A GRADIENT DESCENT INSPIRED APPROACH TO OPTIMIZATION OF PHYSICS QUESTION

Feihong Liu[1] and Yu Sun[2]

1Crean Lutheran High School, 12500 Sand Canyon Ave, Irvine, CA 92618
2California State Polytechnic University,
Pomona, CA, 91768, Irvine, CA 92620

ABSTRACT

Many people believe that the crouch start was the best way to start a sprint [1]. While it seems intuitive, when the process of running is dissected using specific physical and mathematical representations, the question of "what is the best starting position" becomes harder to answer [2]. This paper aims to examine this phenomenon through a computer science approach inspired by gradient descent. Specifically, this paper aims to maximise the distance covered by a runner in ten steps. Assuming that runners do their best on every step and that their motion is not slowed by friction or air resistance, we will generate a hypothetical environment to study what the best strategy is for reaching the furthest distance within ten steps.

KEYWORDS

Gradient Descent.

1. INTRODUCTION

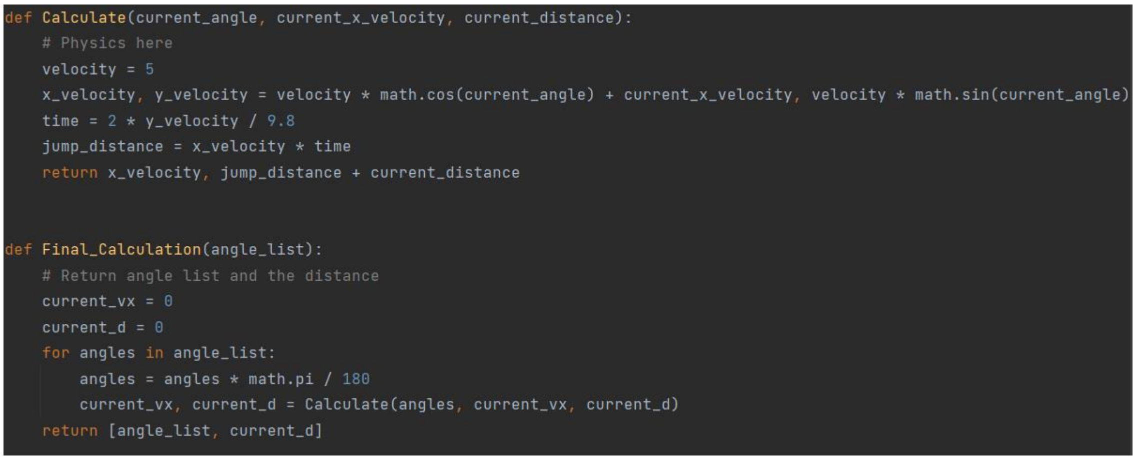
Imagine a runner sprinting ten steps [10]. On each step, they are applying some force on the ground at some angle [3]. Assuming that air resistance and internal forces are negligible, their motion before they hit the ground can be modelled roughly by a projectile motion [4]. Considering if the runner is trying to maximise their distance, they will not hold back on their force and will try their best on every single step. These reasoning give us our first two conditions: a constant max force and an angle in which this force will be applied for every step.

Realistically, a runner will be affected by various forces such as friction, air resistance, and internal forces. Because these forces can vary significantly (they can vary due to weather, heat, among many other factors), those forces will be ignored in our study. We aim to only explore how the different combinations of angles in which forces are applied will result in varying total distances. Thus, we will assume that horizontal velocity is not lost between every step (the velocity from the previous step will carry forward in the next step) and that the vertical velocity will be reset between every step (the runner will not bounce between each step).

With these conditions in mind, the rest of the paper is organised as follows: Section 2 introduces our initial experimentation; Section 3 analyses the results of Section 2 and proposes potential solutions; Section 4 explores the methods proposed in Section 3; Section 5 discusses the implementation of these methods. Finally, Section 6 gives concluding remarks, as well as pointing out the future work of this project.

2. INITIAL EXPERIMENTATION

To get a better sense of how to approach this problem, our situation is simplified to using only 5 angles. To understand a potential approach to this problem, different sequences of angles are generated with their output distance. For example, the list [5, 5, 5, 5, 5] will represent the motion of a runner who applies their force at a degree of 5 at every step. Using a series of algorithms, the distance resulting from this sequence of angles will be generated. Likewise, the list [5, 5, 5, 5, 10] will represent the motion of a runner who applies their force at a degree of 5 for every step other than the last one, which will be an angle of 10. Through an iterative process, all possible combinations of acute angles that are divisible by 5 are generated, and written in a data file. This means that the data point with the angle list [5, 5, 5, 5, 5] is first generated, followed by [5, 5, 5, 5, 10], [5, 5, 5, 5, 15], and so on. To do this, two helper functions are defined to find the distance with a particular list of angles.



```python
def Calculate(current_angle, current_x_velocity, current_distance):
    # Physics here
    velocity = 5
    x_velocity, y_velocity = velocity * math.cos(current_angle) + current_x_velocity, velocity * math.sin(current_angle)
    time = 2 * y_velocity / 9.8
    jump_distance = x_velocity * time
    return x_velocity, jump_distance + current_distance


def Final_Calculation(angle_list):
    # Return angle list and the distance
    current_vx = 0
    current_d = 0
    for angles in angle_list:
        angles = angles * math.pi / 180
        current_vx, current_d = Calculate(angles, current_vx, current_d)
    return [angle_list, current_d]
```

