Snakes and Ladders: Markov Chain Monte Carlo Simulation of a System with Jumps

John Burkardt
Department of Scientific Computing
Florida State University

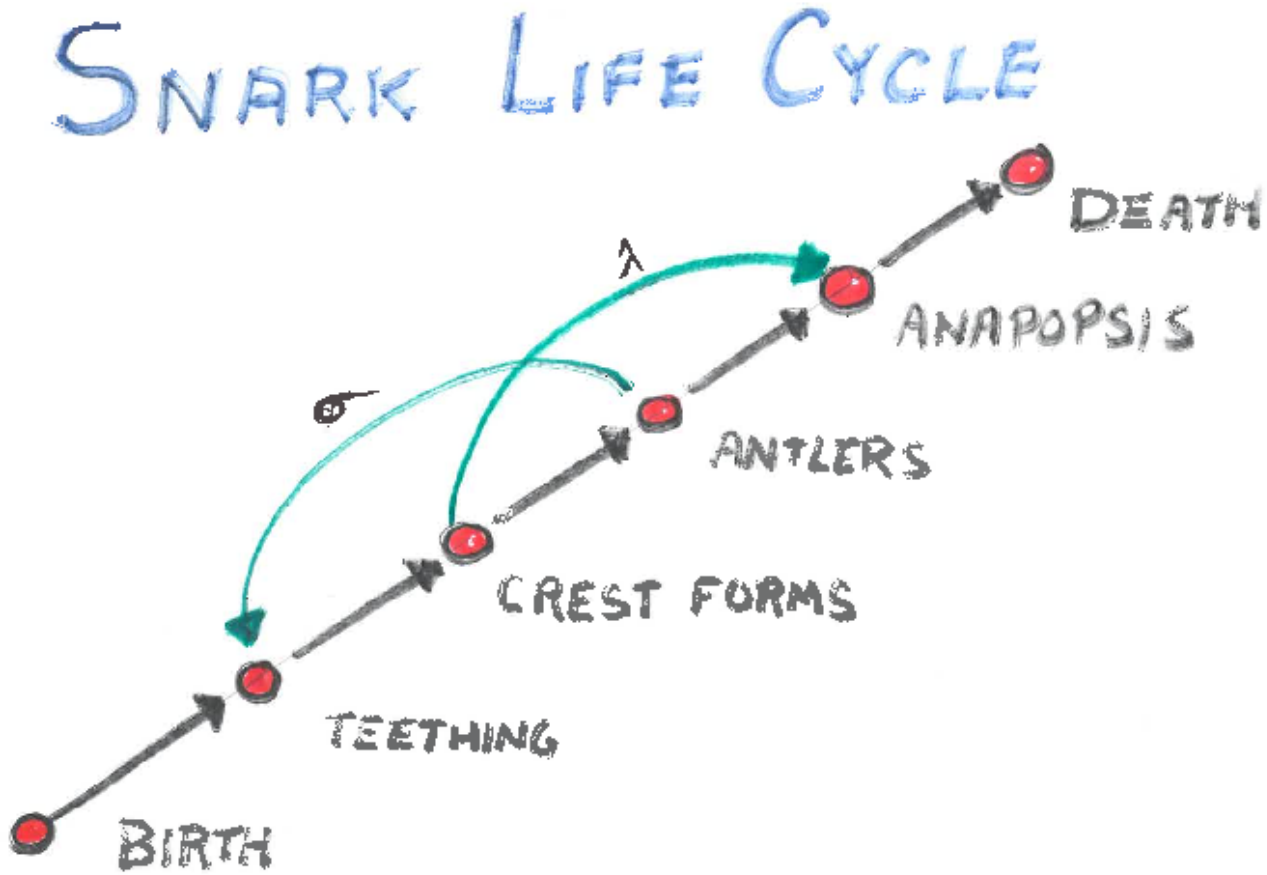
.....

11:00-11:50, 16 September 2014
Max Gunzburger's Group Meeting

.....

https://people.sc.fsu.edu/~jburkardt/presentations/...
snakes_2014_fsu.pdf

MCMC Simulation of Systems with σ and λ jumps



A σ Jump: We Need a 1000 Level Course!

