

Editors: Isabel Beichl, isabel.beichl@nist.gov
 Julian V. Noble, jvn@virginia.edu



THE FULL MONTE

By Julian V. Noble

THIS IS MY INAUGURAL COLUMN AS COEDITOR OF COMPUTING PRESCRIPTIONS, ALTHOUGH NOT MY FIRST APPEARANCE HERE (I WROTE A GUEST COLUMN A COUPLE OF

years ago¹). So, a word of self-introduction is in order. Back in the summer of 1960, while interning at Grumman Aircraft, I was assigned to learn the new language Fortran so that I could program the company's brand-new IBM 704. The rules of engagement were arcane—you wrote out the program on an official IBM programmer's pad, submitted it to the keypunch operators, and (if the Force were with you) got back a punched deck several days later. Then the fun began. Imagine debugging on a system with three-day job turnaround!

Luckily, the next summer I got to program a Burroughs 220 (in machine language) and an IBM 7090 (in Fortran II) in a university setting. We were actually allowed to touch the keypunch machines ourselves (or in the Burroughs's case, the tape punch). And—oh, joy!—the turnaround time was “only” 24 hours! If you want some idea of what prehistoric computing was like, read Fred Hoyle's sci-fi novels, *The Black Cloud* (Lightyear, 1998) and *Ossian's Ride* (Harper, 1959)—both have chapters whose “action” takes place in computer centers.

The personal computer revolution began in the late 1970s. I soon discovered that getting a program to run using interactive Basic on a desktop machine (such as the HP-1000 or the NorthStar) was faster than on the big CDC-6600 mainframe using Fortran. I even published a paper² whose computations were all done on a Sinclair ZX-81 computer that ran overnight. It had a 4-MHz, 8-bit Z80 chip, with an interpretive ROM Basic and 16 Kbytes of RAM. It displayed 16 lines of 64 characters on a TV set and used a tape cassette player for storage. The personal computer was lots slower than a mainframe, but the answers came out much faster—a paradox we all live with now.

When I learned to program, computer science departments lay far in the future. For good or ill, I learned about programming through reading and practice. The upside is

that I acquired no linguistic prejudices along the way and felt free to try anything. The downside is that I am still ignorant of things for which I did not have an immediate use. In 1985, mainly through force of circumstance, I began to use Forth for most programming, preferring it to the various dialects of Fortran and Basic that had served me until then. Although I still occasionally dabble in C and Lisp, I now mostly use Forth and machine code.^{3,4} Although some might consider this eccentric, to me it seems no more so than the artificial machine language MIX that Donald Knuth illustrates algorithms with.⁵ Hold the flames; program fragments in this column will always be pseudocode—I won't slight anyone's pet language.

Forthcoming columns will explore prescriptions for a novel way to solve the Laplace equation, the right angle on keeping orthogonal functions orthogonal, how to do algebra on a computer, finding roots of analytic functions, and so on. These ideas are not engraved in stone: I welcome suggestions and comments, up to and including guest columns, and I am sure my coeditor, Isabel Beichl, feels the same.

Fun with uniform variates

This column is about computing based on various forms of random sampling, or Monte Carlo methods.⁶ To get the ball rolling, I illustrate with applications to evaluating integrals, and to simple simulation, before explaining the prescription that inspired the column. (You can find the Forth code for all the applications in this column, including the random-lookup table objects, at www.phys.virginia.edu/classes/551.jvn.fall01 under the link “Forth system and example programs” and sublink “Monte Carlo techniques.”)

Integration

Of course, regular *CiSE* readers already know how to evaluate integrals using Monte Carlo methods, but it can't hurt to begin with something simple. Monte Carlo integration works best for many dimensions (where it beats the heck out of repeatedly applying a quadrature rule), but for clarity, I work in only one dimension here. Consider Figure 1's graph of a continuous function. If you sprinkle points uniformly but randomly over the bounding rectangle (as a

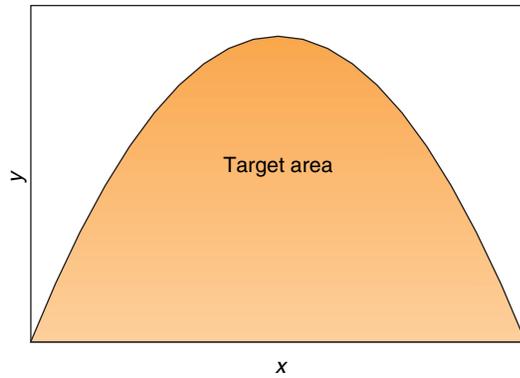


Figure 1. A typical function in its bounding box.

