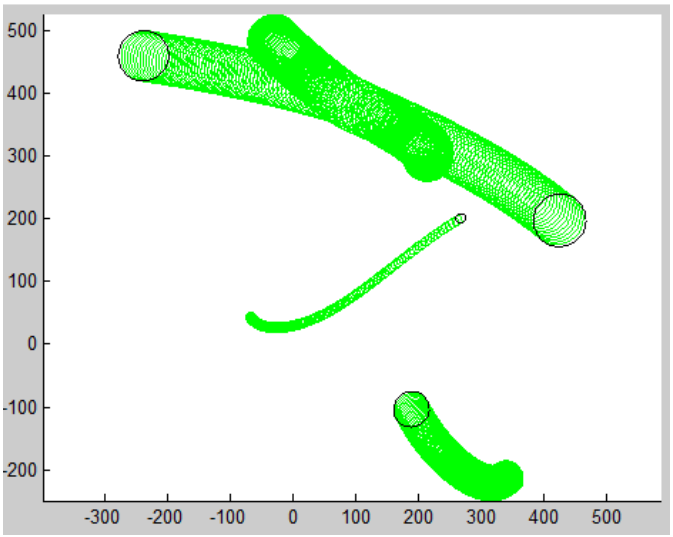
HW6: N-Body Simulation
CSE1010 Fall 2012
Jeffrey A. Meunier
University of Connecticut

1. Introduction

This project is a discrete simulation solution of an *N*-body gravitational force problem. Given some number of bodies in empty space, the bodies are allowed to move freely and interact with each other under gravitational force, although in this simulation you will not simulate collisions between the bodies. The result yields an interesting animation.



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 28, 2012.

4. Objectives

The purpose of this assignment is to give you more practice with the things you have already learned.

Keywords: parallel vectors, loops, functions, parameters, return values, plotting, global variables, pythagorean theorem

5. Background

Be sure you understand how to write and use for loops and while loops.

Be sure you understand how to write and use functions, including parameters, return values, and arguments.

Read all you want about the N-body problem, but you won't need to know any of that information for this project.

6. Assignment

In your CSE1010 folder in Matlab create a new folder for HW6.

6.1. Script nbody.m

Start with the main script file this time. Add your personal information to the comments.

```
% N-Body Simulation
% CSE1010 Project 6, 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
axis equal
hold on
```

Enter these statements into the main script file. These values will determine the characteristics of the simulation.

```
numBodies      = 4;
global G
G              = 100; % not -9.8 in this project!
massFactor     = 40;
minMass        = 5;
distanceFactor = 1000;
velocityFactor = 3;
```

Now create the following vectors:

- **Masses**: A row vector of **numBodies** random values in the range **minMass** to **minMass** + **massFactor**.

- **Xs**: A row vector of **numBodies** random values in the range -**distanceFactor**/2 to +**distanceFactor**/2.
- **Ys**: A row vector of **numBodies** random values in the range -**distanceFactor**/2 to +**distanceFactor**/2.
- **Dxs**: A row vector of **numBodies** random values in the range -**velocityFactor**/2 to +**velocityFactor**/2.
- **Dys**: A row vector of **numBodies** random values in the range -**velocityFactor**/2 to +**velocityFactor**/2.

The **Masses** vector lists the masses of all the bodies.
The **Xs** and **Ys** vectors list the locations on the screen where the bodies are.
The **Dxs** and **Dys** vectors list the velocities of the bodies.

There's a lot more to put in the main script, but you should start by writing a few functions first.

6.2. Function circle

Believe it or not, Matlab does not have a way to draw a circle of an arbitrary size or color. You will write this function.

Create a new function called **circle**. This function has 4 parameters:

1. **x**: this is the x coordinate of the center of the circle
2. **y**: this is the y coordinate of the center of the circle
3. **radius**: this is the radius of the circle
4. **color**: this is the color of the circle

One way to draw a circle in Matlab is to plot a bunch of points in a circle.

