Model for Code Module

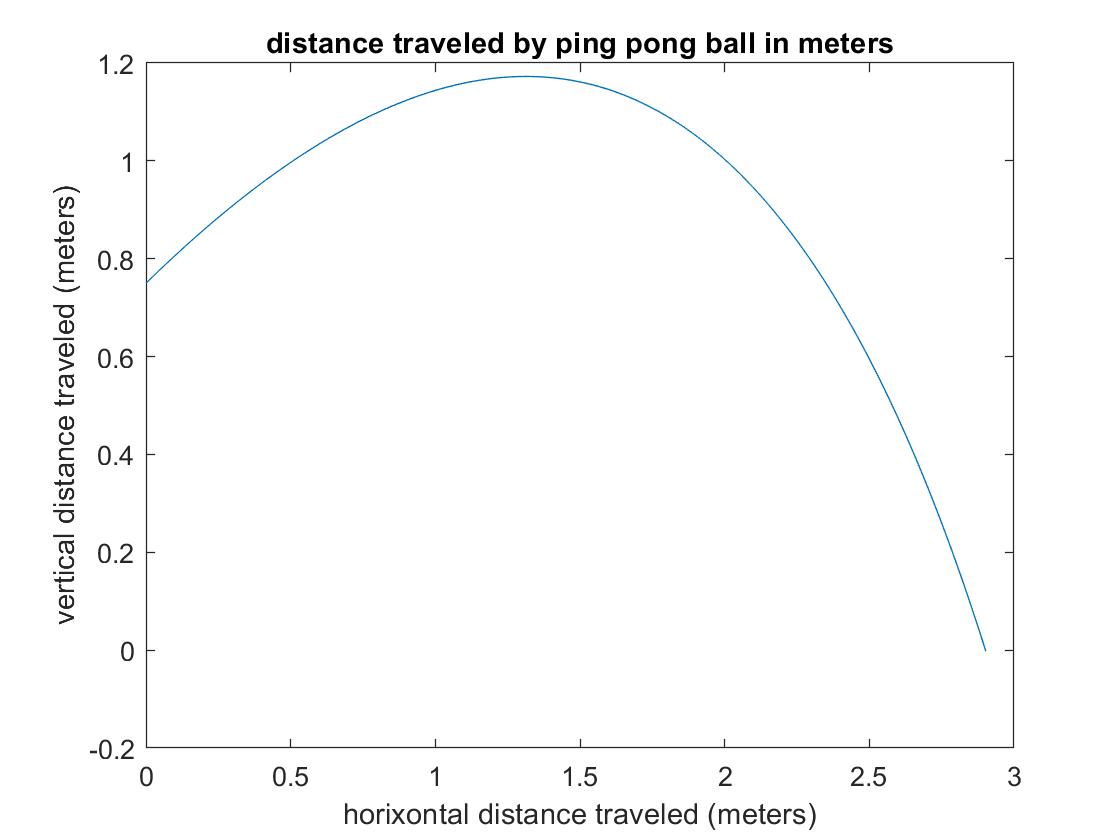
# Code modeling overview

The “controlling code” portion of this project ties many key aspects of device functionality together. The code does, however, have to contribute several key design features in addition to its coordination function. These features include how to handle the user interface, how to structure the handoff of information from the vision system in terms of distance to the rotating base in terms of movement. This may or may not include feedback information from the vision system as to how well the device is aimed. The code needs to also make the user wait until the system is cocked before launching the ball, and of course not launch until the child wishes.

The most technically challenging part of this effort is to convert the distance to the device into a vertical launch angle for the ping pong ball. This document will help describe a way to generate a calibration for this conversion that may or may not be the final method for your device. Either way it will serve as a good system model that will enable you to understand your operating space for movement and build code around it to create basic functionality.

The MATLAB model with drag

In order to model the motion of the ball with air drag you created an iterative model for the movement of the ping pong ball through the air. As the ball traveled wind resistance slowed the motion – a force opposite to the motion of travel. This impacted the downward and horizontally forward movement differently as there is downward acceleration due to gravity that needed to be accounted for. The resulting model showed that for high amounts of air resistance the path of the ball would be far from the perfect parabola of regular projectile motion. This is shown in figure 1 below.