Figure 2.1. Screenshot of Helper methods

The calculate function returns the horizontal velocity and distance with some specifically inputted angle, velocity and distance. This function is called repeatedly in a for loop which will iterate through a list of angles. The final distance will be outputted along with the angle list that outputted this distance. The Final_Calculation function is called repeatedly using different angle lists starting from [5, 5, 5, 5, 5] up to [90, 90, 90, 90, 90] in increments of five. It is assumed that the runner is able to accelerate his velocity by 5 m/s. To save space, the best distances with each unique combination of the four five angles are saved in a data file. For example, in the simplest case, the best angle for the following sequence of angles [90, 90, 90, 90]

Since we have a four-dimensional input and one-dimensional output for every data point, we cannot visualise our data in a two-dimensional plane. Thus, a separate function is run to assign each angle list an x value. The list [5, 5, 5, 5] is given the x-value of 0, [5, 5, 5, 10] is given an x-value of 2, [5, 5, 10, 5] is given 18 ([5, 5, 5, 90] is given 17) and so on. All multidimensional data will be represented in this way throughout this paper. After plotting the data points using matplotlib, the following graph is generated [5]:

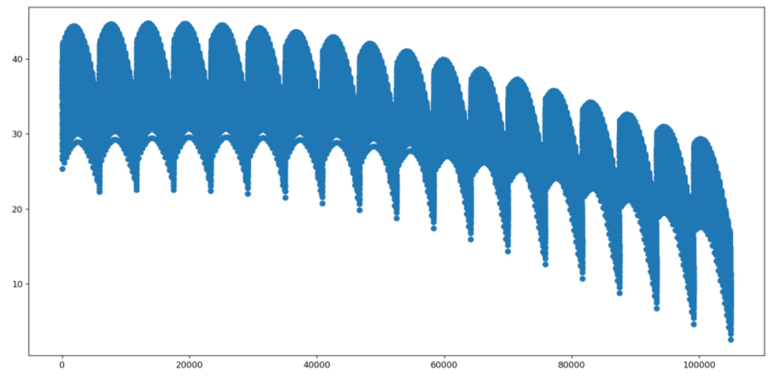


Figure 2.2. Initial Experiment Data

This graph looks roughly like a parabolic curve, composed of smaller curves. Interestingly though, when one zooms in on on curve, we can see that it is made up of smaller parabolic curves similar to this one:

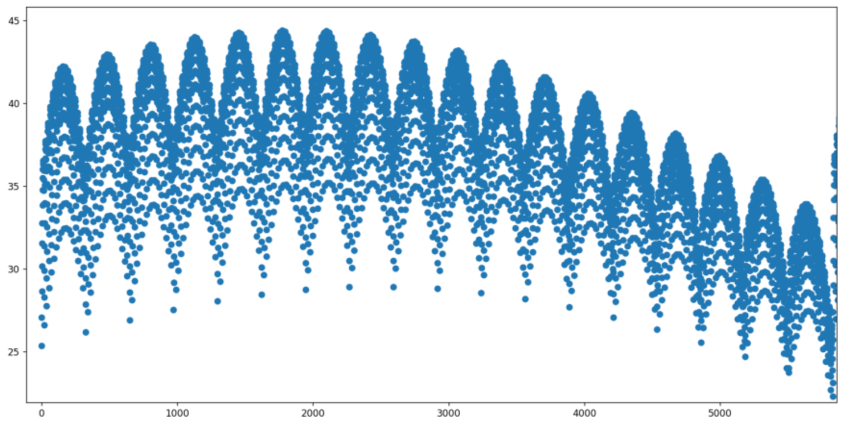


Figure 2.3. Initial Experiment Data (1st Zoom)

If we continue zooming in, we will see the following graphs:

