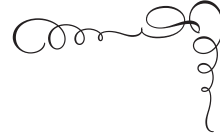
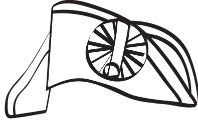
COPYRIGHT NOTICE:

Paul J. Nahin: Digital Dice

is published by Princeton University Press and copyrighted, © 2008, by Princeton University Press. All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher, except for reading and browsing via the World Wide Web. Users are not permitted to mount this file on any network servers.

Follow links for Class Use and other Permissions. For more information send email to: permissions@pupress.princeton.edu





Introduction

Three times he dropped a shot so close to the boat that the men at the oars must have been wet by the splashes—each shot deserved to be a hit, he knew, but the incalculable residuum of variables in powder and ball and gun made it a matter of chance just where the ball fell in a circle of fifty yards radius, however well aimed.
—from C. S. Forester's 1939 novel *Flying Colours* (p. 227),
Part III of *Captain Horatio Hornblower*, the tale of a military man who understands probability

This book is directed to three distinct audiences that may also enjoy some overlap: teachers of either probability or computer science looking for supplementary material for use in their classes, students in those classes looking for additional study examples, and aficionados of recreational mathematics looking for what I hope are entertaining and educational discussions of intriguing probability problems from "real life." In my first book of probability problems, *Duelling Idiots and Other Probability Puzzlers* (Princeton University Press, 2000), the problems were mostly of a whimsical nature. Not always, but nearly so. In this book, the first test a problem had to pass to be included was to be practical, i.e., to be from some aspect of "everyday real life."

This is a subjective determination, of course, and I can only hope I have been reasonably successful on that score.

From a historical viewpoint, the nature of this book follows in a long line of precedents. The very first applications of probability were mid-seventeenth-century analyses of games of chance using cards and dice, and what could be more everyday life than that? Then came applications in actuarial problems (e.g., calculation of the value of an annuity), and additional serious "practical" applications of probabilistic reasoning, of a judicial nature, can be found in the three-century-old doctoral dissertation ("The Use of the Art of Conjecturing in Law") that Nikolaus Bernoulli (1687–1759) submitted to the law faculty of the University of Basel, Switzerland, in 1709.

This book emphasizes, more than does *Duelling Idiots*, an extremely important issue that arises in most of the problems here, that of *algorithm development*—that is, the task of determining, from a possibly vague word statement of a problem, just what it is that we are going to calculate. This is nontrivial! But not all is changed in this book, as the philosophical theme remains that of *Duelling Idiots*:

1. No matter how smart you are, there will always be probabilistic problems that are too hard for you to solve analytically.
2. Despite (1), if you know a good scientific programming language that incorporates a random number generator (and if it is good it will), you may still be able to get numerical answers to those "too hard" problems.

The problems in this book, and my discussions of them, elaborate on this two-step theme, in that most of them are "solved" with a so-called *Monte Carlo simulation*.[1] (To maximize the challenge of the book, I've placed all of the solutions in the second half—look there only if you're stumped or to check your answers!) If a theoretical solution does happen to be available, I've then either shown it as well—if it is short and easy—or provided citations to the literature so that you can find a derivation yourself. In either case, the theoretical solution can then be used to validate the simulation. And, of course, that approach can be turned on its head, with the simulation results being used to numerically check a theoretical expression for special cases.

In this introductory section I'll give you examples of both uses of a Monte Carlo simulation.

But first, a few words about probability theory and computer programming. How much of each do you need to know? Well, more than you knew when you were born—there is no royal road to the material in this book! This is not a book on probability *theory*, and so I use the common language of probability without hesitation, expecting you to be either already familiar with it or willing to educate yourself if you're not. That is, you'll find that words like *expectation*, *random walk*, *binomial coefficient*, *variance*, *distribution function*, and *stochastic processes* are used with, at most, little explanation (but see the glossary at the end of the book) beyond that inherent in a particular problem statement. Here's an amusing little story that should provide you with a simple illustration of the level of sophistication I am assuming on your part. It appeared a few years ago in *The College Mathematics Journal* as an anecdote from a former math instructor at the U.S. Naval Academy in Annapolis:

> It seems that the Navy had a new surface-to-air missile that could shoot down an attacking aircraft with probability 1/3. Some top Navy officer then claimed that shooting off three such missiles at an attacking aircraft [presumably with the usual assumptions of independence] would surely destroy the attacker. [The instructor] asked his mathematics students to critique this officer's reasoning. One midshipman whipped out his calculator and declared "Let's see. The probability that the first missile does the job is 0.3333, same for the second and same again for the third. Adding these together, we get 0.9999, so the officer is wrong; there is still a small chance that the attacking aircraft survives unscathed." Just think [noted the instructor], that student might himself be a top U.S. navy officer [today], defending North America from attack.

If you find this tale funny because the student's analysis is so wrong as to be laughable—even though his conclusion was actually correct, in that the top Navy officer was, indeed, wrong—and if you know how to do the correct analysis,[2] then you are good to go for reading this book. (If you think about the math abilities of the politicians who make

decisions about the viability of so-called anti-ballistic missile shields and the equally absent lack of analytic talent in some of the people who advise them, perhaps you will find this tale not at all funny but rather positively scary.[3])

The same level of expectation goes for the Monte Carlo codes presented in this book. I used MATLAB 7.3 when creating my programs, but I limited myself to using only simple variable assignment statements, the wonderful `rand` (which produces numbers uniformly distributed from 0 to 1), and the common `if/else`, `for`, and `while` control statements found in just about all popular scientific programming languages. My codes should therefore be easy to translate directly into your favorite language. For the most part I have avoided using MATLAB's powerful vector/matrix structure (even though that would greatly reduce simulation times) because such structure is not found in all other popular languages. MATLAB is an incredibly rich language, with a command for almost anything you might imagine. For example, in Problem 3 the technical problem of sorting a list of numbers comes up. MATLAB has a built-in sort command (called—is this a surprise?—`sort`), but I've elected to actually code a sorting algorithm for the Monte Carlo solution. I've done this because being able to write `sort` in a program is not equivalent to knowing how to code a sort. (However, when faced again in Problem 17 with doing a sort I *did* use `sort`—okay, I'm not always consistent!) In those rare cases where I do use some feature of MATLAB that I'm not sure will be clear by inspection, I have included some explanatory words.

Now, the most direct way to illustrate the philosophy of this book is to give you some examples. First, however, I should admit that I make no claim to having written the best, tightest, most incredibly elegant code that one could possibly imagine. In this book we are more interested in problem solving than we are in optimal MATLAB coding. I am about 99.99% sure that every code in this book works properly, but you may well be able to create even better, more efficient codes (one reviewer, a sophisticated programmer, called my codes "low level"—precisely my goal!). If so, well then, good for you! Okay, here we go.

For my first example, consider the following problem from Marilyn vos Savant's "Ask Marilyn" column in the Sunday newspaper

supplement *Parade Magazine* (July 25, 2004):

> A clueless student faced a pop quiz: a list of the 24 Presidents of the 19th century and another list of their terms in office, but scrambled. The object was to match the President with the term. He had to guess every time. On average, how many did he guess correctly?

To this vos Savant added the words,

> Imagine that this scenario occurs 1000 times, readers. On average, how many matches (of the 24 possible) would a student guess correctly? Be sure to guess before looking at the answer below!

The "answer below" was simply "Only one!" Now that is indeed surprising—it's correct, too—but to just say that and nothing else certainly leaves the impression that it is all black magic rather than the result of logical mathematics.

This problem is actually an ancient one that can be traced back to the 1708 book *Essay d'analyse sur les jeux de hazard* (*Analyses of Games of Chance*), by the French probabilist Pierre Rémond de Montmort (1678–1719). In his book Montmort imagined drawing, one at a time, well-shuffled cards numbered 1 through 13, counting aloud at each draw: "1, 2, 3, ..." He then asked for the probability that no card would be drawn with a coincidence of its number and the number being announced. He didn't provide the answer in his book, and it wasn't until two years later, in a letter, that Montmort first gave the solution. Montmort's problem had a profound influence on the development of probability theory, and it attracted the attention of such illuminances as Johann Bernoulli (1667–1748), who was Nikolaus's uncle, and Leonhard Euler (1707–1783), who was Johann's student at Basel.

