FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 4

“Algorithms for unconstrained nonlinear optimization. Stochastic and metaheuristic algorithms”

Performed by

Victor Voronin

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2020

# Goal

The use of stochastic and metaheuristic algorithms (Simulated Annealing, Differential Evolution, Particle Swarm Optimization) in the tasks of unconstrained nonlinear optimization and the experimental comparison of them with Nelder-Mead and Levenberg-Marquardt algorithms**.**

# Formulation of the problem

Generate the noisy data , where k=0…1000, according to the rule:

Where are values of a random variable with standard normal distribution. Approximate the data by the rational function

by means of least squares through the numerical minimization of the following function:

To solve the minimization problem, use Nelder-Mead algorithm, Levenberg-Marquardt algorithm and at least two of the methods among Simulated Annealing, Differential Evolution and Particle Swarm Optimization. If necessary, set the initial approximations and other parameters of the methods. Use ε=0.001 as the precision; at most 1000 iterations are allowed. Visualize the data and the approximants obtained in a single plot. Analyze and compare 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

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.

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.

Simulated annealing

Let be initial approximation. At each iteration :

1. Choose is certain rule
2. If , then if If , then with probability , where Tk is a decreasing non-negative sequence
3. Stop if Tk = 0

Differential evolution

1. Choose CR ∈ [0, 1], the crossover probability, F ∈ [0, 2], the differential weight, and NP ≥ 4, the population size. Let x ∈ Rn denote an agent in the population.
2. Until a termination criterion is met:
   1. Randomly pick NP agents x
   2. Pick three distinct agents a, b, c from the population, different from x.
   3. Compute the trial vector as follows. For i=1…n, pick ri ∈ U(0, 1). If ri < CR, then yi = ai + F(bi − ci), otherwise yi = xi.
   4. If f(y)<f(x), then replace x with the trial vector y, otherwise keep x.
3. Pick the best agent from the population and return it as the best-found solution.

# 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 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.

1. To find an coefficients a and b of linear and rational approximate functions x with Nelder-Mead method and precision ε=0.001, it uses standard function scipy.optimize.minimize() from SciPy library. See code in appendix 3.
2. To find an coefficients a and b of linear and rational approximate functions x with Levenberg-Marquardt and precision ε=0.001, it uses standard function scipy.optimize.minimize() from SciPy library. See code in appendix 4.
3. To find an coefficients a and b of linear and rational approximate functions x with simulated annealing, it uses standard function scipy.optimize.minimize() from SciPy library. See code in appendix 5.
4. To find an coefficients a and b of linear and rational approximate functions x with differential evolution, it uses standard function scipy.optimize.minimize() from SciPy library. See code in appendix 6.

Table 1 – Pivot table for multidimensional direct methods

|  |  |  |
| --- | --- | --- |
| Method | Linear approximate | |
| D(a,b) | Quantity of iteration |
| Nelder-Mead | 136001.2699706518 | 441 |
| LMA | 136006.66133510595 | 250 |
| Simulated annealing | 136001.27760833985 | 6 |
| Differential evolution | 136472.0845665622 | 4 |

Visualization of approximation function with different methods present on figure 1.