Level 1000 courses are offered by university departments as a way of introducing themselves to freshmen who:

- have been vegetating in high school for four years;
- are eager to fill up their schedule with "cake" courses;
- may actually have some interest in the topic;
- don't have any real idea of what the department offers;
- are still looking for a major.

A σ Jump: Endless Constraints

Our department doesn't offer such a course; many undergraduates don't take courses in our department until junior year!

We are trying to develop a 1000 level course which

- is attractive to the "clueless" and "aimless";
- offers some real insight into scientific computing;
- makes scientific computing attractive;
- introduces computing without requiring programming;
- isn't so babified it demeans the students;
- isn't so trivial the instructor is ashamed of it;
- could be taught by graduate students;
- won't be vetoed by the Mathematics or CS departments.

A σ Jump: Brilliant Ideas

A few brainstorming ideas:

- Teach Matlab from Cleve Moler's "Experiments with Matlab" (JB) (vetoed);
- weekly discussion of computers in the news: privacy; breakins; BitCoin; MineCraft (NC);
- an overview of many computer languages, why they exist and what they do (GE);
- a demonstration of how computer programs solve problems, with GUI-based labs (JB) (no programming!);
- "Computers and Art", text analysis, color representation, surface modeling, motion capture (Mike Schneier).
- "Alice", a nonthreatening programming environment from CMU (JP).
- "Computing for Poets", a class at Cornell (JP)

Yet another σ Jump: A Demo

Today's class might not exactly fit under any of these headings.

However, I am trying to feel my way towards a presentation of a freshmanly interesting, computational sciencely-substantial topic, if you will pardon an unpardonable phrase.

You may not learn anything today; but ask yourself (and tell me!):

Would an uncommitted freshman find this presentation engaging?

How can a grade be given for a class with lectures like this?

What might be better?

How to play

SNAKES AND LADDERS

with getting bitten or dizzy!

Game: Snakes and Ladders

The game of Snakes and Ladders may have originated in India.

Game: The rules

Play is on a 10x10 board of squares numbered 1 through 100.

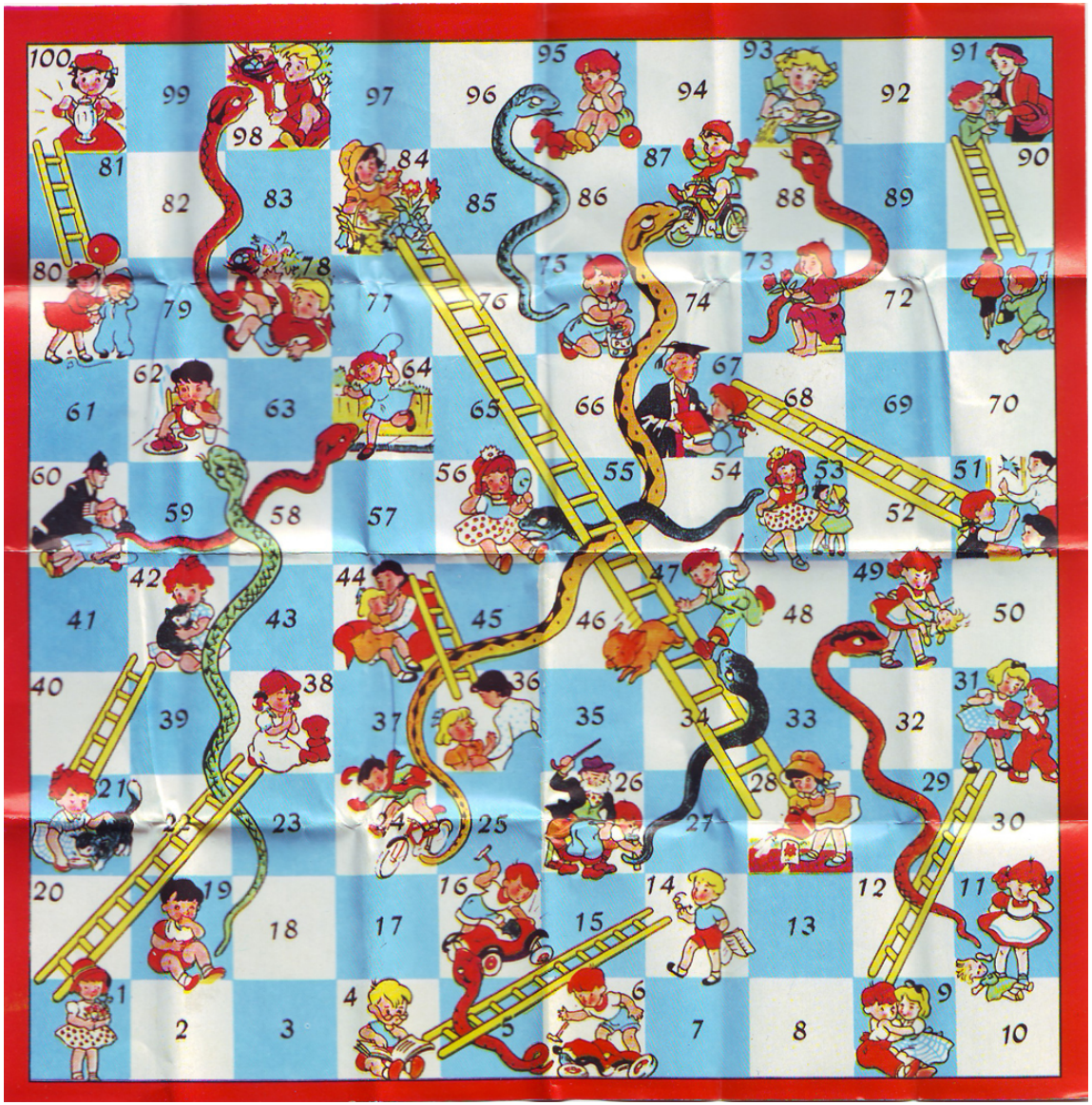
Several players may compete, alternating turns.

