# Interactive Random Search Software Summary

## Background information

The Random Walk Algorithm is a stochastic optimization method that iteratively explores the solution space by making random steps from the current position to seek the optimum of a function. It starts from an initial guess and updates this guess by adding a randomly generated vector, allowing the exploration of the search space without relying on gradient information. This method is characterized by its simplicity, using a prescribed step size and direction to navigate through the search space, which can be particularly effective in complex landscapes or when gradient information is unavailable or unreliable.

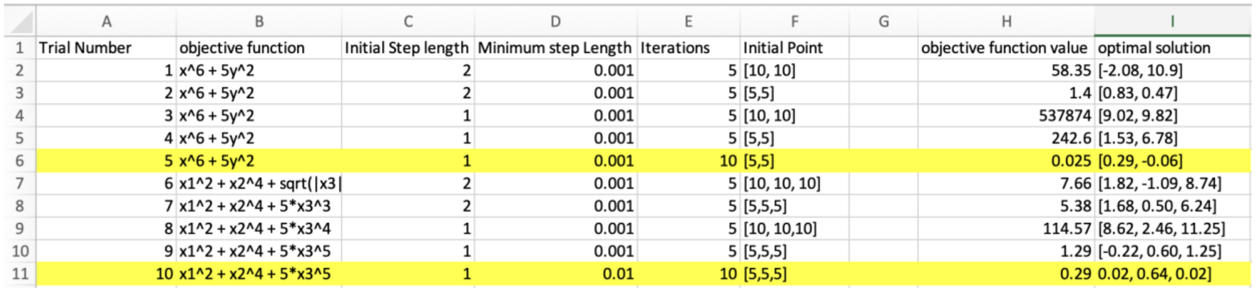
## Software Introduction

Interactive Random Search Software is a program written by Python based on the Random Walk Method. This program allows the clients to completely self-define function and parameters including the number of variables, **objective function**, minimum step length, iteration number, and initial step length. To compare the performance, I have tested several objective functions on Newton's method and the Steepest Descent, where the objective function and parameters need to be modified within the program.

## Experiment Setting and Results

*Experiments on Parameters*

For experiments on the performance of the software, I have used two objective functions x^6 + 5y^2, and x1^2 + x2^4 + sqrt(|x3|) with different parameters. Two objective functions have a minimum value of 0, which is used for comparing the results in different groups of parameters. The results are listed in Fig 1, and the procedure of experiments is put in the appendix.

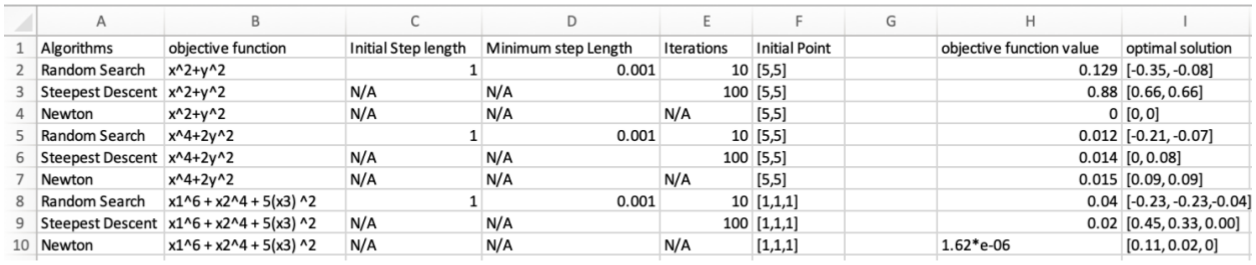


| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Trial Number | objective function | Initial Step length | Minimum step Length | Iterations | Initial Point | | objective function value | optimal solution |
| 2 | 1 | x^6 + 5y^2 | 2 | 0.001 | 5 | [10, 10] | | 58.35 | [-2.08, 10.9] |
| 3 | 2 | x^6 + 5y^2 | 2 | 0.001 | 5 | [5,5] | | 1.4 | [0.83, 0.47] |
| 4 | 3 | x^6 + 5y^2 | 1 | 0.001 | 5 | [10, 10] | | 537874 | [9.02, 9.82] |
| 5 | 4 | x^6 + 5y^2 | 1 | 0.001 | 5 | [5,5] | | 242.6 | [1.53, 6.78] |
| 6 | 5 | x^6 + 5y^2 | 1 | 0.001 | 10 | [5,5] | | 0.025 | [0.29, -0.06] |
| 7 | 6 | x1^2 + x2^4 + sqrt(\|x3\| | 2 | 0.001 | 5 | [10, 10, 10] | | 7.66 | [1.82, -1.09, 8.74] |
| 8 | 7 | x1^2 + x2^4 + 5*x3^3 | 2 | 0.001 | 5 | [5,5,5] | | 5.38 | [1.68, 0.50, 6.24] |
| 9 | 8 | x1^2 + x2^4 + 5*x3^4 | 1 | 0.001 | 5 | [10, 10,10] | | 114.57 | [8.62, 2.46, 11.25] |
| 10 | 9 | x1^2 + x2^4 + 5*x3^5 | 1 | 0.001 | 5 | [5,5,5] | | 1.29 | [-0.22, 0.60, 1.25] |
| 11 | 10 | x1^2 + x2^4 + 5*x3^5 | 1 | 0.01 | 10 | [5,5,5] | | 0.29 | 0.02, 0.64, 0.02] |