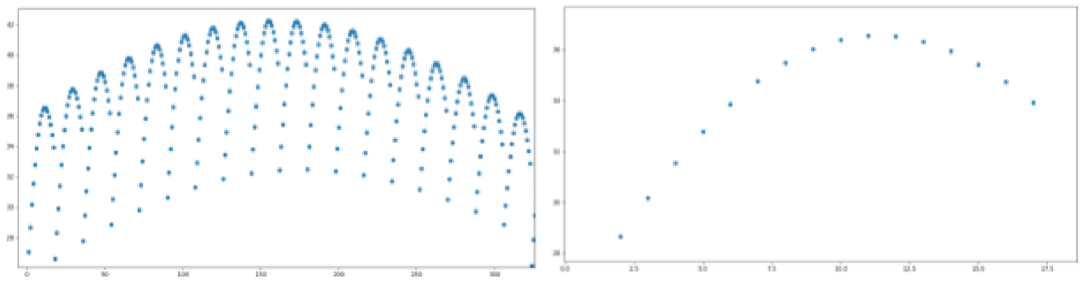


Figure 2.4. Initial Experiment Data (2nd and 3rd Zoom)

Each graph shown above is a zoomed in version of the graph above it. This shows an interesting relationship between each angle. At the smallest level (last image shown above), within each parabola, the first four angles of every point are the same. The last angle of each term is increased by increments of 5, with the first having 5 as its last angle and the last having 90 as its last angle.

Going back up and zooming out from the last graph, we see that each angle corresponds to a specific parabola. The last angle corresponds to the smallest parabola, as shown in the last figure. Similarly, the angle before that corresponds to the second to last figure, which is composed of smaller parabolas. The other angles have similar relationships with the other larger parabolas. It is also notable that there are only four of these parabola patterns because the fifth angle is already taken care of by the way we saved our data, since we are selecting only the optimal fifth angle.

3. Potential Solutions

Considering the example above, we can make several observations. First, we see that there are several smaller parabolas that make up the larger parabola in Figure 2.2 [6]. All data in each one of these smaller parabolas all have the same leading angle. This is represented by the following figure:

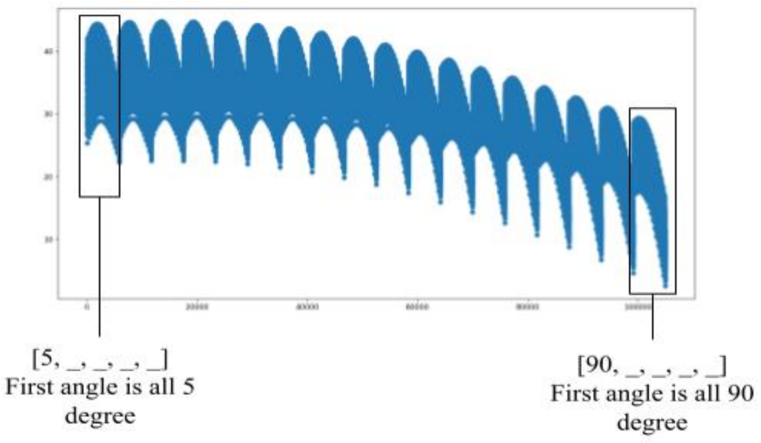


Figure 3.1. Grouping of Overall Data

Considering that these groups come to form a parabolic shape, a good way to represent each large group is to consider the average of its elements. For example, in the five-angle example shown in the previous portion, each group is very complex and composed of many points. By creating a graph of the average value of each group, we have a simpler way of finding the optimal value for this roughly parabolic shape.

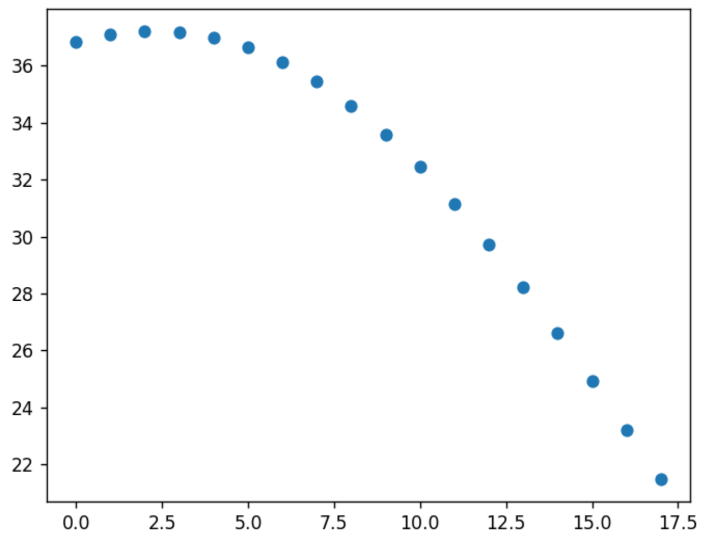


Figure 3.2. Each Group Represented by a Point With a its Average Distance

Thus, utilising this fact, we can generate data with larger increments (i.e. generating data with increments of 30 instead of 5), and find new data points based on calculated data points.

