Brief Report

Automated Two Stage Adjustable Ratio Simplex Algorithm for Two Parameter Optimization

Wenfa Ng *

Posted Date: 7 January 2026

doi: 10.20944/preprints202601.0415.v1

Keywords: Nelder Mead algorithm; simplex optimization; reflection step; number of iterations; contraction and extension steps

 doi:10.20944/preprints202601.0415.v1

Brief Report

Automated Two Stage Adjustable Ratio Simplex Algorithm for Two Parameter Optimization

Wenfa Ng

Citizen scientist, Singapore, Email: ngwenfa771@hotmail.com

Abstract

Multivariable optimization is an essential mathematical exercise in daily engineering design and troubleshooting. To this end, simplex multivariable optimization method is a powerful optimization approach that has served many engineering disciplines well over the years. One such simplex algorithm is the Nelder Mead algorithm. But, the reflection step of the Nelder Mead algorithm may increase the number of iterations needed to arrive at the optimal point, as it reflects the starting point to the opposite side of the function. This work proposes an automated two-stage adjustable ratio simplex optimization method that first search within and around the optimization surface for a good starting point, followed by a narrow and more refined search for the optimal point. For both stages of the new simplex algorithm, only contraction and extension steps are used, and this helps to remove possible oscillatory effects common to other simplex algorithms as the iterations progress. Demonstrative use of the new simplex algorithm on optimizing the coefficients of a quadratic function reveals good accuracy and speed as compared to the Nelder Mead algorithm which uses significantly more iterations. Future testing should be conducted with other optimization functions as well as objective functions. Interested readers are invited to explore and expand on the work reported herein.

Keywords: Nelder Mead algorithm; simplex optimization; reflection step; number of iterations; contraction and extension steps; **Subject areas:** mathematics; biomedical engineering; computer science

Introduction

Simplex is a powerful multivariable optimization method in use in many engineering disciplines to find optimal values for a variety of functions and problems. While there are many variants and implementations of simplex algorithms, the most common remains the Nelder Mead algorithm. [1]

In the Nelder Mead algorithm, the reflection step and shrinkage steps are prominent vertices movement steps to explore the other side of the optimization surface and to shrink the simplex to a good point, respectively.[2,3] This preprint describes an attempt to use an alternative approach to explore the optimization surface from an initial guess.

Specifically, my two-stage adjustable ratio simplex algorithm primarily uses an initial search for a good starting point for the second stage narrow (refined) search for the optimal point. In the first stage, a series of contraction of different percentages, and a series of extension of different multiplier are used to search the optimization surface for a better starting point than the initial guess. This step does suffer from high computational expense, especially if the objective function is to minimize the mean of sum of error.

Following the first stage, an iterative second stage uses a series of contraction and extension steps to further refine the initial good point selected by the first stage to finally narrow down to the optimal point. At the end of each iteration, a simplex is defined around the good point, and function evaluations are made to ascertain if the good point is still some distance away from the optimal point. Finally, a simplex reordering exercise rearranges the vertices such that the point with the smallest function evaluation is made the good point for the start of the next iteration of the narrow search.

 doi:10.20944/preprints202601.0415.v1

Computational implementation of the algorithm

The new simplex optimization is coded in Python using the Spyder 6 editor with Python 3.12.11 as computational engine. Laptop used for this research is a Lenovo Ideapad Slim 3 Windows 11 Home (64 bit) with AMD Ryzen 5 (7520U) processor running at 2.80 GHz, with AMD Radeon graphics (2 GB video RAM), 16 GB of RAM, and 512 GB of SSD storage.