Vos Savant's test-guessing version of Montmort's problem is not well-defined. There are, in fact, at least three different methods the student could use to guess. What I suspect vos Savant was assuming is that the student would assign the terms in a one-to-one correspondence to the presidents. But that's not the only way to guess. For example, a student might reason as follows: If I follow vos Savant's approach, it is possible that I could get every single assignment wrong.[4] But if I select

one of the terms (*any* one of the terms) at random, and assign that same term over and over to each of the twenty-four presidents, then I'm sure to get one right (and all the others wrong, of course). Or how about this method: for each guess the student just randomly assigns a term from all twenty-four possible terms to each of the twenty four presidents. That way, of course, some terms may never be assigned, and others may be assigned more than once. But guess what—the average number of correct matches with this method is still one. Now that's *really* surprising! And finally, there's yet one more astonishing feature to this problem, but I'll save it for later.

Suppose now that we have no idea how to attack this problem analytically. Well, not to worry, a Monte Carlo simulation will save the day for us. The idea behind such a simulation is simple enough. Instead of imagining a thousand students taking a test, let's imagine a million do, and for each student we simulate a random assignment of terms to the presidents by generating a random permutation of the integers 1 through 24. That is, the vector term will be such that term(j), $1 \leq j \leq 24$, will be an integer from 1 to 24, with each integer appearing exactly once as an element in term: term(j) will be the term assigned to president *j*. The correct term for president *j* is term *j*, and so if term(j) = j, then the student has guessed a correct pairing. With all that said, the code of guess.m should be clear (in MATLAB, codes are called m-files and a program name extension is always .m).

Lines 01 and 02 initialize the variables M (the length of the two lists being paired) and totalcorrect (the total number of correct pairings achieved after a million students have each taken the test). (The line numbers are included so that I can refer you to a specific line in the program; when actually typing a line of MATLAB code one does not include a line number as was done, for example, in BASIC. The semicolons at the end of lines 01 and 02 are included to suppress distracting screen printing of the variable values. When we do want to see the result of a MATLAB calculation, we simply omit the terminating semicolon.) Lines 03 and 12 define a for/end loop that cycles the code through a million tests, with lines 04 through 11 simulating an individual test. At the start of a test, line 04 initializes the variable correct—the number of correct pairings achieved on that test—to zero. Line 05 uses the built-in MATLAB command randperm(M)

to generate a random permutation of the integers 1 through M (= 24), and lines 06 and 10 define a for/end loop that tests to see if the condition for one (or more) matches has been satisfied. At the completion of that loop, the value of correct is the number of correct pairings for that test. Line 11 updates the value of totalcorrect, and then another test is simulated. After the final, one-millionth test, line 13 computes the average number of correct pairings.

guess.m

```
01    M = 24;
02    totalcorrect = 0;
03    for k = 1:1000000
04        correct = 0;
05        term = randperm(M);
06        for j = 1:M
07            if term(j) == j
08                correct = correct + 1;
09            end
10        end
11        totalcorrect = totalcorrect + correct;
12    end
13    totalcorrect/1000000
```

When I ran guess.m, the code gave the result 0.999804, which is pretty close to the exact value of 1. But what is really surprising is that this result is independent of the particular value of M; that is, there is nothing special about the $M = 24$ case! For example, when I ran guess.m for $M = 5$, 10, and 43 (by simply changing line 01 in the obvious way), the average number of correct pairings was 0.999734, 1.002005, and 0.998922, respectively. It's too bad that vos Savant said nothing about this. Now, just for fun (and to check your understanding of the Monte Carlo idea), write a simulation that supports the claim I made earlier, that if the student simply selects at random, for each president, a term from the complete list of twenty four terms, then the

average number of correct pairings is still one. You'll find a solution in Appendix 1.

Let me continue with a few more examples. Because I want to save the ones dealing with everyday concerns for the main body of this book, the ones I'll show you next are just slightly more abstract. And, I hope, these examples will illustrate how Monte Carlo simulations can contribute in doing "serious" work, too. Consider first, then, the following problem, originally posed as a challenge question in a 1955 issue of *The American Mathematical Monthly*. If a triangle is drawn "at random" inside an arbitrary rectangle, what is the probability that the triangle is obtuse? This is, admittedly, hardly a question from real life, but it will do here to illustrate my positions on probability theory, Monte Carlo simulation, and programming—and it is an interesting, if somewhat abstract, problem in what is called *geometric probability* (problems in which probabilities are associated with the lengths, areas, and volumes of various shapes; the classic example is the well-known Buffon needle problem, found in virtually all modern textbooks[5]). The 1955 problem is easy to understand but not so easy to analyze theoretically; it wasn't solved until 1970. To begin, we first need to elaborate just a bit on what the words "at random" and "arbitrary rectangle" mean.

Suppose we draw our rectangle such that one of the shorter sides lies on the positive x-axis, i.e., $0 \leq x \leq X$, while one of the longer sides lies on the positive y-axis, i.e., $0 \leq y \leq Y$. That is, we have a rectangle with dimensions X by Y. It should be intuitively clear that, whatever the answer to our problem is, it is what mathematicians call *scale invariant*, which means that if we scale both X and Y up (or down) by the same factor, the answer will not change. Thus, we lose no generality by simply taking the actual value of X and scaling it up (or down) to 1, and then scaling Y by the same factor. Let's say that when we scale Y this way we arrive at L; i.e., our new, scaled rectangle is 1 by L. Since we started by assuming $Y \geq X$, then $L \geq 1$. If $L = 1$, for example, our rectangle is actually a square. To draw a triangle "at random" in this scaled rectangle simply means to pick three *independent* points (x_1, y_1), (x_2, y_2), and (x_3, y_3) to be the vertices of the triangle such that the x_i are each selected from a uniform distribution over the interval (0,1) and the y_i are each selected from a uniform distribution over the interval $(0,L)$.

For a triangle to be obtuse, you'll recall from high school geometry, means that it has an interior angle greater than 90°. There can, of course, be only one such angle in a triangle! So, to simulate this problem, what we need to do is generate a lot of random triangles inside our rectangle, check each triangle as we generate it for obtuseness, and keep track of how many are obtuse. That's the central question here—if we have the coordinates of the vertices of a triangle, how do we check for obtuseness? The law of cosines from trigonometry is the key. If we denote the three interior angles of a triangle by A, B, and C and the lengths of the sides opposite those angles by a, b, and c, respectively, then we have

$$a^2 = b^2 + c^2 - 2bc\cos(A),$$

$$b^2 = a^2 + c^2 - 2ac\cos(B),$$

$$c^2 = a^2 + b^2 - 2ab\cos(C).$$

Or,

$$\cos(A) = \frac{b^2 + c^2 - a^2}{2bc},$$

$$\cos(B) = \frac{a^2 + c^2 - b^2}{2ac},$$

$$\cos(C) = \frac{a^2 + b^2 - c^2}{2ab}.$$

Since the cosine of an acute angle, i.e., an angle in the interval (0,90°), is positive, while the cosine of an angle in the interval (90°,180°) is negative, we have the following test for an angle being obtuse: the sum of the squares of the lengths of the sides forming that angle in our triangle, minus the square of the length of the side opposite that angle, must be negative. That is, all we need to calculate are the numerators in the above cosine formulas. This immediately gives us an easy test for the obtuseness-or-not of a triangle: to be acute, i.e., to not be obtuse, all three interior angles must have positive cosines. The code `obtuse.m` uses this test on one million random triangles.

obtuse.m

```
01  S = 0;
02  L = 1;
03  for k = 1:1000000
04      for j = 1:3
05          r(j) = rand;
06      end
07      for j = 4:6
08          r(j) = L*rand;
09      end
10      d1 = (r(1) - r(2))^2 + (r(4) - r(5))^2;
11      d2 = (r(2) - r(3))^2 + (r(5) - r(6))^2;
12      d3 = (r(3) - r(1))^2 + (r(6) - r(4))^2;
13      if d1 < d2 + d3&d2 < d1 + d3&d3 < d1 + d2
14          obtusetriangle = 0;
15      else
16          obtusetriangle = 1;
17      end
18      S = S + obtusetriangle;
19  end
20  S/1000000
```

