# Generating Random Numbers Efficiently

Erik Postma
Senior Software Architect, Mathematical Software
Maplesoft

Generating (pseudo-)random values is a frequent task in simulations and other programs. For some situations, you want to generate some combinatorial or algebraic values, such as a list or a polynomial; in other situations, you need random numbers, from a distribution that is uniform or more complicated. In this article I'll talk about all of these situations.

## Reproducibility

By default, Maple's random numbers are reproducible: that is, Maple always generates the same sequence of random numbers after a restart of the kernel. This helps a lot when you're developing a new program and you want to track down a problem: it's a nightmare if that problem keeps changing on you! For example, in Maple 18 the first value returned by the rand command is this:

```
restart
rand( )
```

$$395718860534 \quad (1.1)$$

```
restart
rand( )
```

$$395718860534 \quad (1.2)$$

However, there are also situations where you want your random numbers to be unpredictable. In that case you can use the randomize function. It sets the seed of the internal random number generator to the given argument, or if no argument is given, to a value that depends on the system clock.

```
restart
randomize( ) :
rand( )
```

$$425501998829 \quad (1.3)$$

## Combinatorial and algebraic values

The RandomTools package contains many commands related to random number generation.

In particular, it contains the Generate command: the easiest command for generating combinatorial and algebraic values. You give the command a so-called 'flavor' of random objects you are interested in, and it returns a value accordingly. One of the simplest examples is the integer 'flavor', used, for example, as follows to get an integer between $-10$ and $10$:

```
with(RandomTools) :
Generate( 'integer'( 'range' = -10 .. 10) )
```

$$-7 \tag{2.1}$$

Now that's not particularly interesting - but what makes this system powerful is that you can compose multiple flavors into what the documentation calls a structured flavor. That is, there exist 'higher level' flavors such as list: it will generate a list of a specified length, generating independent values according to a flavor you specify. (This is an idea that occurs in a number of situations in Maple -- we have structured types and structured verifications, as well.) For example, you can generate lists of five integers between $-10$ and $10$, as follows:

```
Generate( 'list'( 'integer'( 'range'=-10 .. 10), 5) )
```

$$[9, -2, -9, -1, 4] \tag{2.2}$$

Or we can generate a polynomial in $x$ of degree 5, where each coefficient is one of $a, b, c$:

```
Generate( 'polynom'( 'choose'( [a, b, c]), x, 'degree' = 5) )
```

$$b x^5 + a x^4 + b x^3 + b x^2 + b x + a \tag{2.3}$$

A useful feature of the *Generate* command is its *makeproc* option. This option can be used to save some preprocessing time if you want to frequently generate random structures according to a single flavor. If you supply this option, the *Generate* command doesn't return the random structure that you're interested in, but rather a procedure that can generate such a structure. For example, suppose we want to generate a large number (maybe $10^4$) of lists, each with one integer chosen uniformly between $-10$ and 0, and one integer chosen uniformly between 0 and 10. This is described by the flavor $[integer(range = -10..0), integer(range = 0..10)]$. We could simply call *Generate* many times:

```
> tt := time():
  to 10^4 do
    Generate(['integer'('range' = -10 .. 0), 'integer'('range' = 0
  .. 10)]);
  end do:
  time() - tt;
```

$$0.732 \tag{2.4}$$

On my desktop machine, that took about 0.72 seconds. However, if we ask *Generate* to make a procedure (doing all the preprocessing only once) and then call that procedure $10^4$ times, it's a bit faster:

```
> tt := time():
  p := Generate(['integer'('range' = -10 .. 0), 'integer'('range'
  = 0 .. 10)], 'makeproc'):
  to 10^4 do
    p();
```

```
end do:
time() - tt;
```

0.529 **(2.5)**

About 0.56 seconds. It would be a tiny bit faster if we generate all these lists in one big list; we can do that as follows:

```
> tt := time():
  Generate('list'(['integer'('range' = -10 .. 0), 'integer'
  ('range' = 0 .. 10)], 10^4)):
  time() - tt;
```

0.556 **(2.6)**

However, as we will see in a bit, these methods are still most useful for their convenience, rather than their performance.