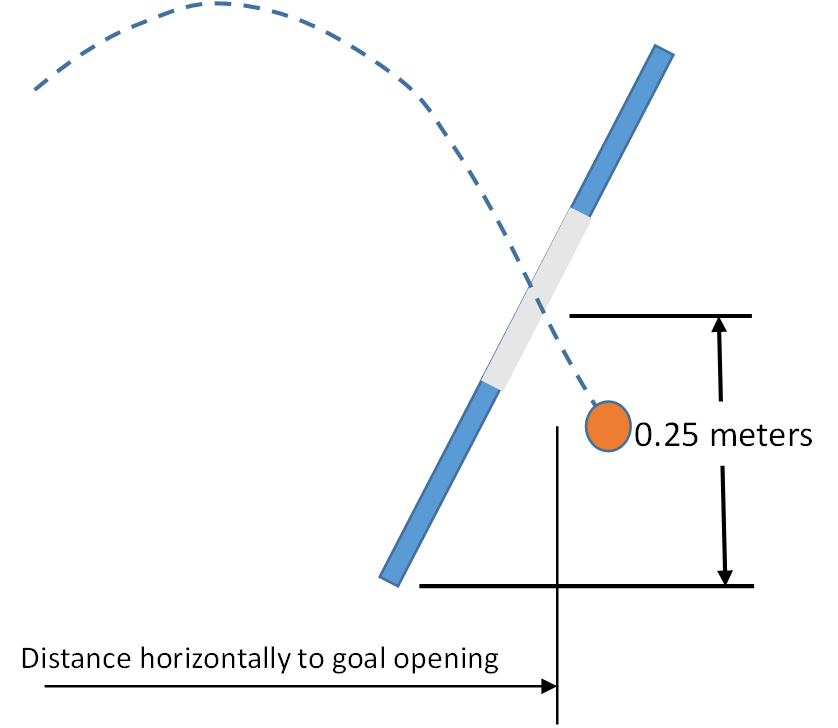


*Figure 1: Model output with drag, 6 m/sec initial velocity, launched from a table at 30 degrees*

Since the mathematics is iterative it is not easy to “reverse” it and move backwards to create an equation. (Note that it is not impossible, it just isn’t easy). A simpler method is to use the model you generated for the motion with drag to create a model to relate distance to needed angle.

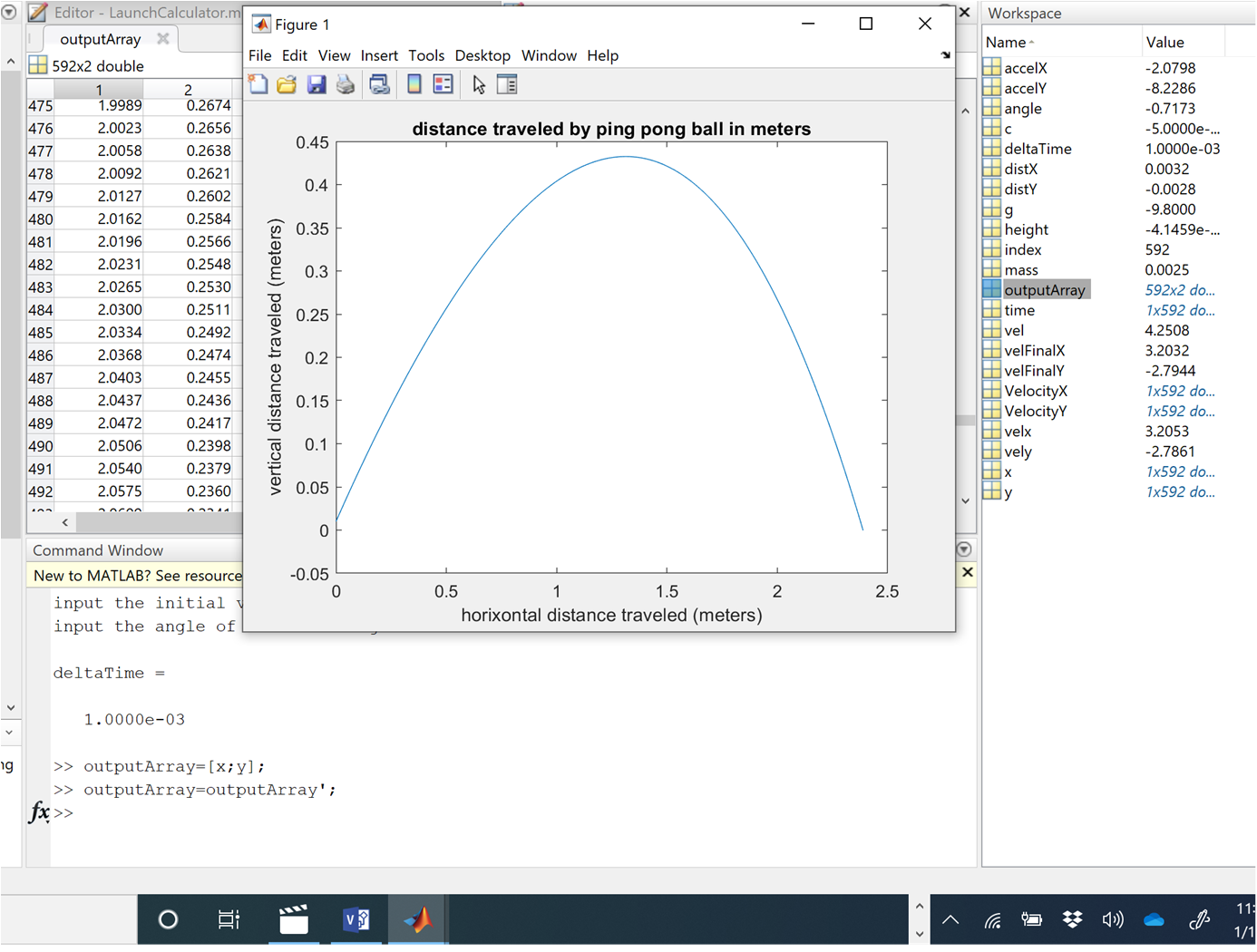
Determine the height of the goal opening

