Random Integers

http://people.sc.fsu.edu/∼jburkardt/isc/week06
lecture_12.pdf

....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 16 June 2011

Functions

- **Introduction**
- The rand() function
- Random Integers
- Rolling a Pair of Dice
- The Random Walk
- Duelling Idiots
- Assignment #5

INTRO: Schedule

Read:

- Today's class covers Sections 5.8 and 5.9

Next Class:

- Arrays

Assignment:

- Programming Assignment #4 is due today.
- Programming Assignment #5 will be due June 23.

Midterm Exam:

- Thursday, June 30th

INTRO: Overview

Today we will continue the study of the random number generator **rand()**, and its use for generating random integers in some range, such as 1 to 6, or 1 to 52, that is useful to us.

We will look at how a histogram or bar graph can be used to make sure that the random integers are well behaved.

We will consider some simple problems in which random integers can be used to simulate the role of chance.

Functions

- Introduction
- **The rand() Function**
- Random Integers
- Rolling a Pair of Dice
- The Random Walk
- Duelling Idiots
- Assignment #5

RAND: The Same Random Values Twice

In the previous class, we discussed how the **rand()** function, which returns random integers between 0 and **RAND_MAX**, can be used to create real numbers between 0.0 and 1.0.

Essentially, you can assume that the computer has a long list of random integers, and that each call to **rand()** returns the next one. The first time your program calls **rand()**, we get the first number on the list, then the second, and so on.

This means that if you run your program twice, you get the same sequence of random integers the second time.

If your program is setting up a game, you want different random numbers each time. To do that, you have to ask for help from the **srand()** function.

RAND: SRAND() Changes the Seed

The starting point of a random number sequence is determined by a number called the seed. We can imagine that the seed tells us what page to start from in the book of random numbers in the computer, and that the default value is page 1.

The function **void srand (unsigned int seed)** allows you, at any time, to "change the page", that is, to jump to a new part of the random number sequence. The input argument has a data type we haven't seen yet, called an **unsigned int**. It is enough right now for you to imagine that this just means we have to input a nonnegative integer.

RAND: Same Seed Means Same Random Numbers

Now, suppose I change my game program to call **srand (123)** at the beginning. The first time I run the program, I'm very happy, because now I see a different string of random values being generated.

The second time, I notice that the same new random numbers have appeared. This is very frustrating! Instead of seeing page 1 over and over, now I see page 123...!

If I really want to see a different stream of random numbers each time I run the program, one approach would be to ask the user to input a value for the seed:

```
cout << "Enter a nonnegative random number seed:";
cin >> seed;
srand ( seed );
```

RAND: TIME() Randomizes the Seed

Our problem is that we need the seed value itself to vary, and we'd prefer the computer to pick it for us.

Luckily, there's a function called **int time (** *something* **)** which returns the number of seconds since January 1st, 1970. The *something* that is an input argument to **time** is too complicated for our purposes, but luckily, we are allowed to use a 0 for it. In that case, **time(0)** will return a big integer that changes every second, and should be good enough to give us a different set of random numbers each time:

```
seed = time ( 0 );
srand ( seed );
```

Now call rand() and be surprised each time!

RAND: Dealing with Unfairness or Bias

We've talked about problems involving flipping a coin. We could simulate this with the raw integers from **rand()** by simply checking whether the integer is even (tails) or odd (heads).

But let's suppose we wanted to use **rand_double()**, which returns a random real value in the interval [0,1]. Then we could still simulate flipping a coin by taking values less than 0.5 to mean tails, and values greater than 0.5 to be heads.

The advantage to thinking about using **rand_double()** is that, if we wish to model a *biased* situation, in which tails comes up 60% of the time, we can take $0 \leq x \leq 0.6$ to correspond to tails, and $0.6 < x \leq 1$ to be heads.

With a little thought, you can also see to model a set of dice using a real random number, whether they are fair (each interval should be 1/6 wide then) or biased.