Fig 1 Experiments with different parameters

*Experiments on different algorithms*

Then, for the comparison between the three algorithms, I tried three different objective functions using the same parameters throughout each model. The three different objective functions consist of three different types: a well-defined smooth 2D function (x^2+y^2), a relatively more sophisticated 2D function x^4+2y^2, and a 3D function x1^6 + x2^4 + 5(x3) ^2. The starting point (or initial guess) for each model is always the same within the three models. I chose initial step sizes of 1 and 10 iteration numbers for the Random Search method and adjusted these parameters for each model accordingly.



| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Algorithms | objective function | Initial Step length | Minimum step Length | Iterations | Initial Point | | objective function value | optimal solution |
| 2 | Random Search | x^2+y^2 | 1 | 0.001 | 10 | [5,5] | | 0.129 | [-0.35, -0.08] |
| 3 | Steepest Descent | x^2+y^2 | N/A | N/A | 100 | [5,5] | | 0.88 | [0.66, 0.66] |
| 4 | Newton | x^2+y^2 | N/A | N/A | N/A | [5,5] | | 0 | [0, 0] |
| 5 | Random Search | x^4+2y^2 | 1 | 0.001 | 10 | [5,5] | | 0.012 | [-0.21, -0.07] |
| 6 | Steepest Descent | x^4+2y^2 | N/A | N/A | 100 | [5,5] | | 0.014 | [0, 0.08] |
| 7 | Newton | x^4+2y^2 | N/A | N/A | N/A | [5,5] | | 0.015 | [0.09, 0.09] |
| 8 | Random Search | x1^6 + x2^4 + 5(x3) ^2 | 1 | 0.001 | 10 | [1,1,1] | | 0.04 | [-0.23, -0.23,-0.04] |
| 9 | Steepest Descent | x1^6 + x2^4 + 5(x3) ^2 | N/A | N/A | 100 | [1,1,1] | | 0.02 | [0.45, 0.33, 0.00] |
| 10 | Newton | x1^6 + x2^4 + 5(x3) ^2 | N/A | N/A | N/A | [1,1,1] | | 1.62*e-06 | [0.11, 0.02, 0] |

Fig 2 Comparison between models

## Analysis

From the experiments of tuning parameters of the random search method, I have the following findings:

1. The performance of the Random Search Algorithm can to some extent depend on the initial point given when the step size is small. From Fig.1 rows 3 and 4 & rows 9 and 10, when we use the initial point closer to the optimal solution, [5,5] and [5,5,5] we can find a better solution than with a 'further away' solution ([10,10], [10,10,10]).
2. (Continued from the first finding) Fig.1 shows that when we have an initial point 'further away' from the optimal solution, the algorithm often performs better when the initial step size is sufficiently large since the larger step size enables the program to search in a larger scope, increasing the possibility of finding the global optimum.
3. With all other information constant, we can find a better result using more iterations since more points will be explored when there are more iterations before the step size decays.

From the comparison experiments between Newton's Method, Steepest Descent, and the Random Search model, there are several findings on each model from different aspects:

1. For the well-defined smooth functions such as x^2+y^2 and x1^6 + x2^4 + 5(x3) ^2 in Fig 2, Newton's method usually performs the best among the three models and the random search method has a better performance than Steepest Descent.
2. Random search shows a distinct advantage in high-dimensional spaces where the constraints on the initial point are less stringent. This contrasts with both Newton's method and the steepest descent. For example, in a 3D optimization problem x1^6 + x2^4 + 5(x3) ^2, the steepest descent and Newton's algorithm failed with an initial point of [5,5,5], suggesting that they are less robust to the initial condition in higher dimensions than the Random Search.

