# CS1132 Spring 2010 Assignment 1

Adhere to the Code of Academic Integrity. You may discuss background issues and general solution strategies with others and seek help from course staff, but the homework you submit must be the work of just you. When submitting your assignment, be careful to follow the instructions summarized in Section 4 of this document.

# 1 That's pretty random...

This exercise will familiarize you with randomization and its use to simulate certain chance events.

## 1.1 Roll die, roll!

First you will simulate the outcome of rolling a fair $n$-faced die. Fair here means that the probability of getting each face is $1/n$. Write your code in a function `dieRoll`:

```
function i = dieRoll(n)
% n:  number of faces on the die
% i:  the outcome of one roll:  an integer from 1 to n, with equal probability.
```

## 1.2 m times

Ok, now let's simulate rolling $m$ $n$-faced dice. Call the function `dieRoll` for each die roll. Write your code in a function `mRoll`:

```
function v = mRoll(n, m)
% n:  number of faces on each die
% m:  number of dice
% v:  the outcome of rolling m dice:  an array of length m in which each entry
% is an integer from 1 to n.
```

## 1.3 Two equal

Let's find out experimentally what is the probability that when three dice are rolled at least two have the same value. Write a function `twoEqual` that simulates the repeated rolling of three 6-faced dice (by calling the function `mRoll`) and returns the fraction of the total rolls for which at least two dice had the same value. As the number of simulations increases (e.g., try 10,000), the fraction will approach the statistical probability. Function `twoEqual` has this specification:

```
function p = twoEqual(t)
% t:  number of simulations
% p:  a real number between 0 and 1 representing the probability of at least two dice
% having the same value when three dice are rolled.
```

## 1.4 Number sandwich

Now, let's find out experimentally what is the probability that the value of the third die is strictly between the values of the first two dice. Write a function `sandwich` that repeatedly simulates rolling three 6-faced dice (by calling the function `mRoll`) and returns the fraction of the total rolls for which the condition holds, i.e., the third value lies strictly between the first two.

```
function p = sandwich(t)
% t:  number of simulations
```

```
% p:  a real number between 0 and 1 representing the probability of the value of the third
% die being strictly between the values of the first two dice.
```

# 2 Square tiles and a coin, please

A coin of diameter 1 inch is thrown onto a surface that is an $n \times n$ array of square tiles. Each square's sides are two inches long. Consider that the lower left corner is at coordinates $(0, 0)$ and the diagonally opposite corner is at $(2n, 2n)$.

## 2.1 Throw it!

Write a function that simulates throwing the coin on the surface. Here we are only interested in the coin's center position; assume that the center of the coin lands randomly, with equal probability, anywhere on the surface. Your function returns the coordinates of the center:

```
function center = coinThrow(n)
% n:  number of tiles on each edge of the surface
% center:  a row vector with two entries, each between 0 and 2n, representing the
% coordinates of the coin center
```

## 2.2 Draw it!

For this part, we provide on the course website a skeleton for the function `draw.m`. Complete the function to draw the surface of the square tiles, call the function `coinThrow` to generate the coin's location, and draw the coin on the surface.
Here is how you draw a line between two points of coordinates $(a, b)$ and $(c, d)$:

```
plot([a, c], [b, d], '-')
```

Type `help plot` in Matlab to see more parameter options for the built-in function `plot`, if you are interested in colors and other formats.

Here is how you draw a circle of radius $r$ and center coordinates stored in the row vector *center*, using $p$ points:

```
theta = linspace(0, 2*pi, p);
x = r * cos(theta) + center(1);
y = r * sin(theta) + center(2);
plot(x, y);
axis square;
```

## 2.3 Landing within a tile

What is the probability that the coin lands entirely within a tile? Use simulation to estimate this probability. Write a function that simulates the throw many times and returns the fraction of times the coin lands entirely within one tile—it is ok if the coin is tangent to the tile border. Use the function `coinThrow` to simulate where the coin lands. Save your code in `within.m`:

```
function p = within(n, t)
% n:  number of tiles on each edge of the surface
% t:  number of trials
% p:  a real number between 0 and 1, representing the fraction of trials for which the
```

```
% coin landed within one tile.
```

Hint: each point $(x, y)$ on a circle satisfies this equation:

$$(x - c_x)^2 + (y - c_y)^2 = r^2$$

where $(c_x, c_y)$ are the coordinates of the center and $r$ is the radius. Note that it is possible to solve this question without using the above formula.

## 2.4 Where's the center?

Write a function that takes as input the coordinates of the center of the coin and returns the row and column numbers of the tile where the coin center landed. Assume that the tile in the lower left corner of the surface is in row 1 and column 1, while the tile in the upper right corner is in row $n$ and column $n$. Save your function in `centerToTile.m`. Here is the function's header:

```
function tile = centerToTile(center)
% center:  row vector with two entries representing the center coordinates of the coin
% tile:  row vector with two entries representing the row and column of the tile on the surface.
```

## 2.5 Aiming for the center

So far in this exercise, we assumed that all locations on the surface are equal candidates for where the coin lands. In other words, the probability is spread uniformly over the surface. In this part, we will explore what happens when one aims for the center of the surface and how we might simulate such a scenario in Matlab.

