

EML 3035: Programming Concepts Project

Spring 2024

Projectile Motion Simulator

Thais Isabelle Parron Ruiz

U15798643

Submission Date: 03/31/2024

Table of Contents

Introduction.....	3
Collecting user information.....	4
calculations.....	6
Plotting mountain.....	7
Plotting target.....	9
Plotting projectile.....	11
Showing results.....	13
Prompt for trying again.....	15
Allow for multiple tries.....	18
Show all results.....	20
Conclusion.....	22

Introduction

The objective of this project is to develop a comprehensive projectile motion simulator. It will prompt users to input parameters such as initial height, target location, launch angle, velocity, and the celestial body they wish to simulate the launch on. The simulator should be robust, allowing users to correct input errors seamlessly.

Additionally, the simulator will generate visual representations, including a detailed plot of the terrain, the target, and the trajectory of the projectile. Emphasis will be placed on accurately depicting the motion, rather than simply showing the line of the path.

Users will have three attempts to hit the target within a 2-meter margin of error. After each attempt, the simulator will provide feedback on the maximum height reached and the distance from the target. Regardless of whether the user completes all three attempts or decides to stop playing, the results of each attempt should be displayed at the end.

IMPORTANT: The codes shown in this report are not completed, there are several missing parts from the original code, it is just a representation of how I did the parts I am talking about in each section. The original code combines all of them in order to make the project work as it should.

Collecting User Information

In this section, users are required to input specific parameters within predefined ranges: starting height should fall between 5 to 20 meters, target placement between 10 to 100 meters, starting angle from 0 to 80 degrees, initial velocity between 10 to 50 meters per second, and the planet choice should be indicated by entering either "1" for Earth, "2" for Mars, or "3" for Jupiter.

To implement this functionality, I utilized conditional statements (if statements) to validate user inputs. If a user provides an incorrect value, the system prompts them to re-enter the respective parameter until a valid value is provided.

Here is a section of the code that shows that:

```
starting_height = input ('Please input a starting height (5 - 20 m):'); %m
if starting_height >= 5 && starting_height <= 20
    target_placement = input ('Where do you want to place the target? (10 - 100 m):');

    if target_placement >= 10 && target_placement <= 100

        else
            disp('This is an invalid starting height. Please try again.');
```

The conditional statement assesses whether the user-provided input falls within the range of 5 to 20. If it does, the code proceeds (details not illustrated in the provided example). If the input falls outside this range, an error message is displayed indicating an invalid value, prompting the user to retry. Subsequently, a while loop, which will be elaborated upon later in this report, facilitates the user in entering a new value.

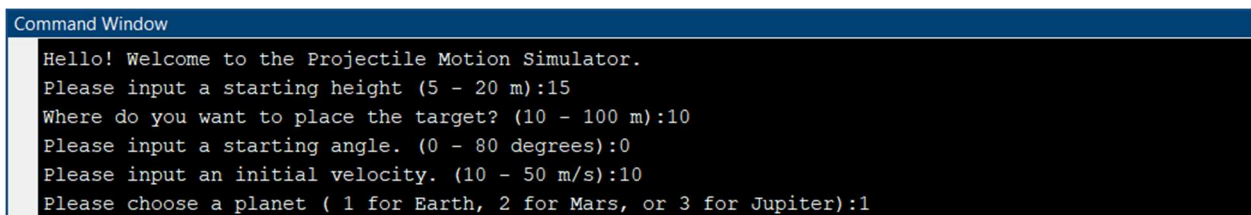
A similar procedure is employed for most user inputs, with adjustments made to the maximum and minimum values accordingly. However, the planet input differs as users are required to input either "1", "2", or "3", rather than any value between two numbers. To handle this, if statements are utilized in a distinct manner.

Here is the section of the code that collects the information about which planet the user wants:

```
Planet = input('Please choose a planet ( 1 for Earth, 2 for Mars, or 3 for
Jupiter):');
if planet == 1
    gravity = 9.81; %m/s^2
elseif planet == 2
    gravity = 3.71; %m/s^2
elseif planet == 3
    gravity = 24.79; %m/s^2
else
    disp('This is an invalid planet selection. Please try again.');
```

That's how I gathered user data and made sure they input valid values for each section. In the next part of this report, I'll dive deeper into the project's code.