1. First you must create a vector of numbers in the range 0 to 2pi in 0.01 increments. Call this vector **Theta**. Use the colon notation.
2. Generate a vector of **X** values for the circle by calling the **cos** function on the **Theta** vector and multiplying the result by radius, and adding **x** to the vector.
3. Generate a vector of **Y** values for the circle by calling the **sin** function on the **Theta** vector and multiplying the result by radius, and adding **y** to the vector.
4. Plot all the points at once using the specified color: **plot(X,Y,color)**

Write comments in the function. Mine look like this. You can copy & paste for this one, but for the rest of the functions you must write your own comments.

```
% Draws a circle at x,y having given radius and color.
% Use: circle(x, y, radius, color)
```

Make sure the help feature works with this function:

```
>> help circle
  Draws a circle at x,y having given radius and color.
  Use: circle(x, y, radius, color)
```

Test the function

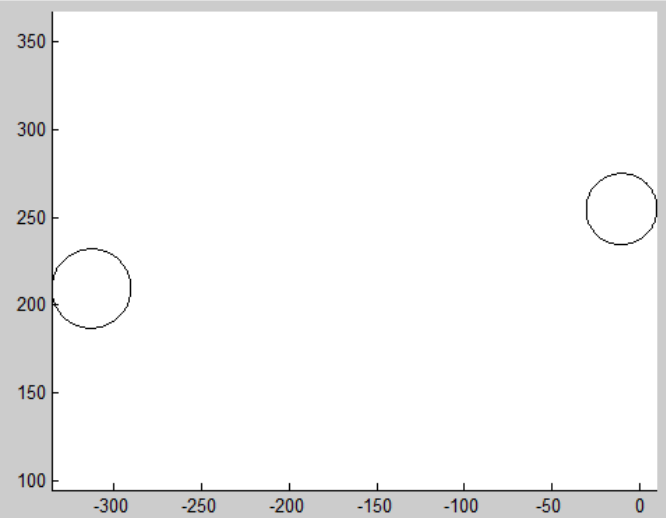
Back in the main script file, you will test the circle function by plotting the first two bodies in the vectors that you generated randomly. Add these lines at the bottom and then run the program:

```
circle(Xs(1), Ys(1), Masses(1), 'k');
```

That statement draws a circle centered at **Xs(1)**, **Ys(1)** with mass (or radius) **Masses(1)** using black. Draw the next one:

```
circle(Xs(2), Ys(2), Masses(2), 'k');
```

Those two statements plot the first two random bodies, showing two circles of different sizes in different locations. Run the program several times to be sure it works correctly. Here's what mine looked like one time:



6.3. Function plotBodies

Instead of calling the circle function once for each body, it would be nice to write a function that calls the circle function for all the bodies.

Create a new function called **plotBodies** that has these parameters:

1. **Xs**: this is the vector of all the bodies' x coordinates

2. **Ys**: this is the vector of all the bodies' y coordinates
3. **Masses**: the vector of masses
4. **color**: the color to plot all the bodies

Determine the number of bodies that are to be plotted (it's the length of either the **Xs**, **Ys**, or **Masses** vectors) and store it in a variable.

Start a for loop that goes from 1 to the number of bodies. Use a loop control variable called something like **b** or **body**, because you're iterating over the number of bodies. I used **b** in my program.

Inside the loop, call the **circle** function. You need to supply 4 arguments when you call that function: the **x** coordinate, the **y** coordinate, the **radius**, and the **color**.

Recall how you tested the circle function:

```
circle(Xs(1), Ys(1), Masses(1), 'k');
```

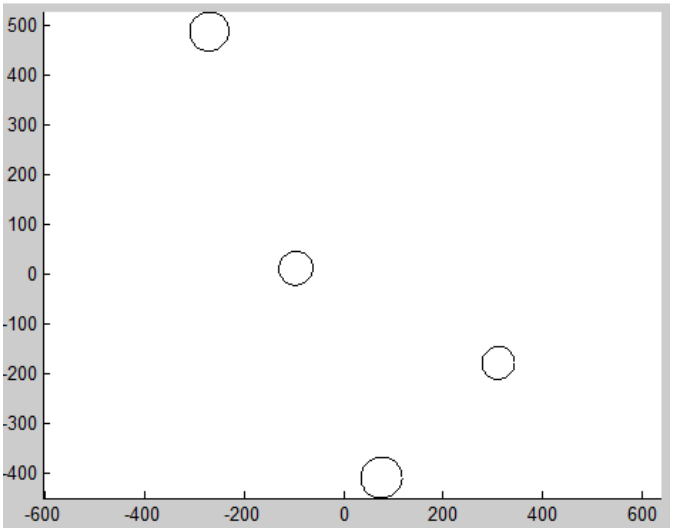
This function call draws the circle for mass 1. Now you want to drawn the circle for mass 1, you want to draw a circle for mass **b**.

