Random Variables in Forth

Matthew M. Burke

There are many applications, particularly in simulation and modelling, where random variables are necessary. This paper presents a highlight of the concepts involved with random variables and their implementation in Forth. The coding is in F-PC 3.53 with the software floating point extension by Robert L. Smith.

Basic Concepts

Although the concept of a random variable has a specific mathematical definition, it is sufficient for our purposes to consider a random variable as a Forth word that takes no parameters and leaves an indeterminate value on the stack (note the departure from Forth tradition by calling a word that leaves a *value* on the stack, rather than an *address*, a "variable". I felt the term "random constant" was somewhat misleading). We will also use the term *generator* for a Forth word (or a procedure in another language) that returns random numbers.

Suppose we were modeling traffic at the corner of Main and Grand in Anytown. We use random variables **nnext** , **enext** , **wnext** , **snext** to give the time until the next north-bound (or respectively east-, west-, south-bound) vehicle. Depending on the level of detail desired, we could use a random variable to determine whether the next vehicle was a truck, car, motorcycle, or whatnot.

Associated with a random variable is a range and a density function. The *range* of a random variable is the set of its possible values. The *density function* gives an indication of how likely the occurrence of a given value is relative to other values. This is also referred to as the distribution of the random variable. Most people are familiar with the uniform distribution where every value is just as likely to occur as every other.

Any function whose definite integral over the variable's range is *one* can be considered a probability density. For the random variable with range [0,1] and uniform distribution, the density function is the constant *one* defined on the range [0,1]. If we wanted random numbers distributed uniformly over the range [0,10], the density function would be the constant function *one-tenth* since the integral of

```
f(x)=0.1
```

over the range [0,10] equals one.

Density functions need not be so simple. Examples of other density functions are the exponential:

```
f(x)=ae**(-ax)
```

where a is the number of events per unit time; the Poisson:

```
f(k)=e**(-l)*l**k/k!
```

where l is the number of events per unit time; and the normal:

```
f(x)=1/(2*pi)**(0.5)*s*e**(-1/2*(x-m/s)**2)
```

where s is the standard deviation and m is the mean.

Complications, Complications

Now for the fun part. The typical programming system has a *uniform random number generator*. It is not that common to find a generator for other probability densities such as the Poisson or exponential. One can, however, generate numbers from most distributions by manipulating a uniform random variable. First, we look at some techniques for generating specific distributions. Then we show how to generate random numbers for any distribution by a technique known as the *rejection method*. In what follows, a random variable with probability density g is called a g-distributed random number. Theoretical justification for the techniques can be found in most good books on statistics[2,3]. It is assumed that the uniform random number generator returns values in the open interval from zero to one since many techniques (particularly for generating exponentially distributed random numbers) blow up at zero.

The exponential distribution is used to simulate time between events that occur with a certain frequency. For example, the time between customers entering a bank queue is usually simulated with an exponential distribution. In the traffic simulation mentioned above, the times between vehicles are typically simulated using exponential distributions. Given a random variable r with uniform distribution, let

```
p = -ln (r) / c
```

where ln is the natural logarithm function. Then p is exponentially distributed (i.e. its density function is

```
c*e**(-cx))
```

In the Forth word, **exponentialvar** (listing 1), frequency is a floating point constant that plays the role of c in the previous equation.

(Listing 1)

```
: exponentialvar  ( -- F: fl )
  uniform fln fnegate frequency f/ ;
```

The Poisson distribution is a discrete distribution, i.e. a Poisson random variable only assumes integer values. The Poisson distribution describes the number of events that occur when the events have a certain frequency of occurrence.

We briefly describe **poissonvar** (listing 2): using an indefinite loop structure we generate a uniformly dis-

tributed random number and multiply it by the previous product while keeping track of the number of iterations. The loop terminates when the product is less than e**(-lambda). The number of iterations is a Poisson-distributed random number.

(Listing 2)

```
: poissonvar  ( --    r1 )
  \ lambda is negative
  0 lambda fexp 1.0
  begin uniform f* fover fover f<
  while 1+
  repeat fdrop fdrop ;
```

The normal distribution is used for simulating things such as the spread of measurements. We will use a method known as the "sum of twelve" to generate normal random numbers. We proceed as follows: sum 12 uniform random numbers and subtract six, this gives a normal distribution around zero with a variance of one. Multiply by the desired standard deviation and add the desired mean. An example is given in listing 3.

(Listing 3)

```
: normalvar  ( -- F: r1 )
  0.0 12 0
  do uniform f+
  loop 6.0 f- sigma f* mean f+ ;
```

A binomial distribution arises from repeated trials of a success/fail event. For example, if we ask what is the probability of k heads when a penny is tossed n times. To generate a binomial distributed random number we need to know the probability of a success and the number of trials. For a penny toss, the success probability is 0.5. Now consider five tosses of a penny. To generate the binomially distributed value, we generate five uniform numbers. If a number is <= 0.5 it is a success. Now, we tally up how many successes (0-5) and this is our binomially distributed value. The binomial distribution, like the Poisson distribution, is a discrete distribution.