However, a single player can also play the game alone.

To move, a player moves ahead as indicated by the roll of a die, with the following special cases:

- players essentially begin on square "0";
- exactly landing on 100 wins the game;
- if the move exceeds 100, some rules allow a win, others forfeit the move;
- landing on a snake moves the player backward;
- landing on a ladder moves the player ahead;

Game: A typical game board



From Game to Model

The game can be fun (or frustrating) for children because of the sudden lucky changes in position. Although someone may seem likely to win, nothing is certain until a winning move has been made.

Let us assume that a computational scientist has noticed this game, and instead of wanting to play it, wants to understand it.

The same techniques used for this example are part of the methods applied to airline scheduling, Google searches, GPS map displays, DNA sequencing, and other problems that we have taught the computer to work on.

Model: a Finite State Machine

We can look at the game as a sort of mechanical process. Any single player starts out off the board, then moves to some square on the board, then another and another. The only thing the player needs to know is the current position.

In order to be consistent, we can say that, at the start of the game, the player is in square 0. That way, we can completely describe the progress of the game for one player by listing the position.

A computational science would call the positions "states"; because there is a limited list of them, the game itself could be described as a finite state machine.

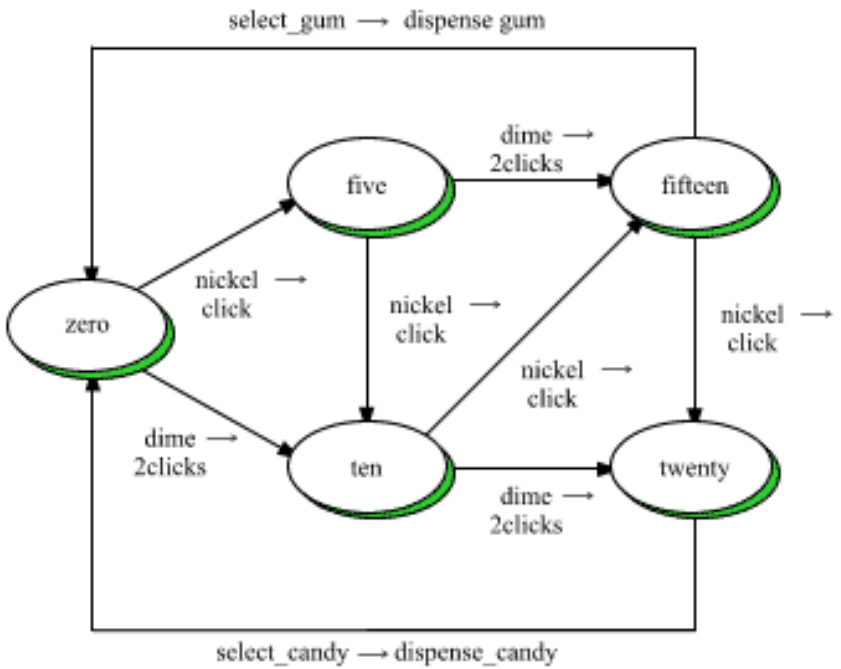
Finite State Machine: a system described by a finite number of states, with rules explaining when and how the system moves from one state to another.