rainstorm would do), it seems intuitively clear that the ratio of the number n falling in the “target area” under the curve to the total number of points N approaches the ratio of the corresponding areas as N increases:

$$\frac{n}{N} \xrightarrow{N \rightarrow \infty} \frac{A}{A_{\text{box}}} \quad (1)$$

Because we know A_{box} , once we know n/N with sufficient precision we’re done. This is sometimes called *brute-force Monte Carlo*. The pseudocode might resemble Figure 2.

If we apply this code to the function $f(x) = x(1-x)$ on the interval $(0, 1)$, within the bounding box $0 \leq f \leq 0.5$, five runs (of 10^4 points each) produce 0.16840, 0.16465, 0.16705, 0.16770, and 0.16525. Their average is 0.16661, and their standard deviation is 0.0016. The exact value is $1/6$; that is, for this relatively smooth function, the standard error for one run is roughly 1 percent, which is just what we expect in a sample of 10^4 .

This brute-force approach is inefficient: it needs too many random numbers. Because a function’s average value over an interval is, by definition, its integral divided by the length of that interval, we have

$$\int_a^b dx f(x) \equiv (b-a)f_{\text{av}} \quad (2)$$

The trick is then to find an independent method for computing f_{av} , which is where random sampling comes in. The most direct approach picks N uniform variates x_k in (a, b) and defines

$$f_{\text{av}} \stackrel{df}{=} \frac{1}{N} \sum_{k=1}^N f(x_k) \quad (3)$$

This method is called *straight sampling* and has the advantage of requiring only one call per point to the built-in (pseudo) random-number generator rather than two calls as in the brute-force algorithm. In n dimensions, this is n versus $n+1$, so the advantage is not so great. (I assume your system contains a reasonably good algorithm for generating random floating-point numbers uniformly distributed on the interval $(0,1)$. By “reasonably good,” I mean it should

have a cycle length of at least 10^9 and should pass the usual tests of serial correlation.)

A second strategy—and this leads toward this column’s theme—is to choose the points where they will matter most. *Importance sampling*⁶ takes variates from some nonuniform distribution (see Equations 4 and 5) that mimics the function we want to integrate. If the known nonuniform distribution consists of step functions, we call the method *stratified sampling*. A related strategy uses *antithetic variates*, in which pairs of variates are anticorrelated with each other. Because

$$\begin{aligned} \text{Var} \left[\frac{1}{2} (f + f') \right] \\ = \frac{1}{4} \text{Var}(f) + \frac{1}{4} \text{Var}(f') + \frac{1}{2} \text{Cov}(f, f'), \end{aligned} \quad (4)$$

where (since they are anticorrelated) the covariance of the pair (f, f')

$$\text{Cov}(f, f') \stackrel{df}{=} \langle (f - \langle f \rangle) \cdot (f' - \langle f' \rangle) \rangle < 0 \quad (5)$$

is negative, the variance can be greatly reduced from straight sampling, meaning we need fewer points for a given probable precision. A physical example is the Buffon needle problem with perpendicular crossed needles, which I often assign as an exercise.

Batting practice

Simple simulations often use uniform distributions to represent the odds for discrete events. For example, we might want to know the incidence of batting slumps in a regular

```

Subroutine: inside?
  leave 1 if point (x,y) is inside the integration volume,
  0 if outside

Main: BruteForce
  Npoints = 0           (initialize)
  Nunder = 0
  BEGIN Npoints < Nmax (TRUE if inequality satisfied)
  WHILE                 (execute up to REPEAT if TRUE)
                        (otherwise jump to DONE)
    randomly choose new point
    Nunder = Nunder + inside?
    Npoints = Npoints + 1
  REPEAT
  DONE

