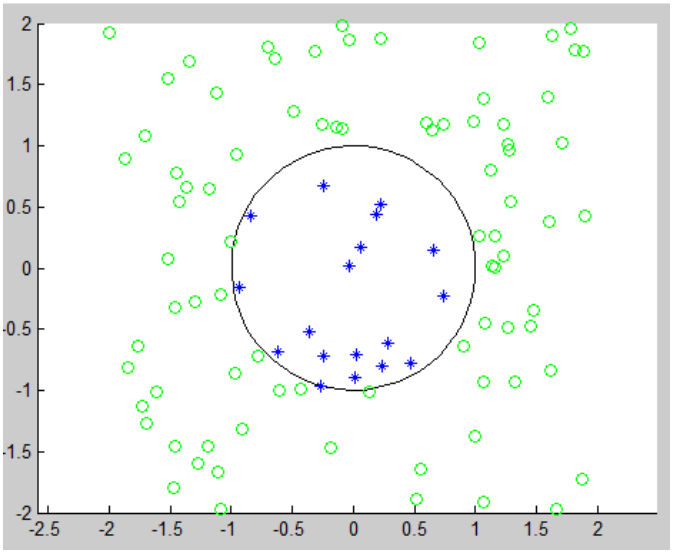
HW4: Monte Carlo Simulation
CSE1010 Fall 2012
Jeffrey A. Meunier
University of Connecticut

1. Introduction

Monte Carlo is a statistical method using random sampling that is used to perform an estimation of something that is difficult or impossible to measure. This process was named after the city of Monte Carlo in Monaco, which is known for its casinos.

We will use the Monte Carlo sampling process to estimate the area of circle, and thus the value of π. Furthermore, we will be creating a simulation of a physical process in which we use Monte Carlo sampling.

Say we have a square field in which there is a perfectly round pond. We know the size of the field and the radius of the pond. What we would like to do is determine using a small number of events what the area of the pond is. What we shall do is fire a cannon at the field a number of times (rather, simulate the firing of a cannon — we're not actually using live firearms). Assuming that the cannon balls land in the field in a uniformly distributed pattern, then we should expect some of them to hit the ground around the pond, and some of them to splash into the pond. I have represented it graphically like this:



This shows one run of my program in which 100 samples were taken (i.e., 100 shots were fired). The pond radius is 1, and the field width is 2 (but Matlab shows more empty space

to the left and right of the plot). The green circles are where the cannon balls hit the ground, and the blue stars are where they splashed into the water. It turns out that the ratio of splashes to thuds is proportional to the area of the pond to the area of the field. The accuracy of the measurement is determined by the number of samples taken, so the greater the number of samples taken, the greater the accuracy is.

In a simulation like this it is not necessary to call the random events cannon shots. You might simulate mathematicians throwing darts at a board while they're arguing in a bar, or a philosopher dropping grains of rice onto a table, or a nuclear physicist firing particles at a metal plate: as long as the samples are distributed uniformly, this algorithm works without modification. If the distribution is not uniform you can still use a Monte Carlo process, but the math would be different.

Q: Why do we need to use sampling to determine the area of that circle? Can't you see that the area is precisely π?
A: Indeed, the area is π. We need to use this information to work backwards to determine the accuracy of our program that uses Monte Carlo sampling. After we know it works, we would be able to apply this technique to find areas of other shapes, some which may be difficult or even impossible to compute analytically.

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 7, 2012.

4. Objectives

The purpose of this assignment is to have you improve your skills in these areas:

- Visualizing data by plotting data on graphs.
- Random numbers.
- Conditional statements.
- Iteration (looping) statements.

Keywords: random numbers, conditional, loop, iteration, monte carlo, simulation, error magnitude, fprintf

5. Background

For this assignment you will need to be familiar with these topics:

- random numbers
- if statements
- loops / iteration
- fprintf function
- plotting

6. Assignment

Read through all of the subsections in this section before you start to make sure you understand how you must proceed with the assignment. In this assignment you will create a program that generates a plot like the one shown in the introduction. You will use the program to run a series of simulations, collect data from them, and write a report on your findings.