The goal board opening likely is not at the floor. More likely your scoring system team member has it at some height above the ground. If that part of the design is still not set, begin your model by assuming it will be about 0.25 meters above the ground. For most goal board designs this is a good first approximation. If the ping pong ball is at about this altitude as it passes the horizontal point where your goal has been theoretically set, then you can consider your model to have scored a goal. That is shown in figure 2 below.



*Figure 2: The critical landing distance is actually when the ping pong ball passes through the goal opening height. For the purposes of this model that value will be assumed to be .25 meters (to the center of the opening when the goal is on a stand tipping it back just a bit). Your design will vary, but if you don’t have a good number to work with, this value is sufficient to get the model started. You can always go back and change that number in your code later.*

In order to determine the horizontal and vertical location of the ball at every point in the trajectory, it is useful to modify your launcher model to have an array variable that has the x and y pairs stored together. That way you can search the array for a given height or a given distance. An example of a command that does that is in the command window in figure 3 below.



*Figure 3: Collecting the x and y values at the bottom of your code can create an array that makes it easier to look at x and y together and see what is happening. This works great for a quick one-time check.*

Creating a Model to help guide calibration

The goal of this document is to help you take the MATLAB model with air drag and turn it into a model that can help you and your team convert distance needed to launch angle. The distance needed will be the distance from the launcher to your target at the height of the goal opening as described in the section above. Since your ball exit velocity will be considered a constant once determined, the model will run for a single exit velocity, but will change the exit angle through a range of angles to help find the best one for this launch. It will also help you get a curve that you can convert into a calibration curve.

Determining the exit velocity of the ball

The good news is that the exit velocity of the ball should be a constant once your team design is finalized. The only variable you will need to worry about for distance is the launch angle. That having been said it is likely that your team has not yet solidified its design for the launch tube, and there may be optimizations in the future.

If you have a fairly consistent launch set-up the best practice is to use actual distance and angle results to calibrate your model. There are two degrees of freedom which can be modified to bring your MATLAB model into closer agreement with your actual device. The most obvious one is the exit velocity. The other is the air drag. The energy calculation done by the team or by the systems integration team member has an estimate for exit velocity. That estimate is the high end of what you actually have as it does not take into account any loss for friction. Guaranteed – you have frictional losses. You might be losing as much as 1/3 of your energy to friction and can still get the ball to go a reasonable distance.

