

If you haven't already, download `MatlabPrograms.zip` from the course Blackboard site and extract all the files into a folder on your disk. Be careful not to overwrite any changes you may have made! That is, if you modified one of the downloaded programs and saved it with the same name, you may overwrite it when you download the Matlab programs again. Now set the path by pulling down File, Set Path, Add Folder ..., and navigate to the folder where you put the files.

Simulating gambling with absorbing states

- 1: Write a program called `GamblingSimulation.m` to simulate gambling with a friend, where you start with \$4 and your friend starts with \$6. Assume that each hand you play is worth \$1 and that you each have the same probability of winning. Store the amounts of money you have after each hand. When one of you runs out of money, stop the simulation. Then plot the record of the amount of money you had over time.
- 2: `GamblingSimulation` Type the name of the program at the Matlab command line to run it. It is helpful to alter the parameters for your simulation from the Matlab command line. You should be able to type something like `GamblingSimulation(10,4)` to simulate gambling with a total of \$10, where you start with \$4.
- 3: `function [void] = GamblingSimulation(total,start);` Add this line to the top of your program. Use `total` in place of the number 10 in your program, and `start` in place of 4.
- 4: `GamblingSimulation(100,40)` Explore how different parameters change the nature of the simulation.

It is also helpful for your program to return numbers relevant to the simulation to Matlab, so they can be used by another program, rather than just printed to the screen. It would be nice to be able to type, at the command line, `[GameLength, FinalWealth] = GamblingSimulation(10,4);` so that the values of the variables `GameLength` and `FinalWealth` would be set by the program.

- 5: `function [GL, FW] = GamblingSimulation(total,start);` Modify the top line of your program this way.
- 6: `GL = length(x);` This is one way to determine the length of the game, after the end state has been reached. Maybe it should be `length(x) + 1`?
- 7: `FW = x(end);` This is one way to determine the final wealth, whether you end with \$0 or \$10. Using the special word `end` inside a vector returns the last entry of that vector.

Now, let's write a program that runs your simulation many times, to determine the statistical properties of the game length and the final wealth. Name the new program `RepeatedGambling.m` and use these lines:

- 8: `function [void] = RepeatedGambling(total,start,numsim)` There will be three arguments, the total amount of money involved, your starting wealth, and the number of simulations to be done.
- 9: `GameLength = zeros(1,numsim);` Prepare a place to store data on the game length.
- 10: `FinalWealth = zeros(1,numsim);` Prepare a place to store data on your final wealth.
- 11: `for i = 1:numsim,` Repeat the following instructions many times.
- 12: `[GL,FW] = GamblingSimulation(total,start);` Run one simulation, store the output values.
- 13: `GameLength(i) = GL;` Store the game length for this simulation.
- 14: `FinalWealth(i) = FW;` Store the final wealth for this simulation.
- 15: `end` End the set of instructions to be repeated.
- 16: `figure(2)` Open a new figure window.
- 17: `clf` Clear the figure window.
- 18: `hist(GameLength,30);` Make a histogram of game lengths with 30 bins (the default is 10).
- 19: `figure(3)` Open a new figure window.
- 20: `clf` Clear the figure window.
- 21: `hist(FinalWealth,2);` Graph the final wealth values.

- 22: `printf('Percentage of time you won is %8.4f\n', 100*length(find(FinalWealth==total))/numsims);`
 Calculate the percentage of time that you ended up with all the money.
- 23: `RepeatedGambling(10,4,100);` Type this at the Matlab command line to run 100 simulations. You might want to comment out the line in the simulation program that makes the graph, since that takes time.
- 24: `mean(GameLength)` Calculate the average game length.

Transition matrix calculations for gambling

- 25: `P = GamblingMatrix(10);` Use the program `GamblingMatrix` to set up the Markov transition matrix for total wealth 10.
- 26: `P` Display the values in the matrix P.
- 27: `GraphTransitionMatrix(P,0:10);` Display the transition matrix in graphical form. The second argument gives names to the states.
- 28: `GraphTransitionMatrix(P^2,0:10);` Graph the 2-step transition matrix.
- 29: `GraphTransitionMatrix(P^10,0:10);` Graph the 10-step transition matrix.
- 30: `GraphTransitionMatrix(P^100,0:10);` Graph the 10-step transition matrix.
- 31: `Q = GamblingMatrix(100);` The transition matrix for a game with more money at stake.
- 32: `GraphTransitionMatrix(Q^200,0:100)` Graph the 100-step transition matrix.
- 33: `shading flat` Remove the grid lines from the graph.
- 34: `A = P^100` The 100-step transition matrix. Can we be sure the game is done after 100 steps?
- 35: `A = P^500` After 500 steps?
- 36: `A(5,1)` Your initial wealth, 4, is encoded as state 5. This is the probability of going from initial wealth 4 to wealth 0 in 500 steps.
- 37: `A(5,11)` This is the probability that you win all the money in 500 steps.
- 38: `AbsorptionTime(P,5);` Use powers of the transition matrix to calculate the probability distribution of the number of steps before the game ends, starting in state 5 (wealth 4).
- 39: `AbsorptionTime(Q,41);` With a total of \$100 at the table, calculate the probability distribution of the number of steps before the game ends, starting in state 41 (wealth 40).