Lines 01 and 02 initialize the variables S, which is the running sum of the number of obtuse triangles generated at any given time, and L, the length of the longer side of the rectangle within which we will draw triangles. In line 02 we see L set to 1 (our rectangle is in fact a square), but we can set it to any value we wish, and later I'll show you the results for both L = 1 and L = 2. Lines 03 and 19 are the for/end loop that cycle the simulation through one million triangles. To understand lines 04 through 09, remember that the notation I'm using for the three points that are the vertices of each triangle is (x_1, y_1) (x_2, y_2), and (x_3, y_3), where the x_i are from a uniform distribution over 0 to 1 and the y_i are from a uniform distribution over 0 to L. So, in lines 04 to 06 we have a for/end loop that assigns random values to the x_i, i.e., $x_1 = r(1)$, $x_2 = r(2)$, and $x_3 = r(3)$, and

in lines 07 to 09 we have a for/end loop that assigns random values to the y_i, i.e., $y_1 = r(4)$, $y_2 = r(5)$, and $y_3 = r(6)$. Lines 10, 11, and 12 use the r-vector to calculate the lengths (squared) of the three sides of the current triangle. (Think of a^2, b^2, and c^2 as represented by d_1, d_2, and d_3.) Lines 13 through 17 then apply, with an if/else/end loop, our test for obtuseness: when obtuse.m exits from this loop, the variable obtusetriangle will have been set to either 0 (the triangle is not obtuse) or 1 (the triangle is obtuse). Line 18 then updates S, and the next random triangle is then generated. After the one-millionth triangle has been simulated and evaluated for its obtusness, line 20 completes obtuse.m by calculating the probability of a random triangle being obtuse. When obtuse.m was run (for L=1), it produced a value of 0.7247 for this probability, which I'll call $P(1)$, while for L=2 the simulation gave a value of $0.7979 = P(2)$.

In 1970 this problem was solved analytically,[6] allowing us to see just how well obtuse.m has performed. That solution gives the theoretical values of

$$P(1) = \frac{97}{150} + \frac{\pi}{40} = 0.72520648\cdots$$

and

$$P(2) = \frac{1{,}199}{1{,}200} + \frac{13\pi}{128} - \frac{3}{4}\ln(2) = 0.79837429\cdots.$$

I think it fair to say that obtuse.m has done well! This does, however, lead (as does the simulation results from our first code guess.m) to an obvious question: What if we had used not a million but fewer simulations? Theoretical analyses of the underlying mathematics of the Monte Carlo technique show that the error of the method decreases as $N^{-1/2}$, where N is the number of simulations. So, going from $N = 100 = 10^2$ to $N = 10{,}000 = 10^4$ (an increase in N by a factor of 10^2) should reduce the error by a factor of about $\sqrt{10^2} = 10$.

To illustrate more exactly just what this means, take a look at Figure 1, where you'll see a unit square in the first quadrant, with an inscribed quarter-circle. The area of the square is 1, and the area of the circular section is $\frac{1}{4}\pi$. If we imagine randomly throwing darts at the square (none of which miss), i.e., if we pick points uniformly distributed

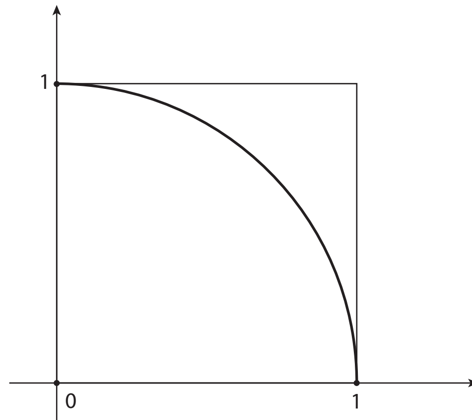


Figure 1. The geometry of a Monte Carlo estimation of π.

over the square, then we expect to see a fraction $\frac{(1/4)\pi}{1} = \frac{1}{4}\pi$ of them inside the circular section. So, if N denotes the total number of random points, and if P denotes the number of those points inside the circular section, then the fundamental idea behind the Monte Carlo method says we can write

$$\frac{P}{N} \approx \frac{1}{4}\pi,$$

or

$$\pi \approx \frac{4P}{N}.$$

We would expect this Monte Carlo estimate of π to get better and better as N increases. (This is an interesting use of a probabilistic technique to estimate a deterministic quantity; after all, what could be more deterministic than a constant, e.g., pi!)

The code pierror.m carries out this process for $N = 100$ points, over and over, for a total of 1,000 times, and each time stores the percentage error of the estimate arrived at for pi in the vector error. Then a histogram of the values in error is printed in the upper plot of Figure 2, thus giving us a visual indication of the distribution of the error we can expect in a simulation with 100 points. The width of the histogram

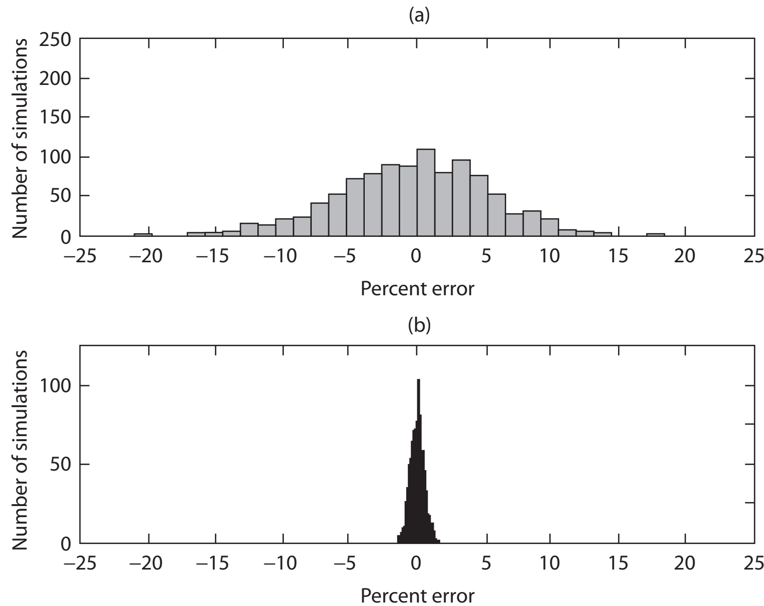


Figure 2. The error distribution in estimating π. a. Number of points per simulation = 100. b. Number of points per simulation = 10,000.

is a measure of what is called the variance of the error. The lower plot in Figure 2 shows what happens to the error variance when N is increased by a factor of 100, to 10,000 points per simulation. As mentioned earlier, theory says the error should decrease in this case by a factor of ten, and that is just what even a casual look at Figure 2 shows has happened. (I have not included in the code the MATLAB plotting and labeling commands that create Figure 2 from the vector `error`.) With high confidence we can say that, with $N = 100$ points, the error made by our Monte Carlo estimate of pi falls in the interval $\pm 15\%$, and with high confidence we can say that, with $N = 10{,}000$ points, the error made in estimating pi falls in the interval $\pm 1.5\%$. The theory of establishing what statisticians call a *confidence interval* can be made much more precise, but that would lead us away from the more pragmatic concerns of this book.

