PSYCH 20A (PROF. FRANE) – LECTURE 8 Part A
Simulating Randomness

Seeding the random number generator (rng)

Matlab can't behave truly randomly, but it can simulate many types of randomness quite well. Before asking Matlab do something "random," we first set something called the *seed*. We can use a specific seed, in which case any time we repeat our program after setting that same seed again, we will get the same "random" results we got the other times we ran the program. A simple way to do this is to set the rng to `default` (typically near the start of the program), and then pick some arbitrary number for the seed (which can be any nonnegative integer less than 2^32; the default seed is 0). For example:

```
rng default % use the default random number generator
rng(9000)   % set the seed to 9000
```

But typically, we don't want the simulated randomness to be the same every time we run the program. For instance, we may be conducting an experiment in which we want to present a set of stimuli in an independently randomized order to each participant. In that case, we can ensure that our randomness is sufficiently random by setting the seed based on whatever the computer's system time (to a billionth of a second) happens to be. We do that by setting the rng to "shuffle," like this:

```
rng shuffle % seed the random number generator based on the current time
```

Note that if we have multiple "random" processes in a program, we don't need to reshuffle the rng for each one. Just shuffling the rng once at the start of the program makes all random processes sufficiently random for the remainder of the Matlab session.

Sampling from a uniform distribution

The `rand` function randomly samples values from a continuous uniform distribution between 0 and 1. You can get a single value, or you can get an array of any size by specifying the dimensions. For example:

`rand`	outputs a single value between 0 and 1
`rand(1, 10)`	outputs a row vector of 10 independently sampled values between 0 and 1
`rand(4, 5, 100)`	outputs a 4 × 5 × 100 array of independently sampled values between 0 and 1

The `unifrnd` function randomly samples values from a continuous uniform distribution between any two values. The first input is the lower bound, the second input is the upper bound, and the third input (optional) is a vector giving the dimensions of the array. For example:

`unifrnd(5, 7)`	outputs a single value between 5 and 7
`unifrnd(5, 7, [1 10])`	outputs a row vector of 10 independently sampled values between 5 & 7
`unifrnd(5, 7, [4 5 100])`	outputs a 4 × 5 × 100 array of independently sampled values btwn 5 & 7

NOTE: The `unifrnd` function requires the "Statistics and Machine Learning" toolbox (which is probably free with your student account). An alternative is to use the `rand` function, multiplying the output by the range of the distribution and then adding the lower bound of the distribution. For example, to output a row vector of 10 independently sampled values from a continuous uniform distribution between 5 and 7:

```
rand(1, 10) * (7 - 5) + 5
```

Simulating a coin-flip (or other binary decisions) using the rand function
"Uniform distribution" means that all values in the given range are equally likely. Hence, when you type `rand`, getting a value between 0 and 0.5 has the same likelihood as getting a value between 0.5 and 1. You can use this fact to make "coin-toss" decisions that have a 50% chance one way or the other:

```
if rand < .5
     headsOrTails = 'heads' ;
else
     headsOrTails = 'tails' ;
end
```

Or more simply, because `rand < .5` will have a value of either "true" (i.e., 1) or "false" (i.e., 0), we can consider 1 as "heads" and 0 as "tails" and simply say:

```
isCoinHeads  =  rand < .5
```

We can also make the coin-toss unfair. For example, for a 75% chance of heads:

```
isCoinHeads  =  rand < .75
```

Sampling from a normal distribution

The `normrnd` function generates an array of values randomly sampled from a normal distribution. The input arguments are the population mean, the population standard deviation, and the dimensions of the array you want to create. For instance, to get a 1 × 10 row vector of values, each independently sampled from a normal distribution with mean 0 and standard deviation 2:

```
mu             = 0                             ; % population mean
sigma          = 2                             ; % population standard deviation
randNormArray = normrnd(mu, sigma, 1, 10) ; % 1x10 sample from norm distribution
```

To sample from the *standard normal distribution* (a normal distribution with mean 0 and standard deviation 1), you can use the same `normrnd` function. Or you can use the `randn` function, which automatically uses population mean 0 and population standard deviation 1, so all you have to input is the dimensions of the array you want to output. For instance, to get a 1 × 10 row vector of values sampled from the standard normal distribution:

```
randStandardNormArray = randn(1, 10) ;
```

Sampling random integers

The `randi` ("random integer") function randomly samples integers from a designated range (with all integers in that range having equal likelihood of being selected).

By default, the lower bound of the range is 1. So you can just input the upper bound. For example, typing `randi(6)` selects an integer between 1 and 6 inclusive (simulating a fair die-roll). Typing `randi(100)` selects an integer between 1 and 100. And so on.

To use a value other than 1 for the lower bound, enter a two-value vector, in which the first value is the lower bound and the second value is the upper bound. For instance, `randi([5 10])` selects an integer between 5 and 10.

To output a whole array of independently sampled integers, add a second input to the `randi` function: a vector giving the dimensions of the array you want. For example, `randi([5 10], [3 2 4])` outputs a 3 × 2 × 4 array of integers, each independently sampled from the set of integers from 5 to 10.

