CS1112 Fall 2022 Project 3 due Wednesday Oct 5 at 11pm

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Ground Rule

As stated before, do *not* use any built-in functions or constructs that have not been discussed in the course.

Objectives

Completing this project will solidify your understanding of user-defined functions and vectors. You will learn how to do a *sensitivity analysis*, which is an important concept and tool in engineering and computational science, and produce scientific graphics.

1 Cornell Tennis Center

Complete Problem **P5.3.7** in *Insight* (page 126), with a few additional requirements/changes described here. Be sure to read *Insight* §5.3 first—it'll help you with this problem! You will submit two m-files in CMS: a *function* file `DrawTennisCourt.m` and a *script* file `CornellTennis.m`.

Function `DrawTennisCourt` draws *one* tennis court as specified in the problem statement in the book. Relative to the example in §5.3, `DrawTennisCourt` has a similar role as the function `DrawFlag`. Your script `CornellTennis` has the role of setting up the figure window and drawing the "tennis center" (by calling your function `DrawTennisCourt`). In your script use the figure window setup commands that are used in `Eg5_3` (the first four commands). Additionally, use the command `hold off` at the end of the script.

Download the function file `DrawRect.m` (same as that used previously in P2 and in Lecture). While you can specify the color for filling the rectangle, `DrawRect` outlines the rectangle in black, which doesn't look right for our tennis court drawing. Therefore, you will modify `DrawRect` so that it does not outline the colored rectangle. This can be done simply by modifying the last statement in `DrawRect`, which calls the built-in function `fill`, to be

```
fill(x,y,c,'EdgeColor','none')
```

Change the function name in the function header to be `DrawRectNoBorder` and save the modified function in a file `DrawRectNoBorder.m`. Call `DrawRectNoBorder` instead of `DrawRect` in drawing the tennis court. You do not need to submit the file `DrawRectNoBorder.m`; we will use our version of the file when we evaluate your code.

Submit your files `DrawTennisCourt.m` and `CornellTennis.m` in CMS (after registering your group).

2 Further Analysis of the Golf Ball Trajectory Model

In Project 2 we simulated the trajectory of a golf ball launched into the air and subject to air drag. Now we will perform a *sensitivity analysis* on the trajectory model. A *sensitivity analysis* is used to explore how the result of a model changes when the inputs to the model are systematically varied, i.e., we ask the question "how sensitive is the result to variations in the model parameters?"

Specifically, in this part of Project 3 you will investigate how the trajectory changes given changes in the launch angle ϕ and the coefficient of friction k. We decompose the problem into two parts: (1) implementing the trajectory simulation model as an independent function and (2) performing the sensitivity analysis by running the model repeatedly with systematically varied inputs and producing visualizations.

2.1 Simulation model as a function

Implement the following function:

```
function [xvec, yvec] = golfTraj(dt,x0,y0,v0,phi,k)
% Simulate the trajectory of a golf ball from launch to landing.
% Input parameters
%   dt:    time step used in simulation, a positive value in seconds
%   x0,y0: scalar initial x- and y-positions, in meters.  y0>=0.
%   v0:    scalar initial velocity, a positive value in m/s
%   phi:   launch angle in radians, a positive value <=pi/2
%   k:     friction coefficient, a positive value <1
% Return parameters
%   xvec,yvec:  vectors of the same length storing the positions of the
%               golf ball:  xvec(i), yvec(i) is the position of the ball
%               after the ith time step.  The last value in yvec should be
%               set to zero.
```

The simulation ends when the golf ball returns to the ground, i.e., when its y position becomes zero or negative. The last iteration of the simulation will likely produce a negative y value instead of an "exact" zero. You will set that last value in `yvec` to zero.

Below are the relevant equations for the trajectory model; the detailed explanations can be found in the Project 2 document.

Given the initial velocity v_0 and launch angle ϕ, the initial velocities in the x- and y-directions are $v_x = v_0 \cos\phi$ and $v_y = v_0 \sin\phi$. Let $\triangle t$ be a discrete time step. At each step of the simulation, compute the new velocities and positions as follows:

$$\begin{aligned} v_x\text{new} &= v_x - \triangle t \cdot k \cdot v_x\sqrt{v_x^2 + v_y^2} \\ v_y\text{new} &= v_y - \triangle t \cdot (k \cdot v_y\sqrt{v_x^2 + v_y^2} + g) \\ x\text{new} &= x + v_x \cdot \triangle t \\ y\text{new} &= y + v_y \cdot \triangle t \end{aligned}$$

where the acceleration due to gravity, g, is 9.81 m/s^2.