The lesson we'll take away from this is *the more simulations the better*, as long as our computational resources and available time aren't overwhelmed. A consequence of this is that Monte Carlo simulations

pierror.m

```
01  numberofpoints(1) = 100;
02  numberofpoints(2) = 10000;
03  for j = 1:2
04      N = numberofpoints(j);
05      for loop = 1:1000
06          P = 0;
07          for k = 1:N
08              x = rand;
09              y = rand;
10              if x^2 + y^2 < 1
11                  P = P + 1;
12              end
13          end
14          error(loop) = (4*P – pi*N)*100/(N*pi);
15      end
16  end
```

of rare events will require large values for N, which in turn requires a random number generator that is able to produce long sequences of "random" numbers before repeating.[7] This isn't to say, however, that running an ever longer simulation is the only way to reduce the statistical error in a Monte Carlo simulation. There are a number of other possibilities as well, falling into a category of techniques going under the general name of *variance reduction*; in Appendix 2 there is a discussion of one such technique. My general approach to convincing you that we have arrived at reasonably good results will be not nearly so sophisticated, however; I'll be happy enough if we can show that 10,000 simulations and 1,000,000 simulations give pretty nearly the same results. Mathematical purists may disagree with this philosophy, but this is not meant to be either a rigorous textbook or a theoretical dissertation. The variability of the estimates is due, of course, to different random numbers being used in each simulation. We could eliminate the variability (but not the error!) by starting each simulation with the random number generator seeded at the same

point, but since your generator is almost surely different from mine, there seems no point in that. When you run one of my codes (in your favorite language) on your computer, expect to get pretty nearly the results reported here, but certainly don't expect to get the *same* results.

Let's try another geometric probability problem, this one of historical importance. In the April 1864 issue of *Educational Times*, the English mathematician J. J. Sylvester (1814–1897) submitted a question in geometric probability. As the English mathematician M. W. Crofton (1826–1915), Sylvester's one-time colleague and protégé at the Royal Military Academy in Woolwich, wrote in his 1885 *Encyclopaedia Britannica* article on probability,[8] "Historically, it would seem that the first question on [geometric] probability, since Buffon, was the remarkable four-point problem of Prof. Sylvester." Like most geometric probability questions, it is far easier to state than it is to solve: If we pick four points at random inside some given convex region **K**, what is the probability that the four points are the vertices of a concave quadrilateral? All sorts of different answers were received by the *Educational Times*: $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, $\frac{3}{8}$, $\frac{35}{12\pi^2}$, and more. Of this, Sylvester wrote, "This problem does not admit of a deterministic solution." That isn't strictly so, as the variation in answers is due both to a dependency on **K** (which the different solvers had taken differently) and to the vagueness of what it means to say only that the four points are selected "at random." All of this variation was neatly wrapped up in a 1917 result derived by the Austrian-German mathematician Wilhelm Blaschke (1885–1962): If $P(\mathbf{K})$ is Sylvester's probability, then $\frac{35}{12\pi^2} \leq P(\mathbf{K}) \leq \frac{1}{3}$. That is, $0.29552 \leq P(\mathbf{K}) \leq 0.33333$, as **K** varies over all possible finite convex regions. For a given *shape* of **K**, however, it should be clear that the value of $P(\mathbf{K})$ is independent of the size of **K**; i.e., as with our first example, this is a scale-invariant problem.

It had far earlier (1865) been shown, by the British actuary Wesley Stoker Barker Woolhouse (1809–1893), that

$$P(\mathbf{K}) = \frac{4M(\mathbf{K})}{A(\mathbf{K})},$$

where $A(\mathbf{K})$ is the area of **K** and $M(\mathbf{K})$ is a constant unique to each **K**. Woolhouse later showed that $M(\mathbf{K}) = \frac{11A(\mathbf{K})}{144}$ and $M(\mathbf{K}) = \frac{289A(\mathbf{K})}{3{,}888}$ if **K** is

a square or a regular hexagon, respectively. Thus, we have the results

$$\text{if } \mathbf{K} \text{ is a square, then } P(\mathbf{K}) = \frac{4 \times 11}{144} = \frac{11}{36} = 0.3055,$$

and

$$\text{if } \mathbf{K} \text{ is a regular hexagon, then } P(\mathbf{K}) = \frac{4 \times 289}{3,888} = \frac{289}{972} = 0.2973.$$

Both of these results were worked out by Woolhouse in 1867.

$M(\mathbf{K})$ was also known[9] for the case of **K** being a circle (a computation first done by, again, Woolhouse), but let's suppose that we don't know what it is and that we'll estimate $P(\mathbf{K})$ in this case with a computer simulation. To write a Monte Carlo simulation of Sylvester's four-point problem for a circle, we have two separate tasks to perform. First, we have to decide what it means to select each of the four points "at random" in the circle. Second, we have to figure out a way to determine if the quadrilateral formed by the four "random" points is concave or convex.

To show that there is indeed a decision to be made on how to select the four points, let me first demonstrate that there is indeed more than one way a reasonable person might attempt to define the process of random point selection. Since we know the problem is scale invariant, we lose no generality by assuming that **K** is the particular circle with unit radius centered on the origin. Then,

Method 1: Let a "randomly selected" point have polar coordinates (r, θ), where r and θ are independent, uniformly distributed random variables over the intervals $(0,1)$ and $(0,2\pi)$, respectively.

Method 2: Let a "randomly selected" point have rectangular coordinates (x, y), where x and y are independent, uniformly distributed random variables over the same interval $(0, 1)$ and such that $x^2 + y^2 \leq 1$.

The final condition in Method 2 is to ensure that no point is outside **K**; any point that is will be rejected. Figure 3 shows 600 points selected "at random" by each of these two methods. By "at random" I think most people would demand a uniform distribution of the points over the

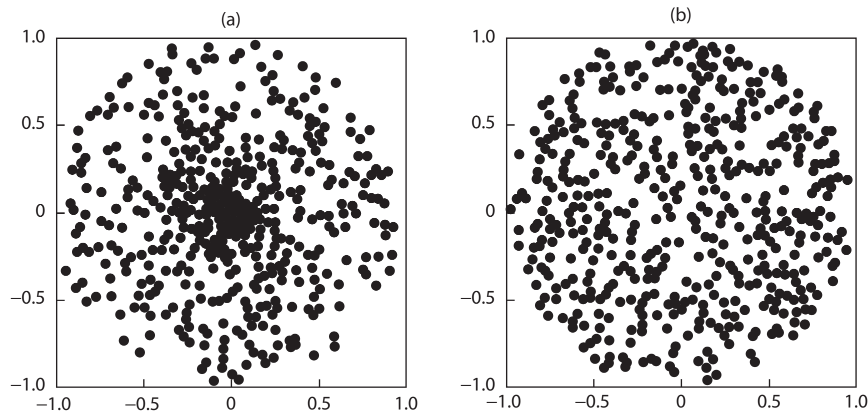


Figure 3. Two ways to generate points "at random" over a circle.
a. Method 1. b. Method 2.

area of the circle, and Figure 3 shows by inspection that this feature is present in Method 2 but is absent in Method 1 (notice the clumping of points near the center of the circle). What "at random" means was still a bit of a puzzle to many in the nineteenth century; Crofton wrote of it, in an 1868 paper in the *Philosophical Transactions of the Royal Society of London*, as follows:

> This [variation] arises, not from any inherent ambiguity in the subject matter, but from the weakness of the instrument employed; our undisciplined conceptions [that is, our intuitions] of a novel subject requiring to be repeatedly and patiently reviewed, tested, and corrected by the light of experience and comparison, before they [our intuitions, again] are purged from all latent error.

What Crofton and his fellow Victorian mathematicians would have given for a modern home computer that can create Figure 3 in a flash!

Method 2 is the way to properly generate points "at random," but it has the flaw of wasting computational effort generating many points that are then rejected for use (the ones that fail the $x^2 + y^2 \leq 1$ condition). Method 1 would be so much nicer to use, if we could eliminate the nonuniform clumping effect near the center of the circle.

This is, in fact, not hard to do once the reason for the clumping is identified. Since the points are uniformly distributed in the radial (r) direction, we see that a fraction r of the points fall inside a circle of radius r, i.e., inside a circle with area πr^2. That is, a fraction r of the points fall inside a smaller circle concentric with **K**, with an area r^2 as large as the area of **K**. For example, if we look at the smaller circle with radius one-half, then one-half of the points fall inside an area that is one-fourth the area of **K** and the other half of the points fall inside the annular region outside the smaller circle—a region that has an area three times that of the smaller circle! Hence the clumping effect near the center of **K**.

