CS/INFO 1305 Programming Exercise 3
Due Sunday, July 29, at 10pm
Submit **either** Level 1 or Level 2.

Different ways to create vectors

Type the following expressions in the MATLAB *Command Window* to see what kind of vectors they create. Write the resulting vectors (and answer the questions) on the blanks.

```
a= zeros(1,4)  %_________________

b= zeros(4,1)  %_________________  What do the arguments specify?_____________________________

c= ones(1,3)   %_________________

d= 10:2:17     %_________________

f= 10:-1:17    %_________________

g= [10 20 40]  %_________________  What does the space separator do?_____________________

h= [10,20,40]  %_________________  What does the comma separator do?_____________________

k= [10;20;40]  %_________________  What does the semi-colon separator do?_________________

m= [a g]       %_________________

n= [b; k]      %_________________

p= [a k]       %ERROR--mismatched dimensions! (Attempt to concatenate a column to a row)

q= b'          %_________________  This operation is called "transpose"

r= [a b']      %_________________
```

1 Level 1

1.1 Evaluate a polynomial

Write a function `evalPoly` to evaluate an n^{th} order polynomial of x:

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

The two input parameters are `coef` and `x`. `coef` is a vector or real values of length $n + 1$ and contains the coefficients of the polynomial. `coef(1)` corresponds to a_0, the coefficient for the term x^0. Input parameter `x` is a real value. Function `evalPoly` returns the value of the polynomial evaluated at `x`.

1.2 Minimum value in a vector

Implement the following function:

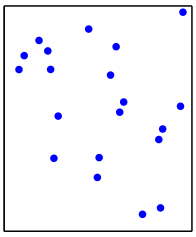
```
function [val, k] = findMin(v)
% Find the minimum value in vector v.  v is a vector of real numbers.  length(v)>0.
% val is the minimum value in v.
% k is the first position at which the minimum value appears.
```

1.3 Biggest rectangle

Implement the following function:

```
function [a,b,c,d] = biggestRectangle(x,y,v,w)
% Find the rectangle with the largest area.
% x,y,v,w are vectors of the same length containing real numbers.  length(x)>0.
% The points (x(1),y(1)) and (v(1),w(1)) are the opposing corners of rectangle 1,
% the points (x(2),y(2)) and (v(2),w(2)) are the opposing corners of rectangle 2,...
% the points (x(k),y(k)) and (v(k),w(k)) are the opposing corners of rectangle k.
% (a,b) and (c,d) are the opposing corners of the biggest rectangle in the set of
% rectangles defined by x,y,v,w.
```

2 Level 2: 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 `moleculesModel.m`. This script is in charge of initializations and calling the function `motion`, which you need to write, to perform the simulation. Read `moleculesModel` 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 `moleculesModel`.

You will submit in CMS three function files: `drawMolecules.m`, `motion.m`, and `checkCollision.m`. In file `drawMolecules.m` you will include a subfunction `DrawDiskNoLine` as specified. You may implement additional *sub*functions in the three files to be submitted as you see fit.

2.1 Drawing Molecules

Let's start with drawing. Having this function allows you to easily check (visually) the different subproblems that you'll solve later. Implement function `drawMolecules` which has the following specifications:

```
function drawMolecules(x,y,r,w,h)
% Draw all molecules with axis limits w (x dir) and h (y dir).
% x, y are vectors of same length: (x(k),y(k)) is 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([0 w 0 h])        % Set axes limits
axis equal manual      % Axes have equal scaling and are frozen at current scale
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 before 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. We 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.

Implement a *subfunction* `DrawDiskNoLine` under function `drawMolecules` to help complete the drawing task. (Recall that implementing a subfunction simply means writing an additional function in the same file of the main function.) `DrawDiskNoLine` has the same function header as `DrawDisk`, a function that we have used many times before (see course website). `DrawDiskNoLine` differs from `DrawDisk` in that the disk drawn does not have a black border. This is easily achieved by calling built-in function `fill` with an extra property '`Linestyle`':

```
fill(xvector,yvector,color,'Linestyle','none')
```

Make sure that your function file `drawMolecules.m` has two (separate) functions in it: function `drawMolecules` and subfunction `DrawDiskNoLine`. Now 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.

2.2 Simulating Gas Molecule Motion

Implement a function `motion` which has the following specifications:

```
function [xFinal,yFinal] = motion(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 `moleculesModel`, which calls your function `motion`. You should see a figure window with a title and just one molecule in magenta. Now start developing the simulation with just one molecule. Look at the overall algorithm given above again. Set up the overall organization, the loop(s) and comments of the "subtasks," in your function.

2.2.1 Simple Motion: Calculating the New Position

The distance traveled in one time step is $v\triangle t$ where v and $\triangle t$ are velocity and 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. Then use the statement `pause(.01)` to pause program execution for 0.01 second to create the visual effect of the molecule in motion.

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

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

2.2.3 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. 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 a function `checkCollision` that has the following specifications:

```
function [vx1,vy1,vx2,vy2] = checkCollision(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.
% Parameters:  at current time, i.e., BEFORE any 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 collision (if any)
%   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] = collide(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.

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

Now that you have function `checkCollision`, you can call it from function `motion`. 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. 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 repetitions: 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 you only need to check these combinations if there are four molecules in total: `12 13 14 23 24 34`.

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 `moleculesModel` 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 large or initial positions overlap), in some 3-molecule (or more) collisions, and in some collisions at corners. Have fun watching the simulation!