min\_test\_function = test\_function([a, b])

        b\_min = b

      iteration += 1

    for a in np.arange(init\_approx[1] - 5, init\_approx[1] + 5, eps/2):

      if test\_function([a, b]) < min\_test\_function:

        if abs(test\_function([a, b]) - min\_test\_function) < eps:

          min\_test\_function = test\_function([a, b])

          a\_min = a

          break

        min\_test\_function = test\_function([a, b])

        a\_min = a

      iteration += 1

  return a\_min, b\_min, iteration

Appendix 8

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

print(scipy.optimize.minimize(D1, [0, 0], method="Nelder-Mead"))

print(scipy.optimize.minimize(D2, [0, 0], method="Nelder-Mead"))