But now suppose that we make the radial distribution of the points vary not as directly with r, but rather as $\sqrt{r}$. Then a fraction r of the points fall inside a circle with area πr (remember, r itself is still uniform from 0 to 1), which is also a fraction r of the area of **K**. *Now* there is no clumping effect! So, our method for generating points "at random" is what I'll call Method 3:

Method 3: Let r and θ be independent, uniformly distributed random variables over the intervals (0,1) and (0,2π), respectively. Then the rectangular coordinates of a point are $(\sqrt{r}\cos(\theta), \sqrt{r}\sin(\theta))$.

Figure 4 shows 600 points generated by Method 3, and we see that we have indeed succeeded in eliminating the clumping, as well as the wasteful computation of random points that we then would reject.

We are now ready to tackle our second task. Once we have four random points in **K**, how do we determine if they form a concave quadrilateral? To see how to do this, consider the so-called *convex hull* of a set of n points in a plane, which is defined to be the smallest convex polygon that encloses all of the points. A picturesque way to visualize the convex hull of a set of points is to imagine that, at each of the points, a slender rigid stick is erected. Then, a huge rubber band is stretched wide open, so wide that all the sticks are inside the rubber band. Finally, we let the rubber band snap tautly closed around the sticks. Those sticks (points) that the rubber band catches are the vertices of the convex hull (the boundary of the hull is the rubber band itself). Clearly, if our four points form a convex quadrilateral, then all

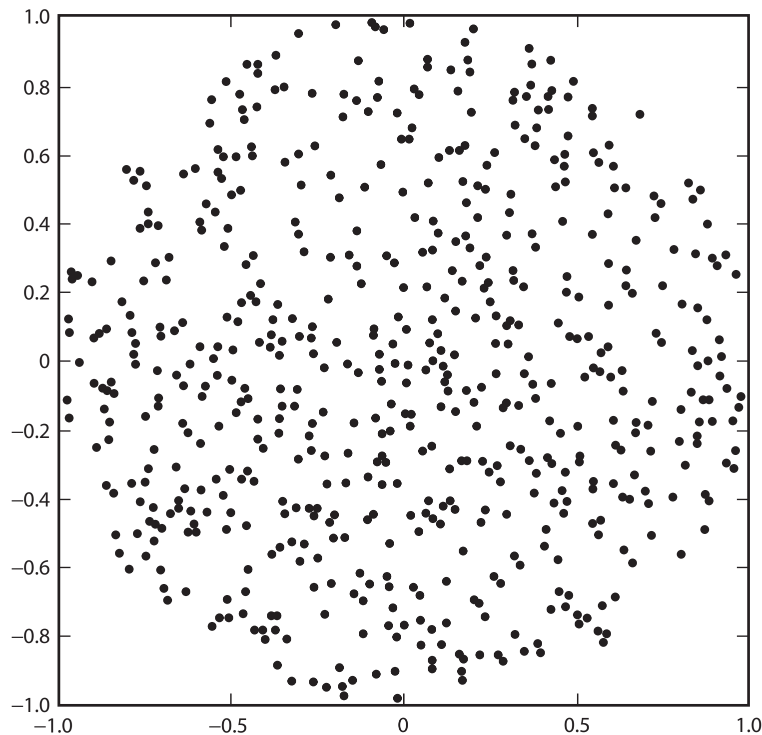


Figure 4. A third way to generate points "at random" over a circle.

four points catch the rubber band, but if the quadrilateral is concave, then one of the points will be inside the triangular convex hull defined by the other three points.

There are a number of general algorithms that computer scientists have developed to find the convex hull of n points in a plane, and MATLAB actually has a built-in function that implements one such algorithm.[10] So, this is one of those occasions where I'm going to tell you a little about MATLAB. Let **X** and **Y** each be vectors of length 4. Then we'll write the coordinates of our $n = 4$ points as (**X**(1), **Y**(1)), (**X**(2), **Y**(2)), (**X**(3), **Y**(3)), and (**X**(4), **Y**(4)). That is, the point with the "name" #k, $1 \le k \le 4$, is (**X**(k), **Y**(k)). Now, **C** is another vector, created from **X** and **Y**, by the MATLAB function convhull; if we write **C** = convhull(**X**, **Y**), then the elements of **C** are the names of the points

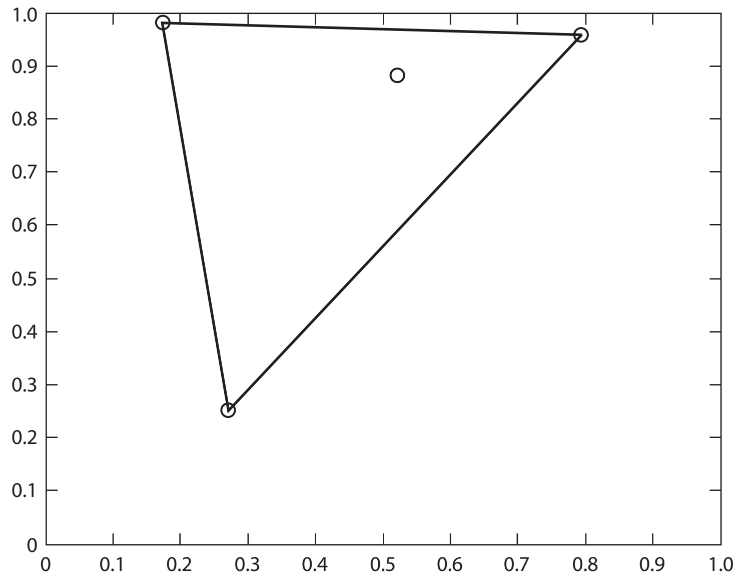


Figure 5. Convex hull of a concave quadrilateral.

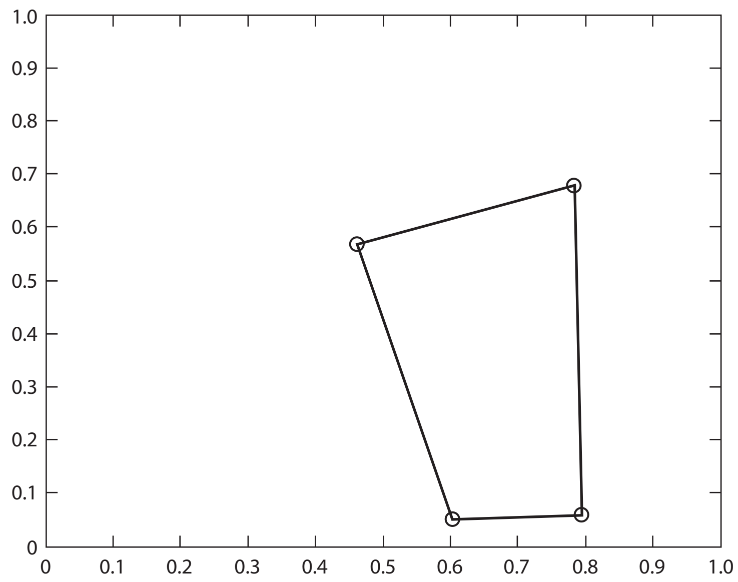


Figure 6. Convex hull of a convex quadrilateral.

on the convex hull. For example, if

$$\mathbf{X} = [0.7948 \quad 0.5226 \quad 0.1730 \quad 0.2714],$$

$$\mathbf{Y} = [0.9568 \quad 0.8801 \quad 0.9797 \quad 0.2523],$$

then

$$\mathbf{C} = [4 \quad 1 \quad 3 \quad 4],$$

where you'll notice that the first and last entry of **C** are the name of the same point. In this case, then, there are only three points on the hull (4, 1, and 3), and so the quadrilateral formed by all four points must be concave (take a look at Figure 5). If, as another example,

$$\mathbf{X} = [0.7833 \quad 0.4611 \quad 0.7942 \quad 0.6029],$$

$$\mathbf{Y} = [0.6808 \quad 0.5678 \quad 0.0592 \quad 0.0503],$$

then

$$\mathbf{C} = [4 \quad 3 \quad 1 \quad 2 \quad 4],$$

and the quadrilateral formed by all four points must be convex because all four points are on the hull (take a look at Figure 6).