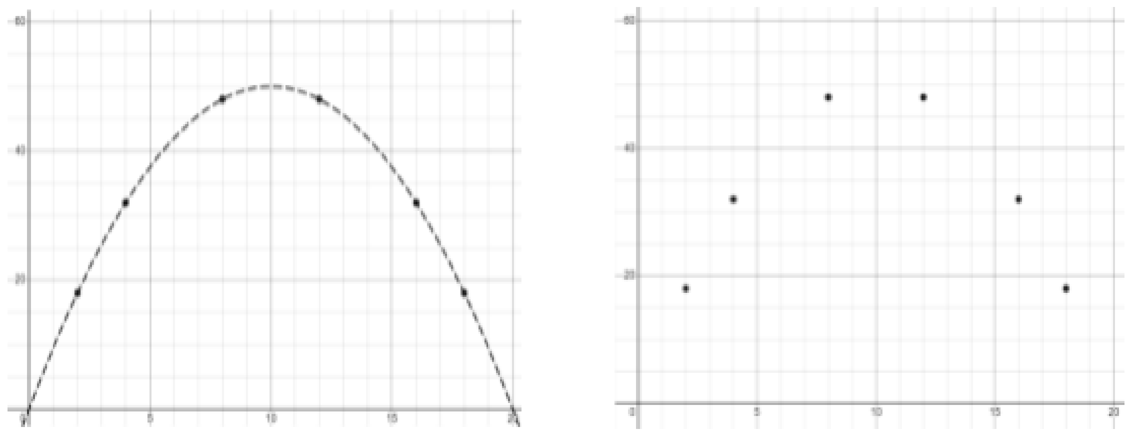


Figure 3.3. Generating Groups With Larger Increments

In the illustration shown above, we have six data points calculated but we are unsure what the optimal value is [7]. This presents us two ways to potentially approximate the optimal solution. We can use quadratic regression to find a quadratic equation that fits the given points or deduce the x value of the optimal by repeatedly comparing the x values of the three data points with the largest y value (this will be explained later on).

Repeating this process for every one of the ten angles. We are able to pinpoint, with relative accuracy and efficiency, the location of the optimal set of angles.

4. Proposed Methods

In this section, we will explore the two methods mentioned in the previous section.

4.1. Regression

Because the general shape for each angle can be roughly modeled by a quadratic curve, recursion seems like the most direct route to finding an optimal solution [8]. However, after experimentation, quadratic regression is shown to have two main issues: relatively large margin or error, and relatively inaccurate models in select situations.

Firstly, when quadratic regression is applied to the average value plot shown in the previous section, we result with the following plot.

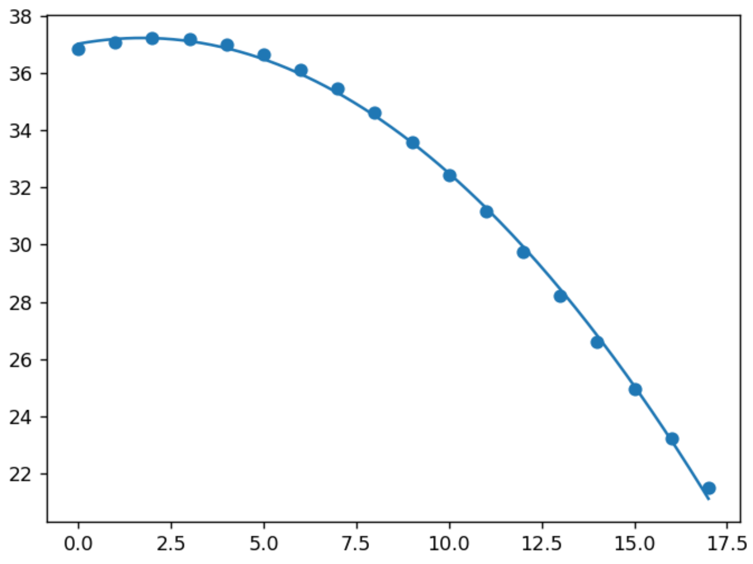


Figure 4.1.1 Regression applied to Figure 3.2

As shown, the plot shown above is slightly inaccurate, predicting that a value around 2.5 (or a sequence of angles beginning with 25 degrees) will produce the optimal. However, the optimal solution produced from the data points appears to be shifted to the right compared to the optimal produced by the regression line. This difference, while seemingly small, has a relatively. important impact in practice. A difference by ten degrees alone could drastically impact the path of the runner. Thus, the strategy of regression is considered to be relatively inaccurate.

4.2. Case Scenarios

Instead of approaching the problem directly with regression, we can take a similar approach by using some observations of several scenarios. For the following case scenarios, consider only the three points with the highest distance value. We are able to ignore the rest because they are unnecessary to find an optimal in these cases. Considering the three angles, we outline two scenarios: when the three are all ascending/descending or when the angle with the largest distance lies between the other two.