```

Figure 2. Pseudocode for brute-force Monte Carlo integration.

```

\ Simulation of batting slumps

\ data structures
0.20e0 FVALUE p          \ batting average
300 VALUE N              \ # times @ bat/season
10 VALUE mu              \ a "slump" is at least mu strikeouts
0 VALUE hitless          \ current # of hitless @ bats
0 VALUE slumps           \ # slumps this season

\ program

function: hit?           \ it's a hit if Rand <= p
  Rand <= p              \ inequalities leave a flag

function: slump?        \ it's a slump if hitless > mu
  hitless > mu

subroutine: end_slump
  slump?                 \ was there a slump going?

  IF slumps = slumps + 1 ENDIF
  hitless = 0            \ reset-slump is ended

subroutine: at_bat
  N 0 DO hit?
    IF end_slump
    ELSE hitless = hitless + 1
    ENDIF
  LOOP

Main: stats (#cases --)
  LOCALS| #cases |
  hitless = 0            \ initialize
  slumps = 0
  #cases 0 DO at_bat
  LOOP
  #cases report         \ output results

```

Normal variates

What if you want to simulate a process for which the events are not representable by uniform variates in the interval (0, 1)? Because feet follow the normal (that is, Gaussian) distribution, a shoe manufacturer's statistician would sample normally distributed random variables. Here are two satisfactory algorithms.

First, choose 12 independent uniform variates $0 < \xi_k < 1$ and define

$$x = \sum_{k=1}^{12} \xi_k - 6. \quad (6)$$

It is easy to see that x has mean 0 and variance 1. The Central Limit theorem, which says that a sum of independent random variables is (to a good approximation) normally distributed, assures that x is a normal variate. In Fortran-ish pseudocode,

```

FUNCTION normal
  sum = -6e0
  DO 1, I=1,12 (add 12 PRNs)
    sum = sum + Rand
  1 CONTINUE
  normal = sum

```

Figure 3. Monte Carlo simulation of batting slumps.

baseball season of 162 games. If a batting slump is defined as μ at bats without a hit, then if a player bats, say, 300 times per season, how many slumps will he experience?

To simulate this Markov process, we assume the player's batting average p is his fixed probability of a hit each time at bat. With each event independent, our program resembles Figure 3. Typical runs for a 0.200 hitter give

```

10 stats
10 trials of 300 at bats, giving 55
  slumps. ok

10 stats
10 trials of 300 at bats, giving 66
  slumps. ok

```

and so forth. A 0.200 hitter has approximately 6 ± 0.8 slumps per season, whereas a 0.350 hitter would only experience 1.2 ± 0.2 . The moral seems to be that he who bats better slumps less—managers take note! Theory predicts roughly $(N - \mu)p(1 - p)^\mu$ per N at bats. For $N = 300$, $\mu = 10$, and $p = 0.2$, the formula predicts 6.2 slumps.

The Box-Muller algorithm⁷ is a useful alternative whose speed is comparable on some machines. Because

$$\iint dx dy e^{-(x^2+y^2)/2} \equiv \int d\theta \int d\rho e^{-\rho^2/2}, \quad (7)$$

if we choose θ uniformly distributed on $[0, 2\pi]$ and

$$\eta = e^{-\rho^2/2}, \quad (8)$$

uniformly distributed on (0,1), we generate two independent, normally distributed random variables:

$$a = \sqrt{-2 \ln \eta} \cos \theta \quad (9a)$$

$$b = \sqrt{-2 \ln \eta} \sin \theta. \quad (9b)$$

The algorithm in this form requires three transcendental functions (log, sin, and cos) and a square root to generate two normal variates, so it is a bit expensive compared with generating a dozen uniform variates. George Box and Mervin Muller discovered a useful trick, based on an earlier

```

Subroutine: Box_Muller
IF      LastCall = TRUE
THEN    return y
        set LastCall = FALSE
ELSE    BEGIN      (loop until u < 1)
        x = -1 + 2 * Rand
        y = -1 + 2 * Rand
        u = x^2 + y^2
        u < 1 UNTIL (end of loop)
        u = sqrt (-(2 * ln (u)) / u)
        x = x * u
        y = y * u
        return x
        set LastCall = TRUE
ENDIF

```

idea of John von Neumann's:⁸ if x and y are independent uniform variates on the interval $(-1, 1)$, the combination

$$u = [x^2 + y^2] \theta(1 - x^2 - y^2) \quad (10)$$

is uniformly distributed over the interval $(0, 1)$. Therefore a and b defined by

$$a = x \sqrt{-2 \ln u / u} \quad (11a)$$

$$b = y \sqrt{-2 \ln u / u} \quad (11b)$$

are independent normally distributed random numbers on the interval $(-\infty, \infty)$, each with mean 0 and variance $\sigma^2 = 1$. Now we only need compute approximately 2.5 uniform random numbers (strictly, $8/\pi$), one transcendental function, and one square root per two normal variates. Figure 4 shows the pseudocode. Because x and y are independent, we compute two variates every other time the subroutine is called. We then return one and save the other (to return on alternate calls). The global flag variable `LastCall` keeps track of where we are. On the average, we compute an unsatisfactory random pair (lying outside the circle) a tad less than 25 percent (that is, $1 - \pi/4$) of the time, so the efficiency is pretty