We thus have an easy test to determine concavity (or not) of a quadrilateral: If **C** has four elements, then the quadrilateral is concave, but if **C** has five elements, then the quadrilateral is convex. The MATLAB function length gives us this information (length(v) = number of elements in the vector v), and it is used in the code sylvester.m, which generates one million random quadrilaterals and keeps track of the number of them that are concave. It is such a simple code that I think it explains itself. When sylvester.m was run it produced an estimate of $P(\mathbf{K} = \text{circle}) = 0.295557$; the theoretical value, computed by Woolhouse, is $\frac{35}{12\pi^2} = 0.295520$. Our Monte Carlo simulation has done quite well, indeed! (When run for just 10,000 simulations, sylvester.m's estimate was 0.2988.)

For the final examples of the style of this book, let me show you two problems that I'll first analyze theoretically, making some interesting arguments along the way, which we can then check by writing Monte Carlo simulations. This, you'll notice, is the reverse of the process we

sylvester.m

```
01  concave = 0;
02  constant = 2*pi;
03  for k = 1:1000000
04      for j = 1:4
05          number1 = sqrt(rand);
06          number2 = constant*rand;
07          X(j) = number1*cos(number2);
08          Y(j) = number1*sin(number2);
09      end
10      C = convhull(X,Y);
11      if length(C) == 4
12          concave = concave + 1;
13      end
14  end
15  concave/1000000
```

followed in the initial examples. Suppose, for the first of our final two problems, that we generate a sequence of independent random numbers x_i from a uniform distribution over the interval 0 to 1, and define the length of the sequence as L, where L is the number of x_i in the sequence until the first time the sequence fails to increase (including the first x_i that is less than the preceding x_i). For example, the sequence 0.1, 0.2, 0.3, 0.4, 0.35 has length $L = 5$, and the sequence 0.2, 0.1 has length $L = 2$. L is clearly an integer-valued random variable, with $L \geq 2$, and we wish to find its average (or *expected*) value, which we'll write as $E(L)$. Here's an analytical approach[11] to calculating $E(L)$.

The probability that length L is greater than k is

$$P(L > k) = P(x_1 < x_2 < x_3 < \cdots < x_k) = \frac{1}{k!}$$

since there are $k!$ equally likely permutations of the k x_i, only one of which is monotonic increasing. If $x_{k+1} < x_k$, then $L = k + 1$, and if $x_{k+1} > x_k$, then $L > k + 1$. In both cases, of course, $L > k$, just as claimed. Now, writing $P(L = k) = p_k$, we have by definition the answer

to our question as

$$E(L) = \sum_{k=2}^{\infty} kp_k = 2p_2 + 3p_3 + 4p_4 + 5p_5 + \cdots,$$

which we can write in the form

$$\begin{aligned} E(L) = \quad & p_2 + p_3 + p_4 + p_5 + \cdots \\ + \; & p_2 + p_3 + p_4 + p_5 + \cdots \\ + \; & \qquad\; p_3 + p_4 + p_5 + \cdots \\ + \; & \qquad\qquad\; p_4 + p_5 + \cdots \\ + \; & \cdots . \end{aligned}$$

The top two rows obviously sum to 1 (since all sequences have *some* length!). Thus,

$$E(L) = 2 + P(L > 2) + P(L > 3) + (P > 4) + \cdots = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots.$$

But, since

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots,$$

we see that $E(L) = 2.718281 \cdots$. Or is it? Let's do a Monte Carlo simulation of the sequence process and see what that says. Take a look at the code called mono.m—I think its operation is pretty easy to follow. When run, mono.m produced the estimate $E(L) = 2.717536$, which is pretty close to the theoretical answer of e (simulation of 10,000 sequences gave an estimate of 2.7246).

This last example is a good illustration of how a lot of mathematics is done; somebody gets an interesting idea and does some experimentation. Another such illustration is provided with a 1922 result proven by the German-born mathematician Hans Rademacher (1892–1969): suppose t_k is $+1$ or -1 with equal probability; if $\sum_{k=1}^{\infty} c_k^2 < \infty$, then $\sum_{k=1}^{\infty} t_k c_k$ exists with probability 1 (which means it is possible for the sum to diverge, but that happens with probability zero, i.e., "hardly ever"). In particular, the so-called random harmonic series (RHS), $\sum_{k=1}^{\infty} \frac{t_k}{k}$, almost surely exists because $\sum_{k=1}^{\infty} \frac{1}{k^2}$ is finite. The question

mono.m

```
01  sum = 0;
02  for k = 1:1000000
03      L = 0;
04      max = 0;
05      stop = 0;
06      while stop == 0
07          x = rand;
08          if x > max
09              max = x;
10          else
11              stop = 1;
12          end
13          L = L + 1;
14      end
15      sum = sum + L;
16  end
17  sum/1000000
```

of the distribution of the sums of the RHS is then a natural one to ask, and a theoretical study of the random harmonic series was done in 1995. The author of that paper[12] wanted just a bit more convincing about his analytical results, however, and he wrote, “For additional evidence we turn to simulations of the sums.” He calculated a histogram—using MATLAB—of 5,000 values of the partial sums $\sum_{k=1}^{100} \frac{t_k}{k}$, a calculation (agreeing quite nicely with the theoretical result) that I’ve redone (using instead 50,000 partial sums) with the code rhs.m, which you can find in Appendix 3 at the end of this book. I’ve put it there to give you a chance to do this first for yourself, just for fun and as another check on your understanding of the fundamental idea of Monte Carlo simulation (you’ll find the MATLAB command hist very helpful—I used it in creating Figure 2—in doing this; see the solution to Problem 12 for an illustration of hist).

Now, for the final example of this Introduction, let me show you a problem that is more like the practical ones that will follow than like the theoretical ones just discussed. Imagine that a chess player,

whom I'll call A, has been challenged to a curious sort of match by two of his competitors, whom I'll call B and C. A is challenged to play three sequential games, alternating between B and C, with the first game being with the player of A's choice. That is, A could play either *BCB* (I'll call this sequence 1) or *CBC* (and this will be sequence 2). From experience with B and C, A knows that C is the stronger player (the tougher for A to defeat). Indeed, from experience, A attaches probabilities p and q to his likelihood of winning any particular game against B and C, respectively, where $q < p$. The rule of this peculiar match is that to win the challenge (i.e., to win the match) A must win two games in a row. (This means, in particular, that even if A wins the first and the third games—two out of three games—A still loses the match!) So, which sequence should A choose to give himself the best chance of winning the match?

What makes this a problem of interest (besides the odd point I just mentioned) is that there are seemingly two different, indeed contradictory, ways for A to reason. A could argue, for example, that sequence 1 is the better choice because he plays C, his stronger opponent, only once. On the other hand, A could argue that sequence 1 is *not* a good choice because then he *has* to beat C in that single meeting in order to win two games in a row. With sequence 2, by contrast, he has two shots at C. So, which is it—sequence 1 or sequence 2?

Here's how to answer this question analytically. For A to win the match, there are just two ways to do so. Either he wins the first two games (and the third game is then irrelevant), or he loses the first game and wins the final two games. Let P_1 and P_2 be the probabilities A wins the match playing sequence 1 and sequence 2, respectively. Then, making the usual assumption of independence from game to game, we have

$$\text{for sequence 1}: \quad P_1 = pq + (1-p)qp$$

and

$$\text{for sequence 2}: \quad P_2 = qp + (1-q)pq.$$

Therefore

$$\begin{aligned} P_2 - P_1 &= [qp + (1-q)pq] - [pq + (1-p)qp] = (1-q)pq - (1-p)qp \\ &= pq[(1-q) - (1-p)] = pq(p-q) > 0 \end{aligned}$$

because $q < p$. So, sequence 2 always, for any $p > q$, gives the greater probability for A winning the challenge match, even though sequence 2 requires A to play his stronger opponent twice. This strikes most, at least initially, as nonintuitive, almost paradoxical, but that's what the math says. What would a Monte Carlo simulation say?