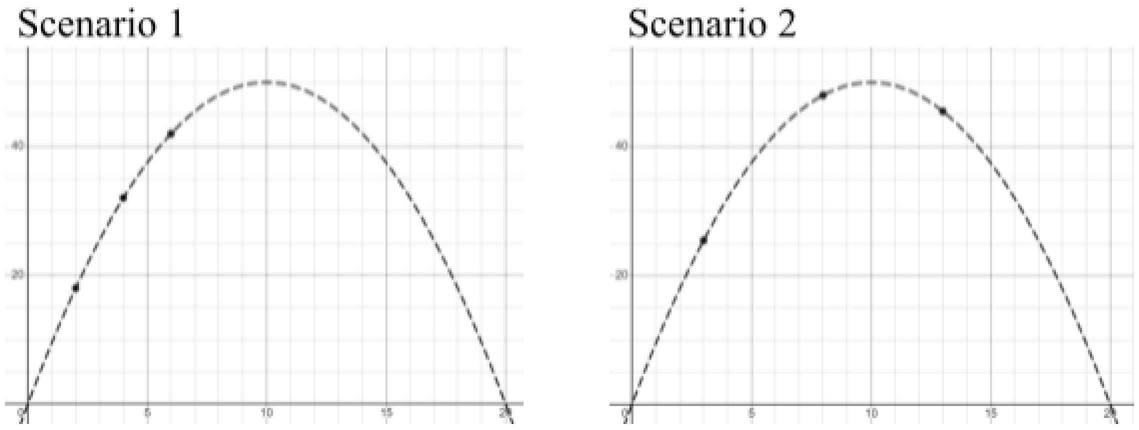
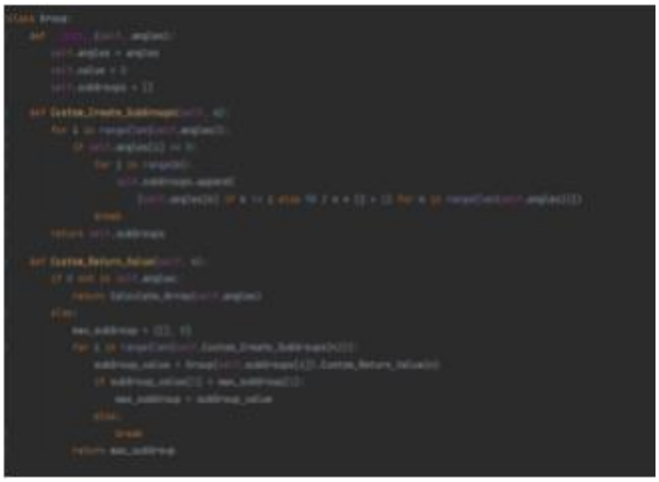


Figure 4.2.1. Two case scenarios

Our goal is to have angles of scenario 2, since we will be able to find another point with the angle equal to the average between the top two. This new data point will, theoretically, have an average value greater than the lesser of the two points. This process of averaging the top two angle measures can be repeated until the margin of error is small enough. However, if it happens that we come across scenario 1, we can simply replace the point with the smallest distance with a point to the right of the point with the largest distance. Through a combination of this process, we will, theoretically, be able to reach an approximate optimal solution with a margin of error of our choosing. The implementation of this method will be discussed below.

5. IMPLEMENTATION

Firstly, a program is run to generate an appropriate amount of data and save it into a data file. This will be important later on to more accurately localize the optimal sequence of angles. Because of the runtime for generating data, increments of 15 are used between each angle. Furthermore, for simplification, only the best cases for unique combinations of the first three angles will be saved. The rest will be optimized through the Group class shown below.



```
class Group:
    def __init__(self, angles):
        self.angles = angles
        self.value = 0
        self.subGroups = []

    def Custom_Create_SubGroups(self, n):
        for i in range(len(self.angles)):
            if self.angles[i] == 0:
                for j in range(n):
                    self.subGroups.append(
                        [self.angles[k] if k != i else 90 / n * (j + 1) for k in range(len(self.angles))])
                break
        return self.subGroups

    def Custom_Return_Value(self, n):
        if 0 not in self.angles:
            return Calculate_Group(self.angles)
        else:
            max_subGroup = [[], 0]
            for i in range(len(self.Custom_Create_SubGroups(n))):
                subGroup_value = Group(self.subGroups[i]).Custom_Return_Value(n)
                if subGroup_value[1] > max_subGroup[1]:
                    max_subGroup = subGroup_value
                else:
                    break
            return max_subGroup
```

Figure 5.1. Screenshot of code 1

Simply, when a group object is initiated, it contains a list of angles that begin with some values. Other values in the list will be defined as zero, which will be important for the it's two functions. The zero values of this array will be values in which the functions will be replaced with other values. For example, if [15, 15, 15, 0, 0, 0, 0, 0] is passed into the function, it will alter the fourth to eighth angles. This is used for the main function, Custom_Return_Value. This function will iterate through all angles incrementing by a value determined by the input n. For example, if a Group object is defined with [15, 15, 15, 0, 0, 0, 0, 0], the function will look through angles [15, 15, 15, 0, 0, 0, 0, 15], [15, 15, 15, 0, 0, 0, 0, 30], etc and return the angles with the furthest distance. However, an additional statement is implemented into this function to save run time. Consider that a ball is climbing up a parabola, a decrease in height will only happen if the ball has passed the vertex. This is essentially the logic of the statement, which will exit the for loop immediately when a decrease in "height" (distance in our case) is detected. The following is the generated data.