Here is how the questions are prompted in the command window:



```
Command Window
Hello! Welcome to the Projectile Motion Simulator.
Please input a starting height (5 - 20 m):15
Where do you want to place the target? (10 - 100 m):10
Please input a starting angle. (0 - 80 degrees):0
Please input an initial velocity. (10 - 50 m/s):10
Please choose a planet ( 1 for Earth, 2 for Mars, or 3 for Jupiter):1
```

Calculations

The formulas utilized in this project were supplied by the professor. I embedded them within each of the if statements corresponding to the planets mentioned earlier. I opted for this approach because these formulas incorporate gravity, and since the gravity value varies based on the planet selected by the user, it necessitated this customization.

Here is the section of the code that does the calculations for the Earth (planet 1):

```
if planet == 1
    gravity = 9.81; %m/s^2

    %calculations of the projectile

    starting_angle = starting_angle*pi/180; %changing to degrees
    time = (-initial_velocity*sin(starting_angle) sqrt((initial_velocity*
        sin(starting_angle))^2 + 2 * gravity * starting_height)) /
        (-gravity); %s
    vertical_position(1,i) = starting_height +
        initial_velocity*sin(starting_angle)*time-(1/2)* gravity*time^2; %m
    horizontal_position(1,i) = initial_velocity*cos(starting_angle)*time; %m
    max_height(1,i) = starting_height + (initial_velocity
        *sin(starting_angle))^2/(2*gravity); %m
    distance_target(1,i) = horizontal_position(1,i) - target_placement; %m
end
```

Under each of the planets' if statements, I duplicated the same code, adjusting solely the gravity variable to ensure accurate calculations aligned with the user's selection. All other variables utilized in this code had been previously defined.

Plotting the mountain

To generate the mountain plot, I utilized the data gathered from the user alongside calculated values. Specifically, factors such as the mountain's height are determined by the initial height input by the user. Additionally, the slope of the lines forming the mountain stripes is influenced by the user's specified starting height.

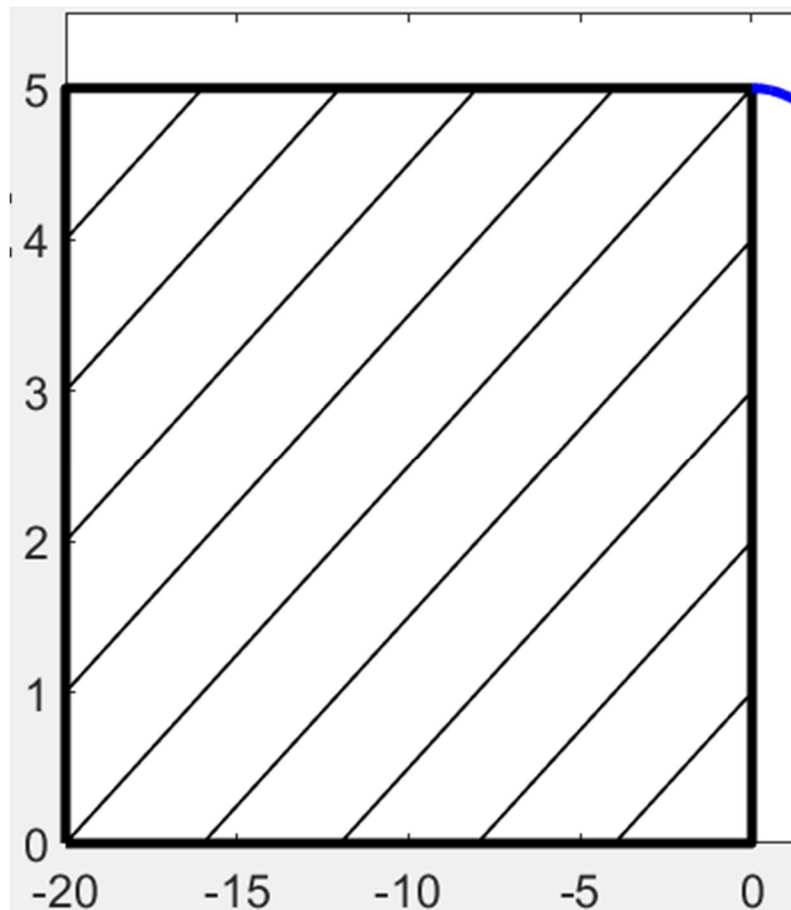
Here is the part of the code that plots the mountain and its stripes:

```
slope = 20/starting_height;
x1 = [-20 0 0 -20 -20];
y1 = [0 0 starting_height starting_height 0];
plot(x1,y1,'Color','k','LineWidth',3);
hold on
xmin = -20;
xmax = 0;
ymin = 0;
ymax = starting_height;
while xmin < 0
    x2 = [xmin 0];
    y2 = [0 ymax];
    plot(x2,y2,'Color','k','LineWidth',1);
    xmin = xmin + slope;
    ymax = ymax - 1;
end
hold on
while ymin < starting_height
    x3 = [-20 xmax];
    y3 = [ymin starting_height];
    plot(x3,y3,'Color','k','LineWidth',1);
    xmax = xmax - slope;
    ymin = ymin + 1;
end
```