Your code from Project 2, correct or corrected, can be the basis for this function `golfTraj`! You may also choose to use the posted example solution of Project 2 as the basis. However, *please be sure to use the appropriate parameter names as specified in the above function header.* Do not modify the given function header above. Furthermore, do not just "dump" all your Project 2 golf ball simulation code into this function: remove any part that is unnecessary for *this* function, i.e., carefully reconsider every part of the simulation as you write the code.

2.2 Sensitivity analysis

You will evaluate the model's sensitivity to changes in the parameters ϕ and k independently. Write a script `golfTrajSensitivity` to perform the sensitivity analysis. Use these constant values: $\triangle t = .05$ second, initial position $x_0 = 0$, $y_0 = 0$ (in meters), and $v_0 = 100$ m/s.

2.2.1 Sensitivity to launch angle ϕ

Run your golf trajectory model, i.e., call your function `golfTraj`, on several values of ϕ in this range: $0 < \phi \leq \pi/2$. Use at least four and no more than six values. (Too many will produce a very busy graph.) In these simulation runs, fix the coefficient of friction, k, at 0.02. Plot in a figure window the trajectories for all the launch angles evaluated. The following code outline shows the relevant graphics commands for plotting the curves:

```
close all         % Close all figure windows
figure            % Start a new figure

%%%% Sensitivity to launch angle phi
subplot(2,1,1)    % First subplot in this figure window
hold all          % See Note 1 below

legendText = {};  % Initialize array variable for storing text.  The curly
                  %   brace notation will be discussed later in the course.
phiVec =          % Vector of launch angle values to evaluate
for i = _____     % Loop over the set of launch angle values

    % Determine the trajectory for the ith launch angle and store the trajectory
    % (the returned vectors of x- and y-coordinates)

    % Call function plot to plot the trajectory, see Note 2 below

    legendText{i} = sprintf('phi=%.2f', ___);  % See Note 3 below

end

legend(legendText) % Make a legend using the text in legendText

% Add a title and labels for the x and y axes

hold off           % Needed before you do the next sensitivity analysis on k
```

Note 1 Users of MATLAB 2015b or a later release may use `hold on` or `hold all`, but users of older versions of MATLAB must use `hold all`, which cycles the plot colors automatically so that each call to `plot` produces a curve in a different color (within the built-in set of colors). `hold on` does not cycle the colors in older versions of MATLAB but does so in the 2015b and later releases.

Note 2 Recall that the returned vectors from function `golfTraj` do not include the beginning coordinates. Therefore you need to *concatenate* to those returned vectors the beginning coordinates (x_0, y_0) in order to plot the complete trajectory. To concatenate is to "glue together." Here are two examples that you should try in MATLAB:

```
a=3;      b=[4 6];     c=[a b] % concatenate a and b horizontally into c (a row
                               %   vector) using the space as the separator
f=[5; 7]; g=[9; 8; 6]; h=[f;g] % concatenate f and g vertically into h (a column
                               %   vector) using the semi-colon as the separator
```

Note 3 On the blank should be the i^{th} launch angle. Note the use of the curly brace, not parenthesis, around the index of the variable `legendText`. We will discuss this kind of array later in the course.

Note 4 Observe the use of the command `subplot`. `subplot` allows us to break up the figure window into separate sections with an independent set of axes in each section. `subplot(2,1,1)` says to break the figure window into two rows by one column (so two sections, one below the other) and use the first section. (In the next part of the sensitivity analysis, you will use `subplot(2,1,2)` to show a plot in the second section.)

2.2.2 Sensitivity to coefficient of friction k

Continue writing code in `golfTrajSensitivity.m` to do a sensitivity analysis on k. Fix the launch angle at $\pi/4$ and vary k over the range of 0.01 to 0.2. Again, choose four to six k values to show the variability of the trajectory due to changes in k. Use a similar graphics framework as shown above to produce a plot of all the trajectories at the chosen k values. This will be the second subplot in the figure, so start this second visualization with the command `subplot(2,1,2)`.

