

Chapter 9

Random Numbers

This chapter describes algorithms for the generation of pseudorandom numbers with both uniform and normal distributions.

9.1 Pseudorandom Numbers

Here is an interesting number:

0.814723686393179

This is the first number produced by the MATLAB random number generator with its default settings. Start up a fresh MATLAB, set `format long`, type `rand`, and it's the number you get.

If all MATLAB users, all around the world, all on different computers, keep getting this same number, is it really “random”? No, it isn't. Computers are (in principle) deterministic machines and should not exhibit random behavior. If your computer doesn't access some external device, like a gamma ray counter or a clock, then it must really be computing *pseudorandom* numbers. Our favorite definition was given in 1951 by Berkeley professor D. H. Lehmer, a pioneer in computing and, especially, computational number theory:

A random sequence is a vague notion ... in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians ...

9.2 Uniform Distribution

Lehmer also invented the *multiplicative congruential algorithm*, which is the basis for many of the random number generators in use today. Lehmer's generators involve three integer parameters, a , c , and m , and an initial value, x_0 , called the *seed*. A

September 16, 2013



sequence of integers is defined by

$$x_{k+1} = ax_k + c \bmod m.$$

The operation “ $\bmod m$ ” means take the remainder after division by m . For example, with $a = 13$, $c = 0$, $m = 31$, and $x_0 = 1$, the sequence begins with

$$1, 13, 14, 27, 10, 6, 16, 22, 7, 29, 5, 3, \dots$$

What’s the next value? Well, it looks pretty unpredictable, but you’ve been initiated. So you can compute $(13 \cdot 3) \bmod 31$, which is 8. The first 30 terms in the sequence are a permutation of the integers from 1 to 30 and then the sequence repeats itself. It has a period equal to $m - 1$.

If a pseudorandom integer sequence with values between 0 and m is scaled by dividing by m , the result is floating-point numbers uniformly distributed in the interval $[0, 1]$. Our simple example begins with

$$0.0323, 0.4194, 0.4516, 0.8710, 0.3226, 0.1935, 0.5161, \dots$$

There is only a finite number of values, 30 in this case. The smallest value is $1/31$; the largest is $30/31$. Each one is equally probable in a long run of the sequence.

In the 1960s, the Scientific Subroutine Package (SSP) on IBM mainframe computers included a random number generator named RND or RANDU. It was a multiplicative congruential with parameters $a = 65539$, $c = 0$, and $m = 2^{31}$. With a 32-bit integer word size, arithmetic mod 2^{31} can be done quickly. Furthermore, because $a = 2^{16} + 3$, the multiplication by a can be done with a shift and an addition. Such considerations were important on the computers of that era, but they gave the resulting sequence a very undesirable property. The following relations are all taken mod 2^{31} :

$$\begin{aligned} x_{k+2} &= (2^{16} + 3)x_{k+1} = (2^{16} + 3)^2 x_k \\ &= (2^{32} + 6 \cdot 2^{16} + 9)x_k \\ &= [6 \cdot (2^{16} + 3) - 9]x_k. \end{aligned}$$

Hence

$$x_{k+2} = 6x_{k+1} - 9x_k \quad \text{for all } k.$$

As a result, there is an extremely high correlation among three successive random integers of the sequence generated by RANDU.

We have implemented this defective generator in an M-file **randssp**. A demonstration program **randgui** tries to compute π by generating random points in a cube and counting the fraction that actually lie within the inscribed sphere. With these M-files on your path, the statement

```
randgui(@randssp)
```

will show the consequences of the correlation of three successive terms. The resulting pattern is far from random, but it can still be used to compute π from the ratio of the volumes of the cube and sphere.

For many years, the MATLAB uniform random number function, `rand`, was also a multiplicative congruential generator. The parameters were

$$\begin{aligned}a &= 7^5 = 16807, \\c &= 0, \\m &= 2^{31} - 1 = 2147483647.\end{aligned}$$

These values are recommended in a 1988 paper by Park and Miller [11].