Algorithmic implementation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
# Defining simplex and simulation variables
shrink_ratio = 0.5
tolerance = 1e-5
num_exploration = 10 # Number of times to explore good point
max_iter = 20 # Number of iterations to perform narrow search
#Define original quadratic function
def original_function(x):
    y = 3*x**2 + 6*x + 10

    return y
# Calculating the original function variation over x value
x_value = []
y_original = []
max_x_value = 100
for i in range (0, max_x_value):
    y_value = original_function(i)

    x_value.append(i)
    y_original.append(y_value)
# Defining the optimization function
def optimize_fun(x, guess_value):
    h = guess_value[0]
    g = guess_value[1]

    y_evaluate = h*x**2 + g*x + 10

    return y_evaluate
#Defining the optimization objective function
def obj_fun(guess):
    k1 = len(x_value)
    error_catalogue = []
    calculations = []
    #Code to calculate mean square error
```

 doi:10.20944/preprints202601.0415.v1


```
    sum_error = 0;
    for i in range (0, k1, 1):
        error = (y_original[i] - optimize_fun(x_value[i], guess))**2
        error_catalogue.append(error)
        calculations.append(i)

        sum_error = sum_error + error

    mean_sum_error = sum_error/k1
    return mean_sum_error
#This function accepts a simplex
#data structure as an input
#and returns another simplex with the
#vertices ordered from good to worst
def order_vertices(simplex):
    simplex_vertices = simplex['vertices']
    simplex_evals = simplex['evals']

    #Index of the good point, where the
    #minimum evaluation is
    index_G = np.argmin(simplex_evals)
    index_W = np.argmax(simplex_evals)
    index_A = 3 - index_G - index_W

    G = simplex_vertices[index_G]
    A = simplex_vertices[index_A]
    W = simplex_vertices[index_W]

    G_eval = simplex_evals[index_G]
    A_eval = simplex_evals[index_A]
    W_eval = simplex_evals[index_W]

    return {
        'vertices' : np.array([G,A,W]),
        'evals' : np.array([G_eval,A_eval,W_eval])
        }

# Putting in the initial guess
initial_guess = [5,10]

#Assigning initial guess to optimization parameters
A_param = initial_guess[0]
```

 doi:10.20944/preprints202601.0415.v1


```
B_param = initial_guess[1]
#Initial simplex
A = np.array([A_param, B_param]) #Initial point of the simplex
B = np.array([A[0] + 0.25*A[0],A[1]])
C = np.array([A[0], A[1]+0.25*A[1]])
simplex = {
    'vertices' : np.array([A,B,C]),
    'evals' : np.array([obj_fun(A),obj_fun(B),obj_fun(C)])
    }
simplex = order_vertices(simplex)
#Defining k_a and k_e plotting variables
A_param_log = []
B_param_log = []
num_cycle = []
#Initialing the variables
G = simplex['vertices'][0]
A = simplex['vertices'][1]
W = simplex['vertices'][2]
G_eval = simplex['evals'][0]
A_eval = simplex['evals'][1]
W_eval = simplex['evals'][2]
G_eval_current = G_eval
A_param_log.append(A_param)
B_param_log.append(B_param)
num_cycle.append(0)
G_eval_previous = G_eval_current
# Attempt to explore the parameter space over longer value ranges
for j   in range (0, num_exploration):

    # Different levels of contraction
    C_20_percent = 0.2*G
    C_20_eval = obj_fun(C_20_percent)
    C_40_percent = 0.4*G
    C_40_eval = obj_fun(C_40_percent)
    C_70_percent = 0.7*G
    C_70_eval = obj_fun(C_70_percent)
    contraction_pt = [C_20_percent, C_40_percent, C_70_percent]
    contraction_pt_eval = [C_20_eval, C_40_eval, C_70_eval]
    index = np.argmin(contraction_pt_eval)
    C = contraction_pt[index]
    C_eval = contraction_pt_eval[index]
    # Different levels of extension
```

 doi:10.20944/preprints202601.0415.v1


```
    E_150_percent = 1.5*G
    E_150_eval = obj_fun(E_150_percent)
    E_200_percent = 2.0*G
    E_200_eval = obj_fun(E_200_percent)
    E_400_percent = 4*G
    E_400_eval = obj_fun(E_400_percent)
    extension_pt = [E_150_percent, E_200_percent, E_400_percent]
    extension_pt_eval = [E_150_eval, E_200_eval, E_400_eval]
    index = np.argmin(extension_pt_eval)
    E = extension_pt[index]
    E_eval = extension_pt_eval[index]
    if C_eval < E_eval:
        G = C
        G_eval = C_eval
        G_eval_current = G_eval
    else:
        G = E
        G_eval = E_eval
        G_eval_current = E_eval