End the for loop.

Write comments in the function.

Test the function

In the main script, replace the two calls to the circle function with a single call to the **plotBodies** function using the correct arguments. Run the program. This is what my plot looks like. Yours will be slightly different, but there should be 4 circles of different sizes placed randomly in the range +/-500 on the x and y axes:



6.5. Function moveBodies

Now that you can draw the *n* bodies on the screen, you need to be able to move them. The velocities are specified in the **Dxs** and **Dys** vectors. In order to move the bodies, the position of each body will be changed by its *dx* & *dy* values found in the vectors, then the bodies will be re-drawn, then moved, then re-drawn, and so on.

Create a new function called **moveBodies**. It has 4 parameters:

1. **Xs**: these are the x positions of all the bodies
2. **Ys**: these are the y positions of all the bodies
3. **Dxs**: these are the x velocities of all the bodies
4. **Dys**: these are the y velocities of all the bodies

This function has two separate return variables, **Xs** and **Ys**. This function is responsible for changing each element in **Xs** by the corresponding amount in **Dxs**, and the amount in **Ys** by the corresponding amount in **Dys**. This function returns the two vectors **Xs** and **Ys** after the vectors have been modified.

Recall that it is allowed to use the same variable name (or names) for a parameter and a return variable.

Notice that the **Dxs** vector contains the amount that the **Xs** vector needs to change. For example, if **Dxs(1)** is 2.3, then **Xs(1)** needs to have 2.3 added to it. If **Dxs(1)** is −0.02, then **Xs(1)** needs to have −0.02 added to it. The same is true for location 2, location 3, and the rest of the vectors. The same is also true for the **Ys** & **Dys** vectors. This is actually much simpler than it sounds, and it does not require the use of a loop.

Notice that you are changing the values of the **Xs** and **Ys** vectors. Recall that when you

change **Xs** and **Ys** inside this function, you're changing copies of those vectors. Thus, these two vectors must be returned from this function.

Write comments in the function.

Test the function

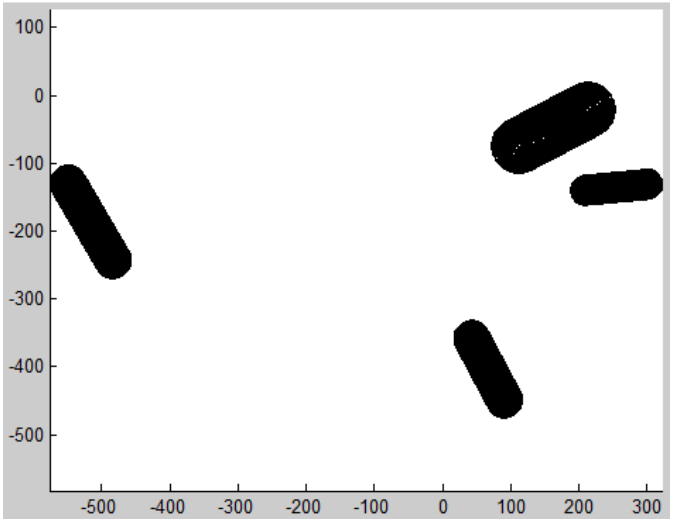
In the main script, after you call **plotBodies**, call the **moveBodies** function and store the result into **Xs** and **Ys**, like this:

```
[Xs Ys] = moveBodies(Xs, Ys, Dxs, Dys);
```

and then call **plotBodies** again right after that to show the result of moving all the bodies. If you run this, it's difficult to see that the bodies have moved at all, so put those two move & plot function calls inside a **for** loop that goes from 1 to 100. You'll end up writing something like this in the main script:

for i = 1:100
*** moveBodies***
*** plotBodies***
end

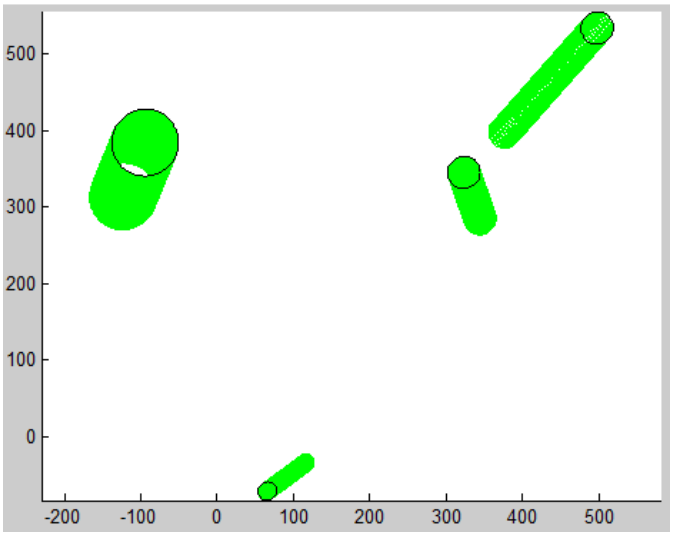
When I run my program this is what I see:



Each body turns into a black streak. This can be fixed by doing this:

- Just before each body is moved, plot the bodies using a color other than black (this draws over the black circles with green ones).
- Move the bodies.
- Plot the bodies using black.

```
for i = 1:100
  plotBodies using green
  moveBodies
  plotBodies using black
end
```



Now it actually looks kind of nice.

6.6. Function calculateForce

This function calculates the gravitational force on a body by another body.

This function needs the following parameters:

1. The vector index number of a body. Let's call it **body1**.
2. The vector index number of another body. Let's call it **body2**.
3. **Xs**: these are the x positions of all the bodies.
4. **Ys**: these are the y positions of all the bodies.
5. **Masses**

This function returns two values: **fx** and **fy**, which are the *x* and *y* components of the resultant force.

The general formula for gravitational force is this:

$$f = G\left(\frac{M_1 M_2}{r^2}\right)$$

where G is the universal gravitational constant, M_1 and M_2 are the masses of the two bodies, and r is the distance between the two bodies.

In this program, G is a global variable that you defined in the main script file. I have found a value for G that, while not realistic, does create a nice simulation. This function can have access to that variable by using this statement:

```
global G
```

But do not re-assign a value to this variable! You mean to use the variable G that already exists, not create a new one.

M_1 and M_2 can be determined by reading the values out of the **Masses** vector at locations **body1** and **body2**.

r is the distance between the two bodies. You know the locations of the two bodies: **Xs(body1)**, **Ys(body1)** and **Xs(body2)**, **Ys(body2)**. Use the Pythagorean theorem to determine the distance between the bodies. It would be useful to create two other variables first: **dx** and **dy**, where **dx** is the difference between the **x** values and **dy** is the difference between the **y** values. You will need to use these variables again below.

Since r will be used as the denominator in the formula, you should prevent r from becoming too small or else the resultant force f will become artificially inflated. In reality, two bodies will never become closer than the sum of their radii (because that's when they crash into each other), but this program does nothing to prevent two bodies from overlapping and even occupying the same point. If this were to happen the distance between them approaches 0 and the force approaches infinity.

This is how you will solve the problem:

Consider the radius of a body to be the same as the body's mass. For example, a mass of 50 has a radius of 50.

Create an **if** statement that checks if r (in this case it's the distance between the centers of the two masses) is less than the sum of the two masses' radii. If it is, set r equal to the sum of the radii.

Now calculate the force f using G, the mass of body 1, the mass of body 2, and the distance squared.

Now that you have the resultant force, you need to decompose it into its x and y

components.

Calculate the angle of the force by calling **atan2(dy, dx)** where **dy** and **dx** are the variables you created previously.

Assign to **fx** the value of calling the **cos** function on the angle.
Assign to **fy** the value of calling the **sin** function on the angle.

Write comments in the function.

Test the function

Put this function call in your main script file and then run the program.

```
calculateForce(1, 2, [0 400], [0 400], [100 100])
```

My program displayed this answer:

```
ans =
    2.2097
```

After you verify your function, remove the function call from the main script.

6.7. Function accelerateBody

This function will apply the gravitational attraction of all other bodies to a single body. This will change the velocity of the body. This is what accelerate means in this context: not necessarily increase the velocity of a body, but change its velocity and direction.

This function has these parameters:

1. A body number. I called mine **b**.
2. **Xs**: these are the x positions of all the bodies
3. **Ys**: these are the y positions of all the bodies
4. **Dxs**: these are the x velocities of all the bodies
5. **Dys**: these are the y velocities of all the bodies
6. **Masses**

This function returns *two* values: **dx** and **dy**, which is the new velocity of the body.

To calculate the total force on a body, you must add up all the separate forces on a body by all bodies other than itself.