Model: Examples of a Finite State Machine

A vending machine is also a finite state machine.

The states are the amount of money accepted. Changes to the current state occur if the user puts in a coin, or selects an item.

(A more realistic model could include current prices, rejected coins, and the user pushing the coin-return...)



Model: The States of Snakes and Ladders

Inspired by the vending machine diagram, we could think of making a similar transition diagram for the game of Snakes and Ladders.

It would consist of 101 circles, labeled "“0" through "100".

We need to draw arrows indicating what jumps or transitions are allowed from one circle to the next.

Since we are rolling a die, roughly speaking our rule should be:
At circle I, draw arrows to circles I+1, I+1, ..., I+6.
but this isn't quite right!

Transition: the change from one state to another.

Model: The Transitions of Snakes and Ladders

Problem #1:
Since we can't go past circle 100, we need to modify our rule:
At circle I, draw arrows to each circle I+1, I+1, ..., I+6 that is no greater than 100.

Problem #2:
Suppose we land on the foot of a ladder. Naturally, as part of the very same turn, we immediately move up the ladder. So if circle I is the foot of a ladder or the mouth of a snake, we never roll a die there, we just move.

1. *If circle I is a regular circle, draw arrows to each circle I+1, I+1, ..., I+6 that is no greater than 100.*
2. *If circle I is a ladder foot or snake mouth, draw a single arrow to the ladder top or snake bottom.*

Model: A 3x3 example

It might help to sketch a small version of our problem:

```
      7 -> 8 -> 9
      ^    |S|
      6 <- 5 <- 4
     |L|         ^
0 -> 1 -> 2 -> 3
```

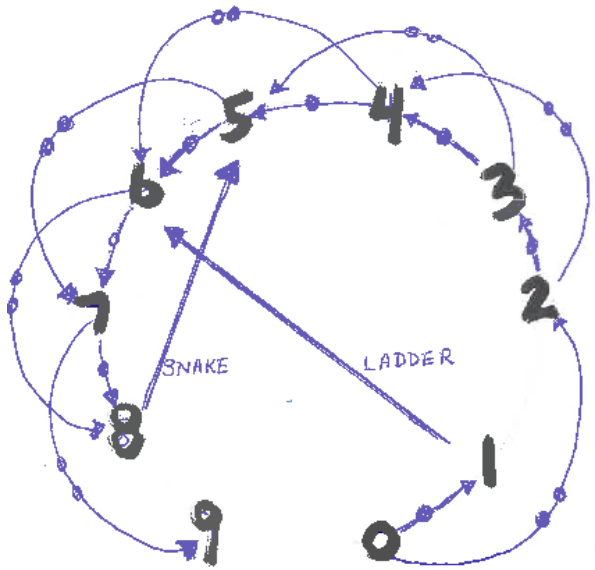
where we only have a 3x3 board, and one ladder, from square 1 to square 6, and one snake, from square 8 back to 5.

To keep the game from ending too quickly, let's suppose our die only returns a 1 or 2.

To record where the snakes and ladders are, we need:

```
           0 1 2 3 4 5 6 7 8 9       Ladder at 1 goes to 6,
connect   |0|6|2|3|4|5|6|7|5|9| <-- Snake at 8 goes to 5
```

Mode: Transition Diagram of a 3x3 example



Model: A 3x3 example

In our diagram, there are two very special states, #0 and #9.

State #0 is special because it only has arrows leading out of it. There is no way for the system to return to this state. It is an initial state.