Functions

- Introduction
- The rand() function
- **Random Integers**
- Rolling a Pair of Dice
- The Random Walk
- Duelling Idiots
- Assignment #5

RANDOM: A Random Integer

It's easy to compute a random integer between 0 and **n-1**:

```
random_int = rand ( ) % n;
```

But if I want random integers between 1 and **n**, I must write:

```
random_int = 1 + rand ( ) % n;
```

For integers in a range starting at **a** and including **n** possible values:

```
random_int = a + rand ( ) % n;
```

Trick question: **What is the formula for a random integer between 10 and 20?**

RANDOM: A Random Int Function

Rather than having to remember that funny formula, we can just write a function called **random_int()** that will remember for us!

```
int random_int ( int a, int b )

// RANDOM_INT returns a random int between a and b.
{
  int range;
  int value;

  range = b - a + 1;              <-- one more number
                                      in range than we
  value = a + rand ( ) % range;      might guess!

  return value;
}
```

RANDOM: Testing RAND_INT

How do we test a random number function? Well, we can at least get some confidence in how it works. We might pretend we are modeling dice, and use the function to return a random integer between 1 and 6.

Doing this 6,000 times, we expect about 1,000 hits per value.

```
int count1 = 0, count2 = 0, count3 = 0;
int count4 = 0, count5 = 0, count6 = 0;
int i, value;
int random_int ( int a, int b ); <-- Declare user function

for ( i = 1; i <= 6000; i++ )
{
  value = random_int ( 1, 6 );   <-- Roll the die
  ...add result to counter, see next page...
}
 cout << "  1  " << count1 << "\n";
 cout << "  2  " << count2 << "\n";
 ...print 4 more lines of data...
```

random_int_test.cpp writes data to **random_int_count.txt**.

RANDOM: Increase Appropriate Counter

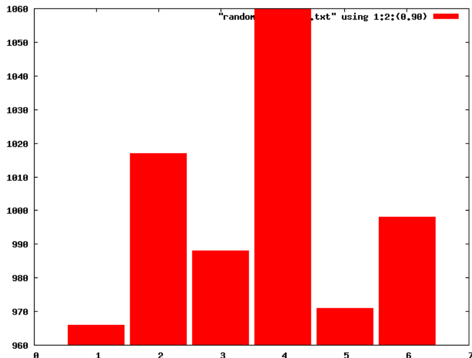
We can use a **switch** statement to update the counters.

```
switch ( value )
{
  case 1:
    count1 = count1 + 1;
    break;
  case 2:
    count2 = count2 + 1;
    break;
  case 3:
    count3 = count3 + 1;
    break;
  case 4:
    count4 = count4 + 1;
    break;
  case 5:
    count5 = count5 + 1;
    break;
  case 6:
    count6 = count6 + 1;
    break;
  default;  <-- Must have default, even if never used!
    break;
}
```

We could also use a sequence of **if/else if/else** statements.
And when we learn about arrays, this will be easier!

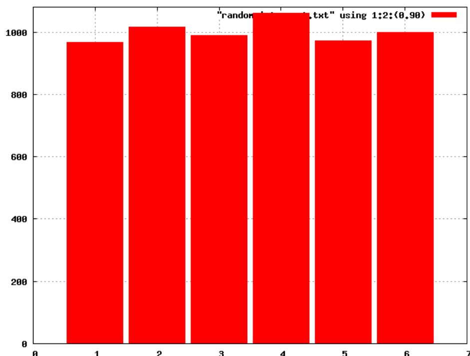
RANDOM: An Alarming Histogram!

```
gnuplot
  set style fill solid
  plot "random_int_count.txt" using 1:2:(0.90) with boxes
```



RANDOM: Show the whole picture!

```
gnuplot
  set style fill solid
  set yrange [0:1080]  <--Show the whole picture!
  plot "random_int_count.txt" using 1:2:(0.90) with boxes
```



- Introduction
- The rand() function
- Random Integers
- **Rolling a Pair of Dice**
- The Random Walk
- Duelling Idiots
- Assignment #5

DICE: Modeling Two Fair Dice

Unless someone has been cheating, we expect that each number of a die is roughly equally likely to show up. We know the number 7 is much easier to score than a 2 or a 12.