The code chess.m plays a million simulated three-game matches; actually, each match is played twice, once for each of the two sequences, using the same random numbers in each sequence. The code keeps track of how many times A wins a match with each sequence, and

chess.m

```
01  p = input('What is p?');
02  q = input('What is q?');
03  prob(1,1) = p;prob(1,3) = p;prob(2,2) = p;
04  prob(1,2) = q;prob(2,1) = q;prob(2,3) = q;
05  wonmatches = zeros(1,2);
06  for loop = 1:1000000
07    wongame = zeros(2,3);
08    for k = 1:3
09      result(k) = rand;
10    end
11    for game = 1:3
12      for sequence = 1:2
13        if result(game) < prob(sequence,game)
14          wongame(sequence,game) = 1;
15        end
16      end
17    end
18    for sequence = 1:2
19      if wongame(sequence,1) + wongame(sequence,2) == 2|...
            wongame(sequence,2) + wongame(sequence,3) == 2
20        wonmatches(sequence) = wonmatches(sequence)+1;
21      end
22    end
23  end
24  wonmatches/1000000
```

so arrives at its estimates for P_1 and P_2. To understand how the code works (after lines 01 and 02 bring in the values of p and q), it is necessary to explain the two entities prob and wongame, which are both 2×3 arrays. The first array is defined as follows: prob(j,k) is the probability A wins the kth game in sequence j, and these values are set in lines 03 and 04. Line 05 sets the values of the two-element row vector wonmatches to zero, i.e., at the start of chess.m wonmatches(1) = wonmatches(2) = 0, which are the initial number of matches won by A when playing sequence 1 and sequence 2, respectively. Lines 06 and 23 define the main loop, which executes one million pairs of three-game matches. At the start of each such simulation, line 07 initializes all three games, for each of the two sequences, to zero in wongame, indicating that A hasn't (not yet, anyway) won any of them. Then, in lines 08, 09, and 10, three random numbers are generated that will be compared to the entries in prob to determine which games in each of the two sequences A wins. This comparison is carried out in the three nested loops in lines 11 through 17, which sets the appropriate entry in wongame to 1 if A wins that game. Then, in lines 18 through 23, the code checks each row in wongame (row 1 is for sequence 1, and row 2 is for sequence 2) to see if A satisfied at least one of the two match winning conditions: winning the first two or the last two games. (The three periods at the end of the first line of line 19 is MATLAB's way of continuing a line too long to fit the width of a page.) If so, then line 20 credits a match win to A for the appropriate sequence. Finally, line 24 give chess.m's estimates of P_1 and P_2 after one million match simulations.

The following table compares the estimates of P_1 and P_2 produced by chess.m, for some selected values of p and q, to the numbers

		Theoretical		*Simulated*	
p	q	P_1	P_2	P_1	P_2
0.9	0.8	0.7920	0.8640	0.7929	0.8643
0.9	0.4	0.3960	0.5760	0.3956	0.5761
0.4	0.3	0.1920	0.2040	0.1924	0.2042
0.4	0.1	0.0640	0.0760	0.0637	0.0755

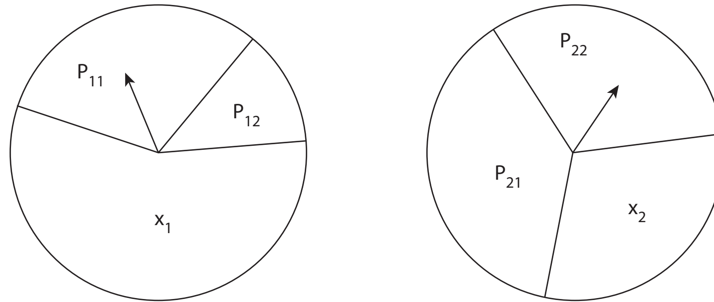


Figure 7. A spin game.

produced by the theoretical expressions we calculated earlier. You can see that the simulation results are in excellent agreement with the theoretical values.

Here's a little Monte Carlo challenge problem, of the same type as the chess player's problem, for you to try your hand at (a solution is given in Appendix 6). Consider the following game, which uses the two spinner disks shown in Figure 7. Suppose a player spins one or the other of the pointers on the disks according to the following rules: (1) if the player spins pointer i and it stops in the region with area p_{ij}, he moves from disk i to disk j (i and j are either 1 or 2); (2) if a pointer stops in the region with area x_i, the game ends; (3) if the game ends in the region with area x_1, the player wins, but if the pointer stops in the region with area x_2 the player loses. What is the probability the player, starting with disk 1, wins? Assume the area of each disk is one, so that $x_1 + p_{11} + p_{12} = 1$, as well as that $x_2 + p_{21} + p_{22} = 1$ (that is, all the x's and p's are actually the probabilities that the pointer of either disk stops in the respective region). This game can be analyzed theoretically (which allows the code to be completely validated), and you'll find that solution in Appendix 6, too. Run your code for the case of $p_{11} = 0.2$, $p_{12} = 0.4$, $p_{21} = 0.3$, and $p_{22} = 0.35$.

I'll end this introduction—so we can get to the fun stuff of the problems—with two quotations. The first is from from Professor Alan Levine (see Problem 13), and it forms the central thesis of this book:

> From a pedagogical point of view, the problem and its solution as we have presented it here illustrate the fact that mathematics

> is discovered in much the same way as any other science—by experimentation (here, simulation) followed by confirmation (proof). All too often, students think mathematics was created by divine inspiration since, by the time they see it in class, all the "dirty work" has been "cleaned up."

In this book, we'll be paying close attention to the "dirty work"!

The second quote is from the mathematical physicist W. Edwards Deming (1908–1993), and it makes a forceful statement on the value of being able to write computer codes like the ones in this book:

> If you can't describe what you are doing as a process [read *process* as *computer code*], you don't know what you are doing.

Right on!

References and Notes

1. Computer programs that use a random number generator to simulate a physical process are called *Monte Carlo codes*, in recognition of the famous gambling casino. This term was used by the pioneers of the method—Stanislaw Ulam (1909–1984), John von Neumann (1903–1957), and Nicholas Metropolis (1915–1999)—from the earliest days, and in his paper, "The Beginning of the Monte Carlo Method," published in *Los Alamos Science* (Special Issue, 1987, pp. 125–130), Metropolis reveals that *he* was the originator of the name. In a paper written some years later ("The Age of Computing: A Personal Memoir," *Daedalus*, Winter 1992, pp. 119–130), Metropolis wrote the astonishing words, "The Monte Carlo method . . . was developed by Ulam and myself without any knowledge of statistics; to this day the theoretical statistician is unable to give a proper foundation to the method." I think the general view among mathematicians is that this last statement is not so. Computer scientists have introduced the general term of *randomized algorithm*, which includes Monte Carlo simulations of physical processes as a subset. A randomized algorithm is any algorithm (computer code) that uses a random number generator. In addition to Monte Carlo codes, there is now another category called *Las Vegas* codes (which are of no interest in this book). You can find more on all of this in a fascinating paper by Don Fallis, "The Reliability of Randomized Algorithms" (*British Journal for the Philosophy of Science*, June 2000, pp. 255–271).

2. I hope no reader thinks I am picking on Annapolis by repeating this tale (one that I suspect is quite popular at the United States Military Academy at West Point). I taught during the academic year 1981–1982 at the Naval Postgraduate School in Monterey, California, and many of my students were Annapolis graduates who were quite good at mathematics. The correct answer to the missile intercept problem is that if a missile successfully intercepts its target with probability 1/3, then of course it *misses* its target with probability 2/3. Three identical, independent missiles all fail to hit the same target, then, with probability $(2/3)^3 = 8/27$. So, at least one of the missiles does hit the target with probability $1 - (8/27) = 19/27 = 0.704$, considerably less than the certainty thought by the "top Navy officer." As the opening quotation makes clear, however, not all naval officers are ignorant of the inherent uncertainty of success in a three-missile attack on a target. An explanation of Hornblower's observation can be found in A. R. Hall, *Ballistics in the Seventeenth Century*, (Cambridge: Cambridge University Press, 1952, p. 55): "Nothing was uniform in spite of official efforts at standardisation; powder varied in strength from barrel to barrel by as much as twenty per cent; shot differed widely in weight, diameter, density and degree of roundness. The liberal allowances for windage, permitting the ball to take an ambiguous, bouncing path along the barrel of the gun, gave no security that the line-of-sight would be the line of flight, even if the cannon had been perfect. There was little chance of repeating a lucky shot since, as the gun recoiled over a bed of planks, it was impossible to return it to its previous position, while the platform upon which it was mounted subsided and disintegrated under the shock of each discharge."