The variables "x1" and "x2" denote the x and y coordinates respectively, forming the interconnected points constituting the mountain. To confine the stripes within the boundaries of the mountain, "ymin", "ymax", "xmin", and "xmax" variables were introduced. Utilizing while loops, I dynamically adjusted the starting and ending points of each stripe by incrementing the y-direction by one meter and altering the slope value in the x-direction. This approach ensured the

creation of parallel stripes meeting the project's specifications. Furthermore, this code is incorporated within the if statements corresponding to each planet.

Here is a picture of the final result of the mount with a starting height of 5m:



Plotting target

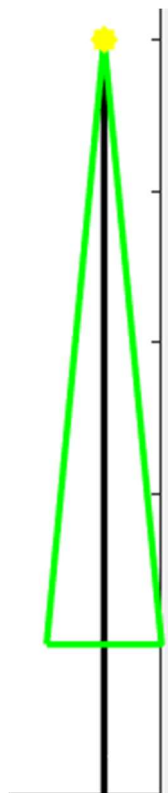
The plotting of the target is intricately tied to the user-provided information and is encapsulated within the if statements corresponding to each planet.

Here is the code that plots the target:

```
hold on
plot([target_placement target_placement],[0 starting_height]
     , 'Color','k', 'LineWidth',2);
hold on
chistmas_tree_x = [target_placement-starting_height/3
                  target_placement+starting_height/3 target_placement target_placement-
                  starting_height/3];
chistmas_tree_y = [starting_height/5 starting_height/5 starting_height
                  starting_height/5];
plot(chistmas_tree_x,chistmas_tree_y, 'Color','g', 'LineWidth',2);
hold on
plot(target_placement, starting_height, 'Color','y', 'LineWidth',6, 'Marker','*');
```

The second line of code handles the plotting of the pole, mirroring the height of the mountain. This is achieved by utilizing the variable "starting_height" as the y-coordinate for the second point. Since the pole should span horizontally at the user-defined distance, both points are assigned the x-coordinate of the variable "target_placement". Following the pole's plotting, I aimed to imbue it with a festive touch resembling a Christmas tree. To accomplish this, I introduced the variables "chistmas_tree_x" and "chistmas_tree_y", which correspond to the x and y coordinates respectively, forming the green triangle. These coordinates are strategically derived from the starting height to ensure proportionality with the pole's size. I determined the optimal ratio through iterative experimentation, selecting values that visually enhanced the appearance across various heights. Lastly, the plot showcases a star marker positioned atop the pole, symbolizing the traditional star ornament adorning the top of a Christmas tree.

Here is the plot of the target with initial height of 5m:



Plotting projectile

This segment of the code is nested within the if statement for each planet, as it directly relies on the gravitational force specific to each celestial body.

Here is the section of the code that plots the projectile path:

```
hold on
w = 1;
for time1 = 0:0.003:time
    y(1,w) = starting_height + initial_velocity*sin(starting_angle)*time1-
        (1/2)*gravity*time1^2;
    x(1,w) = initial_velocity*cos(starting_angle)*time1;
    w = w + 1;
end
for r = 1:w-2
    plot([x(r) x(1+r)], [y(r) y(1+r)], 'Color', 'b', 'LineWidth', 3);
    hold on
    pause(0.001);
end
```