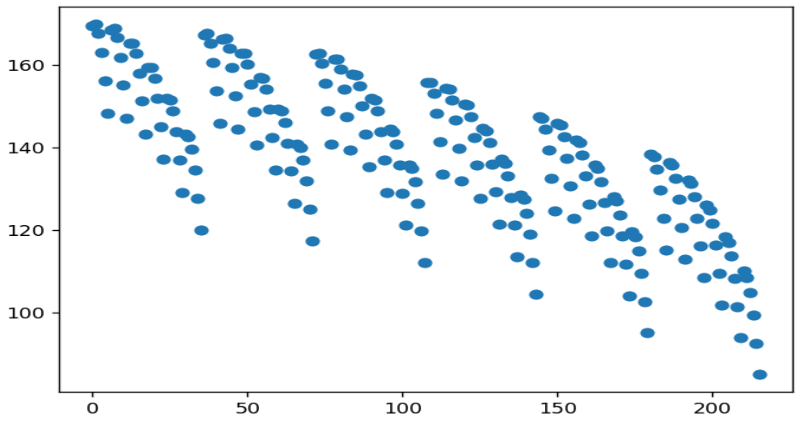
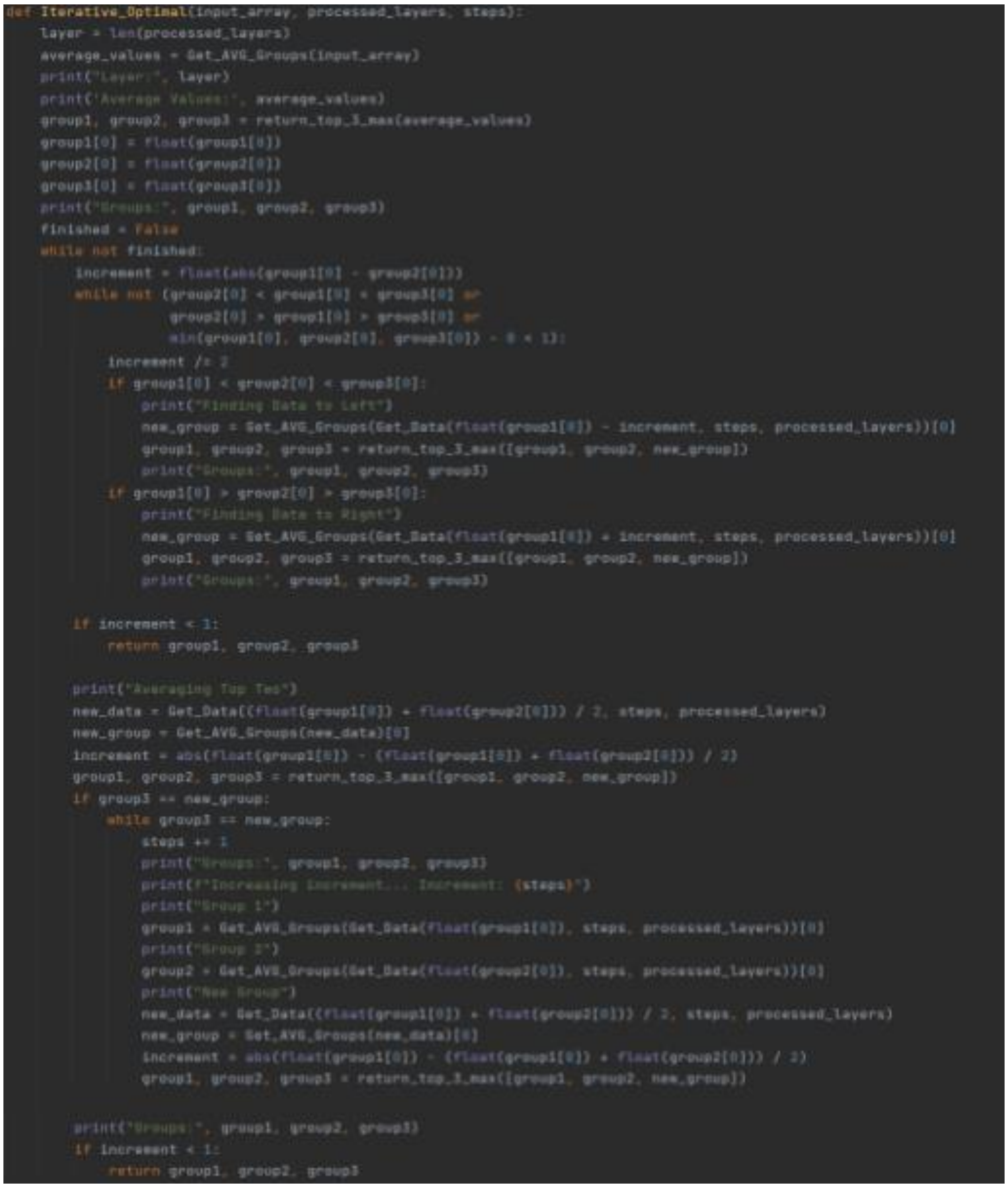