(Listing 4)

```
: binomialvar  ( -- r )
  0 5 0
  do uniform 0.5 f>=
    if 1+ then
  loop ;
```

The above techniques work for specific distributions. But what do you do if you have some *arbitrary* distribution? The rejection method can be used to generate random numbers with respect to any distribution and all that is necessary is a uniform generator. The price is that generating n g-distributed random numbers requires generating an indeterminate number of uniformly distributed random values.

Suppose we have a density function g over the interval [a,b]. Let m be the maximum value of g(x) on [a,b]. Next we generate two uniform random numbers, u1 and u2, on the interval [0,1]. We then scale u1 to the range [a,b] by the following:

u1' = a+(b-a)*u1

and compute

d=g(u1')/m

If u2 <= d then we keep u1' as a random number with respect to density g . If u2 > d then we reject u1 and start over again.

A concrete example will make things clear. We define a density function

g(x) = .04*x*e**(-.02*x**2)

Now we have to cheat slightly since we need a finite range for scaling. The integral of g on the range 0 to infinity is one. We'll limit the range to [0,20] since the integral of g over the range [0,20] is very close to one. We let m = 0.04, the maximum of g on [0,20]. Now we generate two uniform random numbers, say 0.13 and 0.965. We scale 0.13 by multiplying by 20 and compute g(2.6)/0.04 = z, approximately 0.8735. Now since 0.913 > z, we REJECT x. So, we generate two new uniform random numbers, say 0.67 and 0.531. Since 0.531 < g(13.4)/0.04, 0.67*20 = 13.4 is a g-distributed random number.

(Listing 5)

```
: g()  ( F: x -- F: g(x))
  fdup fdup f* -0.02 f* fexp 0.04 f* f* ;
: rejectionvar  ( -- F: r )
  begin uniform uniform 20.0 f* fswap
    fover g() m f/ f>=
  while fdrop repeat ;
```

One of the strong points of Forth is the ease of factoring and we are going to apply that now. What we will do is create defining words so that we can factor out the actual values of the parameters. The Forth words in listing 6 are for use in applications. By using these defining words we can emphasize the similarities between the random quantities used in an application.

(Listing 6)

```
: exponential  ( F: r1 -- F: r2 )
  uniform fln fnegate fswap f/ ;
: poisson ( F: l -- F: r1 )
  \ l is the lambda value .. it must be
  \ negative
  0 fexp 1.0
  begin uniform f* fover fover f<
  while 1+ repeat fdrop fdrop ;
```

```
: binomial  ( F: s; n1 -- n2 )
  \ s is the success fraction
  \ n1 is the number of reps
  0 swap 0
  do uniform fover f< if 1+ then
  loop fdrop ;
: normal ( F: m s -- F: r )
  0.0 12 0
  do uniform f+
  loop 6.0 f- f* f+ ;
: rejection  ( a1 a2 -- F: r )
  \ a1 is address of the scaling word
  \ a2 address of word
  \ that computes g(x)/m
  begin uniform uniform over execute
    fswap fover dup execute f>=
  while fdrop
  repeat 2drop ;
: scale ( F: a b-a -- )
  create f, f,
  does> dup f@ f* f#bytes + f@ f+ ;
: pvar
  create f,
  does> f@ poisson ;
: evar
  create f,
  does> f@ exponential ;
: bvar
  create f, ,
  does> dup f@ f#bytes + @ binomial ;
: nvar
  create f, f,
  does> dup f@ f#bytes + f@ normal ;
: rvar
  create , ,
  does> dup @ >r 2 + @ r> rejection ;

3.3 pvar  customers
4.3 pvar  fish
4.4 evar  eastbound
2.34 evar westbound
3 0.5 bvar heads
0.0 20.0 scale [0,20]

: g(x)/m ( F: u -- F: g(u)/m )
  fdup f* -0.02 f* fexp ;

' [0,20] ' g(x)/m rvar myRandomVar
```

In each case, the defining word creates a header and commas the particular parameter values into the dictionary. The **does>** clause invokes the appropriate generating function. A few words are in order about the defining word for random variables using the rejection technique. The two parameters are the CFAs of a scaling word and a function word. The scaling word takes a uniform random number and scales it to the appropriate range. We give a definition for the Forth word **scale** which is a defining word for scaling words. To create a scaling word for the range [a,b] call **scale** with inputs a and b-a. The functional word computes the density function on its input and must also divide by the maximum of the density function on the random variable's range.