State #9 is special because no arrows leave it. If you reach this state, you are stuck there. It is a final state.

Initial State: a state unreachable from any other state.
Final State: a state which never transitions to any other state.

A finite state machine can have any number of initial or final states, including zero.

From Model to Implementation

The picture of the board speaks to our eye. Looking at it, we can see immediately and easily what is connected to what, where the dangers are, who is playing and so on.

The finite state machine model speaks to our mind. By identifying the states and the transition rules, we know the small scale details. But it's a big mental effort to think through a whole game!

Computers don't work with picture or logical models. Instead, they can handle numbers, which can be stored in variables or lists.

If we can somehow translate our finite state model into commands to move numbers around, the computer can make a legal move in the game. It can just as easily take enough legal moves to win the game. And, if we want, we can ask it to play thousands of games.

Implementation: the translation of a logical model of a system into a series of commands to be carried out by a computer.

Implementation: The Current State

For any player, the only important information is the current state, that is, the player's position on the board.

This is a number between 0 (starting) and 100 (done).

The computer can remember this value if we give it a name.

We'll call this value **I**, and it will start out with the value 0.

```
I = 0  <-- Initialize I to zero, almost any language
```

Initialization statement: a statement in a computer language which indicates that a variable is to be start off with a given value.

Implementation: Rolling the Die

The player's move depends on the roll of a die, a random integer between 1 and 6.

The computer can create such a value on the fly, and we can store it with the name **D**.

```
D = random_integers ( 1, 6 ) <-- How to say this in Python
```

Python: a free popular powerful interactive computer language that is available for free on most computers.

Implementation: How Can a Computer Be Random?

It may seem surprising that a computer can roll dice.

The computer can indeed produce a stream of numbers that seem random, even though they are created by a sequence of logical steps.

In many cases, this process starts from a single, arbitrary, number, called the **seed**. The computer manages to shuffle this initial information by multiplying and adding large numbers, and only keeping the last few digits.

However, starting from the same seed would give us the exact same sequence of "random" values. The fact that these numbers, while looking random, are actually completely predictable, has resulted in the name **pseudorandom**.

Pseudorandom numbers: a technique that allows a computer to produce what seems like random integers or real numbers in some specified range.

Implementation: Making the Move

Having rolled the die, we should consider moving.

Although we might step on a snake or ladder, or go past 100, it's perfectly correct to begin by taking the full step.

Many computer languages use the equals sign as a way to update or change the value of a variable.

```
I = I + D  <-- The VARIABLE on the left is set to
               the VALUE of the formula on the right
```

Implementation: Changing a Value

In mathematics, a statement like

```
I = I + D
```

is an equality statement, saying the quantities on both sides are equal (and hence **D** must be 0!).

In computing, this is an assignment statement, and has an operational meaning:

```
go get the value stored in I;
go get the value stored in D;
compute I + D
put this new value back into I.
```

Assignment statement: a statement in a computer language which indicates that a variable is to be set to a value.

Implementation: Don't Go Too Far

One important rule is that we can't go past square 100.

It's easy to tell if we went to far - just check the value of **i**.

And if we went too far, we just have to pull back to 100.

Here is how the check and correction would be done in the Python language:

```
if ( 100 < I ):   <-- If this condition is true
  I = 100         <-- then move the piece back to 100.
```

Implementation: Snakes? Ladders?

What if our move takes us to a square that is the beginning of a snake or ladder?

In that case, we should change our location to the endpoint of the snake or ladder.

One way to do this is to make a list, for **every** position, of your final destination if you step on that square. For most squares, you will simply stay where you are. Doing it this way makes the check and correction automatic.

```
I = final [ I ]
```

Implementation: Defining the Snakes and Ladders

To implement the snakes and ladders, we start by making an **array** or list of the numbers 0 through 100:

```
final = range ( 0, 101 ) <-- 0, 1, 2, ..., 98, 99, 100.
```

and now, for each square that is a snake or ladder, we replace the square's index by the index of its final destination. The standard board has 19 snakes and ladders, but here is how to set the first five of them:

```
final[1] = 38
final[4] = 14
final[9] = 31
final[16] = 6
final[21] = 42
```

Array: a single variable name that stores a list of values. A value can be accessed by specifying its index on the list.