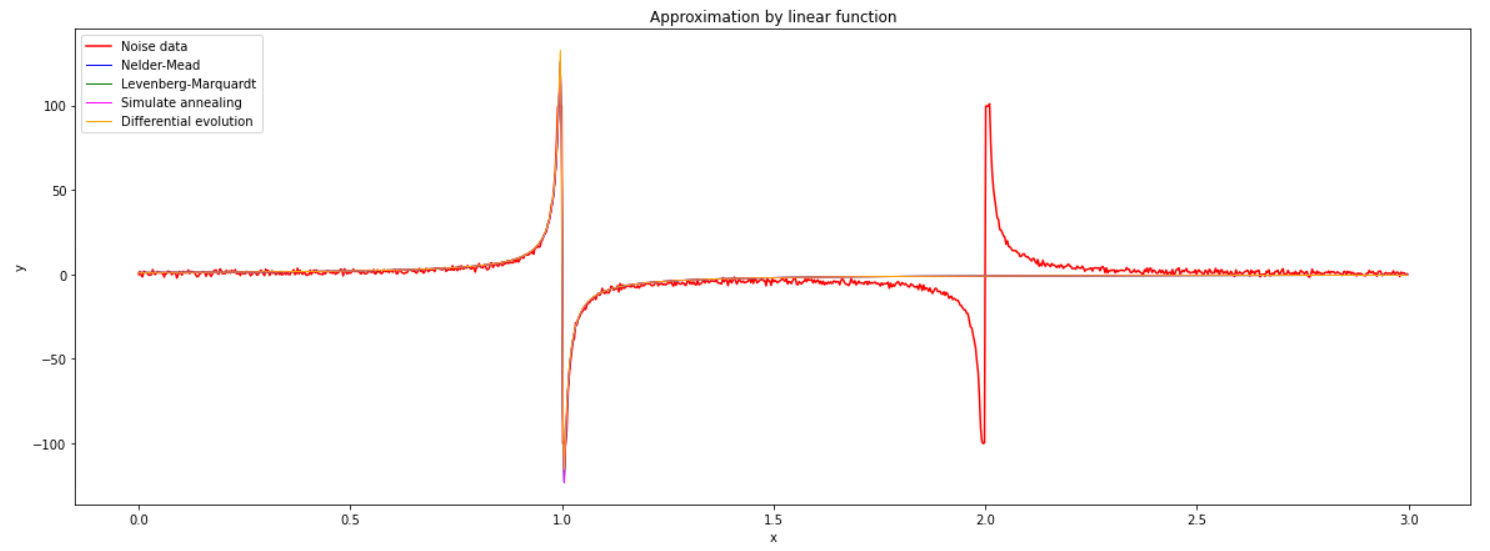


Figure 1 Approximation function obtained with each methods and noise data

All methods showed approximately about the same accuracy and could recognize the first break not noticing the second. Therefore, are all 4 lines merged into one on the final chart. The only difference is in the number of iterations.

In general, stochastic algorithms show faster performance for given objective functions (relatively simple ones) than numerical methods like the Nelder-Mead algorithm.

# Conclusions

During the execution of the task, multi-dimensional methods such Levenberg-Marquardt, Nelder-Mead, simulate annealing, differential evolution 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 = []):

  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 2

def f1(x):

  return 1 / (x \*\* 2 - 3 \* x + 2)

def get\_sample():

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

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

  n = 1000

  x = []

  y = []

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

  np.random.normal(mu, sigma)

  for k in range(n):

    if f1(3 \* k / n) < -100:

      x.append(3 \* k / n)

      y.append(-100 + np.random.normal(mu, sigma))

    elif -100 <= f1(3 \* k / n) <= 100:

      x.append(3 \* k / n)

      y.append(f1(x[k]) + np.random.normal(mu, sigma))

    else:

      x.append(3 \* k / n)

      y.append(100 + np.random.normal(mu, sigma))

  return x, y

def D1(z, \*arg):

  n = 1000

  a,b,c,d=z

  x1 = get\_sample()[0]

  y = get\_sample()[1]

  return sum([(((a\*x1[i]+b)/(x1[i]\*\*2+c\*x1[i]+d))-y[i])\*\*2 for i in range(0,n)])

def D1\_1(z,\*arg):

    n = 1000

    a, b, c, d =z

    x = get\_sample()[0]

    y = get\_sample()[1]

    return [((a\*x[i]+b)/(x[i]\*\*2+c\*x[i]+d))-y[i] for i in range(0,n)]

def approximation(a, b, c, d):

  n = 1000

  x = []

  y = []

  for k in range(n):

    x.append(3 \* k / n)

    y.append((a\*x[k] + b) / (x[k]\*\*2 + c\*x[k] + d))

  return x, y

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

  plt.figure(figsize=(20,7)) #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.plot(x, y, marker="", label = "Noise data", color='red') #drew noise data

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

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

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

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

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

  plt.show() # show plot

Appendix 3

x0=np.array([0.01,0.01,0.01,0.01])

print(scipy.optimize.minimize(D1, x0, method="Nelder-Mead"))

Appendix 4

x0=np.array([0.001,0.001,0.001,0.001])

from scipy.optimize import least\_squares

lm= least\_squares(D1\_1, x0, method='lm')

print(lm)

Appendix 5

x0=np.array([0.01,0.01,0.01,0.01])

sm = scipy.optimize.basinhopping(D2, x0, niter=1000)

print(sm)

Appendix 6

bounds = [(0,1),(0,1),(0,1),(0,1)]

print(scipy.optimize.differential\_evolution(D1, bounds, maxiter=1000))