Modeling, Simulation and Analysis
CS 250
Spring 2018
Homework 1
Due in part **BY 11:59PM** Wednesday, February 28, and
in part **AT THE BEGINNING OF CLASS** Thursday, March 1
(Please see the notes below!)

- The non-programming portions of this homework, including printouts of all code, are due at the beginning of class on March 1.

- The programming portions of this assignment are to be electronically submitted to our course dropbox (using the `submit250` script) by 11:59pm on February 28.

  `submit250 hw1 <your-directory-name>`.

  In your directory for this HW, please have a separate Matlab code file (i.e., a `.m` file) for each exercise; please name each file appropriately for the exercise, and include an identifier such as your name or userid as part of each file name. Please also include a PDF file of your write-up document in the directory. Make sure only the files needed for this HW are submitted!

- For all programming exercises in this course, the code should display answers to the relevant questions to the screen when it is run, not just simply store answers in variables that could be examined in the Matlab *Workspace*. Please use print statements to label output appropriately; as part of that, please look up the `fprintf` and `disp` statements, both of which can be used to display information to the screen. You may often find that `fprintf` yields cleaner output—if so, please use `fprintf` when appropriate!

  All output from programs should be descriptively labeled for readability.

- Common guidelines for good programming style apply:

  - No line should be longer than 80 characters.
  - Comment code effectively! At a minimum, every function or script should have a descriptive comment (e.g., a *contract*); please also add additional comments as needed for clarity and readability. Programs that require *excessive* testing to determine correctness will not receive full credit—the code and documentation (including comments!) should make it straightforward to test if the code works correctly.
  - Use whitespace (tabs, blank lines) for readability.
  - Variable names should make it easy to tell the purpose of each variable.
  - Avoid *magic numbers* in your code! Declare variable names for constant values, etc. This can greatly simplify testing code with different key parameter values.

  - Please avoid `break` statements and related constructs—most of the time, they indicate that more thought is needed in the design of the relevant loop.

- Some guidelines for style in Matlab:
  - Large headings for figures may not fit in the Figure window opened by the `plot` command. Please ensure any text put on figures will fit on a standard Figure window; use line breaks as needed.
  - Putting a `clear` statement (or a `close all` statement) in the middle or at the end of your code can make it impossible to inspect data generated by your code after it has finished its run. Please do not put `clear` statements in your scripts.

- The above guidelines apply to every programming assignment in this course. As usual, not following style guidelines or submission instructions may result in deductions on an assignment. Please feel free to ask me any questions about CS250 guidelines or instructions!

These exercises are intended to be introductions to Matlab, which will be used as the language for implementing projects for the remainder of the semester. Please feel free to ask your Prof. any questions about programming in Matlab as the semester goes along!

1. **Radioactive Decay!** This exercise is based on Exercise 2.2.6 in your textbook; please read the section on *Unconstrained Decay*, pages 28–30 of your textbook. In that section, it is noted that radium-226 has a decay rate of about 0.0427869% per year.

   (a) Write a Matlab script or function to compute and plot the function for what fraction of an original quantity Q_0 of radium-226 remains, as a function of years. (See Figure 2.2.5 in your textbook for an example of a similar function for carbon-14.)

   (b) What fraction of an original quantity Q_0 of radium-226 is left after 500 years? After 5,000 years? (As always, explain your answer.)

   (c) If 60% is left, how old is the radium-226? (As always, explain your answer.)

2. **A Big Piece of Pi!** In this exercise, you'll estimate the value of π! In particular, you'll use simulation techniques involving random number generation for the estimate. (These are called *Monte Carlo simulation* techniques; if you'd like, you can read more about Monte Carlo techniques in Section 9.2 of your textbook, but it's not necessary for this exercise. Indeed, this exercise is very similar to Project 4 from Section 9.2, page 387 of your textbook.)

   Consider a unit circle (i.e., with radius $r = 1$), defined by equation $x^2 + y^2 = 1$. Further, consider the "top right" quarter of that circle, in the quadrant defined by

$x \in [0..1], y \in [0..1]$; note that the area of the entire circle ($\pi r^2 = \pi$, because $r = 1$) is 4 times the area of the circle in that quadrant.