The findings above suggest that when encountering a problem with no explicit optimal solutions, the client can select a random initial point with a sufficiently large initial step size to enable global searching. While Newton's method performs better than the Random Search Method, it has some significant drawbacks that prevent prevalent use. First, Newton's method requires calculation for both hessian and gradient, which would be extremely computationally costly if the function is complicated and not as smooth as in the experiments. Secondly, Newton's method has a sequential calculation process where the next point found is largely dependent on the previous one, this would cause the program less robust and place a heavy burden on the computer storage when performing a large and complicated optimum search. Then for the Steepest Descent, from my experiments, the Steepest Descent did not perform better than the Random Search Method. Additionally, the steepest descent algorithm can sometimes be trapped in the local optimum when the function is sophisticated and random. Therefore, it is reasonable to conclude that the Random Search Algorithm outperforms the existing Newton's method and the Steepest Descent Algorithm from the perspective of cheap computation and global optimum search.

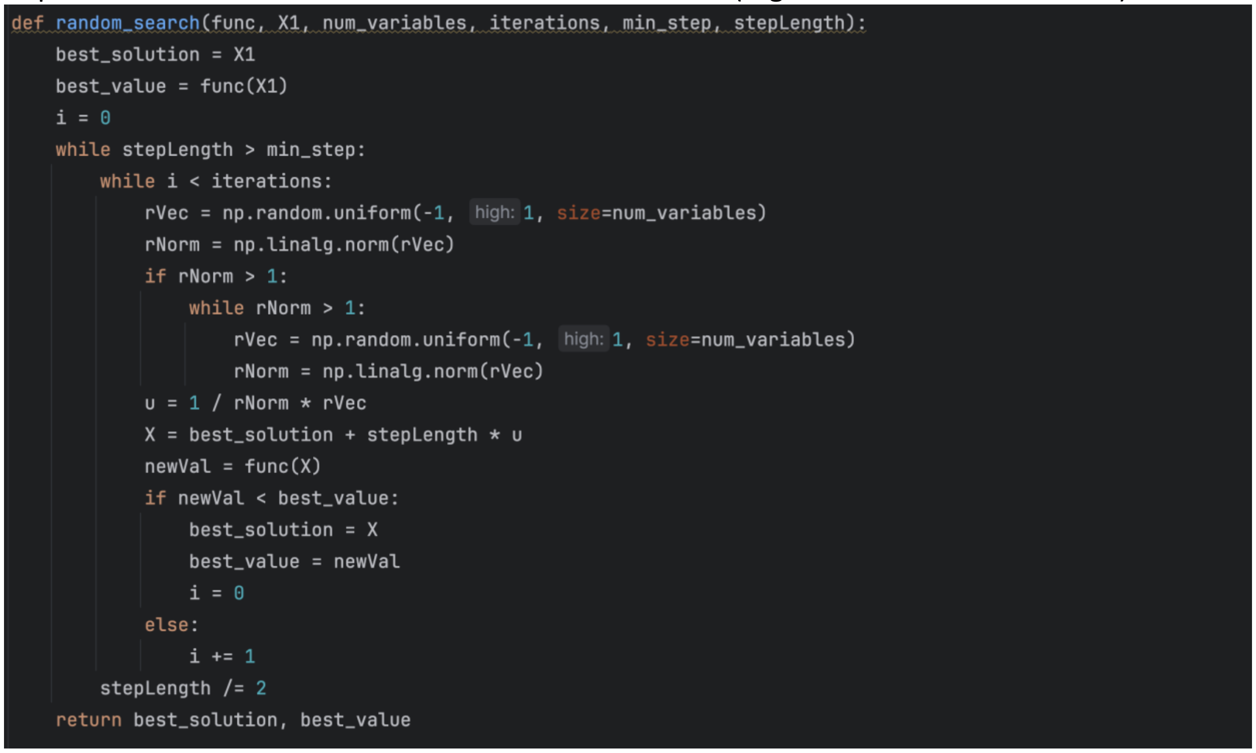
## Further improvement

After experiments, there are some potential improvements for my Interactive Random Search Software from both algorithm design perspective and software design perspective:

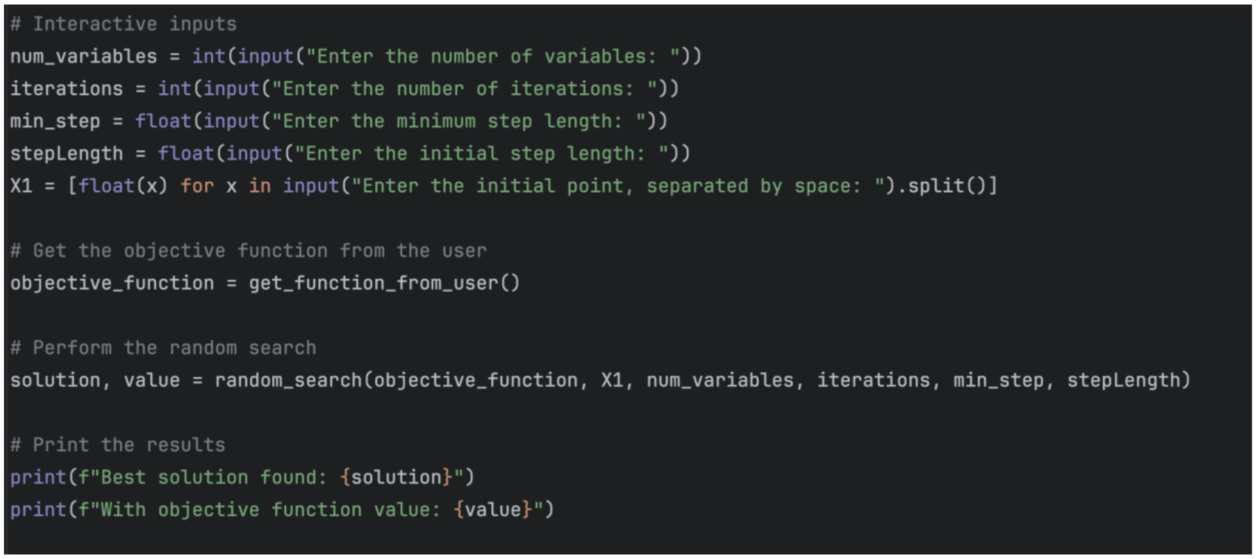
1. **More Iteration and Smaller Minimum Step Size**: During the experiments, I found that randomly generated coefficient vectors will influence the result greatly, I usually need to run it several times to get the 'best' result. For a more reliable and better solution, more iteration and a smaller minimum step size can be used to enable the program to search more exhaustively.
2. **Reproducibility**: The software can be further improved by recording all randomly generated vectors in an array to enable the clients to reproduce the search result and also enable the clients to exhibit their results for others to validate their solution searched.
3. **Adaptive Step Size**: Dynamically adjusting the step size (λ) based on the progress of the search can significantly improve the efficiency of the method. If successive iterations yield better results, the step size may be increased to exploit the direction of descent. Conversely, if no improvement is observed, reducing the step size can help in fine-tuning the search near local optima.
4. **Metaheuristic Enhancements**: Incorporating elements from metaheuristic algorithms like simulated annealing, where the probability of accepting worse solutions decreases over time, can furtherly help the Random Search Algorithm to escape local minima and improve the chances of finding a global optimum.

## Appendix

1. Implementation of Interactive Random Search Software (Algorithm and User Interface)



```python
def random_search(func, X1, num_variables, iterations, min_step, stepLength):
    best_solution = X1
    best_value = func(X1)
    i = 0
    while stepLength > min_step:
        while i < iterations:
            rVec = np.random.uniform(-1, high: 1, size=num_variables)
            rNorm = np.linalg.norm(rVec)
            if rNorm > 1:
                while rNorm > 1:
                    rVec = np.random.uniform(-1, high: 1, size=num_variables)
                    rNorm = np.linalg.norm(rVec)
            u = 1 / rNorm * rVec
            X = best_solution + stepLength * u
            newVal = func(X)
            if newVal < best_value:
                best_solution = X
                best_value = newVal
                i = 0
            else:
                i += 1
        stepLength /= 2
    return best_solution, best_value
```