### 2.5.1 What does it mean to aim?

Typically, when one aims for a certain location, that particular location and the ones around it are more likely to be 'hit' than areas farther away. To describe such events, normal (or Gaussian) distributions are used in statistics to describe the clustering of the values of a variable around a mean value. In Matlab one may generate random numbers from a normal distribution using the built-in function `randn`. We have implemented a function which simulates the throw of the coin on the tile surface as if aiming for the center of the surface—the point of coordinates $(n, n)$. Please note that a large number of different implementations are possible corresponding to degrees of 'aiming accuracy', i.e., how large is the area around the center where most coins land. The code posted on the website (`normalCoinThrow.m`) is just one of many possible implementations. You will use this function to get a sense of how aiming changes the distribution of "coin landings".

### 2.5.2 To aim or not to aim

To observe the effects of aiming, you will write a function `comparison` which takes as input arguments $n$, the number of tiles on each edge of the surface, and $t$, the number of times we simulate the coin throw. Your code must do the following:

1. Simulate $t$ coin throws *without* aiming, i.e., use the function `coinThrow`. For each tile, count the number of coin center hits it gets. Use an $n \times n$ matrix, called $E$, initialized with all zero entries.

2. Then simulate $t$ throws *with* aiming, i.e., use the function `normalCoinThrow`. Note that while the majority of points generated by `normalCoinThrow.m` are on the surface, with the normal distribution there is always a chance of values abnormally 'far' from the mean. This means that the code may produce center values outside the interval $[0, 2n]$, i.e., center lies outside the surface. Your code must check whether the coin center is on the surface. If it is not, those values must be discarded. Use another $n \times n$ matrix, called $N$, to count for each tile the number of coin hits it gets.

3. To see how aiming changes the 'game', we focus on how the distance from the center of the surface changes the probability of hits in that area. For this, we define the following regions: region 1 is the one farthest away from the center, i.e., the border tiles. Region 2 is represented by the next layer closer to the center, and so on until the center region is reached. To clarify, consider the example of a $5 \times 5$ tile surface; in the following diagram, the number in each tile indicates the region:

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 1 |
| 1 | 2 | 3 | 2 | 1 |
| 1 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 |

   For each region, you must compute the number of coins that landed there. Store the results in two vectors: one for the regions of matrix $E$, the other for those of matrix $N$. For example, given the following coin count

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 3 |
| 4 | 0 | 2 | 2 | 1 |
| 5 | 2 | 0 | 1 | 2 |
| 1 | 1 | 7 | 0 | 4 |
| 0 | 3 | 1 | 1 | 0 |

   the per region count is: 32 for region 1, 15 for region 2, and 0 for region 3.

4. Finally, you are ready to compute the effect of aiming. Compute the ratio between the coin counts of $E$ and the coin counts of $N$ for each region. Store the results in a vector *ratio*, such that $ratio(i)$ represents the ratio for region $i$.

Save your code in a function `comparison`:

```
function ratio = comparison(n, t)
% n:  number of tiles on each edge of the surface
% t:  number of trials
% ratio:  a row vector of size ceil(n/2).  ratio(i) is the ration of the coin counts of E
% to the coin counts of N in region i.
```

# 3 Self-check list

The following is a list of the minimum *necessary* criteria that your assignment must meet in order to be considered *satisfactory*. Failure to satisfy any of these conditions will result in an immediate request to resubmit your assignment. Save yourself and the graders time and effort by going over it before submitting your assignment for the first time.
Note that, although all of these are necessary, meeting all of them might still not be *sufficient* to consider your submission satisfactory. We cannot list everything that could be possibly wrong with any particular assignment!

- Δ Comment your code! If any of your functions is not properly commented, regarding function purpose and input/output arguments, you will be asked to resubmit.
- Δ Suppress all unnecessary output by placing semicolons (;) appropriately. At the same time, make sure that all output that your program intentionally produces is formatted in a user-friendly way.
- Δ Make sure your functions names are *exactly* the ones we have specified, *including* case.

- Δ Check that the number and order of input and output arguments for each of the functions matches exactly the specifications we have given.
- Δ Test each one of your functions independently, whenever possible, or write short scripts to test them.
- Δ Check that your scripts do not crash (i.e., end unexpectedly with an error message) or run into infinite loops. Check this by running each script several times in a row. Before each test run, you should type the commands `clear all; close all;` to delete all variables in the workspace and close all figure windows.

# 4 Submission instructions

1. Upload files `dieRoll.m`, `mRoll.m`, `twoEqual.m`, `sandwich.m`, `coinThrow.m`, `draw.m`, `within.m`, `centerToTile.m` and `comparison.m` to CMS in the submission area corresponding to Assignment 1 in CMS.
2. Please do not make another submission until you have received and read the grader's comments.
3. Wait for the grader's comments and be patient.
4. Read the grader's comments carefully and think for a while.
5. If you are asked to resubmit, fix all the problems and go back to Step 1! Otherwise you are done with this assignment. Well done!