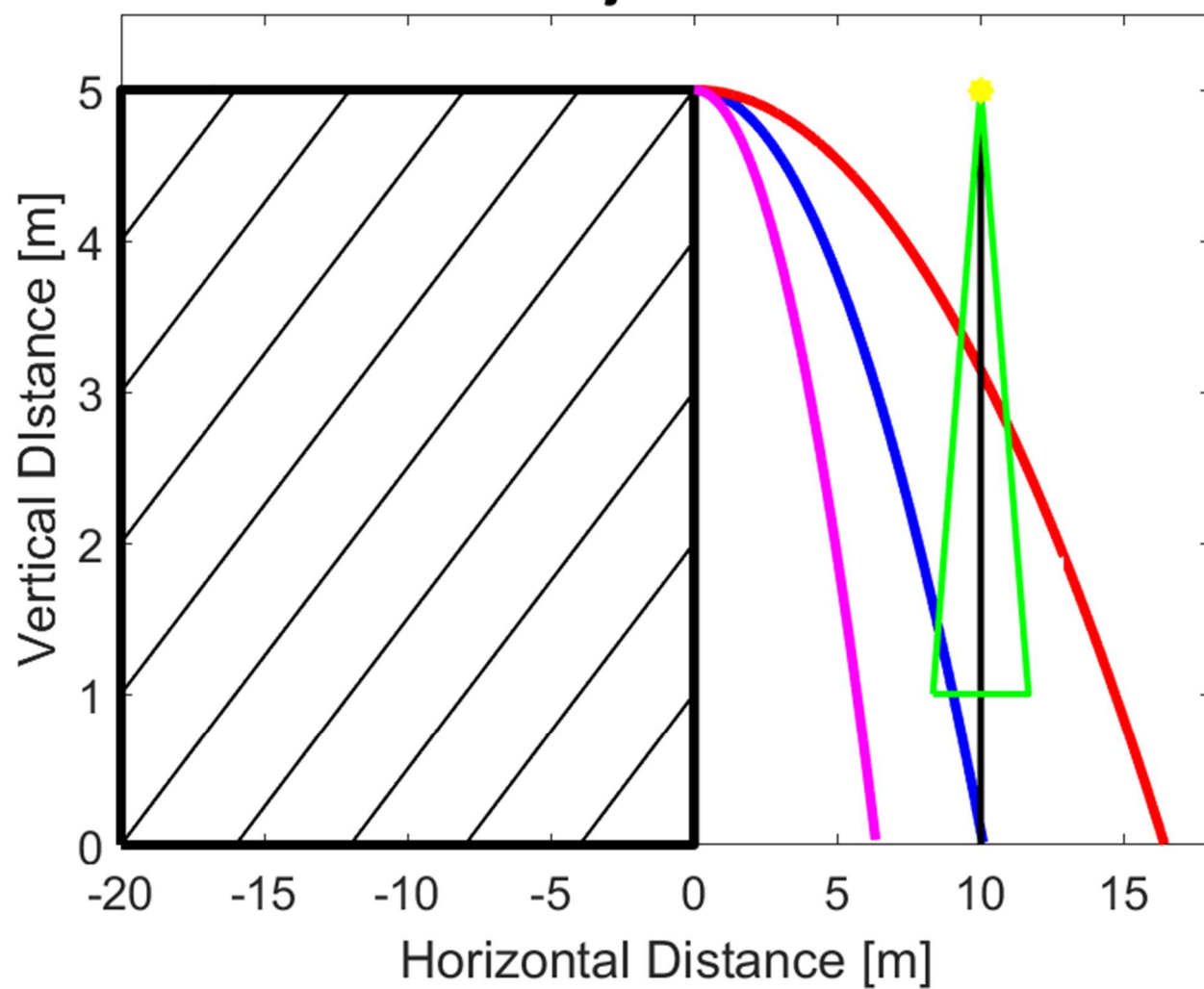
The initial for loop is designed to generate numerous coordinates along the projectile's path and store them in a matrix. These points are spaced at intervals of 0.003 seconds, ensuring a smooth, curved trajectory when connected. This loop iterates until reaching the "time" variable, which signifies the total duration until the projectile impacts the ground.

Subsequently, a secondary for loop is employed to connect these points, effectively rendering the curved trajectory. To facilitate observation of the motion rather than merely the final path, a "pause" function is implemented.

The variable "w" serves a dual purpose in this code: it determines the proper execution of the initial for loop to populate the matrix accurately and also dictates the number of iterations for the second for loop, ensuring alignment with the generated coordinates.

Here is a picture that shows the final plot after the 3 attempts:

Welcome to the Projectile Motion Simulator!



Showing results

After each try the results for that particular try should be displayed to the user, to do that I used the “fprintf” function and if statements.