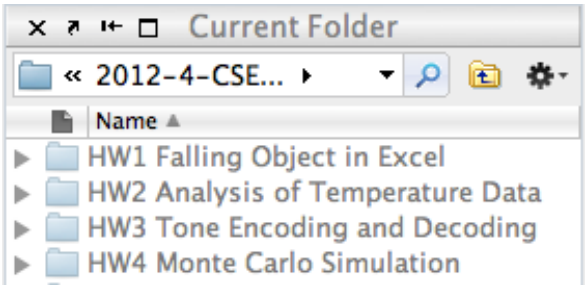
6.1. Create the simulation

In the first part of the assignment you will write a Matlab program that runs a Monte Carlo simulation of firing cannon shots into a field that contains a circular pond.

6.1.1. Create a project folder

For this project you will be creating a Matlab script file and several functions. These should be stored in a new folder made just for this project.

Create a new folder somewhere on your computer for homework project 4. It should be in your CSE1010 folder. I recommend creating a list of folders like this:



The name does not need to be exactly what I show, but I do recommend naming the project folders with some kind of number first because otherwise the folders will appear in alphabetical order instead of chronological order. If you have not yet organized your

folders like this, you can do it now. You can make new folders (right click on the white background, select New Folder), you can drag and drop them, and you can rename them (select a folder and press F2, or hit Enter, depending on your operating system).

If you are using a computer in a lab, place the new folder on your H: drive.

After you create the Monte Carlo folder, double click on it to enter that folder. Now you are ready to add files to the project.

6.1.2. Script monteCarlo

Create a new script file in your project folder called **monteCarlo.m**. Copy and paste this program outline into the file:

```
% Monte Carlo Area calculation
% CSE1010 Project 4, 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
rng('shuffle') % shuffle the random number generator

fieldSize  = 4;
pondRadius = 1;
numShots   = 100;
```

Be sure to change the comments in the comment block at the top of the file.

At the bottom of this script add one more statement that calls a function named **setupField** and supplies one argument: the field size.

6.1.3. Function setupField

Write a function called **setupField** that has these statements in the body:

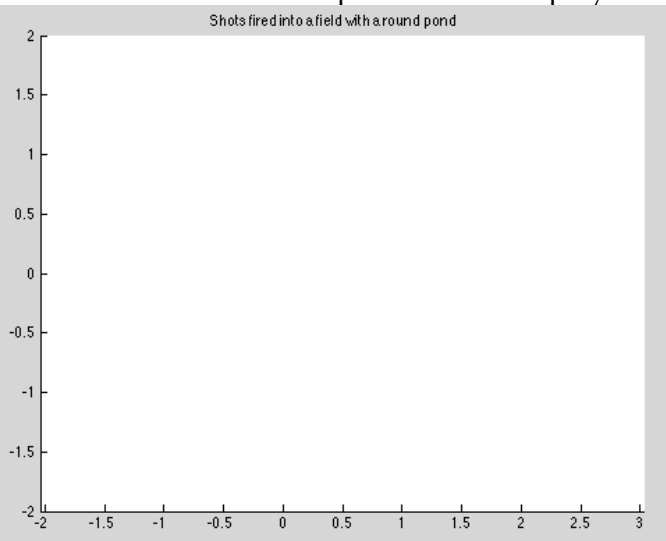
```
  title('Shots fired into a field with a round pond')
  axis([-fieldSize/2, fieldSize/2, -fieldSize/2, fieldSize/2])
  axis equal
  hold on
```

This function does not need to return anything, but it does need a parameter variable.

From the statements above you will be able to determine what the name of the parameter variable should be.

Add a call to this function at the bottom of the **monteCarlo** script file. You need to provide an argument to the **setupField** function when you call it. Can you figure out what it should be?