We can estimate the value of π indirectly by estimating the area of the circle. Here, we'll estimate the area of the quarter of the circle described above, using Monte Carlo techniques. To do this, randomly generate points (x, y) in the range $x \in [0..1], y \in [0..1]$; keep track of how many points are inside the circle and calculate the ratio

$$\frac{number\ of\ points\ inside\ the\ circle}{total\ number\ of\ points\ generated}.$$

Because the points are randomly generated, the fraction of them inside the circle is an estimate of the fraction of the area of the quadrant that is taken up by the circle! (Do you see why?) From that estimated area, we can then estimate the value of π.

So, for this exercise, write code to generate n points (where n is a variable / constant in your code) and estimate the area of π based on the above method. Use that code to do the following:

(a) Run repeated trials with different values of n, to determine the smallest value of n for which you get good estimates of π. Start with $n = 1000$. In your write-up, record all values of n you try, say what "good" value of n you selected, and explain why you selected it.

(b) As part of visualizing the simulation, your code should plot a display showing points inside and outside the circle. One possible display might be something like Figure 1.

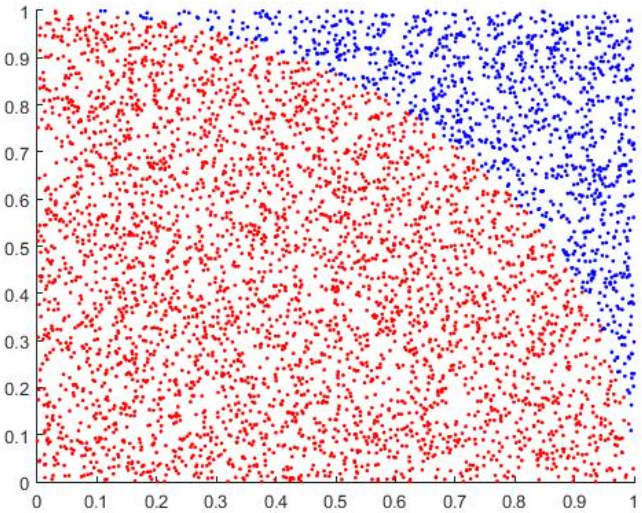

Figure 1: A display of points inside and outside a circle. Note that this particular plot does *not* correspond to an especially good estimate of π!

Look up options to the `plot` command to see how to plot things with different colors, etc. Please be sure to look at the plots for every value of n you test that isn't too large! (For very large n, plotting can be very slow! If you test values of n and decide not to plot them because plotting is too slow, just be sure to indicate that in your write-up.)

(c) With the value of n selected for good estimates, use a loop to run the code 1000 times, and output the maximum, minimum, and average values of π that were estimated. (For this portion of the exercise, be sure not to generate any plots!)

(d) All of this was done with $r = 1$. In your write up, explain what would be different if r were some other (positive) number. Could you make this approach work to estimate π with a different r? If so, how would you change your code to do that? If not, why not? (You do not need to enter or run any new code for this—it's for the write-up only—but if you'd like to include different code in your write-up as part of your answer, you're welcome to do so.)

Optional Practice Exercises

The exercises below are not to be submitted and will not be graded. They are, however, excellent practice exercises, covering useful material for Modeling & Simulation in Matlab. You are encouraged to work on them and discuss them with your classmates or your Prof.!

- **Randomness!** Write a Matlab script or function that will use the random-number generator `rand` to help you determine the number of random numbers needed such that ...

  1. ... the sum of the numbers is at least 10
  2. ... the sum of the numbers is at least 100
  3. ... a number between 0.475 and 0.525 is generated
  4. ... the mean of the numbers generated is within 0.05 of 0.5
  5. ... the mean of the numbers generated is within 0.01 of 0.5

  You are welcome to submit one script that does all of the above, or separate functions for each.

  In addition to submitting your code, please write (as part of your on-paper HW submission) the answers to the above five questions, and also write a paragraph or so that answers the following questions and demonstrates your understanding of programming using random number generators.