This old MATLAB multiplicative congruential generator is available in the M-file `randmzg`. The statement

```
randgui(@randmzg)
```

shows that the points do not suffer the correlation of the SSP generator. They generate a much better “random” cloud within the cube.

Like our toy generator, `randmzg` and the old version of the MATLAB function `rand` generate all real numbers of the form k/m for $k = 1, \dots, m - 1$. The smallest and largest are 0.00000000046566 and 0.99999999953434. The sequence repeats itself after $m - 1$ values, which is a little over 2 billion numbers. A few years ago, that was regarded as plenty. But today, an 800 MHz Pentium laptop can exhaust the period in less than half an hour. Of course, to do anything useful with 2 billion numbers takes more time, but we would still like to have a longer period.

In 1995, version 5 of MATLAB introduced a completely different kind of random number generator. The algorithm is based on work of George Marsaglia, a professor at Florida State University and author of the classic analysis of random number generators, “Random numbers fall mainly in the planes” [6].

Marsaglia’s generator [9] does not use Lehmer’s congruential algorithm. In fact, there are no multiplications or divisions at all. It is specifically designed to produce floating-point values. The results are not just scaled integers. In place of a single seed, the new generator has 35 words of internal memory or *state*. Thirty-two of these words form a cache of floating-point numbers, z , between 0 and 1. The remaining three words contain an integer index i , which varies between 0 and 31, a single random integer j , and a “borrow” flag b . This entire state vector is built up a bit at a time during an initialization phase. Different values of j yield different initial states.

The generation of the i th floating-point number in the sequence involves a “subtract-with-borrow” step, where one number in the cache is replaced with the difference of two others:

$$z_i = z_{i+20} - z_{i+5} - b.$$

The three indices, i , $i + 20$, and $i + 5$, are all interpreted mod 32 (by using just their last five bits). The quantity b is left over from the previous step; it is either zero or a small positive value. If the computed z_i is positive, b is set to zero for the next step. But if the computed z_i is negative, it is made positive by adding 1.0 before it is saved and b is set to 2^{-53} for the next step. The quantity 2^{-53} , which is half of the MATLAB constant `eps`, is called one *ulp* because it is one *unit in the last place* for floating-point numbers slightly less than 1.

By itself, this generator would be almost completely satisfactory. Marsaglia has shown that it has a huge period—almost 2^{1430} values would be generated before it repeated itself. But it has one slight defect. All the numbers are the results of floating-point additions and subtractions of numbers in the initial cache, so they are all integer multiples of 2^{-53} . Consequently, many of the floating-point numbers in the interval $[0, 1]$ are not represented.

The floating-point numbers between $1/2$ and 1 are equally spaced with a spacing of one ulp, and our subtract-with-borrow generator will eventually generate all of them. But numbers less than $1/2$ are more closely spaced and the generator would miss most of them. It would generate only half of the possible numbers in the interval $[1/4, 1/2]$, only a quarter of the numbers in $[1/8, 1/4]$, and so on. This is where the quantity j in the state vector comes in. It is the result of a separate, independent, random number generator based on bitwise logical operations. The floating-point fraction of each z_i is XORed with j to produce the result returned by the generator. This breaks up the even spacing of the numbers less than $1/2$. It is now theoretically possible to generate all the floating-point numbers between 2^{-53} and $1 - 2^{-53}$. We're not sure if they are all actually generated, but we don't know of any that can't be.

Figure 9.1 shows what the new generator is trying to accomplish. For this graph, one ulp is equal to 2^{-4} instead of 2^{-53} .

