HW5: Trajectories
CSE1010 Fall 2012
Jeffrey A. Meunier
University of Connecticut

1. Introduction

In this project you will write a program that calculates and plots the trajectory of a projectile. The user is prompted to enter values for mass, energy, angle, and the initial height of the projectile, and the program will calculate and plot the projectile's trajectory. This simulation will ignore complicating factors such as friction, air resistance, spin, and rebound.

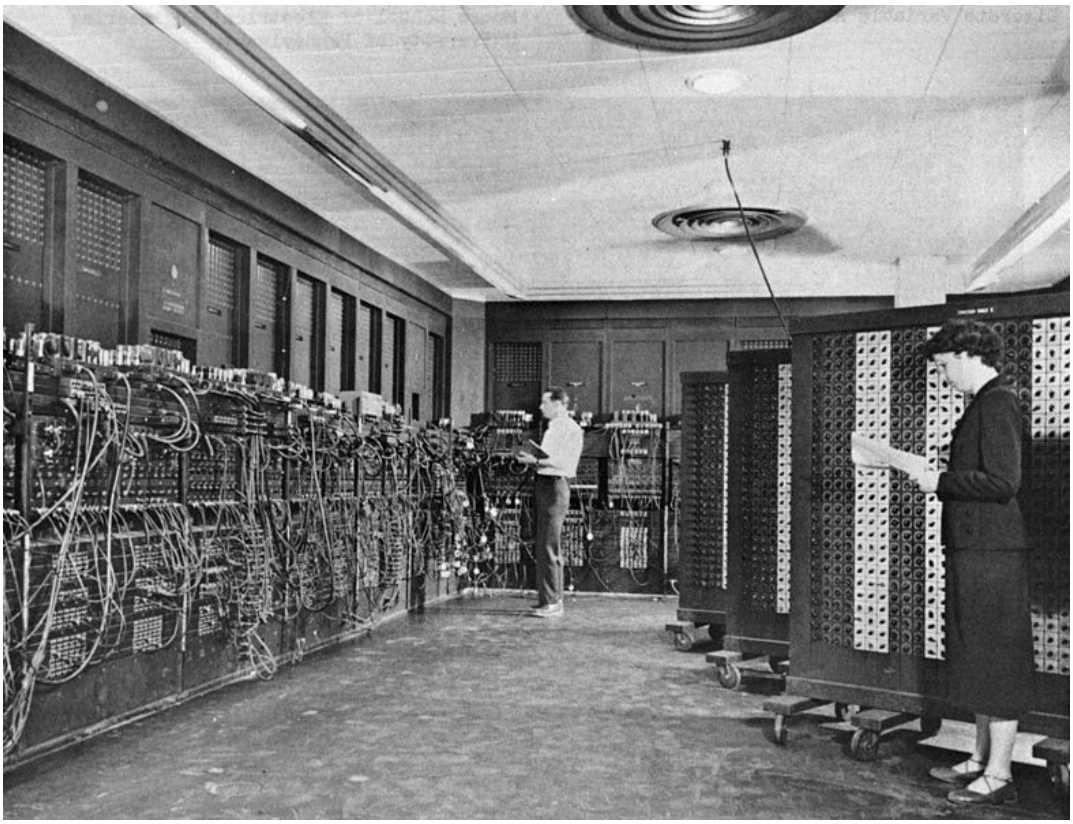
A *trajectory* is the path followed by a projectile.

A *projectile* is an unsupported object that is moving through space under some force (like a rocket) or under its own momentum (like a ball or a rock).

1.1. History

By writing this program you will accomplish in under a week, and using a software tool that cost $99 and a computer that cost approximately $1000 (within an order of magnitude), what took the builders of the ENIAC computer several years and several million dollars to do.

ENIAC is an acronym for *Electronic Numerical Integrator and Computer*. It was designed to calculate ballistics tables for the US Army during World War II. It was programmed by plugging in wires and routing them between panels. This computer was not portable by any means, although it was moved from Pennsylvania to Maryland one time during its operational life.



"Besides its speed, the most remarkable thing about ENIAC was its size and complexity. ENIAC contained 17,468 vacuum tubes, 7,200 crystal diodes, 1,500 relays, 70,000 resistors, 10,000 capacitors and around 5 million hand-soldered joints. It weighed more than 30 short tons (27 t), was roughly 8 by 3 by 100 feet (2.4 m × 0.9 m × 30 m), took up 1800 square feet (167 m2), and consumed 150 kW of power. " "ENIAC... could perform 5,000 simple addition or subtraction operations... every second." (http://en.wikipedia.org/wiki/Eniac)

Compare this to the cell phone in your pocket. The CPU alone contains anywhere from 200 million to 400 million transistors, can perform anywhere from 5 hundred million to 1 billion operations every second and consumes about 1W of power.