Start a for loop that does this:
For each body number from 1 to the number of bodies:

- If the body number is not equal to **b**:
    - Calculate the force on the body by calling **calculateForce**.
    - Calculate the **x** & **y** accelerations. The acceleration is the force divided by the mass. The mass is the mass of body **b**.
    - Add the **x** & **y** accelerations to the **x** & **y** velocities of body **b**.

You must check to see if the body number is equal to b so that you don't calculate the force between body **b** and itself.

Make sure that this function returns the **x** & **y** velocities of body **b**.

Write comments in the function.

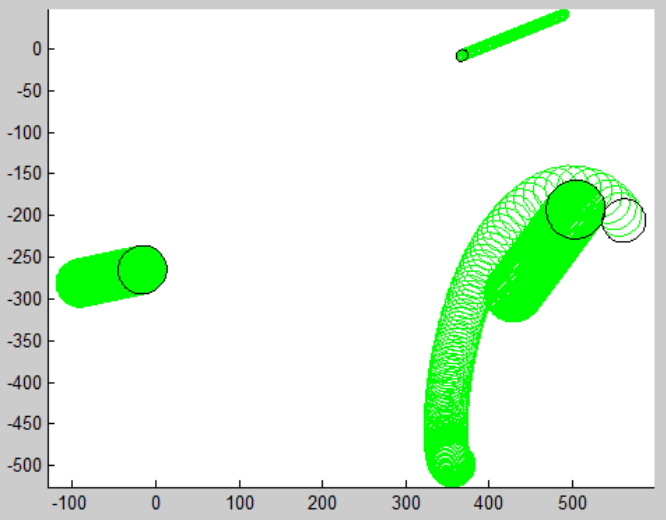
Test the function

In the main script, call the **accelerateBody** function on body number 1 and store the return values back into the **Dxs** and **Dys** vectors:

```
[Dxs(1) Dys(1)] = accelerateBody(1, ___, ___, ___, ___, ___);
```

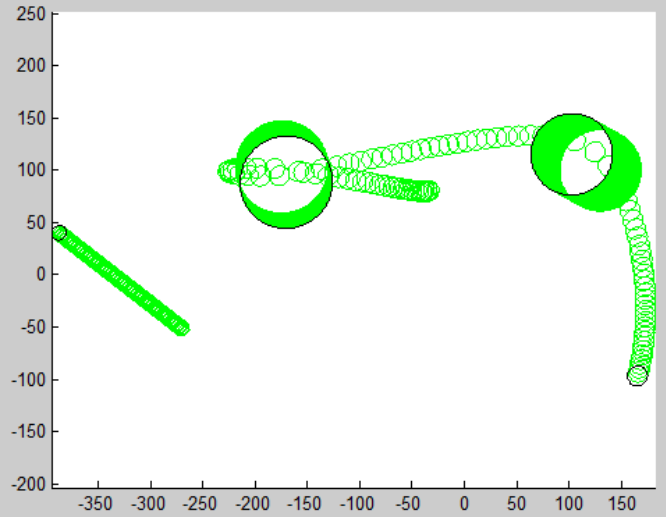
Put that statement inside the loop somewhere. I made it the last statement in the loop. Now run the program to observe the effect. Body 1 should no longer move in a straight line because now it is under the gravitational effects of all the other bodies.

You can tell which body is body 1 because it's the one with the curved trace. Here you can see body 1 accelerating (actually moving faster) and being pulled into an orbit around another more massive body:



I ran the program several times and most plots were usually not this dramatic. In many of them it was difficult to determine if the path of body 1 was changing at all.

Wow, here's an even better one. Body 1 is a very small body that is being thrown around by two very massive bodies:



6.8. Function accelerateBodies

You're almost done! Now that you can accelerate a single body, you can apply the **accelerateBody** function to all bodies.

This function will apply the gravitational attraction to all bodies, changing the velocities of each body. The function has these parameters:

1. **Xs**
2. **Ys**
3. **Dxs**
4. **Dys**
5. **Masses**

The function returns two values: **Dxs** and **Dys**.