Determining a good estimate for the actual exit velocity using your model can give the launcher team valuable feedback as to how much friction could be reduced and efficiency improved. Improving efficiency will not only have your ball launch better (faster and more consistently) but it will also reduce the probability of part breakage because you have had to store less energy in the system. It also will help you in developing a model as to how to correctly aim your launcher.

Determining the landing spot:

Since the coefficient of drag was determined experimentally using a wind tunnel and ping pong ball we can assume it is a pretty good estimate starting out. We now want to create a curve of distance (when 0.25 m above the ground) as a function of angle launched for different launch velocities.

When you run the MATLAB model your “x” variable is your horizontal distance and your “y” variable is your height. Since the model is taking finite steps it is not possible to have MATLAB pull out right where the “y” value comes down below the 0.25 meter mark. It is possible to ask MATLAB to find the index values of where that happens. The bold code below was added to the model just before the end of the loop. You can find the non-bolded code in your program to know where to locate it.

% save distance values

x(index)=x(index-1)+distX;

y(index)=y(index-1)+distY;

height=y(index);

**if distY<0 && flag <=1**

**if height<.25**

**disp(distY)**

**fprintf('goal height detected at distance %.2f meters.',x(index));**

**flag =2;**

**LandingDistance(i)=x(index);**

**end**

**end**

*Code insert1: This set of two nested if statements allows the code check as the ball is falling where it passes the 0.25 meter point. When it does it prints out a statement as to what horizontal distance that happens at and sets a flag so you only get this once per run.*

In order to get this code to work you need to initialize a flag variable (flag=0) before you start into the iterative loop.

Add this to your code and get it working.

Now add an outside loop to iterate through multiple angles:

You would like to create a curve from which you can look at how far you need to go and choose a launch angle to make that happen. Additionally, for the purposes of developing your design, you might want to understand what your range of options are so that you can develop a design that will use the optimum strategy for aiming.

The overall method of doing this will be to create a for loop that walks through a series of angles you would like to test, for example from 20 degrees to 80 degrees in 5 degree increments. Instead of taking the launch angle as an input from the code user, you can replace it with an array of those values and plan to “run the simulation” for each of the angles, compiling the total result at the end.

To do this, comment out the “input” command that asks for an angle, and put in an array of values for angles to test. The following code will do that for the 20 to 80 degree range just discussed. Of course you can modify that range and even the 5 degree resolution to suite your needs.

angle=20:5:80;

angle=angle\*pi./180;

rangeAngle=length(angle);

%angle=input('input the angle of launch in degrees: ');

% convert the degrees into radians so MATLAB likes it

*Code insert 2: At the very top of your code (right after the clear all command) you can set up the array for your angle. The first command uses a sequence command to create the array. The second line converts all of the degrees into radians which MATLAB likes better. What does the third command do? You can see that the original “input” for the angle has been commented out.*

Getting a loop set up:

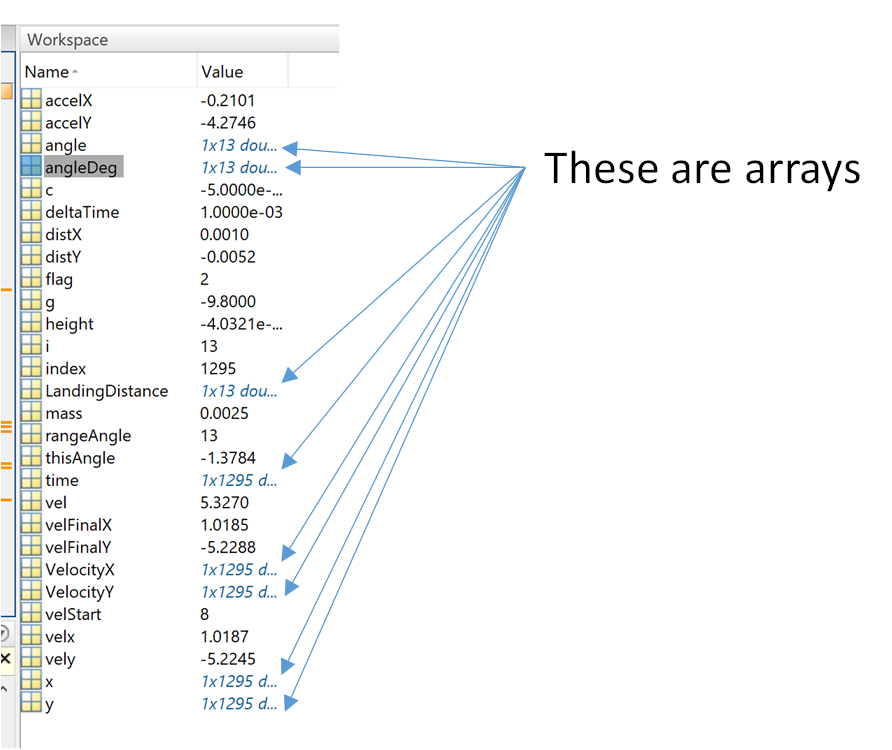
Looping on a loop requires that you make sure that variables, especially those that create arrays during your loop get cleaned up between iterations. You have in your code a while loop that checks to see if the ball has hit the ground yet. That loop will be INSIDE your for loop to change the angle. Before going into the for-loop the initial launch velocity needs to be set. Because the velocity variable (vel) changed this value inside the while loop as the program iterated, the initial value of the velocity will need to be preserved. The input command for the initial velocity has been changed in this example to:

velStart=input('input the initial velocity in m/sec: ');

*Code insert 3: Change the input statement for the initial velocity to give that value a different variable name at the start of the program.*

The for-loop can now be started. The program is to run through and generate a simulation for the first angle, clean up the variables while storing the main results, then run again for the next angle. The for-loop should run one time for each angle variable you are requesting. That is why in code insert 3 the “rangeAngle” variable was created. It counted how many items there were in the array “angle”. That variable can be used as the “counter” for the for-loop.

Once the for-loop is started, all of the key array variables will need to be re-set so that values from a previous iteration of the model don’t mess things up. The easiest way to do that is to use the [ ] command that clears the contents of an array. If you look at your workspace you can easily see the variables that are arrays. Figure 4 shows what arrays look like in the workspace.



*Figure 4: This image of a MATLAB workspace for this program shows which items are arrays. LandingDistance was created to hold the multiple results of multiple runs, so you don’t want to clear it between runs. Other arrays like “time” and “VelocityX” will need to be cleared between runs. Can you see which ones you are going to need to clear?*

If the arrays created each loop always had the same number of items in them, then the loop can be designed to “write over” the previous values. However the while-loop does not give a consistent number of items per array each time, and so clearing them out is the best practice. The following code shows how to set up the variables before going into the while-loop.

for i=1:rangeAngle

% set initial position and time

x=[];

y=[];

time=[];

vel=velStart;

VelocityX=[];

VelocityY=[];

x(1)=0; % meters

y(1)=.01; % meters

time(1)=0; % seconds

mass=.00247; %kg ping pong ball .00247 Kg

g=-9.8; % m/sec^2

c=-0.0005; % coefficient of drag where Re is between 10^3 and 10^5

% so I can load an array for plotting

% start to increment the motion

index=1;

thisAngle=angle(i);

velx=vel\*cos(thisAngle);

VelocityX(1)=velx;

vely=vel\*sin(thisAngle);

VelocityY(1)=vely;

% set a time step

deltaTime=.001 % seconds

height=y(1);

flag=0;

while height>=0 % it has not hit ground yet

*Code insert 4: This code insert shows how to get from the start of the for-loop to the start of the while-loop. Note the arrays that have been reset using the [ ] command. The velocity is reset to the input velocity (velStart). We also want to take just one angle from the array to use in the iteration. The variable “thisAngle” is created to hold just the current angle for the simulation, and so the cosine and sine calculations are modified to use that variable. Also note that this is where the flag is set to zero so that you only get one “result” for distance to target per angle.*

Inside the while loop make sure that the angle being used is also changed to the “thisAngle” variable.

while height>=0 % it has not hit ground yet

index=index+1;

% break velocity into its components

velx=vel\*cos(thisAngle);

vely=vel\*sin(thisAngle);

*Code insert 5: Just inside the while loop you will need to modify the “angle” variable to “thisAngle”*

Now go back and add “hold on” just before the for-loop and put the end of the for-loop right AFTER the plot command. Add “hold off” after that so that. In the code insert below the first “end” is from the while loop. Then you have the plot command, the end of the for loop and the “hold off”.