Run the **monteCarlo** script. It should display an empty plot window that looks like this:



6.1.4. Function plotPond

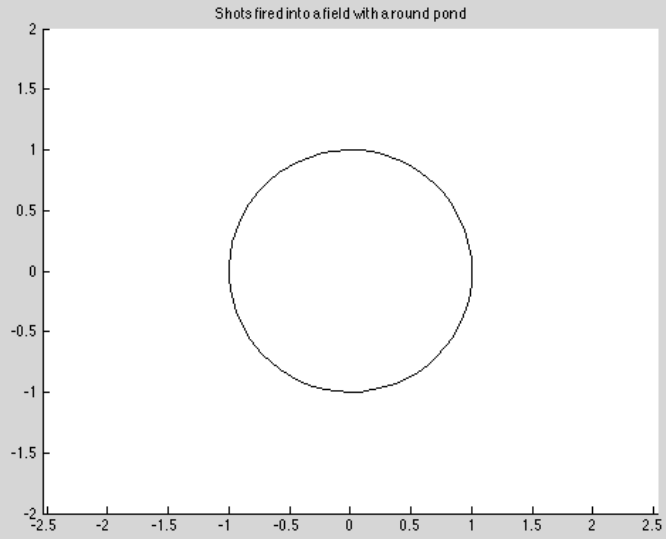
Write a function called **plotPond** that draws a circle in the middle of the field. Refer to lab exercise 4, in which you plotted circles on a Cartesian plane.

This function will have one parameter, which is the radius of the pond circle.

Note that you need only to draw one revolution in order to have a complete circle (or maybe a bit more than one revolution if the ends don't meet fully). In the lab exercise I was having you plot many revolutions. Don't do that here.

Add a call to this function at the bottom of the **monteCarlo** script file. You need to provide an argument to the **plotPond** function when you call it. Can you figure out what it should be?

Run the **monteCarlo** script. It should display a plot window that looks like this:



6.1.5. Function hitPond

Write a new function called **hitPond** that has three parameters:

1. The pond radius.
2. The *x* location of the shot.
3. The *y* location of the shot.

Use the Pythagorean theorem to determine if the shot (which fell at location *x, y*) landed within the pond (i.e., if the distance from 0, 0 to *x, y* is less than the pond radius). This function returns **true** if the shot fell in the pond or **false** if it did not.

Test it:

```
>> hitPond(1, 1, 1)
ans =
     0
>> hitPond(1, 0.5, 0.5)
ans =
     1
>> hitPond(1, 0.7, 0.7)
ans =
     1
>> hitPond(1, 0.7, 0.8)
ans =
     0
```

6.1.6. Function plotShot

This function will place a visual indicator on the plot that shows in color whether a shot landed in the pond or in the field. You may want to scroll down to see how this will look.

This function has three parameters:

1. The pond radius.
2. The *x* location of the shot.
3. The *y* location of the shot.

The function then plots the point *x*, *y* in one of two styles depending on whether it landed in the pond or not.

The **plot** function accepts a third argument that indicates the shape of the point to plot and the point's color. For instance this plots a blue asterisk:

```
plot(x, y, 'b*')
```

and this plots a green circle:

```
plot(x, y, 'go')
```

The statements in the function must do this: If the shot hit the pond, plot a blue star, otherwise plot a green circle. You will need to use an **if/else** statement, and you will need to call the **hitPond** function and the **plot** function.

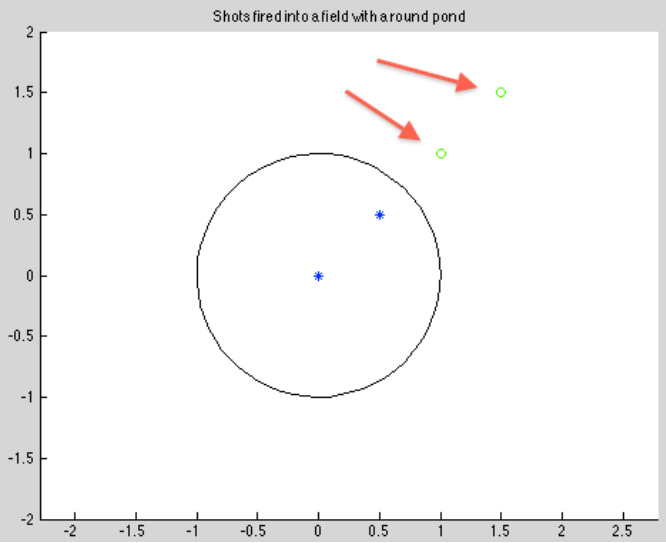
Also, this function must return a result:

- It returns the string **'splash'** if the shot lands in the pond.
- It returns the string **'thud'** if the shot lands in the field.