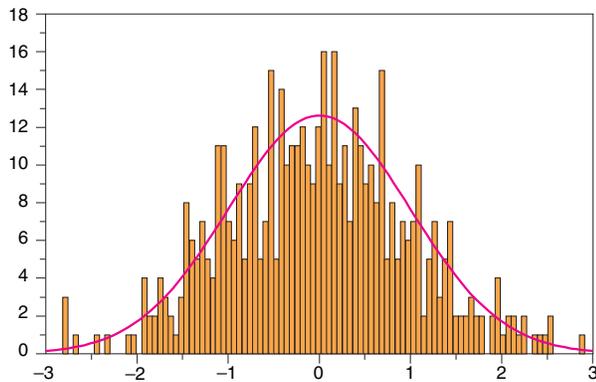


Figure 5. A histogram of 500 normal variates from the Box-Muller algorithm.

Figure 4. The Box-Muller algorithm for normal variates.

good. Figure 5 shows a sample of what the algorithm produces.

Arbitrary variates

Many physical processes are represented neither by uniform nor by normal variates. So, what do we do then? To sample from an arbitrary probability distribution, if the cumulative distributions for two sets of random variables are

$$P(x) = \int_{x_{\min}}^x p(x') dx' \quad (12a)$$

$$Q(\xi) = \int_{\xi_{\min}}^{\xi} p(\xi') d\xi' \quad (12b)$$

and if we set $P(x(\xi)) = Q(\xi)$, then our problem is to find the function $x(\xi)$. Pretend this is a known function, and differentiate both sides with respect to ξ : we get an ordinary differential equation,

$$p(x) = \frac{dx}{d\xi} = q(\xi), \quad (13)$$

that we could, in principle, integrate numerically. However, because the distribution at our disposal is typically the uniform one, the easiest thing to do is solve the transcendental equation

$$\int_{x_{\min}}^{x(\xi)} p(x') dx' = \xi \quad (14)$$

for a uniform variate ξ . Figure 6 is the result of inverting this equation, for the case $p(x) = xe^{-x}$.

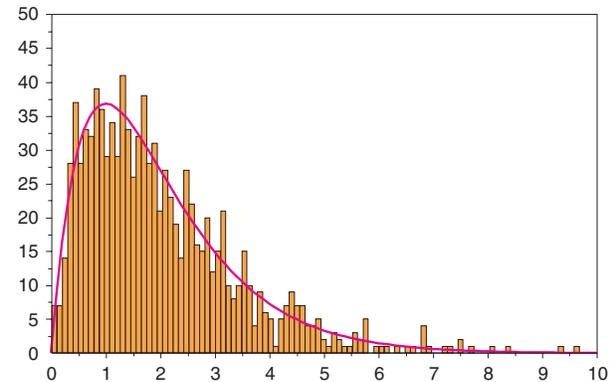


Figure 6. A histogram of 1,000 variates from xe^{-x} using inversion.

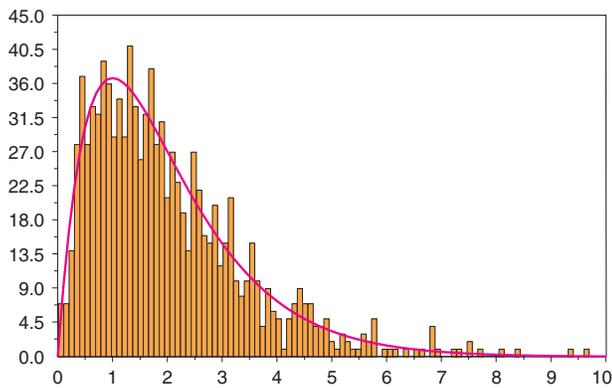


Figure 7. A histogram of 1,000 variates from xe^{-x} using rejection.