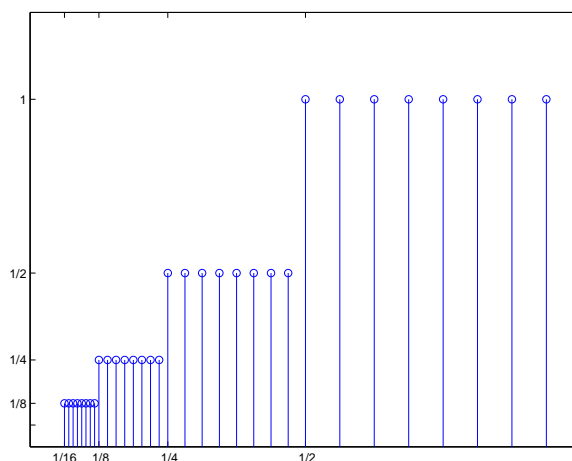


Figure 9.1. *Uniform distribution of floating-point numbers.*

The graph depicts the relative frequency of each of the floating-point numbers. A total of 32 floating-point numbers are shown. Eight of them are between $1/2$ and 1 , and they are all equally likely to occur. There are also eight numbers between $1/4$ and $1/2$, but, because this interval is only half as wide, each of them should occur only half as often. As we move to the left, each subinterval is half as wide as the previous one, but it still contains the same number of floating-point numbers, so their relative frequencies must be cut in half. Imagine this picture with 2^{53} numbers in each of 2^{32} smaller intervals and you will see what the new random

number generator is doing.

With the additional bit fiddling, the period of the new generator becomes something like 2^{1492} . Maybe we should call it the Christopher Columbus generator. In any case, it will run for a very long time before it repeats itself.

9.3 Normal Distribution

Almost all algorithms for generating normally distributed random numbers are based on transformations of uniform distributions. The simplest way to generate an m -by- n matrix with approximately normally distributed elements is to use the expression

```
sum(rand(m,n,12),3) - 6
```

This works because `R = rand(m,n,p)` generates a three-dimensional uniformly distributed array and `sum(R,3)` sums along the third dimension. The result is a two-dimensional array with elements drawn from a distribution with mean $p/2$ and variance $p/12$ that approaches a normal distribution as p increases. If we take $p = 12$, we get a pretty good approximation to the normal distribution and we get the variance to be equal to one without any additional scaling. There are two difficulties with this approach. It requires twelve uniforms to generate one normal, so it is slow. And the finite p approximation causes it to have poor behavior in the tails of the distribution.

Older versions of MATLAB—before MATLAB 5—used the polar algorithm. This generates two values at a time. It involves finding a random point in the unit circle by generating uniformly distributed points in the $[-1, 1] \times [-1, 1]$ square and rejecting any outside the circle. Points in the square are represented by vectors with two components. The rejection portion of the code is

```
r = Inf;
while r > 1
    u = 2*rand(2,1)-1
    r = u'*u
end
```

For each point accepted, the polar transformation

```
v = sqrt(-2*log(r)/r)*u
```

produces a vector with two independent normally distributed elements. This algorithm does not involve any approximations, so it has the proper behavior in the tails of the distribution. But it is moderately expensive. Over 21% of the uniform numbers are rejected if they fall outside of the circle, and the square root and logarithm calculations contribute significantly to the cost.

Beginning with MATLAB 5, the normal random number generator `randn` uses a sophisticated table lookup algorithm, also developed by George Marsaglia. Marsaglia calls his approach the *ziggurat* algorithm. Ziggurats are ancient Mesopotamian terraced temple mounds that, mathematically, are two-dimensional step functions. A one-dimensional ziggurat underlies Marsaglia's algorithm.

Marsaglia has refined his ziggurat algorithm over the years. An early version is described in Knuth's classic *The Art of Computer Programming* [5]. The version used in MATLAB is described by Marsaglia and W. W. Tsang in [7]. A Fortran version is described in [2, sect. 10.7]. A more recent version is available in the online electronic *Journal of Statistical Software* [8]. We describe this recent version here because it is the most elegant. The version actually used in MATLAB is more complicated, but is based on the same ideas and is just as effective.