It is interesting to note that many of the first programmers of the ENIAC, and hence the very first programmers, were women.

2. Value

This program is worth a maximum of 20 points. See the grading rubric at the end for a breakdown of the values of different parts of this project.

3. Due Date

This project is due by 11:59PM on Sunday, October 14, 2012.

4. Objectives

This project introduces global variables. Additionally, it gives you more experience using loops, and of course gives you more practice with functions and everything else you've learned up to this point. You will also need to translate mathematical expressions into Matlab code.

Keywords: global variable, loop, function, plot, trajectory

5. Background

An interesting note about projectiles: A projectile that is fired away from the earth will accelerate back toward the earth under the influence of gravity, and travel in an almost parabolic path. In the absence of drag (air friction), when it returns to the same elevation at which it was launched, it has exactly the same velocity at which it was launched.

In this assignment you will calculate only the primary trajectory of a projectile, and not the subsequent rebound trajectories. In other words, your projectiles will not bounce.

5.1. Formulas

The relationship between an object's energy, mass, and velocity is expressed by this equation:

$$e = \left(\frac{1}{2}\right)mv^2$$

Alternately, solve for m or v if the other terms are known.

Given an object's current horizontal position x and its horizontal velocity v_x, the location of the object x_{next} after some time t can be calculated using this expression:

$$x_{next} = x + v_x t$$

The time t_{ap} it takes a projectile to reach its apogee (the highest point of the trajectory arc) is described by this expression:

$$t_{ap} = -\frac{v_y}{g}$$

where v_y is the object's vertical velocity, and g is the acceleration due to gravity (-9.8 m/s on Earth).

The vertical height y reached by an object after a certain amount of time t when the object has a positive vertical velocity is given by this expression:

$$y = y_0 + v_y t + \left(\tfrac{1}{2}\right) g t^2$$

where y_0 is the current (or initial) height of the object, v_y is the object's vertical velocity, and t is the amount of time the object is allowed to move.

The time t it takes an object to fall from a certain height to the ground is given by this expression:

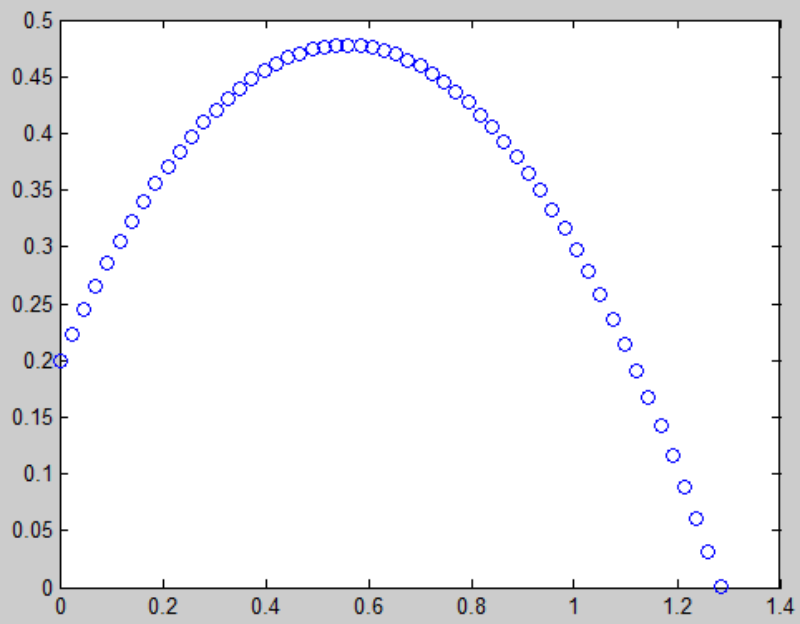
$$t = \sqrt{-\frac{2d}{g}}$$

Where d is the initial distance from the object to the ground.

5.2. Example

Pictured below is the trajectory arc traced by a 45.9 gram projectile (the precise mass of a golf ball) propelled from an initial height of 0.2 meters at an angle of 45 degrees with an energy of 0.5N (N is the scientific abbreviation for *Newton*, where 1N = 1 kG m / s^2). The arc reaches a maximum height (its apogee) of 0.47 meters after 0.23 seconds. It took an additional 0.31 seconds to fall to earth from the apogee and covers a horizontal distance of 1.28m The trajectory points (blue circles) are plotted every 0.01 second starting from 0 seconds.