end

plot(x,y)

title('distance traveled by ping pong ball in meters')

xlabel('horixontal distance traveled (meters)')

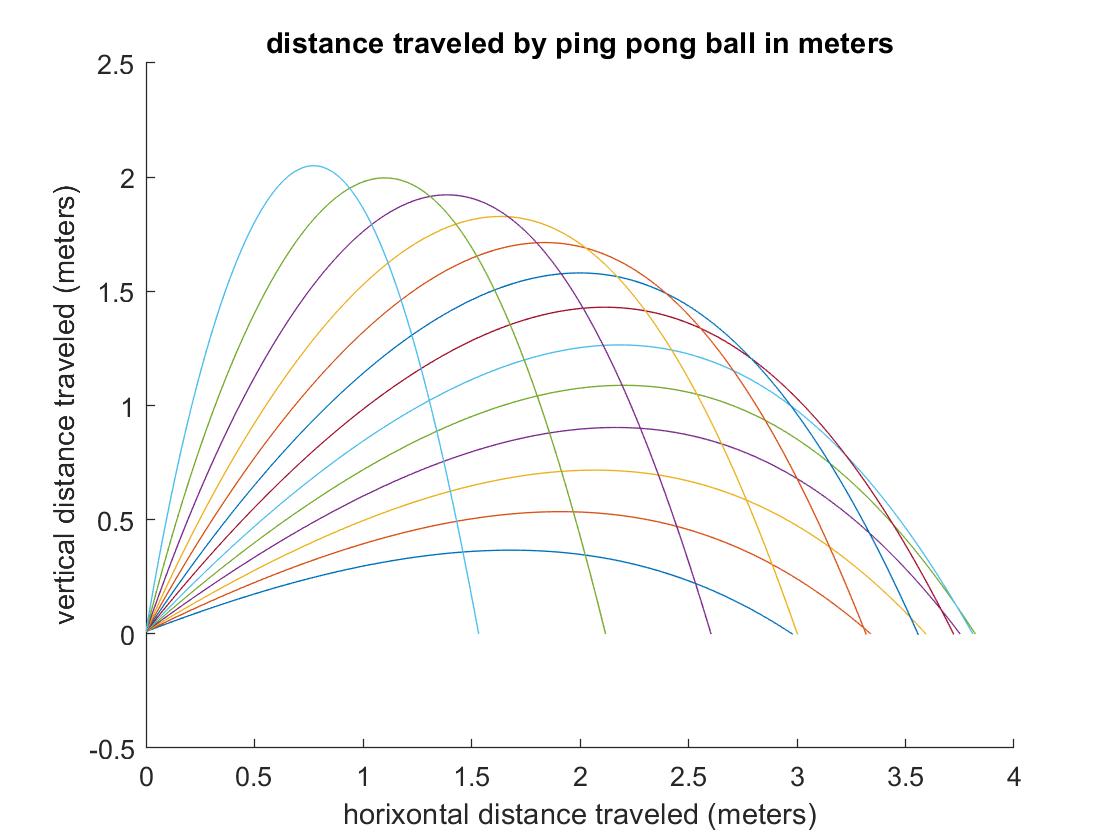
ylabel('vertical distance traveled (meters)')

end

hold off

*Code insert 6: This is how to wrap your old plot command to get a lovely multi-curve plot. Note that you need to add “hold on” before the for-loop starts.*

Run your code to see the results and debug as necessary. Use 8 m/sec as a starting velocity. You should get a result that looks like the graph shown in figure 5.



*Figure 5: A multi-curve plot showing the path of the ping pong ball at 8 m/sec initial velocity for a range of angles. This plot gives an idea of trajectory and how the choice of angle impacts what happens.*

The goal of this model was to create a curve from which a calibration curve could be generated. That will be created as a second figure in this model. The first curve can be preserved by the command “figure” which creates a separate second figure after the hold off. This figure will plot the angle of launch (in degrees) against the distance at the goal height.

end

hold off

figure

angleDeg=(angle.\*180)./pi;

plot(angleDeg,LandingDistance);