Here is the part of the code that does that:

```
if distance_target(1,i) > 2
    fprintf('Your maximum height was of %.2f \nYou overshoot the target by %.2f\n',max_height(1,i),abs(distance_target(1,i)));
elseif distance_target(1,i) < -2
    fprintf('Your maximum height was of %.2f \nYou undershot the target by %.2f\n',max_height(1,i),abs(distance_target(1,i)));
else
    fprintf('Your maximum height was %.2f \nYou were within 2 meters from the target, your distance was %.2f! You win!! \n',max_height(1,i),abs(distance_target(1,i)));
end
```

To win the game, the user must land the projectile within a 2-meter radius of the target. Therefore, the if statements in this section of the code verify this condition. If the distance from the target exceeds 2 meters, indicating an overshoot, the "fprintf" function communicates this outcome. It includes details such as the maximum height reached by the projectile and the distance by which the target was overshoot. Both values are displayed with precision to two decimal places using "%.2f".

Conversely, if the distance from the target is less than -2 meters, indicating an undershoot, the user is similarly informed via the "fprintf" function. This time, the absolute value of the distance is used, as the negative sign is redundant when stating an undershoot. The message also includes details of the maximum height reached and the extent of the undershoot.

Finally, if the distance from the target falls within the 2-meter radius, signifying a successful hit, the user is notified of their victory using the "fprintf" function. Additionally, details such as the maximum height reached and the distance from the target are provided for

reference. I included the distance from the target in this section because, personally, I found it helpful during gameplay. Knowing the distance from the target allowed me to gauge my accuracy and aim for even closer proximity on subsequent attempts.

Here is how the results for the attempt appear in the command window in case the player undershot, overshoot and wins respectively:

```
Your maximum height was of 5.00  
You undershot shot the target by 3.65
```

```
Your maximum height was of 5.00  
You overshoot the target by 6.42
```

```
Your maximum height was 5.00  
You were within 2 meters from the target, your distance was 0.10! You win!!
```

Prompts for trying again

This part deals with potential errors made by the user while inputting values. To handle this, I introduced some "dummy variables." These are utilized within while loops to prompt the user to try again until a valid value is inputted, allowing for multiple attempts if necessary.

Here is part of the code that shows that:

```
h = 0; % to create the loop if the player write the wrong number
t = 0;
a = 1;
v = 1;
p = 1;
while h == 0
    starting_height = input ('Please input an starting height (5 - 20 m):'); %m
    if starting_height >= 5 && starting_height <= 20
        while t == 0
            target_placement = input ('Where do you want to place the target? (10
- 100 m):'); %m
            if target_placement >= 10 && target_placement <= 100
                h = h + 1;
                t = t + 1;
            else
                disp('This is an invalid target placement. Please try again.');
```

In this section, the while loops are initiated when specific variables are set to zero initially. If the user provides an invalid value, the code remains in the loop, continually prompting the user until a correct value is entered. The "else" section of the code displays a message indicating the invalid input, and the variable's value remains unchanged.

Once the user inputs a valid answer for these questions, both variables have 1 added to them, preventing the code from re-entering the loop. These questions are unique because they are asked only once, and after receiving a valid response, they are not asked again.

However, for values such as "starting angle," "initial velocity," and "planet," which may vary with each attempt, a different approach is required. To handle this, another loop is created to manage these changing values and ensure the code's functionality.