In the plot below the projectile does not appear to be fired at a 45 degree angle because of the way Matlab scales the X axis different from the Y axis, even though both axes are in meters.



5.3. Global variables

A global variable is one that is accessible to any script or function that declares the variable as global. It can be declared many times, but it should be defined only once.

```
% script setupGlobals.m
% define G to be a global variable and give it a value:
global G = -9.8
```

If the script **setupGlobals** is run first, then any function that is called later will have access to that value in that variable.

```
function calcTrajectory(...)
  % declare G to be a global variable
  global G
  % now this function can use G, and it will have the value -9.8
  ...
end
```

You can read more about defining global variables on this web page:
http://www.mathworks.com/help/matlab/ref/global.html

6. Assignment

Create the following script and function files.

6.1. Script setupGlobals

This is a script file that defines a single global variable called **G**. The value of **G** must be set to -9.8. Note that for this program **G** *must* be negative: the acceleration due to gravity is negative on the Cartesian plane.

6.2. Function getInputs

This is a function file that has no parameters, and a single return variable called inputs. Do the following:

1. Display the message `'Enter a negative mass to exit the program.'`
2. Ask the user to enter the mass in grams. Store it in a variable.
3. If the mass is negative, set the return variable to the empty vector `[]` and skip over the rest of the function.
4. Otherwise, ask the user to enter the force in Newtons, the angle in degrees, and the initial height in meters.
5. Display a newline using **fprintf**. Just do this: `fprintf('\n')`
6. Store the mass, energy, angle, and initial height into the inputs return variable as a row vector. Remember that I showed you early in the semester how to build a vector using variables.

Here's how mine works:

```
>> i = getInputs
Enter a negative mass to quit the program.
Enter mass in grams: -1
i =
    []

>> i = getInputs
Enter a negative mass to quit the program.
Enter mass in grams: 50
Enter mass in grams: 50
Enter force in Newtons: 0.5
Enter angle in degrees: 45
Enter height in meters: 0.1
i =
    50.0000  0.5000  45.0000  0.1000
```

6.3. Function convertInputs

This function has a single parameter and a single return value. It expects the value in the

parameter to be a vector of 4 values: mass, energy, angle, and height; and does the following conversions:

1. Convert mass in grams to mass in kilograms.
2. Convert angle in degrees to angle in radians.

The function must place a vector of 4 values on the return variable as a row vector, just like the **getInputs** function, only use the values of the mass in kg and angle in radians.

The way I wrote the function is this: Take the 4 values out of the vector and assign them to separate variables. I called the parameter **inputs**, so I wrote statements like this:

```
__ = inputs(1);
__ = inputs(2);
```

and so on, only I used actual variable names in place of the blanks.

Modify those variables that need to be converted. Build the return vector from those variables.

Ask Google if you don't know how to do the conversions.

Assuming that **i** is the vector returned by the **getInputs** function:

```
i =
    50.0000  0.5000  45.0000  0.1000
```

Here's how my function works:

```
>> ci = convertInputs(i)
ci =
    0.0500  0.5000  0.7854  0.1000
```

6.4. Function displayConvertedInputs

This function has a single parameter. It expects the value in that parameter variable to be a vector like the one returned by the **convertInputs** function and displays the values in a nice way:

```
>> displayConvertedInputs(ci)
mass   = 0.05 kg
energy = 0.5 N
```

```
angle  = 0.785398 rad
y0     = 0.1 m
```

Use the **fprintf** statement with the **%g** format specifier to display each number. Make sure the equal signs line up vertically like I show here. After the final **y0** line display an blank line. That means that this function displays 6 lines. To display a blank line using **fprintf**, use the string **'\n'**.

This function does not return anything.

6.5. Function calcDependents

This function calculates a group of dependent variables. They are dependent because they depend on the values of the input variables. The values of these variables are not difficult to calculate, but they can't be calculated until after the user enters the values for the inputs.

This function has a single parameter. It expects the value in that parameter to be a vector that contains 4 values (mass, energy, angle, height (or y0)) and calculates the following values:

1. v: initial velocity; you already know the mass and energy
2. v_x: x-component of the velocity; this is the velocity times the cosine of the angle
3. v_y: y-component of the velocity; this is the velocity times the sine of the angle
4. t_{ap}: time to apogee; you know $\mathbf{v}_y$ in *m/s* and you know G in *m/s2*, so solve for time
5. y_{ap}: you know all the values to solve this formula (where y_{ap} is the same as y):

$$y = y_0 + v_y t + \left(\frac{1}{2}\right) g t^2$$

6. t_{fall}: time it takes the object to fall from the apogee to the ground (hint: the *y* component of the velocity is 0 at the apogee)
8. t: total time, $\mathbf{t}_{ap} + \mathbf{t}_{fall}$
9. $\mathbf{d}_h$: total horizontal distance moved by the object in time t

Use the formulas presented in the background section of this document.