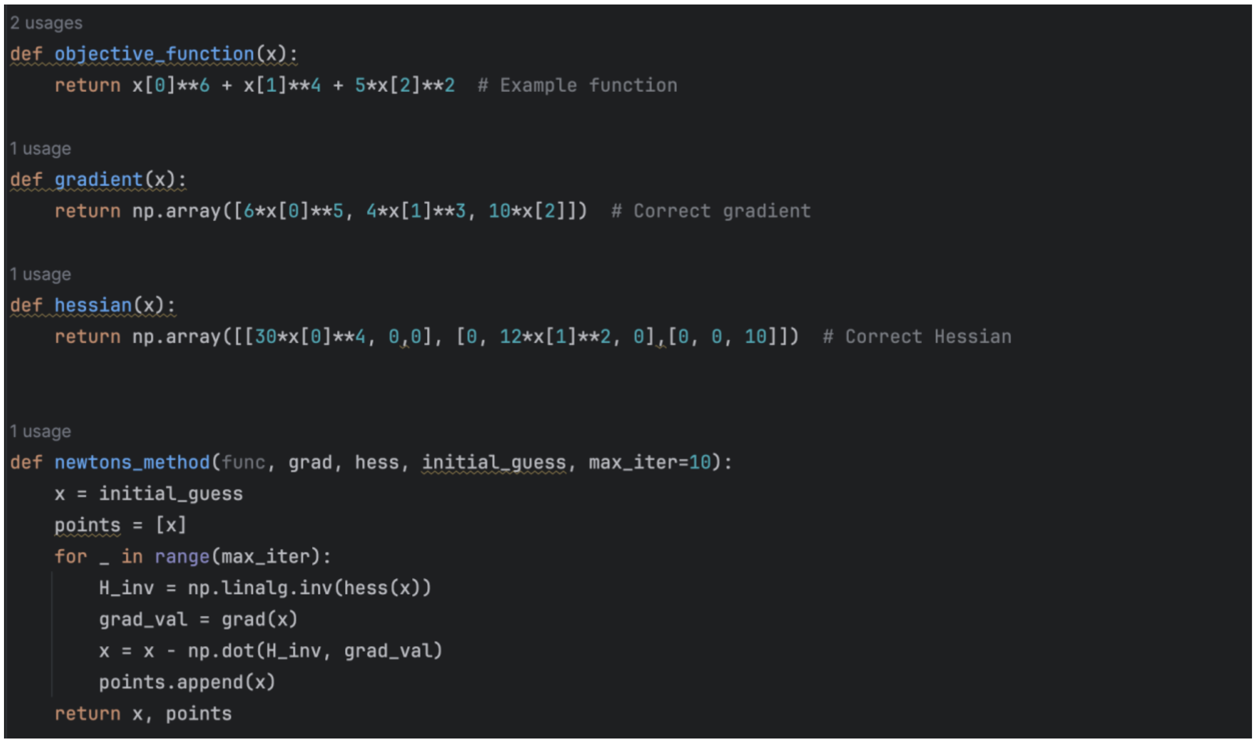
```python
# Interactive inputs
num_variables = int(input("Enter the number of variables: "))
iterations = int(input("Enter the number of iterations: "))
min_step = float(input("Enter the minimum step length: "))
stepLength = float(input("Enter the initial step length: "))
X1 = [float(x) for x in input("Enter the initial point, separated by space: ").split()]

# Get the objective function from the user
objective_function = get_function_from_user()

# Perform the random search
solution, value = random_search(objective_function, X1, num_variables, iterations, min_step, stepLength)

# Print the results
print(f"Best solution found: {solution}")
print(f"With objective function value: {value}")
```