# Starting narrow search for final optimal point
for i in range(1,max_iter):

    G_eval_previous = G_eval_current

    #Keep track of A_param and B_param for the plot

    A_param_log.append(G[0])
    B_param_log.append(G[1])
    num_cycle.append(i)



    #We will try a contraction

    # Different levels of contraction
    C_40_percent = 0.4*G
    C_40_eval = obj_fun(C_40_percent)

    C_60_percent = 0.6*G
    C_60_eval = obj_fun(C_60_percent)
```

 doi:10.20944/preprints202601.0415.v1


```
C_75_percent = 0.75*G
C_75_eval = obj_fun(C_75_percent)

contraction_pt = [C_40_percent, C_60_percent, C_75_percent]
contraction_pt_eval = [C_40_eval, C_60_eval, C_75_eval]

index = np.argmin(contraction_pt_eval)
C = contraction_pt[index]
C_eval = contraction_pt_eval[index]


if (C_eval < G_eval): #Is f(T)<f(R)?
        G = C
        G_eval = C_eval
        G_eval_current = G_eval

        simplex['vertices'][0] = G
        simplex['evals'][0] = G_eval
        print('Contraction')

        # Updated simplex
        G = np.array([G[0], G[1]]) #Initial point of the simplex
        A = np.array([G[0] + 0.25*G[0],G[1]])
        W = np.array([G[0], G[1]+0.25*G[1]])

        simplex = {
            'vertices' : np.array([G,A,W]),
            'evals' : np.array([obj_fun(G),obj_fun(A),obj_fun(W)])
            }

if (G_eval < C_eval):

    E_120_percent = 1.2*G
    E_120_eval = obj_fun(E_120_percent)

    E_150_percent = 1.5*G
    E_150_eval = obj_fun(E_150_percent)

    E_180_percent = 1.8*G
    E_180_eval = obj_fun(E_180_percent)

    extension_pt = [E_120_percent, E_150_percent, E_180_percent]
```

 doi:10.20944/preprints202601.0415.v1


```
            extension_pt_eval = [E_120_eval, E_150_eval, E_180_eval]

            index = np.argmin(extension_pt_eval)
            E = extension_pt[index]
            E_eval = extension_pt_eval[index]


            if (E_eval < G_eval):
                G = E
                G_eval = E_eval
                G_eval_current = G_eval
                simplex['vertices'][0] = G
                simplex['evals'][0] = G_eval
                print('Extension')

                # Updated simplex
                G = np.array([G[0], G[1]]) #Initial point of the simplex
                A = np.array([G[0] + 0.25*G[0],G[1]])
                W = np.array([G[0], G[1]+0.25*G[1]])

                simplex = {
                    'vertices' : np.array([G,A,W]),
                    'evals' : np.array([obj_fun(G),obj_fun(A),obj_fun(W)])
                    }


        #re-order the simplex vertices
        simplex = order_vertices(simplex)

        #Check convergence
        error = G_eval_current - G_eval_previous
        error_magnitude = np.abs(error)

        if error_magnitude < tolerance:
            break

print(G, ' in ', i+1, ' iterations')
#Use of Nelder Mead for curve fitting
#Use the built-in Nelder-Mead function
result = minimize(obj_fun, initial_guess,\
                  method='Nelder-Mead')
print(result.x, ' in ', result.nit, ' iterations')
```

 doi:10.20944/preprints202601.0415.v1