3. Consider, for example, this quotation (*Washington Post*, July 22, 2005) from Air Force Lt. Gen. Henry Obering, director of the U.S. Missile Defense Agency: "We have a better than zero chance of successfully intercepting, I believe, an inbound warhead." This is no doubt true—but of course a high-flying eagle lost in a snowstorm with a mininuke clamped in its beak could make the same claim. After all, though "rather small," it is still true that $10^{-\text{googolplex}} > 0$.

4. The probability of that happening—of getting no pairings of presidents and terms correct—is 0.368, which is not insignificant. In general, the probability of getting m correct pairings when assigning M terms to M presidents, where the M terms are each uniquely assigned to a president, is given by $\frac{1}{m!}\sum_{k=0}^{M-m}\frac{(-1)^k}{k!}$. For $m = 0$ and $M = 24$ this formula gives the probability of zero correct pairings as very nearly $e^{-1} \approx 0.368$. See, for example, Emanuel Parzen, *Modern Probability Theory and Its Applications* (New York: John Wiley & Sons, 1960, pp. 77–79). You can find a scholarly, readable history of Montmort's problem, including detailed discussions of how the greats of yesteryear calculated their solutions to the problem, in L. Takács, "The Problem of Coincidences" (*Archive for History of Exact Sciences*,

21 [no. 3], 1980, pp. 229–244). The standard textbook derivation of the above formula uses combinatorial arguments and the inclusion-exclusion principle of probability. There is, however, a very clever way to numerically calculate the probability of no pairings, to any degree of accuracy desired, by using recursion. See Appendix 4 for how to do that, along with a MATLAB code that estimates the probability of no pairings with a Monte Carlo simulation. The inclusion-exclusion principle is discussed in Appendix 5, both theoretically and experimentally (i.e., it is illustrated with a simple MATLAB code).

5. In addition to one or more of those texts, you can find some interesting discussion on the Buffon needle problem in "Nineteenth-Century Developments in Geometric Probability: J. J. Sylvester, M. W. Crofton, J.-É. Barbier, and J. Bertrand" (*Archive for History of Exact Sciences*, 2001, pp. 501–524), by E. Senata, K. H. Parshall, and F. Jongmans.

6. Eric Langford, "A Problem in Geometric Probability" (*Mathematics Magazine*, November–December 1970, pp. 237–244). This paper gives the general solution for all $L \geq 1$, not just for the special cases of $L = 1$ and $L = 2$. The analysis in that paper inspired the following related question. If two points are independently and uniformly located in the unit interval, they divide that interval into three segments. What is the probability that those three segments form an obtuse triangle? You can find a theoretical analysis of this question in *Mathematics Magazine* (November–December 1973, pp. 294–295), where the answer is given as $\frac{9}{4} - 3\ln(2) = 0.170558\cdots$. The Monte Carlo code obtuse1.m produced the estimate 0.170567 using one million simulations of randomly dividing the unit interval into three parts. The variable S in line 01 is the same

obtuse1.m

```
01  S=0;
02  for k=1:1000000
03      point1=rand;
04      point2=rand;
05      if point1>point2
06          temp=point1;
07          point1=point2;
08          point2=temp;
09      end
10      a=point1;
11      b=point2-point1;
12      c=1-point2;
13      if a+b>c&a+c>b&b+c>a
14          d1=a^2;
15          d2=b^2;
```

(continued)

(continued)

```
16            d3=c^2;
17            if d1<d2+d3&d2<d1+d3&d3<d1+d2
18                obtusetriangle=0;
19            else
20                obtusetriangle=1;
21            end
22            S=S+obtusetriangle;
23        end
24    end
25    S/1000000
```

S as in obtuse.m. Lines 03 and 04 define the variables point1 and point2, and lines 05 through 09 ensure that their values are such that 0 < point1 < point2 < 1. Thinking of point1 and point2 as the two points selected at random in the unit interval, then lines 10, 11, and 12 calculate the values of a, b, and c as the lengths of the three sides of a would-be triangle. Line 13 determines whether those sides do, in fact, satisfy the triangle inequalities that must be satisfied if and only if a triangle is possible (in a triangle, the sum of the lengths of any two sides is greater than the length of the remaining side); if they do, then the rest of the code is simply that of obtuse.m, which determines whether the triangle is an obtuse triangle.

7. For more on random number generators, see my book, *Duelling Idiots* (Princeton, N. J.: Princeton University Press, 2000, pp. 175–197). The generator in MATLAB 7.3, for example, will produce $2^{1492} > 10^{449}$ random numbers before repeating. If that generator had begun producing numbers at the prodigious rate of one trillion per second from the moment the universe was created (famously known as the Big Bang), about fifteen billion years ago, then it would have produced about 4.5×10^{29} numbers up to now. This is an infinitesimal fraction of the 7.3 generator's cycle length. To gain some historical appreciation of modern random number generators, consider the following complaint made by Lord Kelvin (Scottish engineer, physicist, and mathematician William Thomson [1824–1907]) in a lecture given in April 1900 at the Royal Institution of Great Britain (you can find his talk in the July 1901 issue of *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* under the title, "Nineteenth Century Clouds over the Dynamical Theory of Heat and Light"). When studying a problem in theoretical thermodynamics, Kelvin performed what may be the very first Monte Carlo simulation of a physical process. To select, with equal probability, from among 200 possibilities, he used 100 cards numbered from 0 to 99 (subjected before each drawing to a "very thorough shuffling") and coupled the result with the outcome of a coin toss. Alas, any real coin is almost certainly not

fair, but Kelvin said nothing about how he accounted for that bias—this can be done for any coin; do you see how? The answer is in Appendix 7. In a footnote Kelvin says he also tried to replace the cards with small pieces of paper drawn from a bowl, but found "In using one's finger to mix dry [pieces] of paper, in a bowl, very considerable disturbance may be expected from electrification [i.e., from static electricity!]." Kelvin would have loved MATLAB's `rand`, but truly practical Monte Carlo had to wait until the invention of the high-speed electronic computer more than forty years later (as well as for advances in theoretical understanding on how to build random number generators in software). For a nice discussion on the modern need to achieve very high-speed generation of random numbers, see Aaldert Compagner, "Definitions of Randomness" (*American Journal of Physics*, August 1991, pp. 700–705).

8. For more on Crofton's contributions to geometric probability, see B. Eisenberg and R. Sullivan, "Crofton's Differential Equation" (*American Mathematical Monthly*, February 2000, pp. 129–139).

9. Richard E. Pfiefer, "The Historical Development of J. J. Sylvester's Four Point Problem" (*Mathematics Magazine*, December 1989, pp. 309–317).

10. So, unlike my argument about `sort`, I am not going to create the detailed code for implementing a convex hull algorithm. And why not, you ask? Well, I have to leave *something* for you to do! See, for example, R. L. Graham, "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set" (*Information Processing Letters*, 1972, pp. 132–133).

11. Harris S. Shultz and Bill Leonard, "Unexpected Occurrences of the Number e" (*Mathematics Magazine*, October 1989, pp. 269–271). See also Frederick Solomon, "Monte Carlo Simulation of Infinite Series" (*Mathematics Magazine*, June 1991, pp. 188–196).

12. Kent E. Morrison, "Cosine Products, Fourier Transforms, and Random Sums" (*American Mathematical Monthly*, October 1995, pp. 716–724) and Byron Schmuland, "Random Harmonic Series" (*American Mathematical Monthly*, May 2003, pp. 407–416). The probabilistic harmonic series is interesting because the classic harmonic series, where $c_k = +1$, always, diverges. For more discussion on this and on the history of $\sum_{k=1}^{\infty} 1/k^2$, see my book, *An Imaginary Tale: The Story of* $\sqrt{-1}$ (Princeton, N. J.: Princeton University Press, 1998, 2006 [corrected ed.], pp. 146–149).