The probability density function, or *pdf*, of the normal distribution is the bell-shaped curve

$$f(x) = ce^{-x^2/2},$$

where $c = 1/(2\pi)^{1/2}$ is a normalizing constant that we can ignore. If we generate random points (x, y) , uniformly distributed in the plane, and reject any of them that do not fall under this curve, the remaining x 's form our desired normal distribution. The ziggurat algorithm covers the area under the pdf by a slightly larger area with n sections. Figure 9.2 has $n = 8$; actual code might use $n = 128$. The top $n - 1$ sections are rectangles. The bottom section is a rectangle together with an infinite tail under the graph of $f(x)$. The right-hand edges of the rectangles are at the points $z_k, k = 2, \dots, n$, shown with circles in the picture. With $f(z_1) = 1$ and $f(z_{n+1}) = 0$, the height of the k th section is $f(z_k) - f(z_{k+1})$. The key idea is to choose the z_k 's so that all n sections, including the unbounded one on the bottom, have the same area. There are other algorithms that approximate the area under the pdf with rectangles. The distinguishing features of Marsaglia's algorithm are the facts that the rectangles are horizontal and have equal areas.

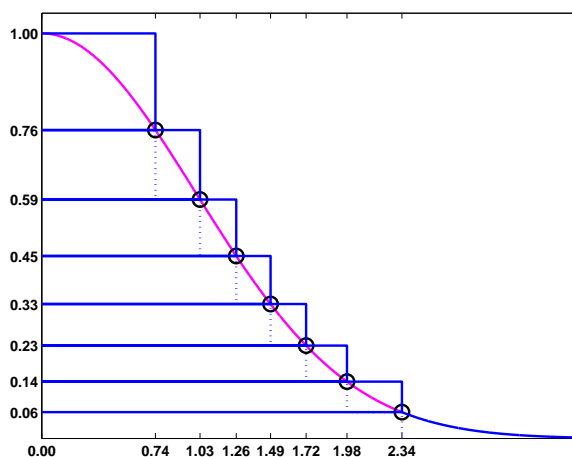


Figure 9.2. *The ziggurat algorithm.*

For a specified number, n , of sections, it is possible to solve a transcendental equation to find z_n , the point where the infinite tail meets the last rectangular section. In our picture with $n = 8$, it turns out that $z_n = 2.34$. In an actual code with $n = 128$, $z_n = 3.4426$. Once z_n is known, it is easy to compute the common

area of the sections and the other right-hand endpoints z_k . It is also possible to compute $\sigma_k = z_{k-1}/z_k$, which is the fraction of each section that lies underneath the section above it. Let's call these fractional sections the *core* of the ziggurat. The right-hand edge of the core is the dotted line in our picture. The computation of these z_k 's and σ_k 's is done in initialization code that is run only once.

After the initialization, normally distributed random numbers can be computed very quickly. The key portion of the code computes a single random integer, j , between 1 and n and a single uniformly distributed random number, u , between -1 and 1 . A check is then made to see if u falls in the core of the j th section. If it does, then we know that uz_j is the x -coordinate of a point under the pdf and this value can be returned as one sample from the normal distribution. The code looks something like this.

```
j = ceil(128*rand);
u = 2*rand-1;
if abs(u) < sigma(j)
    r = u*z(j);
    return
end
```

Most of the σ_j 's are greater than 0.98, and the test is true over 97% of the time. One normal random number can usually be computed from one random integer, one random uniform, an if-test, and a multiplication. No square roots or logarithms are required. The point determined by j and u will fall outside the core less than 3% of the time. This happens if $j = 1$ because the top section has no core, if j is between 2 and $n-1$ and the random point is in one of the little rectangles covering the graph of $f(x)$, or if $j = n$ and the point is in the infinite tail. In these cases, additional computations involving logarithms, exponentials, and more uniform samples are required.