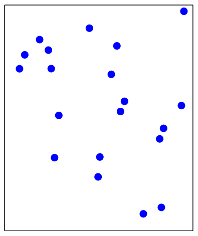
Finally let's change the size of the figure window so that the two subplots are more spaced out. Write the following statement immediately below the `figure` command that was used to start the figure window:

```
set(gcf, 'Units', 'normalized', 'Position', [.3 .2 .5 .6])
```

This `set` statement positions the lower left corner of the figure window at 30% from the left edge of the screen and 20% from the bottom edge of the screen, and sets the width and height of the figure window to be 50% and 60% of the screen's width and height, respectively. This and a select set of graphics format options are presented in *Appendix A: Refined Graphics* in our textbook. You can also search the MATLAB documentation for even more options!

Submit your files `golfTraj.m` and `golfTrajSensitivity.m` on CMS (after registering your group).

3 Gas Molecules Simulation



You will develop a simulation of the movement of gas molecules in a confined 2-dimensional space. (Or you can think about it as the motion of rigid, frictionless billiard balls.) The molecules will bounce off walls and one another. In our simplified model, the collisions are fully "elastic"—there is no loss of energy or momentum.

You will simulate the molecules over T time periods. The simulation has this organization:

- *Given initial positions and velocities*
- *Draw all the molecules*
- *For each time period*
  - *For each molecule*
    - *Check for bouncing against border; update velocities and positions as necessary*
    - *Check for collision with other molecules; update velocities as necessary*
    - *Calculate new position*
  - *Draw all the molecules*

Using good program development practice, we first lay out the organization (as above) and then we can work on one subproblem at a time. Another good program development strategy is to solve a simplified problem first, so we will start with just one molecule.

Download from the course website the file `gasMotionModel.m`. This script is in charge of initializations and calling the function `inMotion`, which you need to write, to perform the simulation. Read `gasMotionModel` carefully. The script is set up to initially include only one molecule in the simulation. As you develop your simulation you will need to change `gasMotionModel`.

You will submit in CMS four function files: `DrawDiskNoLine.m`, `drawMolecules.m`, `inMotion.m`, and `checkImpact.m`. You may implement additional *sub*functions in these files as you see fit, but this is not necessary.

3.1 Drawing Molecules

Let's start with drawing. Having this functionality allows you to easily check visually the different subproblems that you'll solve later. You'll need two functions: `DrawDiskNoLine` and `drawMolecules`.

3.1.1 `DrawDiskNoLine`

You will make a simple modification to the given `DrawDisk` function: draw a colored disk `without` a black outline. First change the function name (and therefore file name) to be `DrawDiskNoLine`. Next read the original function body and note that `fill` is the graphics command that draws a polygon outlined in black and fills it with a color. To draw a disk without a black outline, you will modify the last statement to call `fill` as follows:

```
fill(x,y,c,'LineStyle','none')
```

`LineStyle` is one of many properties of graphics objects in MATLAB. In addition to `'none'` used here, in previous projects you experimented with other `LineStyle` property values, including `'--'` for a dashed line and `':'` for a dotted line. There is no need to memorize such graphics options; look them up in the MATLAB documentation whenever you need them! For example, type in the the Command Window `doc fill` if you wish to learn how to control the properties used by function `fill`.

3.1.2 `drawMolecules`

Implement function `drawMolecules` as specified:

```
function drawMolecules(x,y,r,w,h)
% % Draw all molecules with axis limits w (x direction) and h (y direction).
% x, y are vectors of the same length: (x(k),y(k)) is the position of the kth molecule.
% All molecules have radius r.
% Assume r<w/2, r<h/2, and all the molecules lie completely inside the borders.
% The first molecule is magenta; all other molecules are blue.
```

Start your function with these graphics commands to maintain the correct axes throughout the simulation:

```
cla                      % Clear axes (i.e., remove all drawn objects)
axis equal manual        % Axes have equal scaling and are frozen at current scale
axis([0 w 0 h])          % Set axes limits: x-axis ranges from 0 to w; y-axis ranges from 0 to h
set(gca, 'xtick', [])    % No x-axis tickmarks
set(gca, 'ytick', [])    % No y-axis tickmarks
box on                   % Draw border
hold on                  % Subsequent plot/fill commands appear on current axes
```

Remember to put the `hold off` command at the end of this function. If you later have errors in your simulation you may find it useful to see the axis tickmarks—simply temporarily comment out the two `set` commands given above. Notice that the function starts with the command `cla` to remove previously drawn objects. Write your code to draw one of the molecules in a different color so that later when you have a large number of molecules it is easier to track the motion of that one molecule. Make effective use of function `DrawDiskNoLine`.