```
print(result.x[0])
print(result.x[1])
#Reconstruct current values using parameters from fast simplex optimization
y_value_NM = []
k1 = len(x_value)
for j in range(0, k1):
    NM_prediction = optimize_fun(x_value[j], result.x)
    y_value_NM.append(NM_prediction)
#Plotting functions
plt.figure(1)
plt.plot(x_value,y_original,'b.')
plt.plot(x_value, y_value_NM, 'r-')
plt.xlabel("X value")
plt.ylabel("Y value")
plt.show()
plt.figure(2)
plt.plot(num_cycle, A_param_log,'b.-', label = "A_param value variation")
plt.plot(num_cycle, B_param_log, 'r-', label = "B_param value variation")
plt.xlabel("Number of iterations")
plt.ylabel('Variation of A_param and B_param')
plt.title("Variation of A_param and B_param with iterations")
plt.legend()
plt.show()
```

Results and Discussion of demonstrative use of algorithm

The two stage adjustable ratio simplex algorithm was tasked to optimize the coefficients of a quadratic function of $3x^2 + 6x + 10$ with h as the optimization parameter of x^2 and g as optimization parameter of x. The comparative algorithm for this exercise is Nelder Nead simplex algorithm from the Scipy optimize library of Python.

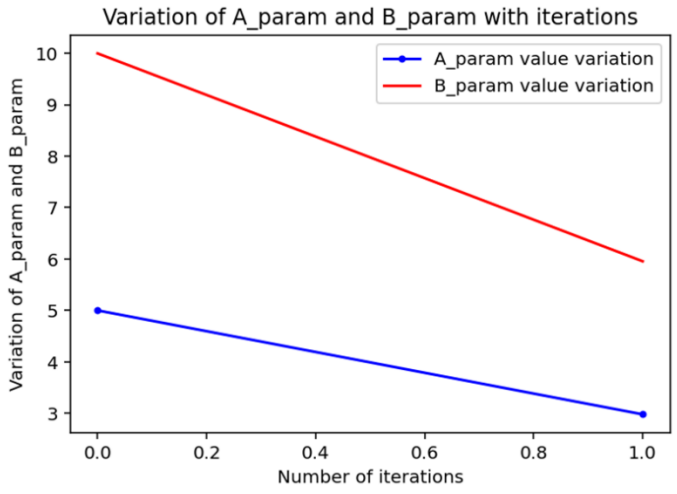

Figure 1. Variation of A parameter (h) and B parameter (g) during the narrow search phase of the two stage adjustable ratio simplex algorithm developed in this work.

 doi:10.20944/preprints202601.0415.v1

Demonstrative use of the two stage adjustable ratio simplex algorithm with initial guess of h = 5 and g = 10, reveals a 2 iteration narrow search after a 10 iteration search for a good starting point. Final optimal value obtained is h = 2.97, and g = 5.96, which is comparable to the h = 2.99, and g = 6.00 values obtained by the Nelder Mead algorithm after 64 iterations.

Conclusion

This work attempts to develop a two-stage adjustable ratio simplex algorithm different from the Nelder Mead algorithm to help reduce the number of iterations incurred due to the reflection steps in the Nelder Mead algorithm. The new simplex algorithm works by first using a series of contraction and extension steps to find a good starting point for a second stage more refined and narrower search for the optimal point. Such an approach allows the simplex algorithm to at least search a significant portion of the optimization surface before settling on a good starting point for a narrow search for the optimal point. Demonstrative use of the two stage adjustable ratio simplex method on optimizing the coefficients of a quadratic function demonstrates good performance in comparison to the robust Nelder Mead algorithm. But, more test cases and optimization functions are needed to further examine the functional properties of the new simplex algorithm. Interested readers are invited to explore and expand on the work reported herein.

Funding: No funding was used in this work.

Conflicts of interest: The author declares no conflicts of interest.

References

1. A. Galántai, "The Nelder–Mead Simplex Algorithm Is Sixty Years Old: New Convergence Results and Open Questions," *Algorithms*, vol. 17, no. 11, p. 523, 2024, doi: 10.3390/a17110523.
2. F. Gao and L. Han, "Implementing the Nelder-Mead simplex algorithm with adaptive parameters," *Comput. Optim. Appl.*, vol. 51, no. 1, pp. 259–277, Jan. 2012, doi: 10.1007/s10589-010-9329-3.
3. P. C. Wang and T. E. Shoup, "Parameter sensitivity study of the Nelder–Mead Simplex Method," *Adv. Eng. Softw.*, vol. 42, no. 7, pp. 529–533, July 2011, doi: 10.1016/j.advengsoft.2011.04.004.