Implementation: Defining the Snakes and Ladders

```
final[ 1]=38 final[28]=__ final[56]=__ final[87]=__
final[ 4]=14 final[36]=__ final[62]=__ final[93]=__
final[ 9]=31 final[48]=__ final[64]=__ final[95]=__
final[16]= 6 final[49]=__ final[71]=__ final[98]=__
final[21]=42 final[51]=__ final[80]=__
```

From Implementation to Game

Now that we know how to make one move, we can play the whole game. Ordinarily, a computer will implement a sequence of commands in the order in which they are given. Thus, the following program takes the first move.

(statements that define final go here)

```
I = 0
D = random_integers ( 1, 6 )
I = I + D
if ( 100 < I ):
  I = 100
I = final [ I ]
```

Game: Taking More Moves

To take a second move, we could cut and paste another set of move commands. But repetition is an important part of computation. There is a **while** command that forces the computer to go back and repeat the commands until some condition is true.

```
(statements that define FINAL go here)
I = 0
while ( I < 100 ):    <-- Until reaching square 100
  D = random_integers ( 1, 6 )
  I = I + D
  if ( 100 < I ):
    I = 100
  I = final [ I ]
```

This is how it would be done in the Python language.

Game: Conditional Statements

Both the **if():** and **while():** statements are examples of conditional statements. Such statements indicate that a set of statements that follow are to be carried out only if a certain condition is true.

The **if():** statement allows us to reduce the value of **l** to 100, but only if we need to, because it exceeded 100.

The **while():** statement allows us to repeat the following statements as long as the game hasn't been won, that is, as long as **l** is less than 100.

Conditional statement: a statement in a computer language which indicates that other statements are to be carried out only if some condition is true.

Game: A Computer Simulation

Now we have implemented a sequence of computer statements that "do the same thing" as Snakes and Ladders. There is no board involved, no one rolls a die, but if we wanted to, we could keep track of the changes in the variable **l**, and show that they corresponded to a legal game.

What we have done, therefore, is simulated the game of Snakes and Ladders.

Computer simulation: the use of a computer program to represent,
analyze or predict the behavior of a real life system or situation.

Game: Questions We Can Investigate

Now that we have taught the computer how to play Snakes and Ladders, it is possible to think of questions we would like it to answer for us.

- what is the shortest possible game?
- can a game go on forever?
- what is the average length of a game?
- what square is visited most often?
- what ladder shortens the game the most?
- does adding another snake always lengthen the game?
- what happens if the board wraps around so square 100 is followed by square 1?
- in a game of two players, does the first player have a tiny advantage?
- what happens if the end of a snake is the mouth of another snake?
- what happens if a snake and a ladder "wrap around"?

Game: Questions We Can Investigate

Several of our questions ask about the number of steps (least, most, average) it takes to win a game.

To answer such a question, we need to keep track of the number of steps in one game, and we need to play many games.

This will lead us to the idea of writing our own **functions**.

Function: a collection of computer statements that carries out a particular computation. It may accept initial data as input, and it may return a result, called output.

Game: Counting the Steps

```
def game_length():  <-- begin a function
  ( statements that define FINAL )
  S = 0             <-- step starts at 0
  I = 0
  while ( I < 100 ):
    S = S + 1       <-- take another step
    D = random_integers ( 1, 6 )
    I = I + D
    if ( 100 < I ):
      I = 100
    I = final [ I ]

  return S          <-- value returned by the function
```

Game: Counting the Steps

Here are some important facts about our new function:

- the function plays a random game, and reports the number of steps:
- to use it with a statement like **n = game_length()**;
- the **def game_length():** statement begins the function;
- the **return S** statement terminates the function;
- the function serves as a package, which carries out the complicated task of playing the game, while making it easy to ask a simple question.

Why Simulate?

How can we estimate the average number of steps in Snakes and Ladders?

We have seen that the computer can count the number of moves it takes in order for one player to play the game to completion.

It's as easy for the computer to play the game thousands of times, and record the number of moves every time.

By dividing the total number of moves by the number of games, we get an estimate for the average number of moves.