Alternatively, we could sample the distribution $p(x)$ using the rejection method:⁸ choose a random variable η from a uniform distribution on $(0, 1)$, and choose a value of x in the domain of $p(x)$ by appropriate scaling. Then, if

$$\eta p_{\max} \leq p(x), \tag{15}$$

we keep the point x ; otherwise, we reject it. (This resembles the Box-Muller algorithm, and works for the same reason.) Figure 7 shows a histogram of 1,000 such variates from the same distribution xe^{-x} .

Manifestly, the inverse and rejection methods both generate acceptable distributions. To illustrate, Figure 8 is output from a simulation of the Young two-slit experiment, showing how a diffraction pattern builds up from individual photons. (I have done versions using either inversion or rejection—there seems little difference in execution speed, although I had expected rejection to be significantly faster.) The probability distribution, in the direction perpendicular to the slits, is proportional to $\cos^2 x$. The screen shot contains 500 “photons.”

Random objects

All the preceding was preamble: I am now about to Prescribe. In a large simulation, both the rejection and inverse algorithms are generally far too slow. The solution is a random-lookup table. This is equivalent to replacing the actual distribution function $p(x)$ with a step-function approximation to it, as Figure 9 shows.

For many simulations, a histogram representation of the nonuniform distribution is perfectly adequate (the

technical term is “good enough for government work”). Depending on the available storage, we could sufficiently fine-grain the histogram so as not to incur any loss of “physical significance.” To construct such a representation, it is enough to create a table of N variates X_k by solving the equation

$$\int_{x_{\min}}^{X_k} p(x') dx' = \frac{k}{N}, k = 0, \dots, N - 1. \tag{16}$$

Now, in what sense does such a table represent the desired random process? If we sample it by generating random indices into the table, the variates we randomly sample will approximate variates drawn from the original distribution. For example, Figure 10 shows a histogram of 1,000 samples from the distribution $p(x) = xe^{-x}$, retrieved from a table with only 64 entries. The results are quite acceptable. And, of course, by doing it this way, we only have to generate one random integer and no expensive functions whenever we want a random variate. Although the table used in Figure 10 has only $64 = 2^6$ entries, my random-number generator has a cycle length of 2^{31} , so there are $2^{25} \approx 3 \times 10^7$ possible ways for it to traverse the table (a lot fewer than the $64! \approx 10^{89}$ possible orderings of a 64-entry table but big enough for most purposes). Even if you want a more fine-grained table, it gets constructed once, so the time consumed in solving the transcendental equation N times is unimportant.



Figure 8. A screen shot of 500 “photons” passing through parallel slits.

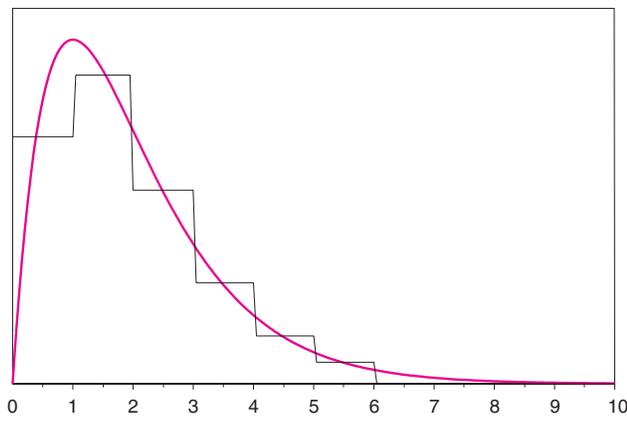


Figure 9. A histogram approximation of $p(x) = xe^{-x}$ by its average over equal intervals.

In Fortran or C, we must construct each random-lookup table by hand, define an array to hold the table, and then define a function that uses that array. The array must be filled before the program proper is executed by another subroutine written to do that. By contrast, extensible languages such as Lisp and Forth, as well as object-oriented ones such as C++, SmallTalk, or Forth, provide tools for defining constructors that can generate as many (independent) named lookup tables as we like. Each such table becomes an object—data packaged with code that automatically performs the table lookup—including, if desired, its own random-number generator. By using different generators (or different seeds) for different tables (even if they represent the same distributions), we can eliminate problems arising from serial correlations (that is, a given uniform generator might not be random enough, even though it passes standard tests).