It is important to realize that, even though the ziggurat step function only approximates the probability density function, the resulting distribution is exactly normal. Decreasing n decreases the amount of storage required for the tables and increases the fraction of time that extra computation is required, but does not affect the accuracy. Even with $n = 8$, we would have to do the more costly corrections almost 23% of the time, instead of less than 3%, but we would still get an exact normal distribution.

With this algorithm, MATLAB 6 can generate normally distributed random numbers as fast as it can generate uniformly distributed ones. In fact, MATLAB on an 800 MHz Pentium laptop can generate over 10 million random numbers from either distribution in less than one second.

9.4 randtx, randntx

Our NCM M-file collection includes textbook functions **randtx** and **randntx**. For these two functions, we reproduce the behavior of the corresponding built-in functions **rand** and **randn** that were available in version 5 of MATLAB. The two textbook functions use the same algorithms and produce the same results (to within roundoff

error) as those two built-in functions. All four functions—**rand** with or without an **n** and with or without a **tx**—have the same usage. With no arguments, the expression **randtx** or **randntx** generates a single uniformly or normally distributed pseudorandom value. With one argument, the expression **randtx(n)** or **randntx(n)** generates an **n**-by-**n** matrix. With two arguments, the expression **randtx(m,n)** or **randntx(m,n)** generates an **m**-by-**n** matrix.

The version 5 random number generators are available in newer versions of MATLAB by using the statement

```
rand('state',0)
```

or by using the function **rng**. See the documentation of **rng** for details.

It is usually not necessary to access or set the internal state of any of the generators. But if you want to repeat a computation using the same sequence of pseudorandom numbers, you can reset the generator state. By default, a generator starts at the state set by **randtx('state',0)** or **randntx('state',0)**. At any point during a computation, you can access the current state with **s = randtx('state')** or **s = randntx('state')**. You can later restore that state with **randtx('state',s)** or **randntx('state',s)**. You can also set the state with **randtx('state',j)** or **randntx('state',j)**, where **j** is a single integer in the range $0 \leq j \leq 2^{31} - 1$. The number of states that can be set by a single 32-bit integer is only a tiny fraction of the total number of states.

For the uniform generator **randtx**, the state **s** is a vector with 35 elements. Thirty-two of the elements are floating-point numbers between 2^{-53} and $1 - 2^{-53}$. The other three elements in **s** are small integer multiples of **eps**. Although they cannot all be reached from default initial settings, the total number of possible bit patterns in the **randtx** state is $2 \cdot 32 \cdot 2^{32} \cdot 2^{32 \cdot 52}$, which is 2^{1702} .

For the normal generator **randntx**, the state **s** is a vector with two 32-bit integer elements, so the total number of possible states is 2^{64} .

Both generators have setup calculations that are done only when the generator is first used or reset. For **randtx**, the setup generates the initial floating-point numbers in the state vector one bit at a time. For **randntx**, the setup computes the breakpoints in the ziggurat step function.

After the setup, the principal portion of the uniform generator **randtx** is

```
U = zeros(m,n);
for k = 1:m*n
    x = z(mod(i+20,32)+1) - z(mod(i+5,32)+1) - b;
    if x < 0
        x = x + 1;
        b = ulp;
    else
        b = 0;
    end
    z(i+1) = x;
    i = i+1;
    if i == 32, i = 0; end
```



```

    [x,j] = randbits(x,j);
    U(k) = x;
end

```

This takes the difference between two elements in the state, subtracts any carry bit **b** from the previous calculation, adjusts the result if it is negative, and inserts it into the state. The auxiliary function `randbits` does an XOR operation between the fraction of the floating-point number **x** and the random integer **j**.