The Sample Application

We now describe a sample application using random variables. The Forth words **sim** and **sim'** in listing 7 are stochastic population simulations with **sim'** using a more complicated growth rate. The guiding design principle is that only one event (a birth or a death) is allowed to occur at any given instance. Thus, if b and d are the birth and death rates respectively, then the net growth rate is b-d and the number of events per unit time is b+d. An exponential random variable with parameter l=b+d gives the time until the next event. At this point, we use a binomial random variable (n=1) to determine whether the event was a birth or a death, and increment or decrement the population as appropriate.

In **sim'** the birth and death rates are bN and dN, respectively, where N is the current population. Note in both cases, the s uccess probability for event type is the same since it is the ratio of births to total events. This is b/(b+d) for **sim** and bN/(bN+dN)=b/(b+d) for **sim'**.

Listing 7

```
anew popsim
fload random floats
fvariable time
5.0 fconstant maxtime
variable population
10 constant population0
\ initial population

6.9 fconstant births \ per unit time
2.4 fconstant deaths \ per unit time
births deaths f+ fconstant lambda
\ lambda = events/time

: lambda' ( -- F: fl )
  \ more realistic (complicated)
  births population @ ifloat f*
  \ model
  deaths population @ ifloat f* f+ ;

births lambda f/ fconstant success%
\ births / total events

lambda evar nextevent
' nextevent >body constant paddr success%
1 bvar eventtype
\ 1 = birth, 0 = death
```

```
\ only increase/decrease population
\ if it's non-zero
: +pop ( -- )
  population @ 0>
  if population incr then ;
: -pop ( -- )
  population @ 0>
  if population decr then ;
: .report ( -- )
  time f@ 3 9 f.r population @ 13 .r cr ;
: .heading ( -- )
  cls cr 5 spaces
  ." Time" 3 spaces
  ." Population" cr 5 spaces
  ." ----" 3 spaces
  ." ----------" cr .report ;

: sim ( -- )
  population0 population !
  0.0 time f! .heading
  begin time f@ nextevent f+ fdup
    time f! maxtime f<
  while eventtype
    if +pop else -pop then .report
  repeat .report ;

: sim' ( -- )
  population0 population !
  lambda' paddr f!
  0.0 time f! .heading
  begin time f@ nextevent f+ fdup
    time f! maxtime f<
  while eventtype
    if +pop else -pop then .report
    lambda' paddr f!
  repeat .report ;
```

Conclusion

This article is a brief overview of generating techniques for random variables. No theory has been developed and there are several areas for improvement. For instance, the "sum of twelve" technique is not recommended for serious applications. It was included, however, for ease of implementation. Additionally, the above words could be better factored and more efficiently written, and the material could be rewritten to use fixed-point, rather than floating point, arithmetic for faster operation. As a final caveat, the above techniques give random sequences that are only as good as the uniform random numbers that they are based upon. Thus, it is important to have the best uniform generator that you can (see Dagpunar 88 for a good discussion of uniform random number generation).

Random variables represent a good example of the power of Forth. We have easily added a family of new data types that allow s us to hide the complexities of implementation and emphasize the similarities between different random quantities.

References

1. Bulgren, William, A Computer-Assisted Approach to Elementary Statistics: Examples and Problems, Wadsworth Publ ishing, Co., Belmont, California, 1971.

2. Dagpunar, John, Principles of Random Variate Generation, Clarendon Press, Oxford, 1988.

3. Devroye, Luc, Non-uniform Random Variate Generation, Springer-Verlag, New York, 1986.

Matthew Burke is a graduate student in both Computer Science and Mathematics at Washington State University. His main research interest is Ecological Modeling. Current Forth projects include: developing a library for generating random variables using state-of-the-art techniques and a system for teaching abstract algebra by hands-on experimentation. He can be reached by conventional mail at: Computer Science Department, Washington State University Pullman, WA 99164-2752; or by email as mburke@yoda.eecs.wsu.edu.

continued from page 4:

SIGForth '92 Program Highlights:

Wednesday, March 4th, pre-conference tutorials:
8:30 am- 12 noon *ShBoom:* "Damn fast and Dirt Cheap" 100+ Mhz stack-based RISC processor.
1:30 pm- 5:00 pm *Open Boot:* CPU independent Forth technology from Sun Microsystems.

Thursday, March 5th, CSC/CSE/SIGForth Joint Sessions:
10:15 am Tutorial: Introduction to Forth.
1:30 pm Keynote address by Charles Moore, Forth
3:30 pm inventor, followed by technical papers. Panel session: "From the Classroom to the Real World".
7:00 pm Informal Working Groups.

Friday, March 6th:
8:30 am Guest Speaker, Mr. Mike Wong of IBM on Forth Project Management, followed by technical papers.
3:30 pm ANS Forth Roundtable: The State of the Standard.
5:15 pm SIGForth Business Meeting.
7:00 pm Social Session.

Saturday, March 7th:
8:30 am Technical Papers.
7:00 pm Survivors Session.

For paper submission and information contact:

Dr. Paul Frenger, Program Chair
(address / phone number given on inside front cover)