I first used this technique to simulate the passage of high-energy particles through large atomic nuclei. An incident particle interacts differently with neutrons and protons; they in turn have their own momentum and spin distributions. What happens depends not only on the initial conditions but also on the branching probabilities for the various types of reactions that can occur. It gets very complex very quickly, and you'd have to repeat the "experiment" millions of times to obtain adequate statistics. It was helpful to be able to create named lookup objects for each different statistical distribution, which, when invoked, would return the appropriate sample variate. I hope you find this trick equally helpful. Happy computing!

As an afterword, I should note that at least two earlier Computing Prescriptions have discussed methods for generating variates from nonuniform distributions.^{9,10} Of particular relevance is Alastair Walker's "alias" method,^{11,12} explained by Donald Knuth¹³ and featured in Isabel Beichl and Francis Sullivan's column,⁹ because it uses a precomputed lookup table. ❧

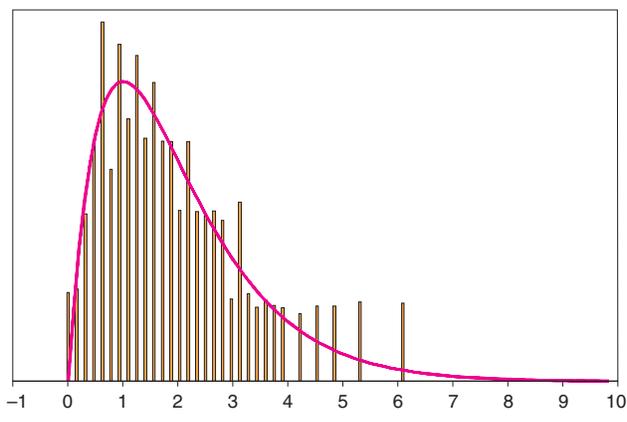


Figure 10. A histogram of 1,000 variates from xe^{-x} using a 64-entry random-lookup table.

References

1. J.V. Noble, "Gauss-Legendre Principal Value Integration," *Computing in Science & Eng.* vol. 2, no. 1, Jan. 2000, pp. 92–95.
2. J.V. Noble, "Missing Longitudinal Strength Cannot Reappear at Very High Energy," *Phys. Rev. C*, vol. 27, no. 1, Jan. 1983, pp. 423–425.
3. J.V. Noble, *Scientific Forth: A Modern Language for Scientific Computing*, Mechum Banks Publishing, Ivy, Va., 1992.
4. J.V. Noble, "Technology News & Reviews: Adventures in the Forth Dimension," *Computing in Science & Eng.*, vol. 2, no. 5, Sept. 2000, pp. 6–10.
5. D.E. Knuth, *The Art of Computer Programming*, 3rd ed., vol. 1, Addison-Wesley, Boston, 1997, p. 124.
6. J.M. Hammersley and D.C. Handscomb, *Monte Carlo Methods*, Methuen & Co., London, 1964, p. 57.
7. G.E.P. Box and M.E. Muller, *Annals Math. Statistics*, vol. 29, 1958, p. 610.
8. J. Von Neumann, *National Bureau of Standards Applied Mathematics Series*, vol. 12, 1951, p. 36.
9. I. Beichl and F. Sullivan, "Pay Me Now or Pay Me Later," *Computing in Science & Eng.*, vol. 1, no. 4, July/Aug. 1999, pp. 59–62.
10. W.J. Thompson, "Poisson Distributions," *Computing in Science & Eng.*, vol. 3, no. 3, May/June 2001, pp. 79–82.
11. A.J. Walker, "New Fast Method for Generating Discrete Random Numbers with Arbitrary Frequency Distributions," *Electronics Letters*, vol. 10, no. 8, 1974, pp. 127–128.
12. A.J. Walker, "An Efficient Method for Generating Discrete Random Variables with General Distributions," *ACM Trans. Math. Software*, vol. 3, no. 3, Sept. 1977, pp. 253–256.
13. D.E. Knuth, *The Art of Computer Programming*, 2nd ed., Addison-Wesley, Boston, 1981, vol. 2, pp. 115–119.

Julian Noble is a professor of physics at the University of Virginia's Department of Physics. His interests are eclectic, both in and out of physics. His philosophy of teaching computational methods is "no black boxes." He received his BS in physics from Caltech and his MA and PhD in physics from Princeton. He is a member of the American Physical Society, the ACM, and Sigma Xi. Contact him at the Dept. of Physics, Univ. of Virginia, PO Box 400714, Charlottesville, VA 22904-4714; jvn@virginia.edu.