After the setup, the principal portion of the normal generator `randntx` is

```

R = zeros(m,n);
for k = 1:m*n
    [u,j] = randuni;
    rk = u*z(j+1);
    if abs(rk) < z(j)
        R(k) = rk;
    else
        R(k) = randntips(rk,j,z);
    end
end
end

```

This uses a subfunction `randuni` to generate a random uniform **u** and a random integer **j**. A single multiplication generates a candidate result **rk** and checks to see if it is within the “core” of the ziggurat. Almost all of the time it is in the core and so becomes an element of the final result. If **rk** is outside the core, then additional computation must be done by the auxiliary subfunction `randtips`.

9.5 Twister

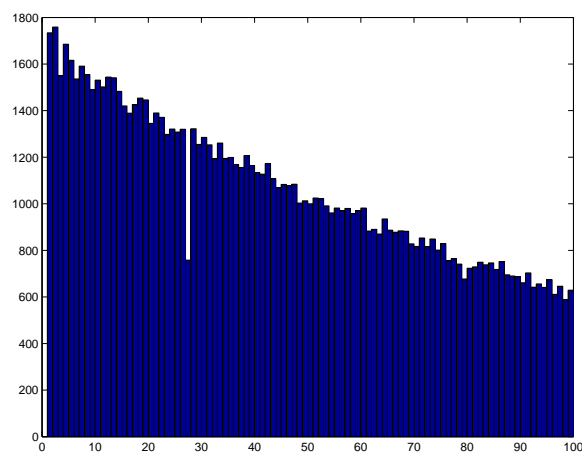


Figure 9.3. Histogram of run lengths.

Let's do a run length test using the uniform random number generator that we have described in this chapter and that is implemented in `randtx`.

```
rand('state',0)
x = rand(1,2^24);
delta = .01;
k = diff(find(x<delta));
t = 1:99;
c = histc(k,t);
bar(t,c,'histc')
```

This code sets the generator to its initial state and generates a few million uniformly distributed random numbers. The quantity `delta` is a cutoff tolerance. The quantity `k` is a vector containing the lengths of runs of random numbers that are all greater than the tolerance. The last three statements plot a histogram of the lengths of these sequences. The result is figure 9.3. The histogram generally confirms our expectation that longer runs are less frequent. But surprisingly, in this experiment, runs of length 27 are only about half as frequent as they ought to be. You can experiment with other initial states and tolerances and verify that this anomalous behavior is always present.

Why? The gap shown in figure 9.3 is a property of the “subtract-with-borrow” algorithm employing a cache of length 32 and a generator with offsets 20 and 5,

$$z_i = z_{i+20} - z_{i+5} - b.$$

The values 32 and 5 conspire to deplete runs of length 27. This is a curious, but probably serious, property of the algorithm.

Nevertheless, starting with version 7.4 in 2007, the default uniform random number generator in MATLAB uses an algorithm known as the *Mersenne Twister*, developed by M. Matsumoto and T. Nishimura, [10]. The generator uses a cache of size 624 and has a incredible period of length $2^{19937} - 1$. It produces floating point values in the closed interval

$$[2^{-53}, 1 - 2^{-53}]$$

In other words, it does not produce an exact 0 or an exact 1. For more details, enter

```
help rand
```

and see the papers referenced on the Mersenne Twister Web page, [10].

Exercises

9.1. The number 13 is usually regarded as unlucky. However,

```
rand('state',13)
randgui(@rand)
```

produces a surprisingly lucky result. What is that result?

- 9.2. Modify `randgui` so that it computes π using a circle inside a square instead of a sphere inside a cube.
- 9.3. In `randgui`, change the statement

```
X = 2*randfun(3,m)-1;
```

to

```
X = 2*randfun(m,3) '-1;
```

We've interchanged 3 and `m`, and inserted a matrix transpose operator. With this change,

```
randgui(@randssp)
```

no longer demonstrates the defect in `randssp`. Explain why.

- 9.4. A very fast random number generator can be based on an irrational number such as the golden ratio

$$\phi = \frac{1 + \sqrt{5}}{2}.$$

The sequence x_n in the interval $0 < x_n < 1$ is generated simply by

$$x_n = \text{fractional part}(n\phi).$$

You can generate this sequence by repeatedly executing the statement

```
x = rem(x + phi, 1)
```

This generator passes a few statistical tests, but does poorly on most.