When a formula refers to the gravitational constant g, use the global variable **G**. In order to

use **G** in this function you must declare it as a global variable inside this function. DO NOT RE-DEFINE **G** IN THIS FUNCTION (meaning, do not assign a value to **G** in this function. **G** already has a value, you simply need to give this function access to it).

The function returns all 8 values in a single vector. Here's how mine works:

```
>> d = calcDependents(ci)
d=
  Columns 1 through 7
    3.1623 2.2361 2.2361 0.2282 0.3551 0.2692 0.4974
  Column 8
    1.1122
```

6.6. Function displayDependents

This function works similar to the **displayConvertedInputs** function. It takes a single vector of 8 values and displays them. Here's how mine works:

```
>> displayDependents(d)
Velocity              = 3.16228 m/s
Vx                    = 2.23607 m/s
Vy                    = 2.23607 m/s
Time to apogee        = 0.22817 s
Height at apogee      = 0.355102 m
Time to fall to earth = 0.269202 s
Total time            = 0.497372 s
Distance traveled     = 1.11216 m
```

Also display a blank line after the last **Distance traveled** line.

6.7. Function calcTrajectory

This function takes the inputs and dependent values and generates a matrix of x/y values that will later be used to plot the trajectory points.

This function must have two parameters: it takes both the vector of **inputs** and the vector of **dependents**.

Inside this function you will need to extract from the inputs and dependents for only those values that you will need. You won't need all of them. For example, if the parameters are named inputs and dependents, you would have several statements like this:

```
__ = inputs(__);
__ = dependents(__);
```

Create a vector of time values from 0 to *t* counting by 0.01 seconds. Recall that *t* is the total time, and you can find that value in the vector of dependent values. Store this new vector of times in a variable.

Create a vector of *x* values that go from 0 to **dist** (the total distance traveled). It must have the same number of values as the times vector does that you just created. In other words, you don't know the spacing between each of the *xs* (*xs* is the plural of *x*) in the vector, but you do know how many there are. I think you'd be better off using the **linspace** function than using the colon notation to create this vector.

Create a vector of y values that depend on the values in the times vector. Each value in the ys vector relates to each time value with this expression:

$$y = y_0 + v_y t + \left(\frac{1}{2}\right) g t^2$$

The values for y_0, v_y, and g (use **G**) are found in the parameter vectors. The values for t are found in the times vector, but how do you plug in an entire vector into that expression?

It turns out that by using the **.^** operator to raise a vector to a power (type **help power** or **doc power**), you can plug the vector into that equation directly in both places for the variable t.

You can use either * (star) or **.*** (dot-star) to multiply v_y by t. Matlab detects that one is a scalar and uses **.*** automatically. Type **help times** for information about the **.*** operator.

Return the **xs** and **ys** vectors from this function in the form of a two row matrix with **xs** in the first row and **ys** in the second row. This means that you need to figure out how to construct a 2-dimensional matrix from two 1-dimensional vectors.

```
>> t = calcTrajectory(i, d)
t=
  Columns 1 through 7
         0  0.0227  0.0454  0.0681  0.0908  0.1135  0.1362
    0.1000  0.1219  0.1428  0.1627  0.1816  0.1996  0.2165
  Columns 43 through 49
    0.9533  0.9760  0.9987  1.0214  1.0441  1.0668  1.0895
```

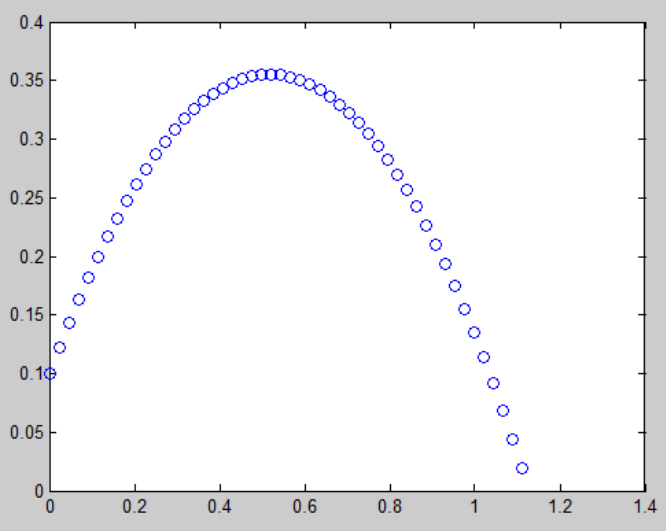
```
    0.1748   0.1555   0.1352   0.1140   0.0918   0.0685   0.0444
  Column 50
    1.1122
    0.0192
```

6.8. Function plotTrajectory

This is a simple function. It has a single parameter that expects a two row matrix, where the first row is the *x* values and the second row is the *y* values.