Simulating a fair coin-flip using the randi function
We can simulate a fair coin-flip by typing `randi(2)` and considering heads vs. tails as 1 vs. 2.

Randomly ordering integers

The `randperm` ("random permutation") function generates a vector of integers from 1 to a designated upper bound, shuffled into a random order. For example, typing `randperm(3)` would output either [1 2 3] or [1 3 2] or [2 1 3] or [2 3 1] or [3 1 2] or [3 2 1] with equal likelihood.

By entering a second input to the `randperm` function, we can select a random *subset* of integers from the given range. For example, typing `randperm(3, 2)` would output only 2 values from the set 1:3. That is, it would output either [1 2] or [1 3] or [2 1] or [2 3] or [3 1] or [3 2] with equal likelihood.

Randomizing stimuli
A common use for `randperm` is shuffling the order of stimuli. For instance, imagine you have a list of 5 words in a string array, and you want to present them in random order. You can use `randperm` to create a shuffled vector of the integers 1 through 5, and then use that vector to index the word list (thus shuffling the words):

```
wordList          = ["pencil", "leaf", "table", "hallway", "lunch"] ; % words
wordShuffleOrder = randperm(5) ; % integers 1 thru 5 in random order
wordListShuffled = wordList(wordShuffleOrder) ; % all 5 words in shuffled order
```

Or to randomly shuffle only 3 words, randomly selected from the list of 5:

```
wordList          = ["pencil", "leaf", "table", "hallway", "lunch"] ; % words
wordShuffleOrder = randperm(5, 3) ; % 3 random integers from the set 1:5
wordListShuffled = wordList(wordShuffleOrder) ; % 3 selected words (shuffled)
```

Randomizing condition order
Suppose we want to present a series of 8 stimuli, exactly half in "condition 1" and exactly half in "condition 2." In that case, we can randomize the condition-order like this:

```
conditionTemp = [1 1 1 1 2 2 2 2] ; % unshuffled vector of condition-codes
conditionShuffleOrder = randperm(8) ; % integers 1 thru 8 in random order
condition = conditionTemp(conditionShuffleOrder) ; % shuffled condition-codes
```

Then our `condition` vector will be a random ordering of four 1s and four 2s, e.g.: `2 1 1 2 1 2 2 1`

Constraining the possible random orders
There are various ways to put constraints on what orders are possible. Often the simplest way is to use a while-loop and the `strfind` function (note that the `strfind` function can be applied to numeric arrays as well as strings or character arrays). For instance, to modify the above example so that it doesn't allow orders that have three 1s or three 2s in a row:

```
conditionTemp = [1 1 1 1 2 2 2 2] ; % unshuffled vector of condition-codes
condition     = conditionTemp     ; % initialize shuffled condition-code vector

% keep shuffling until condition vector doesn't have 3 like conditions in a row
while any(strfind(condition, [1 1 1])) || any(strfind(condition, [2 2 2]))
    conditionShuffleOrder = randperm(8) ;
    condition             = conditionTemp(conditionShuffleOrder) ;
end
```

PSYCH 20A – Lecture 8 Part B
Timing and Waiting

Using `tic` and `toc`

The simplest way to measure elapsed time is to execute the `tic` function when you want to start timing, and then execute the `toc` function when you want to stop timing. `toc` gives you the time elapsed time (in seconds) since `tic`. The following code records how long it takes the user to answer a question:

```
tic % start timing age response
age             = input('What is your age in years? ') ; % input age
ageResponseTime = toc ; % number of seconds user took to enter age
```

Instead of timing just one interval, we might want to time a series of intervals. We could still use `tic` and `toc` as above, because each `toc` would reference the most recent `tic`. But what if we want to time *overlapping* intervals? For example, we might want to time each response in a questionnaire, and also time the whole questionnaire overall. To do that, we save each `tic` as a variable, and then make each `toc` refer to a particular corresponding `tic`. For example:

```
totalTimeStart   = tic ; % start timing the questionnaire overall

ageTimeStart     = tic ; % start timing the response to the age question
age              = input('What is your age in years? ') ; % input age
ageResponseTime  = toc(ageTimeStart) ; % seconds user took to enter age

nameTimeStart    = tic ; % start timing the response to the name question
name             = input('What is your name? ', 's') ; % input name
nameResponseTime = toc(nameTimeStart) ; % seconds user took to enter name

totalTime        = toc(totalTimeStart) % total seconds elapsed in questionnaire
```

The value of `tic` is a *timestamp* representing the computer's current system time (to a billionth of a second). `toc` simply gets a new timestamp and compares it to `tic` in order to output the elapsed time.

Using `pause`

Waiting for a key-press:
If we just type `pause` with no inputs, then Matlab waits for the user to press a key from the command window. For example:

```
    fprintf('\nPress any key to continue\n') ;
    pause % wait for key-press before continuing
```

Waiting for a specified duration:
If we input a number of seconds to the `pause` function, then Matlab will wait for that amount of time instead of waiting for a key-press. For example:

```
    fprintf('\nGet ready. The experiment will begin in 10 seconds.\n') ;
    pause(10) % wait 10 seconds before continuing
```