Investigating a system with inherent randomness in this way is called the Monte Carlo method. It is popular for problems when there is no obvious mathematical way to compute an answer,

Monte Carlo method: repeatedy running a computer simulation which includes a random influence, in order to estimate the average outcome.

Game: Averaging Many Games

It's time to realize the advantage of having made a computer version of the game. If we wish to estimate the average number of moves by averaging over 1000 games:

- We don't have to actually play the board game 1000 times.
- We don't have to play the computer version 1000 times.
- We don't have to type the sequence of commands the define a game 1000 times.
- We can simply call **game_length()** 1000 times, keeping a running total and averaging at the end.

But even this is too much work. Can't we make the computer do everything for us?

Game: Average Number of Moves

```
games = 0
total = 0

while ( games < 1000 ):
  games = games + 1
  n = game_length ( )
  total = total + n

average = total / 1000
print 'Average number of moves is ' + repr ( average )
```

repr() makes a printable string from the value of **average**.

Output statement: a statement like **print** which causes text or the value of variables to be displayed on the screen.

Game: Average Number of Moves

We said that the **while():** command repeats the commands after it until the condition is true.

In this program, we only want the next three commands to be repeated, but the computation of the average should only be carried out when the repetition is done.

In the Python language, we indicate the commands to be repeated by indenting them. Because the indenting stops when we reach the command **average = total / 1000**, the **while():** command will stop the repetition just before that point.

Indentation: using horizontal spacing as a way to group statements. Other languages use parentheses, or special **end** statements.

Game: A Thousand Simulations, Ten Times

It seems like 1000 Monte Carlo simulations should be enough for good estimates. Comparing ten trials, the longest game data seems to vary. (Is there a longest possible game?)

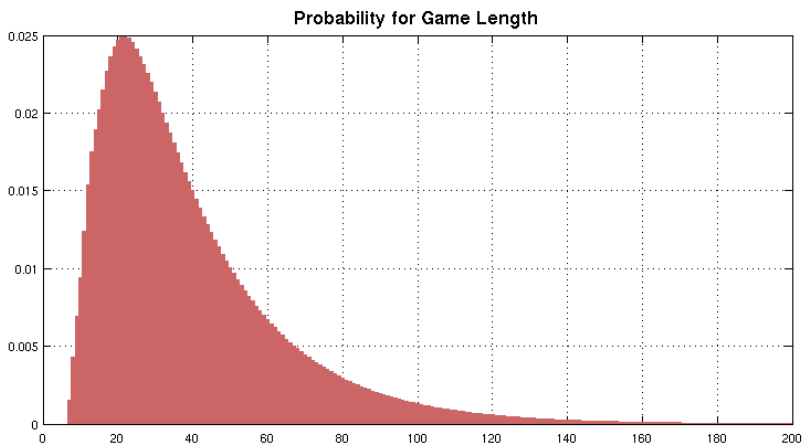
Trial	Average	Shortest	Longest
1	39.8	7	175
2	39.2	7	185
3	38.7	7	158
4	39.5	7	205
5	38.5	7	187
6	38.3	7	198
7	39.5	8	207
8	38.8	7	176
9	39.9	7	185
10	39.0	7	242

Long Tail: low probability events, hard to detect by simulation.

Game: The Probability Distribution Function

In fact, if we are more careful about keeping our records, we can estimate the probability that the game will take any number of moves.

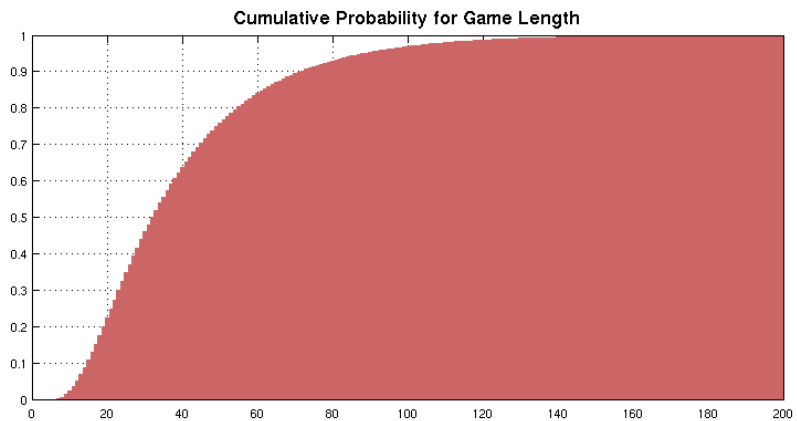
That is, if we play the game 10,000 times, we can ask for the probability that the game will take exactly 39 moves to win (which is about the average number.) If we observe a 39 move game 247 times in our sample, we estimate the probability at $\frac{247}{10{,}000} \approx 0.025 = 2.5\%$ chance.