Plot the **xs** and **ys** using blue circles.

```
>> plotTrajectory(t)
```



6.9. Script trajectory

Create a script called trajectory. This is the main script file that will be used to run your program. Start the file with the usual comment block:

```
% Trajectories
% CSE1010 Project 5, Fall 2012
% (your name goes here)
% (the current date goes here)
% TA: (your TA's name goes here)
% Section: (your section number goes here)
% Instructor: Jeffrey A. Meunier

clc   % clear command window
clear % clear all variables
clf   % clear plot area
```

Now do the following, using the functions you have already written:

1. Set up the global variables.
2. Get the inputs and store them in a variable.
3. If the inputs are empty (use the **isempty** function; type **help isempty** or **doc isempty** for help), execute the **break** statement to terminate the program (type **help break**).
4. Convert the inputs, store the result in a variable.
5. Display the converted inputs.
6. Calculate the dependent variables from the converted inputs. Store in a variable.
7. Display the dependent variables.
8. Calculate the trajectory. Store in a variable.
9. Plot the trajectory.

Verify that this works for a mass value greater than zero, and that the program terminates for a negative mass value.

Now place all the steps 1-9 inside a **while true** loop. So your script file used to look like this, with just a bunch of statements one after the other:

```
___
___
___
```

Now it will look like this, with a bunch of statements inside a loop:

```
while true
  ___
  ___
  ___
end
```

This will cause your program to run repeatedly until the user enters a mass that is less than 0. Make sure that you indent the statements in the loop correctly.

6.10. Comment all the source code files

Place "help" comments inside each function, like this:

```
function setupGlobals
  % Defines global variables needed in this project
  % Use: setupGlobals
  ...
end
```

7. Report

Create a Microsoft Word or OpenOffice Word file (or some other format that your TA agrees on -- ask him or her if you are not sure). Save the file with the name **Project2** with a .doc or .docx format.

At the beginning of the document include this information:

Trajectories
CSE1010 Project 5, Fall 2012
Your name goes here
The current date goes here
TA: *Your TA's name goes here*
Section: *Your section number goes here*
Instructor: Jeffrey A. Meunier

Be sure to replace the parts that are underlined above.

Now create the following five sections in your document.

1. Introduction

In this section copy & paste the text from the introduction section of this assignment. (It's not plagiarism if you have permission to copy something. I give you permission.)

2. Test runs

Run the program using the numbers shown in the example in section 5.2. Copy and paste the console output (including the number that the user entered) into the document, and insert the plot into the document. Run the program two more times using different numbers. Copy and paste the console output and insert the graph.

3. Source code

Copy & paste the contents of your **trajectories.m** file, then after it paste each function that you wrote. You don't need to write anything with the functions this time because the functions already contain comments.

8. Submission

Submit the following things things on HuskyCT:

1. All the .m files for this project stored in a Zip file. Do not upload each separate .m file on HuskyCT. I have made tutorial videos available on HuskyCT describing how to make a Zip file.
2. The MS Word document.

If for some reason you are not able to submit your files on HuskyCT, email your TA before

the deadline. Attach your files to the email.

9. Notes

Here are some notes about working on this project:

- I can't think of anything right now.

10. Grading Rubric

Your TA will grade your assignment based on these criteria:

- (10 points) The program displays the correct answers and calculates them in the correct way, and the plots are correct for the inputs that were entered.
- (5 points) The program is formatted neatly. Follow any example in the book, or see the web site here: https://sites.google.com/site/jeffscourses/cse1010/matlab-program-formatting
- (5 points) The document contains all the correct information and is formatted neatly.

11. Getting help

Start your project early, because you will probably not be able to get help in the last few hours before the project is due.

- If you need help, send e-mail to your TA immediately.
- Go to the office hours for (preferably) your TA or any other TA. I suggest you seeing your own TA first because your TA will know you better. However, don't let that stop you from seeing any TA for help.
- Send e-mail to Jeff.
- Go to Jeff's office hours.