2. Newton’s Method Implementation



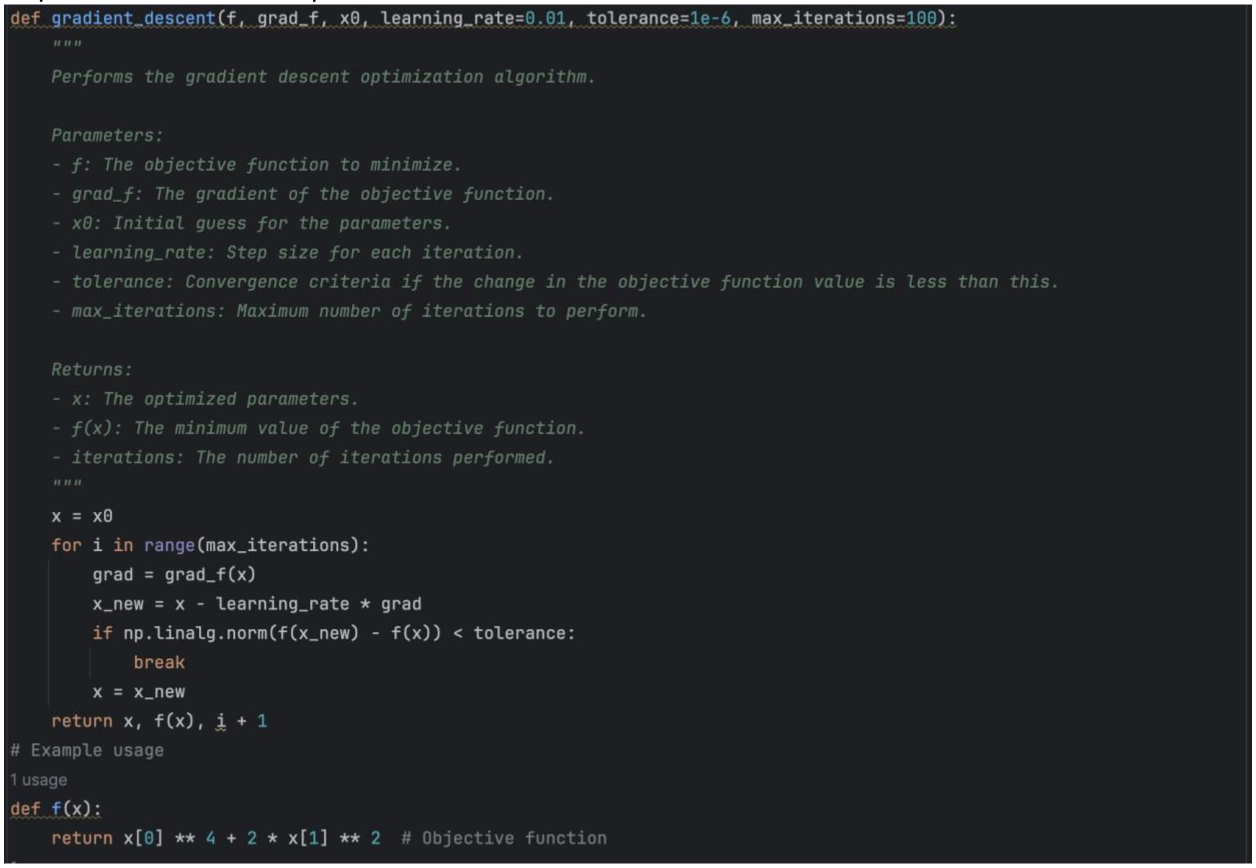
```
2 usages
def objective_function(x):
    return x[0]**6 + x[1]**4 + 5*x[2]**2  # Example function

1 usage
def gradient(x):
    return np.array([6*x[0]**5, 4*x[1]**3, 10*x[2]])  # Correct gradient

1 usage
def hessian(x):
    return np.array([[30*x[0]**4, 0,0], [0, 12*x[1]**2, 0],[0, 0, 10]])  # Correct Hessian


1 usage
def newtons_method(func, grad, hess, initial_guess, max_iter=10):
    x = initial_guess
    points = [x]
    for _ in range(max_iter):
        H_inv = np.linalg.inv(hess(x))
        grad_val = grad(x)
        x = x - np.dot(H_inv, grad_val)
        points.append(x)
    return x, points
```

3. Implementation of Steepest Descent

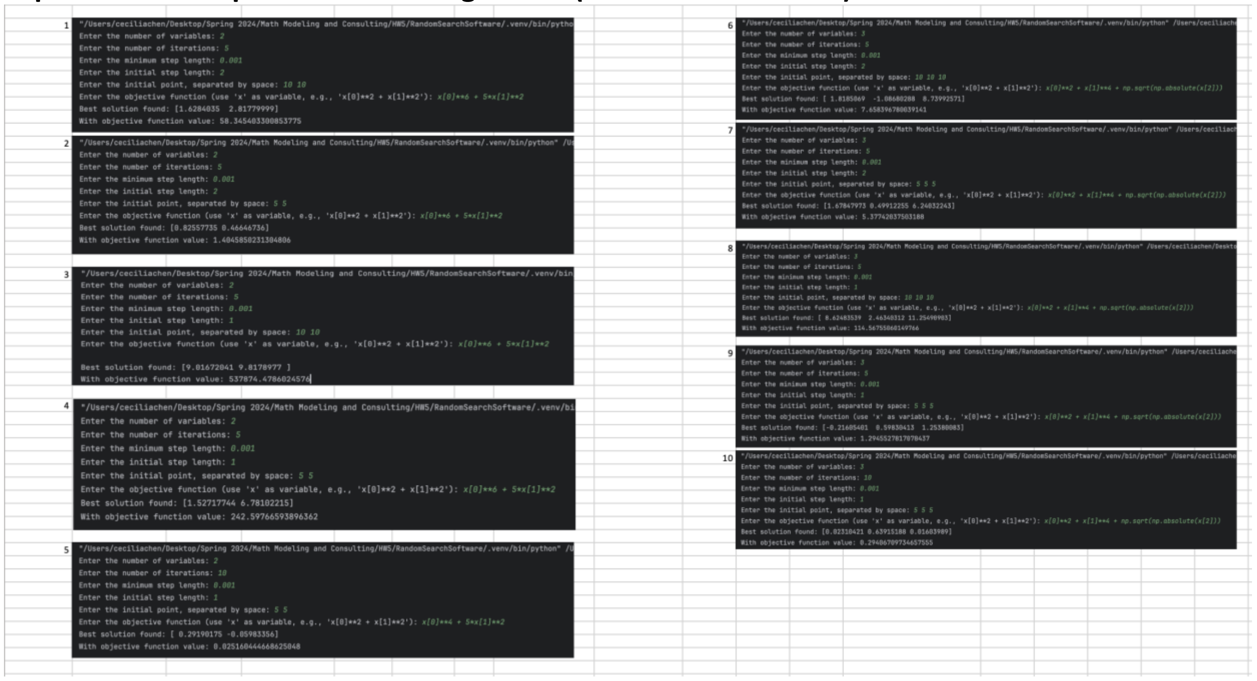


```
def gradient_descent(f, grad_f, x0, learning_rate=0.01, tolerance=1e-6, max_iterations=100):
    """
    Performs the gradient descent optimization algorithm.

    Parameters:
    - f: The objective function to minimize.
    - grad_f: The gradient of the objective function.
    - x0: Initial guess for the parameters.
    - learning_rate: Step size for each iteration.
    - tolerance: Convergence criteria if the change in the objective function value is less than this.
    - max_iterations: Maximum number of iterations to perform.

    Returns:
    - x: The optimized parameters.
    - f(x): The minimum value of the objective function.
    - iterations: The number of iterations performed.
    """
    x = x0
    for i in range(max_iterations):
        grad = grad_f(x)
        x_new = x - learning_rate * grad
        if np.linalg.norm(f(x_new) - f(x)) < tolerance:
            break
        x = x_new
    return x, f(x), i + 1
# Example usage
1 usage
def f(x):
    return x[0] ** 4 + 2 * x[1] ** 2  # Objective function
```