Figure 5.2. Generated data (not averaged by groups)

After this data is generated, an iterative processing function is called. This function will first go through the angles, and select the one being altered, from the first to the last angle. With each angle, the top three distances are found and the corresponding angle in which this occurred is stored into three groups. Afterwards, our second method from the last section will begin processing these angles (see Method 2 of Experimentation). These processes will keep iterating until the margin of error is less than 1. The following code (and helper function) demonstrates this process.

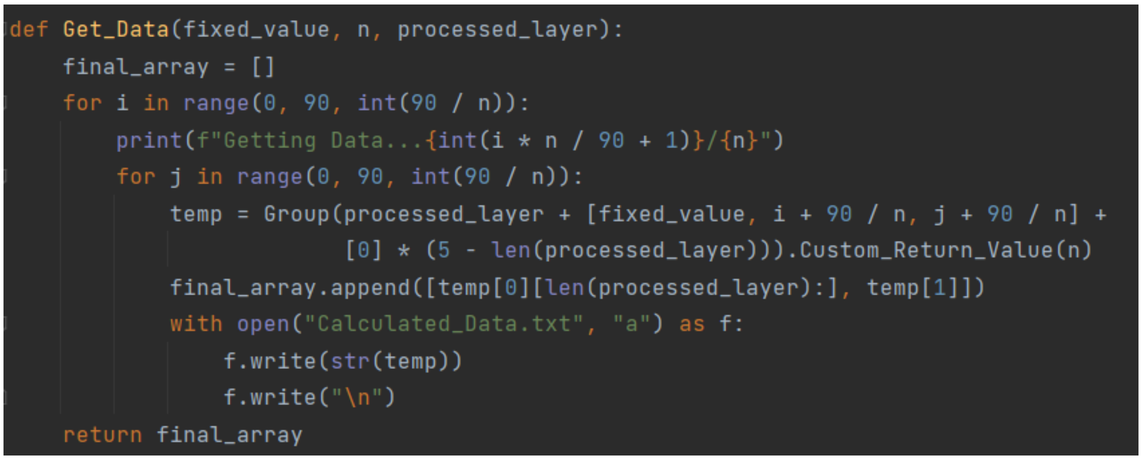


```python
def Iterative_Optimal(input_array, processed_layers, steps):
    layer = len(processed_layers)
    average_values = Get_AVG_Groups(input_array)
    print("Layer:", layer)
    print('Average Values:', average_values)
    group1, group2, group3 = return_top_3_max(average_values)
    group1[0] = float(group1[0])
    group2[0] = float(group2[0])
    group3[0] = float(group3[0])
    print("Groups:", group1, group2, group3)
    finished = False
    while not finished:
        increment = float(abs(group1[0] - group2[0]))
        while not (group2[0] < group1[0] < group3[0] or
                   group2[0] > group1[0] > group3[0] or
                   min(group1[0], group2[0], group3[0]) - 0 < 1):
            increment /= 2
            if group1[0] < group2[0] < group3[0]:
                print("Finding Data to Left")
                new_group = Get_AVG_Groups(Get_Data(float(group1[0]) - increment, steps, processed_layers))[0]
                group1, group2, group3 = return_top_3_max([group1, group2, new_group])
                print("Groups:", group1, group2, group3)
            if group1[0] > group2[0] > group3[0]:
                print("Finding Data to Right")
                new_group = Get_AVG_Groups(Get_Data(float(group1[0]) + increment, steps, processed_layers))[0]
                group1, group2, group3 = return_top_3_max([group1, group2, new_group])
                print("Groups:", group1, group2, group3)

        if increment < 1:
            return group1, group2, group3

        print("Averaging Top Two")
        new_data = Get_Data((float(group1[0]) + float(group2[0])) / 2, steps, processed_layers)
        new_group = Get_AVG_Groups(new_data)[0]
        increment = abs(float(group1[0]) - (float(group1[0]) + float(group2[0])) / 2)
        group1, group2, group3 = return_top_3_max([group1, group2, new_group])
        if group3 == new_group:
            while group3 == new_group:
                steps += 1
                print("Groups:", group1, group2, group3)
                print(f"Increasing Increment... Increment: {steps}")
                print("Group 1")
                group1 = Get_AVG_Groups(Get_Data(float(group1[0]), steps, processed_layers))[0]
                print("Group 2")
                group2 = Get_AVG_Groups(Get_Data(float(group2[0]), steps, processed_layers))[0]
                print("New Group")
                new_data = Get_Data((float(group1[0]) + float(group2[0])) / 2, steps, processed_layers)
                new_group = Get_AVG_Groups(new_data)[0]
                increment = abs(float(group1[0]) - (float(group1[0]) + float(group2[0])) / 2)
                group1, group2, group3 = return_top_3_max([group1, group2, new_group])

        print("Groups:", group1, group2, group3)
        if increment < 1:
            return group1, group2, group3
```



```python
def Get_Data(fixed_value, n, processed_layer):
    final_array = []
    for i in range(0, 90, int(90 / n)):
        print(f"Getting Data...{int(i * n / 90 + 1)}/{n}")
        for j in range(0, 90, int(90 / n)):
            temp = Group(processed_layer + [fixed_value, i + 90 / n, j + 90 / n] +
                         [0] * (5 - len(processed_layer))).Custom_Return_Value(n)
            final_array.append([temp[0][len(processed_layer):], temp[1]])
            with open("Calculated_Data.txt", "a") as f:
                f.write(str(temp))
                f.write("\n")
    return final_array
```