One way to check what is happening is to simulate this process on a computer. When two dice are rolled, we have to model each die as a separate chance event, and then consider the sum. The code would look something like this:

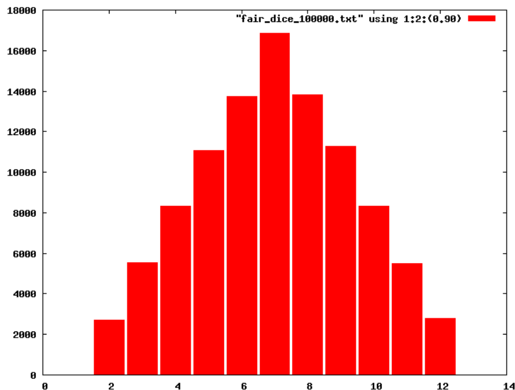
```
die1 = random_int_ab ( 1, 6 );
die2 = random_int_ab ( 1, 6 );
score = die1 + die2;
```

A program tracks the scores 2 through 12, simulates many roll
prints the results so that **gnuplot** can display them.

fair_dice_simulation.cpp

DICE: 100,000 Rolls of Two Dice

```
gnuplot
  set yrange [0:18000]
  set style fill solid
  plot "fair_dice_10000.txt" using 1:2:(0.90) with boxes
```



DICE: A Table Explains It

	1	2	3	4	5	6
1	1+1	1+2	1+3	1+4	1+5	1+6
2	2+1	2+2	2+3	2+4	2+5	2+6
3	3+1	3+2	3+3	3+4	3+5	3+6
4	4+1	4+2	4+3	4+4	4+5	4+6
5	5+1	5+2	5+3	5+4	5+5	5+6
6	6+1	6+2	6+3	6+4	6+5	6+6

probability of 7 $=$ 6 ways out of 36 $= \frac{6}{36} \approx 0.17$
probability of 2 $=$ 1 way out of 36 $= \frac{1}{36} \approx 0.03$

DICE: The Game of Craps

The book presents an example of the dice game called “craps”.

A player’s turn begins with one roll of the dice. The player wins immediately with 7 or 11, and loses immediately with 2, 3 or 12. Otherwise, the score rolled by the player is called the point, and the player continues.

Thereafter, the player continues to roll until getting the point again for a win, or a 7, for a loss.

Look at the program **craps.cpp**

DICE: Sample Games

Here are some sample games:

```
craps
  1:  1 + 1 = 2
  The house wins.

craps
  1:  3 + 2 = 5
  The player must roll until making 5 again.
  2:  3 + 6 = 9
  3:  1 + 2 = 3
  4:  4 + 2 = 6
  5:  3 + 5 = 8
  6:  1 + 5 = 6
  7:  2 + 1 = 3
  8:  4 + 1 = 5
  The player wins.

craps
  1:  2 + 4 = 6
  The player must roll until making 6 again.
  2:  6 + 2 = 8
  3:  6 + 5 = 11
  4:  3 + 4 = 7
  The house wins.
```

DICE: Program Outline

The skeleton of the program looks like this:

```
seed = time ( 0 );
srand ( seed );
score = roll_two_dice ( );  <-- Call a user function;

switch ( score )            <-- First roll is special.
{
  case 7:
  case 11:
    result = +1;
    break;
  case 2:
  case 3:
  case 12:
    result = -1;
    break;
  default:
    result = 0;
    point = score;
    break;
}
while ( result == 0 )        <-- Keep rolling?
{
  score = roll_two_dice ( );
  if ( score == point )
  {
    result = +1;
  }
  else if ( score == 7 )
  {
    result = -1;
  }
}
```