  – What gives you confidence in your answers?
  – How many times did you run the script before you settled on an answer for each of the 5 questions, and why?
  – Which of those 5 answers were easiest to predict without programming (if any), and why?

- **A Staggering Exercise?** This exercise asks you to implement 1-dimensional *random walks*: In a random walk, a walker starts at a point and moves in some randomly chosen way at each timestep. (You can read about random walks in Module 9.5 in your textbook, but that is not necessary for this exercise.) Note that because this is a 1-dimensional walk, there are only 2 directions in which a walker can move, positive or negative.

  Here, your walkers will all start at the origin (point 0) and walk inside an area with boundaries at $+B$ and $-B$ units from the origin (where B is a parameter for the simulation). At each timestep, each walker takes a step of randomly chosen amount, as selected by the `randn` function; so, code for a walk of N steps by a single walker might look something like

```
steps(1) = 0;
for i = 2:N
    steps(i) = steps(i-1) + randn;
end
```

  For purposes of this exercise, we will say the first step is the one with index 1, and it is at position 0, for all walkers. (One might in principle say, then, the walkers *actually* take $N - 1$ steps, not counting the initial placement as the first step. By our conventions, though, the initial placement is the first step.)

  If a walker hits or exceeds boundary distance B, it should stay frozen at its position for the remainder of the simulation.

  Write a Matlab script (or function) to do the following:

  1. Simulate W random walks, each with *numSteps* steps, and boundary distance B. Set up your simulation to have $W = 20$, *numSteps* $= 5$, and $B = 5$, but **avoid the use of "magic numbers"**—explicitly have W, *numSteps*, and B as variables / constants to make it straightforward to run the same simulations with different parameter values. (Please feel free to ask your Prof. any questions about what this entails, or about the terminology involved!)
  2. Plot the data of every walk, showing the position of each walker at each step. One way to do this is a figure something like Figure 2, but you are welcome to visualize data differently, as long as your method is effective—experiment and find what works for you! Be sure to label your figure(s) to ensure readability.
  3. Calculate the average number of steps before a walker "collides"—i.e., reaches or exceeds the boundary distance—for those walkers that do collide.
  4. Plot the number of surviving (non-frozen) walkers as a function of the simulation step. As always, please label the axes / plot to ensure easy readability.
  5. For the above two items of data—the average number of steps before a collision (for walkers that collide) and the number of surviving walkers at each step—how do those data vary with the number of steps in the simulation? With the value

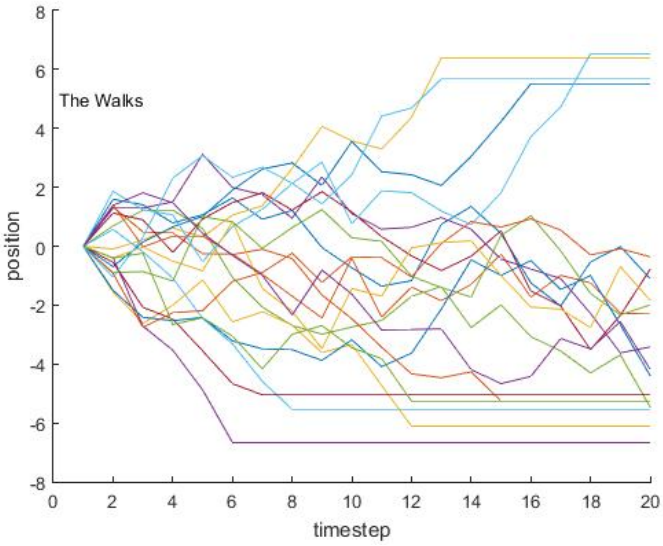

Figure 2: A display of random walks, graphing position at each timestep.

of B? With the ratio $\frac{numSteps}{B}$? Run several simulations and describe your findings. (Be sure to describe what simulations you ran to arrive at your findings, as part of your write-up!)

Parts (a)–(d) above are coding exercises and need not have separate on-paper answers; part (e) above should be answered in your on-paper write-up.