Be sure to test your function. In the Command Window call your function `drawMolecules`: `drawMolecules(1,3,.2,6,4)` should draw one molecule, in magenta, 1/6 the box width from the left and 3/4 the box height from the bottom. Try other values to make sure that your function is correct.

3.2 Simulating Gas Molecule Motion

Implement function `inMotion` as specified:

```
function [xFinal,yFinal] = inMotion(x,y,vx,vy,r,w,h,T)
% Simulate the motion of molecules in a 2-d space with width w and height h
%   over T time steps.  All molecules have radius r.
% x and y are vectors where (x(k),y(k)) is the position of the kth particle.
% vx and vy are vectors:
%   vx(k) is the x-velocity of the kth particle.
%   vy(k) is the y-velocity of the kth particle.
% Return parameters:
%   xFinal and yFinal store the positions of the molecules after T time steps:
%     (xFinal(k),yFinal(k)) is the final position of the kth particle.
```

The first action (statement) in this function is to draw the molecules at their initial locations: just a call to function `drawMolecules`. To start testing, run the given script `gasMotionModel`, which calls your function `inMotion`. You should see a figure window with a title and just one molecule in magenta. Next, start developing the simulation with just one molecule. Look at the overall algorithm given on page 5 again. Set up the overall organization, the loop(s) and comments of the "subtasks," in your function.

3.2.1 Simple Motion: Calculating the New Position

The distance traveled in one time step is $v\triangle t$ where v and $\triangle t$ are the velocity and the change in time, respectively. Consider each time period a unit time, so $\triangle t = 1$. Throughout this simulation we work with the x and y components independently, so we have

$$x' = x + v_x \quad \text{and} \quad y' = y + v_y$$

where x and y are the current coordinates, x' and y' are the coordinates after the time step, and v_x and v_y are the x- and y-components of the current velocity of the molecule. Remember that a simulation is just an approximation—the new position calculated may be outside the borders. So in general a realistic-looking simulation would need low velocities (or short time steps) so that such approximation errors are small.

Draw the molecules at the end of a time period. Insert a pause (in code execution) of .01 seconds to create the visual effect of the molecule in motion.

Run `gasMotionModel` for testing. You should see the molecule moving in a straight line and eventually moving off the box (disappearing).

3.2.2 Bouncing Off the Wall

Literally. When a molecule hits a vertical wall, its x-velocity reverses—just a sign change. Similarly, the y-velocity reverses when a molecule hits a horizontal wall. Check for bouncing off walls, and if appropriate update the velocity components, one direction at a time. When does a molecule hit a wall? When the *edge* of the molecule is currently at or beyond the border. In order to reduce the appearance of a molecule entering a wall, if the edge of the molecule is beyond a border, update the appropriate coordinate so that the edge is exactly at the border.

Run `gasMotionModel` for testing. Now the single molecule should bounce around inside the box! You should increase the number of time steps in `gasMotionModel` now so that you can see several bounces off walls. Increase the number of molecules, `n` in `gasMotionModel`, to 2. You will see that the molecules "pass through" one another.

3.2.3 Collisions

A discussion of simple collision between two objects can be found in most elementary Physics textbooks.[1] For our purposes only the "simplified" equations for calculating the post-collision velocities of two molecules numbered 1 and 2 are given here:

$$\begin{aligned}
v'_{1x} &= \frac{1}{d_x^2 + d_y^2}\left(v_{2x}d_x^2 + v_{2y}d_xd_y + v_{1x}d_y^2 - v_{1y}d_xd_y\right)\\
v'_{1y} &= \frac{1}{d_x^2 + d_y^2}\left(v_{2x}d_xd_y + v_{2y}d_y^2 - v_{1x}d_xd_y + v_{1y}d_x^2\right)\\
v'_{2x} &= \frac{1}{d_x^2 + d_y^2}\left(v_{1x}d_x^2 + v_{1y}d_xd_y + v_{2x}d_y^2 - v_{2y}d_xd_y\right)\\
v'_{2y} &= \frac{1}{d_x^2 + d_y^2}\left(v_{1x}d_xd_y + v_{1y}d_y^2 - v_{2x}d_xd_y + v_{2y}d_x^2\right)
\end{aligned}$$