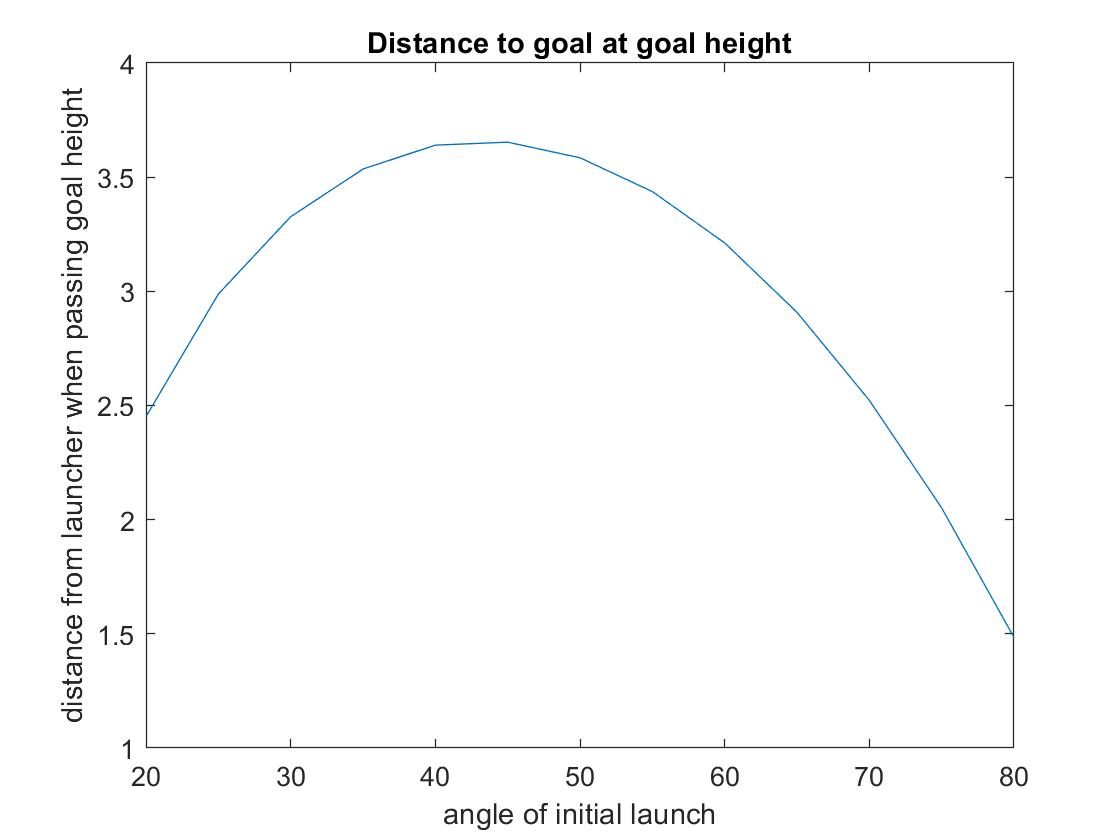
title('Distance to goal at goal height')

xlabel('angle of initial launch')

ylabel('distance from launcher when passing goal height')

*Code insert 7: The “end” is the closing of the for-loop and the hold off stops the graph making for the multi-curve graph shown in figure 5. The next set of code creates a new figure, converts the angle which are in radians back into degrees and plots the “LandingDistance” (the distance from the launcher to the place where the ball is at goal opening height) against the angle (measured in degrees).*

Running this code with an exit velocity of 8 m/sec gave a graph from code insert 7 that is shown in figure 6.



*Figure 6: The graph of the distance to the goal as a function of the initial launch angle for a given exit velocity.*

Getting to a calibration curve:

The last step in the process is to use the model to get to a calibration curve. Your calibration curve will have an input of the distance to the goal and an output of the vertical angle that the launcher needs to be set at. This part of the process is not part of the proof of concept modeling but here are a few steps to help get there.

1. Create a single array with the variables used to create the plot in figure 6. **You do this once you and your team are comfortable in the right exit velocity to use for your system.** You can use the code shown in figure 3, changing the variables used to create the array to angleDeg and LandingDistance.
2. You can use the MATLAB fit command or export to excel (xlswrite) and use their graphing functions that will let you try curve equations for a best fit. Remember to have excel put the equation on the graph so you have it. **This is just like you did for the mini-project in EGR 102.**
3. Write into your final control code the equation generated by the above 2 steps. Note that this simulation code is MODELING and is not part of your final control code.