DICE: Program Outline

The function **roll_two_dice ( )** looks like:

```
int roll_two_dice ( )              <-- No input to function.
//
//  ROLL_TWO_TWICE simulates the roll of two dice,
//  prints the values and returns the sum.
//
{
  int die1;
  int die2;
  int random_int ( int a, int b );  <-- Declare helper function;
  int score;

  die1 = random_int ( 1, 6 );
  die2 = random_int ( 1, 6 );

  score = die1 + die2;

  cout << "  " << die1 << " + " << die2 << " = " << score;

  return score;
}
```

The main program includes the declaration of **roll_two_dice (**
roll_two_dice ( ) includes the declaration of **random_int()**.
And the text of **random_int()** must be included in the file.

DICE: Project Ideas

If you are playing craps, and get a point on the first roll, some points are better than others. Can you estimate your chances of winning, once you know the point you have to make?

Is it possible to model a situation in which the dice are unfair, that is, that some numbers are more likely than others? Of course, but then we have to come up with a replacement for **random_int()**, which simply picks each possibility with equal likelihood.

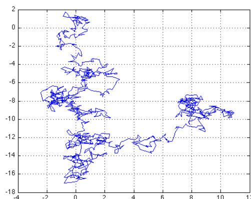
In the game of Yahtzee, you have five dice. You roll them all once, and can roll some of them again two more times. You are trying to match certain patterns, such as 4 two's, or a small straight (2-3-4-5) or a full house (3-3-5-5-5). Some patterns are worth more, and you only get credit for matching each pattern once. How well can a simple program score?

These are more examples of possible projects for you to pick to work on in July.

Functions

- Introduction
- The rand() function
- Random Integers
- Rolling a Pair of Dice
- **The Random Walk**
- Duelling Idiots
- Assignment #5

WALK: A Physical Observation



In 1827, Scottish botanist Robert Brown was studying pollen grains which he had mixed in water. Although the water was absolutely still, the pollen grains seemed to quiver and move about randomly. He could not stop the motion, or explain it. He carefully described his observations in a paper.

In 1905, the same year that he published his paper on special relativity, Albert Einstein wrote a paper explaining the motion. Each pollen grain, he said, was constantly being jostled by the motions of the water molecules on all sides of it. Random imbalances in these forces would cause the pollen grains to twitch and shift.

WALK: A Mathematical Model

Since then, many physical processes have been observed which can be modeled by a similar idea, that a large scale motion is occurring because of the effects of many small influences. Physicists found this model so important that they made a simplified mathematical version called the **random walk**.

In the one dimensional version, a particle starts at the origin, and then is free to take a step left or right, but chooses its direction at random. Many steps follow, each taken at random. Surprisingly, after many steps have been taken, the particle often has not gotten very far.

WALK: 100 Drunken Sailors

We'll introduce the random walk with a story about a captain whose ship was carrying a load of rum. The ship was tied up at the dock, the captain was asleep, and the 100 sailors of the crew broke into the rum, got drunk, and staggered out onto the dock.

The dock was 100 yards long, and the ship was at the 50 yard mark. The sailors were so drunk that each step they took was in a random direction, to the left or right. They were only able to manage one step a minute. Two hours later, the captain woke up.

"Oh no!" he said, "There's only 50 steps to the left or right and they fall into the sea! And two hours makes 120 steps!"

But he was surprised to see that 70 of the crew were actually within 12 steps to the left or right, and that **all** of the crew was alive and safe, though in sorry shape.

WALK: 100 Drunken Sailors

1D Random Walkers - 100 walkers, 120 steps

Number of walkers

X

WALK: N Steps Moves Us *sqrt*(N) Units

What explains the huge disparity between how far the sailors could have gotten, and how far they did get? Taking **N** random steps is like adding up random +1's and -1's. While it's possible for all the steps to be in the same direction, it's likely that most of them will cancel.

In fact, roughly speaking, taking **N** random steps is likely to move us about $\sqrt{N}$ units from where we start.