Figure 5.3. Screenshot of code 2

This process is iterated through all the angles, finding the approximate optimal solution (with a 1 degree margin of error). However, because of the nature of how the Group class is set up, the last four angles are not returned. Thus, a final function is used to iterate through all combinations for the last four angles with an increment of 1. All the angles are rounded and returned. This means that, overall, this approach will return an angle list with the first six angles as approximations of an optimal solution with a margin of error of 1, and the last four angles are optimal integer solutions. This finally produces the following angles list. [8, 16, 25, 34, 42, 52, 57, 65, 73, 81].

6. Conclusions And Future Works

The approach outlined in the paper demonstrated a more runtime-efficient algorithm that approximates an optimal solution with a margin of 1 degree for the first six angles [9]. This is an acceptable margin of error mainly due to two reasons. Firstly, people in reality are not able to have pinpoint accuracy in the direction they apply their force, meaning that this 1 degree of error will have little impact in a real life situation. Second, it is clearly shown in numerous figures in this paper that the overall shape of this scenario resembles a parabola, meaning that little margins of error towards the optimal solution will result in little to no change in the total distance travelled (change in vertical distance increases the
further away a point is horizontally from the vertex).

Although this situation assumes many conditions that are too good to be true for real life, the general result found in these projects add reason to many strategies in running. For example, Olympic runners start off their run with pedals that allow them to exert a smaller angle of force. Observers often see them straightening out within a few steps. Our results demonstrate that increasing angles of force does indeed allow runners to travel an optimal distance (although our program does not account for time efficiency).

Among many issues that the current algorithm has, most notable to me is the issue of time efficiency. This algorithm is not very time-efficient, and will not work nearly as well if the number of angles were increased. Thus, modifications and better approaches can be considered to significantly improve this aspect of our project. Additionally, adding conditions that are closer to reality could also be beneficial for this program, as it can give us a strategy that we can use in real life.

REFERENCES

[1] Salo, Aki, and Ian Bezodis. "Athletics: Which starting style is faster in sprint running standing or crouch start?." Sports Biomechanics 3.1 (2004): 43-54.
[2] Graber, Kenneth C. The effect of starting body position on velocity in running. Diss. 2013.
[3] Frishberg, Barry A. "An analysis of overground and treadmill sprinting." Medicine and science in sports and exercise 15.6 (1983): 478-485.
[4] Parker, G. W. "Projectile motion with air resistance quadratic in the speed." American Journal of Physics 45.7 (1977): 606-610.
[5] Barrett, Paul, et al. "matplotlib--A Portable Python Plotting Package." Astronomical data analysis software and systems XIV. Vol. 347. 2005.
[6] Richmond, Bettina, and Tom Richmond. "How to recognize a parabola." The American Mathematical Monthly 116.10 (2009): 910-922.
[7] Zhang, Xinpeng. "Reversible data hiding with optimal value transfer." IEEE Transactions on Multimedia 15.2 (2012): 316-325.
[8] DeVORE, Ronald A., and Zheng Yan. "Error analysis for piecewise quadratic curve fitting algorithms." Computer aided geometric design 3.3 (1986): 205-215.
[9] Huang, Xiaoqiu, and Webb Miller. "A time-efficient, linear-space local similarity algorithm." Advances in applied mathematics 12.3 (1991): 337-357.
[10] Young, W. A. R. R. E. N., B. R. I. A. N. Mc Lean, and J. A. M. E. S. Ardagna. "Relationship between strength qualities and sprinting performance." Journal of sports medicine and physical fitness 35.1 (1995): 13-19.