- (a) Write a MATLAB function `randphi`, in the style of `randmcg` and `randssp`, that implements this generator.
- (b) Compare the histograms of `randmcg`, `randssp`, and `randphi`. Use 10,000 samples and 50 bins. Which of the generators has the “best” uniform distribution?
- (c) How well does

```
randgui(@randphi)
```

compute π ? Why?

- 9.5. The M-files `randtx.m` and `randntx.m` include an internal function `randint` that employs a sequence of bit shifts to produce random integers.
 - (a) Write a MATLAB function `randjsr`, in the style of `randmcg`, that uses the shift register integer generator to produce uniformly distributed floating-point numbers.
 - (b) Compare the histogram of your `randjsr` with the histogram of `randtx`. You should find that the two histograms have the same appearance.
 - (c) Verify that

```
randgui(@randjsr)
```

does a good job of computing π .

- 9.6. Write an M-file `randnpolar.m` that generates normally distributed random numbers using the polar algorithm described in section 9.3, Normal Distribution. Verify that your function produces the same kind of bell curve shaped histograms as `randn` and `randntx`.
- 9.7. The NCM M-file `brownian` plots the evolution of a cloud of particles that starts at the origin and diffuses in a two-dimensional random walk, modeling the Brownian motion of gas molecules.
 - (a) Modify `brownian.m` to keep track of both the average and the maximum particle distance from the origin. Using `loglog` axes, plot both sets of distances as functions of n , the number of steps. You should observe that, on the log-log scale, both plots are nearly linear. Fit both sets of distances with functions of the form $cn^{1/2}$. Plot the observed distances and the fits, using linear axes.
 - (b) Modify `brownian.m` to model a random walk in three dimensions. Do the distances behave like $n^{1/2}$?
- 9.8. The term *Monte Carlo simulation* refers to the use of pseudorandom numbers in computational models of stochastic or probabilistic phenomena. The NCM M-file `blackjack` provides an example of such a simulation. The program simulates the card game, either one hand, or thousands of hands, at a time, and collects payoff statistics.

In blackjack, face cards count 10 points, aces count 1 or 11 points, and all other cards count their face value. The objective is to reach, but not exceed, 21 points. If you go over 21, or “bust,” before the dealer, you lose your bet on that hand. If you have 21 on the first two cards, and the dealer does not, this is “blackjack” and is worth 1.5 times the bet. If your first two cards are a pair, you may “split” the pair by doubling the bet and use the two cards to start two independent hands. You may “double down” after seeing the first two cards by doubling the bet and receiving just one more card. “Hit” and “draw” mean take another card. “Stand” means stop drawing. “Push” means the two hands have the same total.

The first mathematical analysis of blackjack was published in 1956 by Baldwin, Cantey, Maisel, and McDermott [4]. Their basic strategy, which is also described in many more recent books, makes blackjack very close to a fair game. With basic strategy, the expected win or loss per hand is less than 1% of the bet. The key idea is to avoid going bust before the dealer. The dealer must play a fixed strategy, hitting on 16 or less and standing on 17 or more. Since almost one third of the cards are worth 10 points, you can compare your hand with the dealer’s under the assumption that the dealer’s hole card is a 10. If the dealer’s up card is a 6 or less, she must draw. Consequently, the strategy has you stand on any total over 11 when the dealer is showing a 6 or less. Split aces and split 8’s. Do not split anything else. Double down with 11, or with 10 if the dealer is showing a 6 or less. The program displays the recommended basic strategy play for each situation in red. The complete basic strategy is defined by three arrays, `HARD`, `SOFT`, and `SPLIT`, in the code.

A more elaborate strategy, called *card counting*, can provide a definite mathematical advantage. Card-counting players keep track of the cards that have appeared in previous hands, and use that information to alter both the bet and the play as the deck becomes depleted. Our simulation does not involve card counting.