Test it:

```
>> monteCarlo
>> plotShot(1, 0, 0)
ans =
splash
>> plotShot(1, 0.5, 0.5)
ans =
splash
>> plotShot(1, 1, 1)
ans =
thud
>> plotShot(1, 1.5, 1.5)
ans =
thud
```

Here is what the plot should look like:



Note that I have drawn two red arrows to show where two shots fell outside the pond area and were plotted in green. They might be difficult to see.

6.1.7. Function fireShot

The **fireShot** function will have two parameters: the size of the field and the radius of the pond.

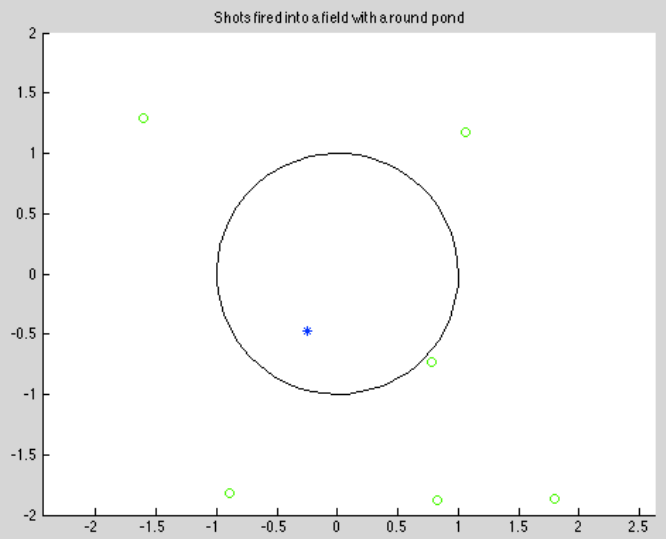
The function generates two random numbers that fall within the field (that is from −2 to +2, but don't use −2 and +2, use the field size variable). Use these two numbers as the *x*, *y* point location of the shot (one number is *x*, the other is *y*). Call the **plotShot** function to plot the shot. Return the value that is returned by the **plotShot** function call.

Test it:

```
>> monteCarlo
>> fireShot(4, 1)
ans =
thud
```

Look carefully for the point in the plot window. It may be hard to find.

After calling **fireShot(4, 1)** a total of 7 times, this is what my plot looks like:



One shot landed in the pond, 6 others landed outside the pond. Your shots may be in different locations. Note that it's still pretty hard to see the green points outside the pond area, but they're there.

6.1.8. Function fireShots

This function will fire multiple shots and count how many of them land in the pond. This function has three parameters:

1. The number of shots to fire.
2. The size of the field.
3. The radius of the pond.

This function is not long, but it incorporates several new things.

- The function must *iterate*, counting down from some number to 0, and stopping when it gets to 0.
- The function must count events: every time something happens then a variable must be incremented.
- Strings must be compared for equality.

Let's start by iterating. Place a **while** loop in the function that iterates while the number of shots to fire is greater than 0. Inside the loop, display the number of shots and then decrement the number of shots (meaning, subtract 1 from the variable):

```
___ = ___ - 1;
```

Test the function:

```
>> fireShots(5, 4, 1)
5
4
3
2
1
```

Yours may look a little different, but as long as it counts from 5 to 1 then it is correct so far. After you determine that it works, remove the statement that displays the number.

Inside the loop do this: call the **fireShot** function using the field size and pond radius as arguments. Examine the result of this function call. If the result is a string that's equal to **'splash'**, then increment the number of splashes that have been counted so far.

Be careful: you must still decrement the number of shots, but make sure that it happens whether the shot hit the pond or the field.

In order to compare two strings for equality, you should not use the double equal operator: that will treat the two strings as vectors, which will work *only* if the two strings have exactly the same length. This is what would happen:

```
>> 'splash' == 'splash'
ans =
     1     1     1     1     1     1
```