Run length distribution

- 40: `RunLength` Simulate coin flips and keep statistical information on the lengths of runs.
- 41: `RunLengthDist` Set up a Markov transition matrix and calculate probabilities of absorption and absorption time.

Use these programs to explore the frequency of runs of varying lengths, the time until you get a run of heads of a certain length, etc.

Can you modify the transition matrix so that it will keep track of any type of runs, whether it be a run of heads or a run of tails?

Equilibrium location distribution in Monopoly

- 42: `Monopoly` Set up the transition matrix for Monopoly and calculate the equilibrium distribution.
- 43: `GraphTransitionMatrix(P,1:40)` Graph the transition matrix.
- 44: `shading flat` Remove grid lines.
- 45: `GraphTransitionMatrix(P^10,1:40)` Interpret this. Note that the grayscale is automatically rescaled; black no longer means a probability of 1.
- 46: `GraphTransitionMatrix(P^100,1:40)` Interpret this.

If you modify state 31 (Go to Jail) to be absorbing, you can explore the distribution of the length of time until you are sent to jail for the first time.

Simulating molecular diffusion

- 47: `DiffusionSimulation` Simulate molecular diffusion in a box with reflecting walls. `randn` generates a normally distributed random number, a number usually between -3 and 3, but most often between -1 and 1. Explain the effect of the variable `affinity`. What happens when the particle reaches the edge of the box?
- 48: `affinity(34:36) = 5*ones(1,3)`; Uncomment this line. How does it change the motion of the particle? What interpretation would you give? How can this happen to a molecule?

Generating random numbers with specified probabilities

At the end of the last set of instructions, we generated a random sequence using commands like:

- 49: `show(1+(rand(75,1)>0.5), 'HT')` Coin flips

This works well enough for simple experiments in which each trial results in just two outcomes. But for experiments with trials that have more outcomes, especially when the outcomes are not equally likely, a program called `rando.m` is more useful.

- 50: `p=[1 2 3 4 5 6]/21` A row vector (1 by 6 matrix) of probabilities for the outcomes of rolling a die.
- 51: `sum(p)` Make sure these sum to 1!
- 52: `rando(p)` Use these probabilities to simulate rolls of a die. Use the up arrow to repeat this command several times. You should get 5's and 6's more often than 1's and 2's.

Writing a Matlab program to simulate RNA sequences

You are going to create a new Matlab program called `RNARo11.m`, which will simulate a sequence of A,C,G,U which is supposed to look like RNA.

Above the command window is a blank white sheet; click on that to open a new editor window. In that window, type the commands below on separate lines, then save the program as `RNARo11.m` by clicking on the disk icon. In OS X, you may need to remove the directory information before the filename in order to save the file in Macintosh HD. In fact, it would be good to save the program in the folder `MatlabPrograms`.

- 53: `freq=[1 2 2 1]/6`; Probabilities for each of the four letters. C and G will occur more often than A and U.
- 54: `x=[]`; Clear out any old value of `x`.
- 55: `for i=1:400` What comes next will be repeated 400 times.
- 56: `x(i) = rando(freq)`; Generate a number from 1 to 4.
- 57: `end` The end of the commands to repeat 400 times.
- 58: `show(x, 'ACGU')` Translate the numbers into these letters.

Back in the command window, type `RNARo11` to run the program, then the up arrow to run it again and again. If there is an error, check to see that you have typed everything correctly. As before, you may wish to pull down File, Preferences, Command Window, and check the box labeled Wrap lines.

- 59: `show(x, 'A---')` For fun, display just the A entries. Can you get just the G's?

Modeling RNA

Many organisms have RNA that is similar to corresponding RNA from related organisms, but shows some unpredictable variations from organism to organism. It is important to understand these variations and to be able to measure the degree of variation. This is a difficult task, but we can begin with some simple probability concepts and Matlab programs.

- 60: `load 23SData` Load the sequence data from the 23S RNA. This loads a variable named `y`.
- 61: `show(y, 'ACGU')` Display the RNA sequence.
- 62: `FrequencyCount(y)` Run a program that calculates the frequencies with which each outcome occurs in `y`. Note that A, C, G, and U do not occur with equal frequencies.
- 63: `freq = [0.2600 0.2505 0.3065 0.1830]`; Modify the first line of `RNARo11.m` this way, then run it again.