where $d_x = x_2 - x_1$, $d_y = y_2 - y_1$, and the apostrophe, or prime symbol, indicates *post*-collision (i.e. v'_{1x} is the post-collision x-component of the velocity of particle 1). It does not matter which molecule is numbered 1. Of course, you apply these equations only if two molecules are colliding. In our model, two molecules collide if there is any overlap between the molecules. Implement function `checkImpact` as specified:

```
function [vx1,vy1,vx2,vy2] = checkImpact(x1,y1,vx1,vy1,x2,y2,vx2,vy2,r)
% Check for collision between two molecules and update their velocities
%   as appropriate.  If there is no collision the velocities do not change.
```

[1]An excellent and concise discussion (that assumes some knowledge of vector algebra) can be found at `http://www.vobarian.com/collisions/2dcollisions2.pdf`.

```
% Parameters:  at current time, i.e., BEFORE checking for collision
%   x1,y1 are scalars representing the position of molecule 1.
%   vx1,vy1 are scalars representing the x- and y-velocities of molecule 1.
%   x2,y2 are scalars representing the position of molecule 2.
%   vx2,vy2 are scalars representing the x- and y-velocities of molecule 2.
%   r is a scalar representing the radius of each molecule.
% Return parameters:  AFTER checking for and possibly handling a collision
%   vx1,vy1 are scalars representing the x- and y-velocities of molecule 1.
%   vx2,vy2 are scalars representing the x- and y-velocities of molecule 2.
```

Test your function by calling it in the Command Window. For example, the following call should have two molecules traveling horizontally toward each other collide head-on:

```
[vx1,vy1,vx2,vy2] = checkImpact(3,5,15,0,3.4,5,-15,0,.2)
```

The returned values should indicate that the x-velocities of the two molecules switch and the y-velocities remain 0.

Now that you have function `checkImpact`, you can call it from function `inMotion`. We will deal with *pair-wise* collisions only: for each molecule, check against all *other* molecules. (This is not the most efficient algorithm, but it is fine for our simulation.) Be careful with how you set up your loops. Suppose I have four molecules; examine the following fragment:

```
for k= 1:4
   for j= 1:4
     fprintf('%d%d ', k, j)
   end
end
```

The output is `11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44` . That's all combinations of the indices but with duplications: combination `12` is the same as combination `21`, for example. Additionally the combinations "with itself" (`11`, `22`, ..., etc.) are included. To check all pairwise collisions, in the case of four molecules in total, you only need to check these combinations: `12 13 14 23 24 34`. *Consider carefully how you should set up your loops to include only unique—non-duplicate—combinations.*

Note that we are not concerned with the error in a collision of more than two molecules given our simple algorithm. For example, suppose molecules 1, 2, and 4 are currently positioned such that molecules 1 and 2 collide and molecules 1 and 4 collide. Given our simple pair-wise algorithm, the velocities of molecules 1 and 2 are first updated using the post-collision velocity equations shown above. So by the time molecules 1 and 4 are processed, the velocities of molecule 1 are no longer the original velocities and our code will be calculating the post-collision velocities between molecules 1 and 4 incorrectly. We are not concerned with such errors.

Run `gasMotionModel` to see the result. If two molecules seem to move correctly then change `n` to 3 (number of molecules) and run the program again.

Our collision approximation scheme is far from exact, but it gives reasonable results most of the time, especially for a small number of molecules. Visible problems (like multiple molecules stuck and tumbling together) are seen sometimes when molecules have significant overlap before we detect a collision (velocity values are too large or initial positions overlap), in some 3-molecule (or more) collisions, and in some collisions at corners. We are not concerned with these errors in this project. Have fun watching the simulation!

Submit your files `DrawDiskNoLine.m`, `drawMolecules.m`, `inMotion.m`, and `checkImpact.m` on CMS.