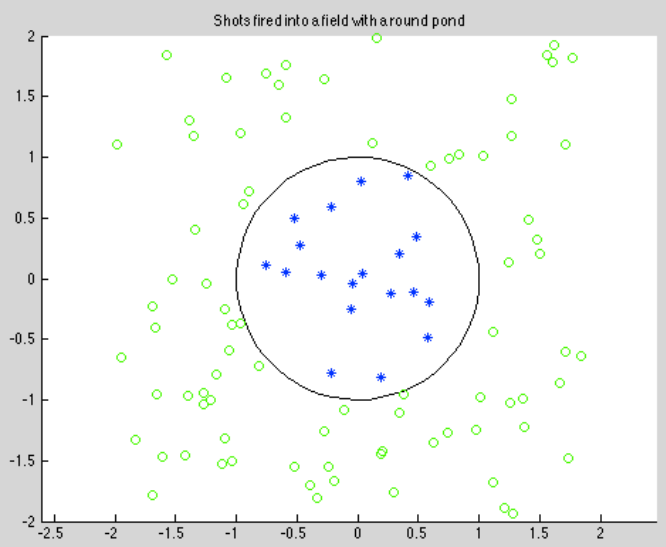
A vector of all 1s is acceptable as a logical value, however:

```
>> 'splash' == 'thud'
Error using  ==
Matrix dimensions must agree.
```

The solution is to use the **strcmp** function built into Matlab. Use Matlab's built-in help to learn how to use the **strcmp** function.

Test it:

```
>> monteCarlo
>> fireShots(100, 4, 2)
ans =
  19
```



Thus, out of 100 shots, 19 have landed in the pond.

Add a statement to the bottom of the **monteCarlo** script so that the **fireShots** function is called automatically. Replace the numbers 100, 4, and 2 with the appropriate variables that you defined in the **monteCarlo** script. Now when you type **monteCarlo** at the command prompt, the plot is drawn and 100 shots are fired. Also, the number of shots landing in the pond is returned and displayed. Change that so the return value is stored in a variable instead, and display two message like this:

```
Number of shots = 100
Number of splashes = 19
```

Format each message on one line. To do this you will need to use two **fprintf** statements (although it's possible to do it using one **fprintf** statement). Use the **%g** format specifier for each number, and also use the **\n** format command. See pages 42 and 43 in the book for information about using the **fprintf** statement, or use Matlab's built-in help.

6.1.9. Function estimatePondArea

Write a function called **estimatePondArea** that has three parameters:

1. The number of splashes.
2. The total number of shots.
3. The size of the field.

The estimated area of the pond is the ratio of splashes to shots multiplied by the field area. Note that the field *size* is not the same as its *area*. The field is square.

Return the estimated area from this function.

Test it:

```
>> estimatePondArea(19, 100, 4)
ans =
    3.0400
```

Call this function from the **monteCarlo** script. Store the result in a variable and display a message like this:

```
Estimated pond area = 3.04
```

6.1.10. Function estimatePi

Write a function called **estimatePi** that has two parameters:

1. The estimated pond area.
2. The actual pond radius.

Note that we can estimate the area of a shape just by firing shots into it, but we can't calculate the estimated value of *pi* without knowing the pond's actual radius. If this were a real-life experiment with a real pond in a real field, we would have to be sure that the pond was perfectly round, and we would need to measure the pond's size accurately.

The estimated value of *pi* is the estimated pond area divided by the square of the actual pond radius. Return the estimated value of *pi* from this function.

Test it:

```
>> estimatePi(3.04, 1)
ans =
    3.0400
```

Call this function from the **monteCarlo** script. Store the result in a variable and display a message like this:

```
Estimated value of pi = 3.04
```

6.1.11. Function piErrorPercent

Write a function called piErrorPercent. This function has one parameter: the estimated value of *pi*.

Inside this function calculate the error magnitude by taking the absolute value of the difference between the estimated value of *pi* and the actual value of *pi*. Use the **abs** function.

Then calculate the error percentage by dividing the error magnitude by the actual value of *pi* and multiplying by 100.

Return the error percentage.

Test it:

```
>> piErrorPercent(3.04)
ans =
    3.2338
```

Thus, the number 3.04 differs from Matlab's value of *pi* by about 3¼ percent.