## 4. Experiment with parameters tuning Result (User end exhibition)



1
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/pytho
Enter the number of variables: 2
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 2
Enter the initial point, separated by space: 10 10
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**6 + 5*x[1]**2
Best solution found: [1.6284035  2.81779999]
With objective function value: 58.345403300853775
```

2
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /U
Enter the number of variables: 2
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 2
Enter the initial point, separated by space: 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**6 + 5*x[1]**2
Best solution found: [0.82557735 0.46646736]
With objective function value: 1.4045850231304806
```

3
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin
Enter the number of variables: 2
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 10 10
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**6 + 5*x[1]**2

Best solution found: [9.01672041 9.8178977 ]
With objective function value: 537874.4786024576
```

4
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bi
Enter the number of variables: 2
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**6 + 5*x[1]**2
Best solution found: [1.52717744 6.78102215]
With objective function value: 242.59766593896362
```

5
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /U
Enter the number of variables: 2
Enter the number of iterations: 10
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**4 + 5*x[1]**2
Best solution found: [ 0.29190175 -0.05983356]
With objective function value: 0.025160444668625048
```

6
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /Users/ceciliache
Enter the number of variables: 3
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 2
Enter the initial point, separated by space: 10 10 10
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**2 + x[1]**4 + np.sqrt(np.absolute(x[2]))
Best solution found: [ 1.8185069  -1.08680288  8.73992571]
With objective function value: 7.658396780039141
```

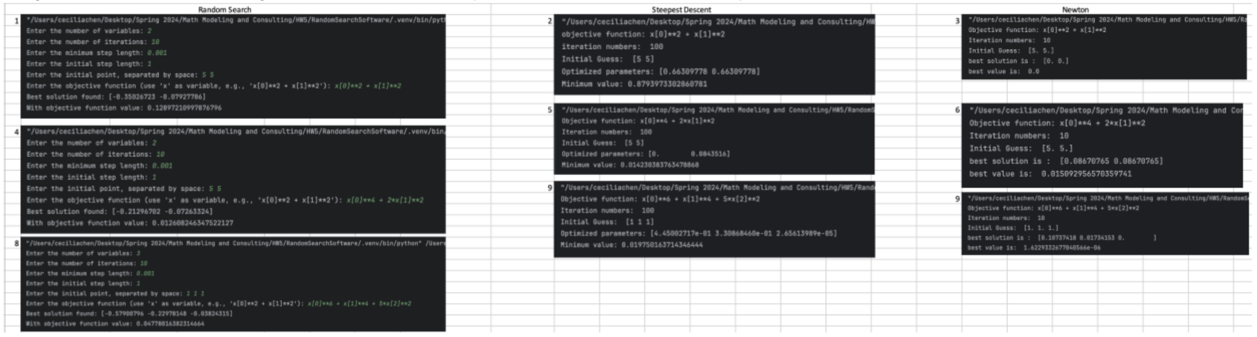
7
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /Users/ceciliach
Enter the number of variables: 3
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 2
Enter the initial point, separated by space: 5 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**2 + x[1]**4 + np.sqrt(np.absolute(x[2]))
Best solution found: [1.67847973 0.49912255 6.24032243]
With objective function value: 5.37742037503188
```

8
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /Users/ceciliachen/Deskto
Enter the number of variables: 3
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 10 10 10
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**2 + x[1]**4 + np.sqrt(np.absolute(x[2]))
Best solution found: [ 8.62483539  2.46348312 11.25490903]
With objective function value: 114.56755860149766
```