Thus, for the sailors to be in serious danger, they'd have to take about N = 50 * 50 = 2,500 steps, just to have a strong chance of ending up about 50 feet away (and hence falling off the pier!)

WALK: A Random Walk Program

random_walk_1d.cpp: A program for modeling a 1D random walk:

beginning stuff

```
int main ( )
{
  int i, n = 1000, pos, seed;
  int random_int ( int a, int b );
  seed = time ( 0 );
  srand ( seed );

  pos = 0;
  for ( i = 1; i <= n; i++ )
  {
    pos = pos + random_int ( -1, +1 );
    if ( pos < -50 )
    {
      cout << "Sailor fell off at -50.\n";
    }
    else if ( 50 < pos )
    {
      cout << "Sailor fell off at +50.\n";
    }
  }
  cout << "Sailor is safe at " << pos << " after " << n << " steps.\n";
  return 0;
}
```

Text of **random¨int()** *needed here!*

WALK: Walks in 2D

A random walk in 2D can be set up in the same way, except for some minor changes.

Our position is now an (x, y) value.

Our choices should include (-1,0,+1) for x, and for y.

We might be interested in the time that it takes for a particle, starting at (0,0), to reach the boundary of a box when the walls are 20 units away.

WALK: A Random Walk in 2D

"fred.txt" using 2:3

20
15
10
5
0
-5
-10
-15
-20

-20 -15 -10 -5 0 5 10 15 20

WALK: Walks in 2D

The program **random_walk_2d.cpp** puts a fly and a spider in a square box. The fly starts at the origin (0,0), and hopes to reach the safety of one of the walls, which are 20 units away. The spider starts at a random point.

The fly makes a random move, then the spider.

The fly wins if it steps on the spider, or escapes if it reaches the wall.

The spider wins by stepping on the fly.

WALK: A Random Walk in 2D

The distance between the spider and the fly is as low as one step. In this example, the fly escapes.

"fred3.txt" using 1:6

25
20
15
10
5
0

0 20 40 60 80 100 120 140 160 180 200

Functions

- Introduction
- The rand() function
- Random Integers
- Rolling a Pair of Dice
- The Random Walk
- **Duelling Idiots**
- Assignment #5

DUEL: A Fight to the Death

The dueling idiots problem assumes that players A and B have agreed to a duel, in which A will fire at B, then B at A, and so on, until one of them has been hit.

We happen to know the accuracy, that is, the probability that, on a given shot, each player will hit the other. Let's start by assuming that this is 50% for each player.

It seems obvious that A has an advantage by going first, but is it a huge or small advantage? One way to try to see this is to simulate the duel.

DUEL: A Program

duel_once.cpp: A program for modeling a duel:

beginning stuff

```
int main ( )
{
  int seed, shot, winner;

  seed = time ( 0 );
  srand ( seed );

  winner = 0;
  shot = 0;

  while ( winner == 0 )
  {
    shot = shot + 1;
    if ( ( rand ( ) % 2 ) == 0 )
    {
      winner = 1;
      break;
    }
    shot = shot + 1;
    if ( ( rand ( ) % 2 ) == 0 )
    {
      winner = 2;
      break;
    }
  }
  cout << "Duel was won by " << winner << " after " << shot << " shots.\n";
  return 0;
}
```

DUEL: Estimating Probabilities

To try to estimate the probability of survival, we would want to run this program many times, record who won. The program **duel_many.cpp** does that, essentially by using **duel_once** as a function which returns the index of the winner.

In this simple case, a mathematical argument can tell you that the exact answer is that A has a $\frac{2}{3}$ chance of winning.

If the two players have different accuracies, then the problem becomes harder. It's possible that a less accurate player might be more likely to survive, if only given the chance to go first.

DUEL: A Three Way Fight

Now suppose three people, A, B and C, all want to have a duel, and that A is the best player, B second best, and C the worst.

On A's turn, who should be the target? Presumably B.

When it is B's turn, who should be the target? Presumably A.