Call this function from the **monteCarlo** script. Store the result in a variable and display a message like this:

```
Error in estimation of pi = 3.23379%
```

Make sure the percent sign is shown after the number. To get the percent sign to be displayed, just place a double percent inside the string that you use in the **fprintf** statement: `'... %g%%\n'`

6.1.12. Get user input

Modify the monteCarlo script file so that instead of assigning constant scalars to the three variables, like this:

```
fieldSize  = 4;
pondRadius = 1;
numShots   = 100;
```

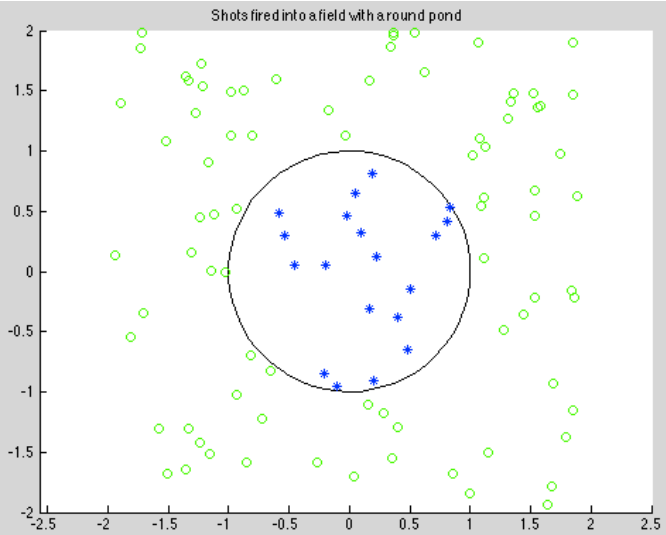
you call the input function once for each variable, like this:

```
fieldSize = input('Enter the size of the field: ');
```

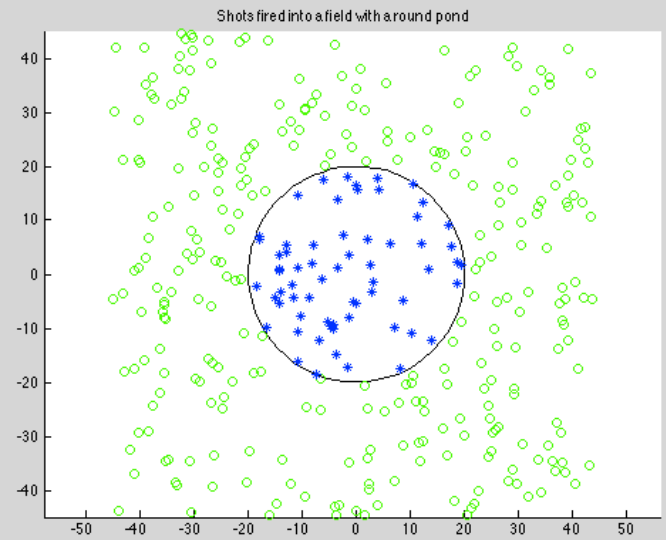
This allows you to change the characteristics of the experiment each time you run the

program.

```
Enter the size of the field: 4
Enter the radius of the pond: 1
Enter the number of shots: 100
Number of shots = 100
Number of splashes = 19
Estimated pond area = 3.04
Estimated value of pi = 3.04%
```




```
Enter the size of the field: 90
Enter the radius of the pond: 20
Enter the number of shots: 350
Number of shots = 350
Number of splashes = 64
Estimated pond area = 1481.14
Estimated value of pi = 3.70286%
```



6.1.13. Comment the functions

Add comment lines inside each function that you wrote. There are a few examples of this on page 87. Use at least two comment lines:

1. A brief description of what the function does.
2. Show how to call the function (you can just copy & paste the function definition without the word **function**).

Here is what the comments look like in my setupField function:

```
function setupField(fieldSize)
  % Sets up the plot window for the simulation.
  % Use: setupField(fieldSize)
```

6.2. Run the simulation

In the second part of the assignment you will perform some experiments by running the simulation, collecting data, and creating graphical visualizations of the data.

6.2.1. Create spreadsheet data

You have just created a very useful simulation. Now it is time to use the simulation to run a few experiments (this is the *science* part of *Computer Science and Engineering*). What you will do is run the program a number of times and collect the data into a spreadsheet.