9
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /Users/ceciliache
Enter the number of variables: 3
Enter the number of iterations: 5
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 5 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**2 + x[1]**4 + np.sqrt(np.absolute(x[2]))
Best solution found: [-0.21605401  0.59830413  1.25380083]
With objective function value: 1.2945527817070437
```

10
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /Users/ceciliache
Enter the number of variables: 3
Enter the number of iterations: 10
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 5 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**2 + x[1]**4 + np.sqrt(np.absolute(x[2]))
Best solution found: [0.02310421 0.63915188 0.01603989]
With objective function value: 0.29406709734657555
```

## 5. Experiment of Comparisons Result (User end exhibition)



Random Search

1
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/pyt
Enter the number of variables: 2
Enter the number of iterations: 10
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**2 + x[1]**2
Best solution found: [-0.35026723 -0.07927786]
With objective function value: 0.12897210997876796
```

4
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin
Enter the number of variables: 2
Enter the number of iterations: 10
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 5 5
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**4 + 2*x[1]**2
Best solution found: [-0.21296702 -0.07263324]
With objective function value: 0.012608246347522127
```

8
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSearchSoftware/.venv/bin/python" /Users
Enter the number of variables: 3
Enter the number of iterations: 10
Enter the minimum step length: 0.001
Enter the initial step length: 1
Enter the initial point, separated by space: 1 1 1
Enter the objective function (use 'x' as variable, e.g., 'x[0]**2 + x[1]**2'): x[0]**6 + x[1]**4 + 5*x[2]**2
Best solution found: [-0.57900796 -0.22970148 -0.03824315]
With objective function value: 0.04778016382314664
```

Steepest Descent

2
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW
objective function: x[0]**2 + x[1]**2
iteration numbers:  100
Initial Guess:  [5 5]
Optimized parameters: [0.66309778 0.66309778]
Minimum value: 0.8793973302860781
```

5
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomS
Objective function: x[0]**4 + 2*x[1]**2
Iteration numbers:  100
Initial Guess:  [5 5]
Optimized parameters: [0.         0.0843516]
Minimum value: 0.014230383763478868
```

9
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/Rand
Objective function: x[0]**6 + x[1]**4 + 5*x[2]**2
Iteration numbers:  100
Initial Guess:  [1 1 1]
Optimized parameters: [4.45002717e-01 3.30868460e-01 2.65613989e-05]
Minimum value: 0.019750163714346444
```

Newton

3
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/Ra
Objective function: x[0]**2 + x[1]**2
Iteration numbers:  10
Initial Guess:  [5. 5.]
best solution is :  [0. 0.]
best value is:  0.0
```

6
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Co
Objective function: x[0]**4 + 2*x[1]**2
Iteration numbers:  10
Initial Guess:  [5. 5.]
best solution is :  [0.08670765 0.08670765]
best value is:  0.015092956570359741
```

9
```
"/Users/ceciliachen/Desktop/Spring 2024/Math Modeling and Consulting/HW5/RandomSe
Objective function: x[0]**6 + x[1]**4 + 5*x[2]**2
Iteration numbers:  10
Initial Guess:  [1. 1. 1.]
best solution is :  [0.10737418 0.01734153 0.        ]
best value is:  1.6229332677048566e-06
```

**Random Walk Algorithm**

```
1. Initialize best_solution to X1
2. Initialize best_value to func(X1)
3. Set i to 0
4. While stepLength is greater than minimum_step_length do:
   a. While i is less than max_iterations do:
      i. Generate a random vector rVec of dimension num_variables with elements
uniformly distributed between -1 and 1
      ii. Normalize rVec to unit vector u
      iii. Generate new solution X by adding stepLength * u to best_solution
      iv. Calculate newVal by applying func to X
      v. If newVal is less than best_value then:
         - Update best_solution to X
         - Update best_value to newVal
         - Reset i to 0
      vi. Else, increment i by 1.
   b. Halve the stepLength
5. Return best_solution and best_value
```