Here is part of the code for the other questions:

```
while i < 3
    a = a - 1; % so it enters the while loops for the questions again on later
    attempts
    v = v - 1;
    p = p - 1;
    i = i + 1; % to count the tries
    while a == 0 % while loops to allow mistakes
        starting_angle = input('Please input a starting angle. (0 - 80 degrees:');
        %rad
        if starting_angle >= 0 && starting_angle <=80 %if there is no mistake goes
        to the next question
            while v == 0
                initial_velocity = input('Please input an initial velocity. (10 - 50
m/s:'); %m/s
                if initial_velocity >= 10 && initial_velocity <= 50
                    while p == 0
                        planet = input('Please choose a planet ( 1 for Earth, 2 for
Mars, or 3 for Jupiter:');
                        % different gravity for each planet

                        if planet == 1
                            gravity = 9.81; %m/s^2

                            % to stop the loops

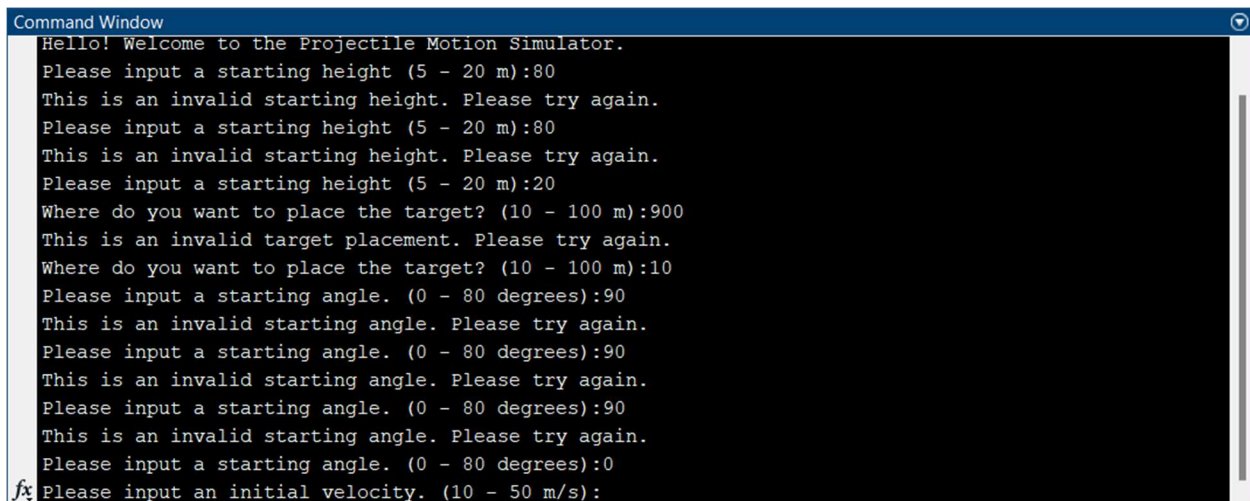
                            a = a + 1;
                            v = v + 1;
                            p = p + 1;
```

In the code provided earlier, the variables for this section start with a value of 1. The initial loop utilizes a variable "i" to denote the current attempt, beginning at 0 and incremented by 1 upon entering the loop, indicating the user's first try. The other variables begin at 1 so that they can be subtracted within the loop, allowing them to equal 0 and progress to the next loop, which solicits user input.

In the event of an invalid input, the else statements display an error message, with no value added to any variable. This allows the loop to continue until the user provides valid values for all questions, at which point 1 is added to the variables to exit the loop.

If the player chooses to play again, the code loops back to the initial while loop, subtracting the 1 previously added from the variables, and resetting them to 0. This enables the program to re-enter the loops prompting for user input. This iterative process continues until the user exhausts all 3 attempts or decides to stop playing.

Here is how that looks on the command window:



```
Command Window
Hello! Welcome to the Projectile Motion Simulator.
Please input a starting height (5 - 20 m):80
This is an invalid starting height. Please try again.
Please input a starting height (5 - 20 m):80
This is an invalid starting height. Please try again.
Please input a starting height (5 - 20 m):20
Where do you want to place the target? (10 - 100 m):900
This is an invalid target placement. Please try again.
Where do you want to place the target? (10 - 100 m):10
Please input a starting angle. (0 - 80 degrees):90
This is an invalid starting angle. Please try again.
Please input a starting angle. (0 - 80 degrees):90
This is an invalid starting angle. Please try again.
Please input a starting angle. (0 - 80 degrees):90
This is an invalid starting angle. Please try again.
Please input a starting angle. (0 - 80 degrees):0
fx Please input an initial velocity. (10 - 50 m/s):
```

Allow for multiple tries

The user is allowed to try to reach the target 3 times and to allow that I used the while loop discussed in the previous section and a few more “dummy variables”.

Here are parts of the code that show that:

```
kk = 0;
i = 0; %Number of tries
while i < 3
    i = i + 1; % to count the tries
    if i == 3 % to end the game on the tird attempt
        break
    end
    while kk == 0 % loop to ask if the player whats to go again
        play_again = input ('Do you want to try again? Type 1 for yes and 0 for
no:');
        if play_again == 1
            break
        elseif play_again == 0
            i = i + 3; % so it won't enter the loop again
            break
        else
            disp('This is an invalid answer. Please try again.')
        end
    end
end
```

The code enters the first if statement only after the user has completed three rounds of play. Upon entering this if statement, the "break" function is employed to prevent further inquiries about whether the user wishes to continue playing, as they have exhausted their allotted attempts.