Start Excel or some other spreadsheet program and enter these row and column headings (make the headings bold):

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**mean**	**log(mean)**
1							
10							
100							
1000							
10000							
100000							

Run your program 5 times, each time with 1 shot. Use field size 4 and pond radius 1. Enter only the error percentage in each cell in the row labeled 1. You'll fill in the **mean** and **log(mean)** columns afterward. My row looks like this, your numbers might be different:

	Run 1	Run 2	Run 3	Run 4	Run 5
1	409.2958	100	100	100	100

Run the program 5 more times, each time with 10 shots. Enter the error percentages in the row labeled 10.

Do it 5 times with 100 shots, and 5 times with 1000 shots. For 10000 and 100000 shots the simulation becomes very slow. The problem is not so much that there are too many iterations, but that you are asking Matlab to plot those points. The loop would run very quickly without all the plotting. Since all you need is the number at the end and not the plot, you can comment out the part of the program that makes it slow. Find the two plot statements inside the **plotShot** function and comment them out:

```
%plot(x, y, 'b*')

%plot(x, y, 'go')
```

When you run the program the plot window will still appear, but the shots will not be drawn. As a result the program will run quickly.

6.2.2. Complete spreadsheet

Now it's time to enter the formulas for the **mean** and **log(mean)** columns.

In cell G2 enter the formula `=AVERAGE(B2:F2)`
Make sure you type the equal sign in that formula.

In cell G3 enter the formula `=AVERAGE(B3:F3)`

Do the same for G4, G5, G6, and G7. Make sure you change the range appropriately for each entry: `B4:F4`, `B5:F5`, and so on.

In cell H2 enter the formula `=LOG(G2)`
In cell H3 enter the formula `=LOG(G3)`
Do the same for H4, H5, H6, and H7.

6.2.3. Plot log(mean)

Next you will create a plot in Matlab of the **log(mean)** column. Here's how you do it using

copy & paste:

1. In the Excel spreadsheet, highlight the 6 numbers in the **log(mean)** column and press Ctrl-C (or select Copy in the ribbon strip under the Home tab) to copy the numbers to the clipboard in Windows (use Cmd-C on the Mac).
2. Click once in the Matlab workspace window in the upper right and type Ctrl-N (Cmd-N on Mac) to create a new variable. Call it **logMean**.
3. Double click on the **logMean** variable to open the variable editor.
4. Press Ctrl-V (Cmd-V) to paste the numbers copied from Excel.
5. Close the variable editor window.

This creates a column vector in the **logMean** variable. Now go to the command window and type **plot(logMean)**. Save this plot (use *File* → *Save as*) because later you will insert it into the report document.

The fact that this plot is largely linear shows that the error in the calculation in *pi* depends linearly on the number of shots fired. For instance, doubling the number of shots doubles the accuracy.

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:

Monte Carlo Simulation
CSE1010 Project 4, 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 3 times. For all three runs use field size 4, pond radius 1, and for the number of shots use 10, then 100, then 1000. For each test run, copy & paste the output from the command window into the document, and save the plot and insert it into the document. Label each test run & plot, something like this:

Test run of field size 4, pond radius 1, 10 shots

```
Enter the size of the field: 4
Enter the radius of the pond: 1
Enter the number of shots: 10
```

etc.

3. Plot of log(mean)

Insert the **log(mean)** plot that you created in section 6.2.3. Write a brief description of what the plot is and what it means. Why is the plot linear? Why does the plot go in the direction it goes?

4. Spreadsheet data

Copy and paste the spreadsheet data into this section, or insert a cropped screen shot of your spreadsheet. Add a sentence or two describing what the values in the spreadsheet are.

5. Source code

Copy & paste the contents of your **monteCarlo.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 two 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. These are the files you must include in the Zip file:
    - estimatePi.m
    - estimatePondArea.m
    - fireShot.m
    - fireShots.m
    - hitPond.m
    - monteCarlo.m
    - piErrorPercent.m
    - plotPond.m
    - plotShot.m

- randomShot.m
- setupField.m

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:

10. Grading Rubric

Your TA will grade your assignment based on these criteria:

• (1 point) The comment block at the beginning of the **monteCarlo.m** file is correct.

• (10 points) The program displays the correct answers and calculates them in the correct way.

• (4 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, and you have used correct grammar and spelling where appropriate (this will be mostly in your program comments: you don't need to write full sentences, but make sure that what you write is otherwise correct). Non-native speakers of English will be granted a bit of leniency on grammar.

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.