Our **blackjack** program has two modes. The initial bet for each hand is \$10. “Play” mode indicates the basic strategy with color, but allows you to make other choices. “Simulate” mode plays a specified number of hands using the basic strategy and collects statistics. One graph shows the total stake accumulated over the duration of the simulation. Another graph shows the observed probabilities of the ten possible payoffs for each hand. These payoffs include zero for a push, win \$15 for a blackjack, win or lose \$10 on a hand that has not been split or doubled, win or lose \$20 on hands that have been split or doubled once, and win or lose \$30 or \$40 on hands that have been doubled after a split. The \$30 and \$40 payoffs occur rarely (and may not be allowed at some casinos) but are important in determining the expected return from the basic strategy. The second graph also displays with $0.xxxx \pm 0.xxxx$ the expected fraction of the bet that is won or lost each hand, together with its confidence interval. Note that the expected return is usually negative, but within the confidence interval. The outcome in any session with less than a few million hands is determined more by the luck of the cards than by the expected return.

- (a) How many decks of cards are used in our **blackjack** program? How is the deck represented and how is it shuffled? How are the cards dealt? What role does **rand** play?
- (b) What is the theoretical probability of getting blackjack from a freshly shuffled deck? In other words, the player has 21 on the first two cards and the dealer does not. How does this compare with the probability observed in the simulation?
- (c) Modify **blackjack** so that blackjack pays even money instead of 1.5 times the bet. How does this affect the expected return?
- (d) In some casinos, a “push” is regarded as a loss. Modify **blackjack** to use such a rule. How does this affect the expected return?
- (e) Modify **blackjack** to use four artificial 56-card decks that have twice as many aces as usual. How does this affect the expected return?
- (f) Modify **blackjack** to use four artificial 48-card decks that do not have any kings. How does this affect the expected return?

Bibliography

- [1] G. FORSYTHE, M. MALCOLM, AND C. MOLER, *Computer Methods for Mathematical Computations*, Prentice–Hall, Englewood Cliffs, NJ, 1977.
- [2] D. KAHANER, C. MOLER, AND S. NASH, *Numerical Methods and Software*, Prentice–Hall, Englewood Cliffs, NJ, 1989.
- [3] C. MOLER, *Numerical Computing with MATLAB*,
Electronic edition: The MathWorks, Inc., Natick, MA, 2004.
<http://www.mathworks.com/moler>
Print edition: SIAM, Philadelphia, 2004.
<http://www.ec-securehost.com/SIAM/ot87.html>
- [4] R. BALDWIN, W. CANTEY, H. MAISEL, AND J. McDERMOTT, *The optimum strategy in blackjack*, Journal of the American Statistical Association, 51 (1956), pp. 429–439.
- [5] D. E. KNUTH, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, Addison–Wesley, Reading, MA, 1969.
- [6] G. MARSAGLIA, *Random numbers fall mainly in the planes*, Proceedings of the National Academy of Sciences, 61 (1968), pp. 25–28.
- [7] G. MARSAGLIA AND W. W. TSANG, *A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions*, SIAM Journal on Scientific and Statistical Computing 5 (1984), pp. 349–359.
- [8] G. MARSAGLIA AND W. W. TSANG, *The ziggurat method for generating random variables*, Journal of Statistical Software, 5 (2000), pp. 1–7.
<http://www.jstatsoft.org/v05/i08>
- [9] G. MARSAGLIA AND A. ZAMAN, *A new class of random number generators*, Annals of Applied Probability, 3 (1991), pp. 462–480.
- [10] M. MATSUMOTO AND T. NISHIMURA, Mersenne Twister Home Page,
<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>
- [11] S. K. PARK AND K. W. MILLER, *Random number generators: Good ones are hard to find*, Communications of the ACM, 31 (1988), pp. 1192–1201.