The variable "kk" serves as the trigger for the loop asking the player if they want to play again. Its value remains unchanged throughout the code due to the presence of the "break" function. The user is prompted to input 0 to indicate their desire to stop playing or 1 to signify their wish to play again. Selecting 1 breaks out of the most recent while loop, returning the code to the third line, thereby allowing the user to initiate another round of play. Conversely, selecting

0 results in adding 3 to the variable "i," which tracks the number of attempts. This action ensures that the code does not re-enter the loop as "i" exceeds 3, effectively ending the program.

Here is how that looks on the command window:

```
Do you want to try again? Type 1 for yes and 0 for no:3  
This is an invalid answer. Please try again.  
Do you want to try again? Type 1 for yes and 0 for no:9  
This is an invalid answer. Please try again.  
Do you want to try again? Type 1 for yes and 0 for no:8  
This is an invalid answer. Please try again.  
Do you want to try again? Type 1 for yes and 0 for no:0  
Thanks for playing!
```

Show all results

To display all results once the player finishes, I integrated if statements and "fprintf" functions into the previously discussed code segment.

```
if i == 3 % to end the game on the third attempt
    disp('Thanks for playing!')
    fprintf(['Your results are as follows: \nAttempt ' ... %player's results
            '#1: max height = %.2f, Distance from Target = %.2f.\nAttempt #2: ' ...
            'max height = %.2f, Distance from Target = %.2f.\nAttempt #3: ' ...
            'max height = %.2f, Distance from Target =
            %.2f.\n'],max_height(1,1),abs(distance_target(1,1)), ...
            max_height(1,2),abs(distance_target(1,2)),max_height(1,3),abs(distance_target(1,3)));
    break
end
```

This if statement resides within the overarching while loop mentioned earlier. The "break" function is employed to halt the code execution, preventing further prompts asking the player if they wish to play again.

Here is the section of the code that shows the results if the player decides to stop playing before the 3 attempts:

```
elseif play_again == 0
    disp('Thanks for playing!')
    if i == 1
        fprintf(['Your results are as follows: \nAttempt ' ... % showing the player
                their results
                '#1: max height = %.2f, Distance from Target =
                %.2f.\n'],max_height(1,1),abs(distance_target(1,1)));
    elseif i == 2
        fprintf(['Your results are as follows: \nAttempt ' ...
                '#1: max height = %.2f, Distance from Target = %.2f.\nAttempt #2: ' ...
                'max height = %.2f, Distance from Target =
                %.2f.\n'],max_height(1,1),abs(distance_target(1,1)), ...
                max_height(1,2),abs(distance_target(1,2)));
    end
    i = i + 3; % so it won't enter the loop again
    break
```

This section distinguishes whether the player was on their first or second attempt using if statements and the variable "i," which tracks the attempts. Inside these if statements,

corresponding "fprintf" functions inform the player of their results. The "break" function halts the code execution, and adding 3 to the variable prevents the code from re-entering the main loop.

Here is how it looks after stopping on the first try:

```
Thanks for playing!  
Your results are as follows:  
Attempt #1: max height = 5.00, Distance from Target = 3.65.
```

This next picture shows how it appears in the command window if the player stops in the second attempt:

```
Thanks for playing!  
Your results are as follows:  
Attempt #1: max height = 5.00, Distance from Target = 3.65.  
Attempt #2: max height = 5.24, Distance from Target = 2.60.
```

This last picture shows how it appears in the command window if the player uses all of the attempts:

```
Thanks for playing!  
Your results are as follows:  
Attempt #1: max height = 5.00, Distance from Target = 3.65.  
Attempt #2: max height = 5.00, Distance from Target = 2.70.  
Attempt #3: max height = 6.18, Distance from Target = 3.47.
```

Conclusion

In conclusion, this project proved to be quite time-consuming, and I encountered numerous challenges along the way. One significant hurdle was enabling the player to rectify multiple mistakes. Initially, I attempted to use the "while 1" function followed by a break statement, but this only resulted in breaking out of the latest while loop, causing the other questions to be continuously prompted. Additionally, I struggled with implementing the motion of the projectile's path, requiring several attempts to resolve errors and ensure smooth execution.

Despite these obstacles, I found the project to be engaging and rewarding. It provided valuable insights into coding principles and logic. Overall, while it was a lengthy endeavor with its share of difficulties, I believe I gained a deeper understanding of coding through the process.