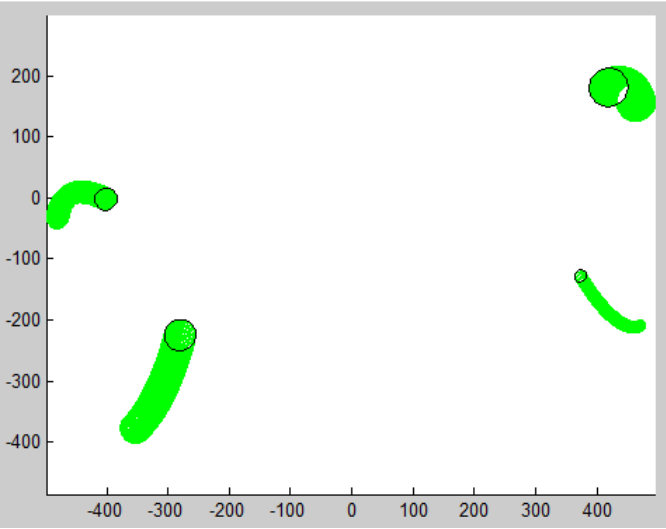
Write a for loop that iterates over all the bodies. It calls the **accelerateBody** function on each body, and stores the results back into the **Dxs** and **Dys** vectors for that body.

Write comments in the function.

Change the main script to call this function instead of the **accelerateBody** function. Make sure you store the results back into the **Dxs** and **Dys** vectors:

```
[Dxs Dys] = accelerateBodies(___, ___, ___, ___, ___);
```

Here's what mine looks like:



6.9. Animate the program

This static display is nice, but wouldn't it be nice to watch these bodies move around in real time?

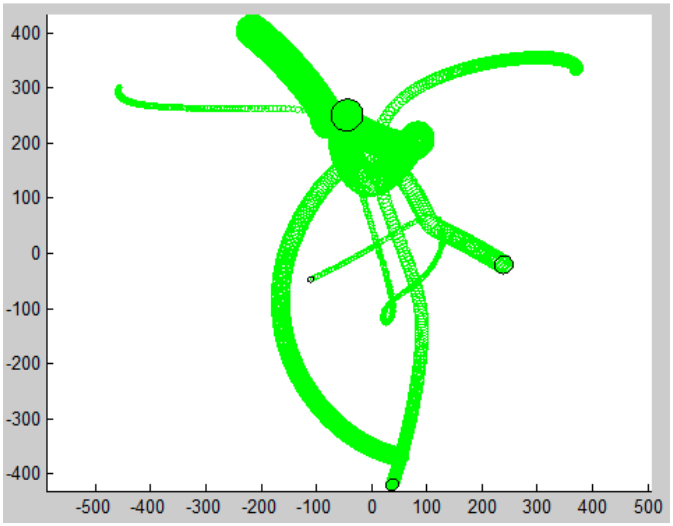
Do this in the main script:

- Change the **for** loop to a **while** true loop. Do not run the program yet. You will not see any output.
- Add the statement **pause(eps)** to the bottom of the loop, inside the loop. The eps value is a built-in constant that is the smallest number that Matlab can represent accurately. This causes Matlab to update the display and pause for about 2×10^{-16} second. While your program pauses, Matlab will update the display.

How to stop the program

Since this program runs in an unbounded while loop, you must interrupt the program manually by typing Control-C (hold the Control key and press c key) in either the plot window or the command prompt window.

Here's what mine looks like after running for a minute or so.



You'll notice that the program gets slower and slower the longer it runs. This has something to do with the way Matlab stores and draws graphics on the screen. Furthermore, not only is Matlab getting slower, but it's consuming significantly more memory the longer it runs.

You don't have to let the program run for very long, but test it several times to be sure it's working properly.

6.10. Comment all the files

Make sure you place "help" comments inside each function, like this:

```
function circle(___)
  % Draws a circle...
  % Use: circle(___)
  ...
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 a .doc or .docx format.

At the beginning of the document include this information:

N-Body Simulation
CSE1010 Project 6, 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 sections in your document.

1. **Introduction**
   In this section copy & paste the text from the introduction section of this assignment. You do not need to copy the plot images, but you can if you want to. Rewrite it a bit to make it flow better. Keep in mind that the introduction should explain the project to someone who does not have the actual assignment document handy: this could be you next semester or next year. (Do not discard any of your work!)
2. **Output**
   Run the program 4 times. Each time let the program run for a minute or so, then save an image of the plot window. Paste the 4 plot images into the document.
3. **Source code**
   Copy & paste the contents of your .m file(s) here. You do not need to write anything.

8. Submission

Submit the following things things on HuskyCT:

1. All the .m files for this project stored in a single 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:

- (2 points) The comment block at the beginning of the script file is correct, and each function has a brief comment section in it.
- (12 points) The program works correctly and contains all the necessary functions.
- (3 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
- (3 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.