- 64: `FrequencyCount(x)` Now `RNARoll.m` generates A, C, G, and U with roughly the observed frequencies. If you change `RNARoll.m` to generate a longer sequence (change 400 to 4000, say), the observed frequencies will agree better with `freq`.

Matrices in Matlab

- 65: `M=rand(6,6)` A 6 by 6 matrix of random numbers.
 66: `M(2,:)` Pull out the second row of the matrix.
 67: `M(:,4)` Pull out the fourth column of the matrix.
 68: `diag(M)` Pull out the diagonal entries of the matrix.
 69: `sum(M)` Sum the columns of the matrix.
 70: `sum(M')'` Sum the rows of the matrix. The apostrophe denotes **transpose**.

Markov chain for RNA sequences

In the program `RNARoll.m`, each letter in the sequence has no relation to or dependence on the letters before it, just like when you flip a coin or roll a die. The letters are said to be **independent**.

However, we should be able to set up a situation in which an A is usually followed by an A but never by a U. To do this, we could generate a sequence this way: After the letter A, we could use the frequency vector `[0.6 0.2 0.2 0]` to choose the next letter. After the letter C, we could use a **different** frequency vector, etc.

To do this, we'll create a new program called `RNAMarkov.m`. Open a new editor window and type the following commands. Note that the vector for the Markov chain is called `Pi`, as in the textbook.

- 71: `A=[0.6 0.2 0.2 0]; [0 0.6 0.2 0.2]; [0.2 0 0.6 0.2]; [0.2 0.2 0 0.6]];` A is a 4 by 4 matrix. Each row tells the probabilities we are going to use for the next letter. Row 1 tells what probabilities to use after the number 1 (corresponding to the letter A), row 2 tells what probabilities to use after the number 2 (for C), and so on.
 72: `Pi(1) = randi([1 1 1 1]/4);` Choose the first state randomly.
 73: `for i=1:399,`
 74: `Pi(i+1) = randi(A(Pi(i),:));` Pull out row `Pi(i)` of the matrix A and use these as the probabilities to generate the next state.
 75: `end`
 76: `show(Pi,'ACGU');`

Run the program. Do you ever see an A followed by a U? How many AA pairs do you see?

After you run this a few times, you might want to be able to generate longer strings of letters all at once. A good way to do that is to make `RNAMarkov.m` into a **Matlab function**, so that you can call it with a parameter (pass an argument to it). To do this, follow the next two steps:

- 77: Add the following line to the top of `RNAMarkov.m`:
`function [Pi] = RNAMarkov(n)` Then `RNAMarkov` is a function which needs a parameter `n` (the length of the sequence to generate) and returns a vector `Pi`.
 78: Change 399 to `n-1`.

For a function, the variables it introduces, like `Pi`, are internal to the function unless they are explicitly passed back when the function is done. Just like the functions `sin` and `rand`, the function `RNAMarkov` gives back a value, the vector `Pi`. When you go back to the command window to run it, type this:

- 79: `Pi=RNAMarkov(4000);` Semicolon to suppress displaying `Pi` itself.
 However, `RNAMarkov` uses `show` to display `Pi`. You might want to "comment out" that line by putting a `%` at the beginning of it:
 80: `%show(Pi,'ACGU');`

Counting actual transitions

The transition matrix A says that the letter C should be followed by G 60% of the time, but in any given sequence, the actual observed frequency will be slightly different from that. The

program `TransitionCount.m` will count up how many times each transition occurs and display the frequency of each transition. It also displays the frequency of each individual letter.

```
81: TransitionCount(Pi, 'ACGU');
```

Running `RNAMarkov` with a large argument (like 10000) will make the actual transition frequencies close to those in the matrix `A`.

You might want to add `TransitionCount(Pi, 'ACGU')` to the end of `RNAMarkov`.

Using RNA data to specify the parameters in a RNA model

`TransitionCount` is especially useful when you have a set of actual RNA data and want to make a Markov model for it. You should go through the data, count transitions, and make a transition matrix `A` based on that.

```
82: TransitionCount(y, 'ACGU')           Count transitions for the 23S data.
83: A = [[0.2755 0.2531 0.3133 0.1580]; [0.2261 0.2638 0.3101 0.2000]; [0.2879 0.2192
      0.2784 0.2145]; [0.2381 0.2817 0.3393 0.1409]];           Add this line below
the original definition of A in RNAMarkov.m (all on one very long line), then run the program
again and analyze the transition frequencies.
```

Now we are able to generate a random sequence `Pi` which matches the 23S sequence in two statistical features. The transition counts are just about right, and the frequencies of the letters in `Pi`, which you can check, are also just about right.

Of course, this is not the end of the story!