Game: The Cumulative Distribution Function

Another useful piece of information is called the cumulative probability density function. If we want to know the probability that the game will be over in 25 moves or less, we simply add together the probabilities that it will be over in 1 move, 2 moves, ..., 25 moves.

This means that the result will be an ever increasing function which reaches 1 (certainty) at the end.



Conclusion: An Overview of the Investigation

Our investigation involved a number of steps:

We first needed to make a mental model of the game, that identified it as a system in which transitions were made from one square to certain other squares.

We were then able to write out the rules for one move, and then for the whole game.

We were able to ask the computer to report the game length.

Then we were able to make the computer play many games, and to compute some statistics about game length which answered our question.

Conclusion: Other Games are Different

For any system you study, the method of modeling and implementation will be different. Each new situation requires new thinking.

In Snakes and Ladders:

- no skill or choice is involved (unlike chess, bridge)
- players do not compete against each other (unlike Parcheesi)
- there are no prizes or properties (unlike Monopoly)
- all players see all information (unlike Stratego or Poker)
- the odds don't change over time (unlike for cardcounters in blackjack)

Conclusion (of ISC 1066)

We looked at the game of Snakes and Ladders because it was a simple example with clear rules and some questions that made sense to ask.

The point is, though, that the very same ways of looking and analyzing a problem are used when a computational scientist tries to make a model of

- the spread of disease;
- the difference between writings by Shakespeare or Francis Bacon;
- the spiral pattern of kernels on a corn cob;
- the common family tree of the many varieties of frogs;
- the explosion of stars;
- the properties of high-temperature superconductors;
- the underground transport of nitrogen from Tallahassee to Wakullah Springs;
- industrial methods for designing nano-molecules.

CLASS DISMISSED!

Please do not harm the teaching assistants.

Conclusion (of This Talk)

Some things this talk could **not** discuss:

- This is an example of a Markov Chain;
- The transition matrix A is sparse; nifty MATLAB commands can set it up;
- If the vector w represents the current state of the board, A'*w computes the probability that any square will be occupied on the next move;
- Powers of A indicate multiple moves;
- There is one eigenvalue of 1, and one corresponding eigenvector.
- There is an exact formula for the average number of moves
- Adding ladders doesn't always shorten the game!
- Making the game wrap around opens up a new set of questions.

Conclusion (of This Talk)

A talk like this has several aims:

- Fill up 50 minutes of time;
- Allow the instructor to feel that something useful is being imparted;
- Distract the students from their cellphones;
- Awaken, where possible, an interest in computing;
- Teach to the "middle level" of students, while offering something for all,
- Provide some "testable material" (the vocabulary words?)

We must judge how well this talk succeeeded!

References

Steve Althoen, Larry King, Kenneth Schilling, *How long is a game of Snakes and Ladders?*, The Mathematical Gazette, Volume 77, Number 478, March 1993, pages 71-76.

Nick Berry, A mathematical analysis of snakes and ladders, http://www.datagenetics.com/blog/november12011/index.html

Desmond Higham, Nicholas Higham, **MATLAB Guide**, SIAM, 2005, ISBN13: 9780898717891.

http://people.sc.fsu.edu/~jburkardt/... m_src/snakes_and_ladders/snakes_and_ladders.html or py_src/snakes_and_ladders/snakes_and_ladders.html

Remarks from the Audience

- "This is enough material for three lectures."
- "Cumulative density function...you've got to be kidding."
- "Without an overview of the lecture, students will get confused. For instance, the discussion of pseudorandom numbers seemed like a change of topic. It was not clear that this was part of the same discussion at first."
- "Audience participation through the use of clickers?"
- "You need more pictures."
- "You're aiming for the middle, but all your audience will fall below or above your talk."