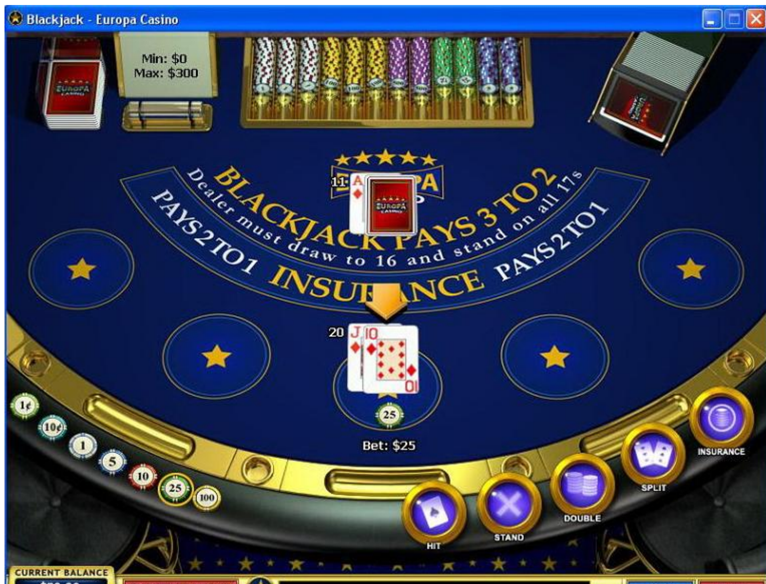
When it's C's turn, who should be the target? If both A and B are still alive, would it make sense for C to simply fire in the air? That way, A and B will keep shooting at each other...and if A hits B or B hits A, then C gets the first shot in a two-way duel with the survivor.

This is another example of a topic that can be a final project. Please let me know if this is what you want to work on.

Functions

- Introduction
- The rand() function
- Random Integers
- Rolling a Pair of Dice
- The Random Walk
- Duelling Idiots
- **Assignment #5**

ASSIGNMENT #5: Blackjack



ASSIGNMENT #5: Rules of Blackjack

In the game of Blackjack, each player tries to get a perfect score of 21, or as near as possible without going over.

The score is based on the player's cards, with aces counting 1 or 11, face cards 10, and other cards having their normal value.

Each player starts with two cards, and may ask for more. A common strategy is to keep requesting another card until a score of 17 or more is reached.

Of course, the new card may create a score greater than 21, which causes an instant loss.

We would like to know, for a player using this strategy:

- the probability of going over 21;
- the probability, for a given starting score, of going over 21;

ASSIGNMENT #5: Simplifications

1. Although the game normally involves many players, we only consider one player.
2. Although aces normally can be used as 1 or 11, whichever is convenient, we will always count them as 1's. This is to keep the program simple.
3. Because the dealer uses a very large deck, we will assume that the probability of drawing a particular card doesn't change over time.
4. Because the suits of the cards don't matter, we will assume that we are simply drawing a numbered card, between 1 and 13, but that cards 11, 12 and 13 all count for 10 points.

ASSIGNMENT #5: Estimating The Change of a Bust

To estimate the probability of going over 21:

Start by picking two cards, and computing the score. As long as the score is less than 17, pick another card. The only reasons to stop are if the new card takes you over 21, (you lose), or it takes you to between 17 and 21, (we'll say you win).

Try this for **n**=10,000 times, count **m**, the number of times you go over 21, and estimate the probability as $\frac{m}{n}$.

ASSIGNMENT #5: The Effect of the Starting Score

Some starting scores are better than other ones. 16 is a very dangerous score, for instance!

To estimate the probability of a bust, given a particular starting score of 16 or less:

- For each starting score from 2 to 16:
  - play 10,000 games of blackjack, count the times you bust;
  - print the starting score, and the probability of a bust.

ASSIGNMENT #5: Things to Turn in

You may want to write two separate programs, one to do the overall calculation, and the second to analyze what happens for each starting score.

Email to Detelina:

- your program's results;
- a copy of your programs.

The program and output are due by Thursday, June 23.