Hidden Markov models

Chapter 3 of *Biological Sequence Analysis* discusses how DNA appears to switch back and forth between CpG islands and regular DNA. In CpG islands, the base frequencies are different, as are the transition probabilities from letter to letter.

As a simple example of such a phenomenon, we will write a program that simulates a casino switching between a fair and loaded die. Because switching the dice is hard to do without the gamblers noticing, it isn't done that often; usually the fair die is used several times in a row, then the loaded die, then the fair die, etc. This can be modeled as a Markov chain having two states, `F` and `L`, for Fair and Loaded.

Type the following lines into a program called `HMMDice.m`. Or, save `RNAMarkov.m` as `HMMDice.m` and make changes.

```
84: A = [[.95 .05]; [.1 .9]];           F is followed by F 95% of the time.
85: Pi=[];                               Clear out any old value of Pi
86: Pi(1) = rando([1 1]/2);             Start in a randomly chosen state.
87: for i=1:74,
88:   Pi(i+1) = rando(A(Pi(i),:));
89: end
90: show(Pi, 'FL')
```

Run `HMMDice` to see how the die alternates slowly between fair and loaded.

Now the gamblers don't get to see whether the fair or loaded die is being used. Instead, on each gamble, whichever die is being used will be rolled, and all the gamblers get to see is what number was rolled. When the fair die is being used, outcomes 1, 2, 3, 4, 5, 6 are equally likely, but when the loaded die is being used, 6 occurs more often than the other outcomes. The matrix `E` will be used to store the probabilities of the various outcomes of rolling the two dice. The first row will be for the fair die, the second for the loaded die. Modify the program to read:

```
91: A = [[.95 .05]; [.1 .9]];
92: E = [[1 1 1 1 1 1]/6; [1 1 1 1 1 5]/10]; A loaded die lands on 6 with probability 0.5.
93: Pi=[];                               Clear out any old value of Pi
94: x=[];                               Clear out any old value of x
95: Pi(1) = rando([1 1]/2);
```

```

96: x(1) = rando(E(Pi(1),:)); Use row Pi(1) of E to get the probabilities when rolling the
    die.
97: for i=1:74,
98:   Pi(i+1) = rando(A(Pi(i),:));
99:   x(i+1) = rando(E(Pi(i+1),:));
100: end
101: show(Pi, 'FL')
102: show(x, '123456')

```

When you run the program, you will see a sequence of F's and L's representing which die was used and directly below these, the number which was rolled at each step.

The Viterbi algorithm

The Viterbi algorithm is described on page 55 of *Biological Sequence Analysis*. Given the matrices **A** and **E** and an observed sequence (the rolls of the die), it estimates what the hidden states were (fair or loaded).

`ViterbiDice.m` is a program that switches between fair and loaded dice, rolls the dice, then runs the Viterbi algorithm to guess whether a fair or loaded die was used, and marks the errors with `*`.

```

103: ViterbiDice(75,1)                                75 die rolls.
104: ViterbiDice(75,2)    75 die rolls, but at each step, the die is rolled twice. This makes it
    easier to tell whether the die was fair or loaded.
105: ViterbiDice(75,5)                                5 rolls at each step.

```

Stochastic context free grammars

RNA molecules have **long-range dependence** in their base sequences, because bases that are far away in the sequence are physically close together when the RNA folds. So, for instance, an A at one point in the sequence will be matched by a U at some other point far away in the sequence.

Markov models alone are unable to simulate this long-range dependence, but stochastic grammars can.

106: **SGGenerator** Run this program to see how a simple RNA loop with mostly canonical base pairs can be generated. Keep pressing a key until the program terminates.

A stochastic grammar is a way of writing and rewriting a sequence, making random changes at certain points, until no further changes are called for. At each step, certain elements of the sequence, called **terminals**, remain fixed forever after, but other elements, called **non-terminals**, can change from one step to the next. The nonterminals change into additional nonterminals or into terminals.

In `SGGenerator`, the letters a, c, g, u are the terminals, and the numbers 1 to 9 are the nonterminals. Nonterminals 1 to 4 simply become a, c, g, or u after the next rewrite. Nonterminals 5 to 8 become 91, 92, 93, or 94 after the next rewrite. But nonterminal 9 can become a wide range of things, such as 18, 27, 36, and 45. Taking 18 as an example, after the next rewrite, it will be a94, and after the next, the 4 will change to a u, so the transition

9 -> 18 -> a94 -> a(9)u

is a way of inserting an au pair into the sequence. Note that, at the last step, the 9 will have changed into something else, as a way to continue generating the sequence.

When the middle of the sequence finally is replaced with terminals, the rewriting process stops.

Note that it is possible to have the transition 9 -> 99, which will begin generating *two* loops simultaneously, making a three-way junction.