# Numbers

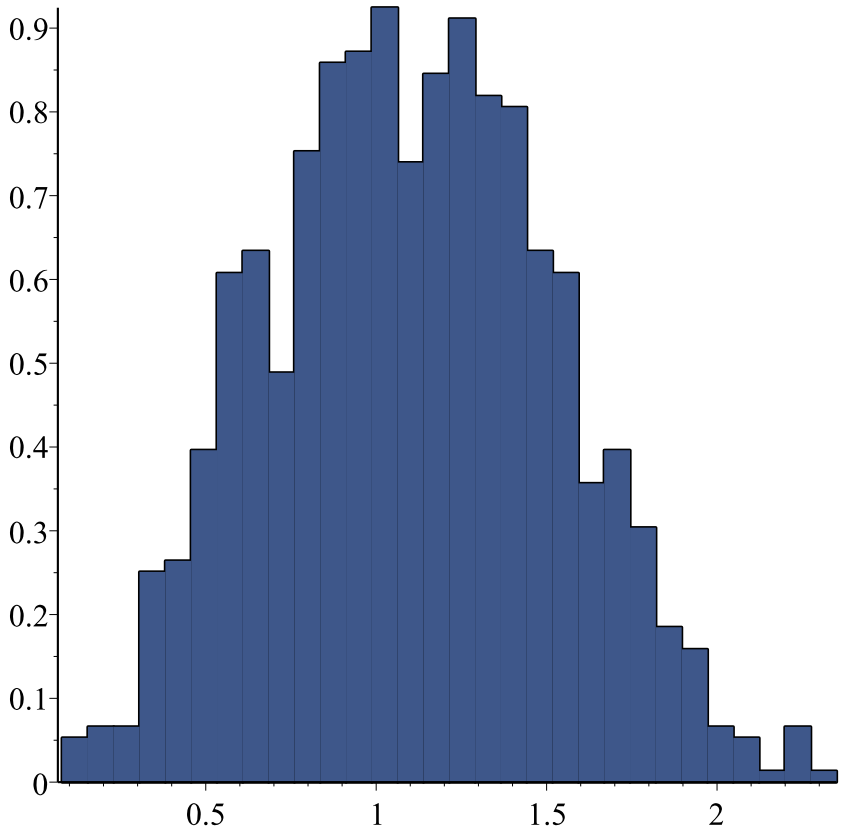
The best code for generating pseudorandom numbers according to a probability distribution is in the *Statistics* package. The Sample command takes a probability distribution or random variable, and a requested number of values, and creates a Vector (or Matrix or Array) and fills it with the generated numbers. Here's an example, generating 3 numbers uniformly in the interval from 0 to 1:

```
with(Statistics):
Sample('Uniform'(0, 1), 3)
```

$$\left[\begin{array}{ccc} 0.302160150919349 & 0.107134882186657 & 0.0718947197794679 \end{array}\right] \quad \textbf{(3.1)}$$

For a more complicated example, you can define a random variable and then generate samples of functions of that variable. For example, if $Z$ is distributed according to the chi-squared distribution with parameter 2, then we can generate samples of $\sqrt[3]{Z}$ as follows:

```
Z := RandomVariable('ChiSquare'(2)):
s := Sample(Z^(1/3), 1000):
Histogram(s)
```




If you want only a single number, you can of course index the resulting Vector by the number 1.

*Sample*( *'Uniform'*(*0*, *1*), *1*)[*1*]

0.769150119312673 **(3.2)**

However, if you do this many times, there is quite a bit of overhead. Let's see how much exactly.

```
> tt := time():
  to 10^4 do
    Sample('Uniform'(0, 1), 1)[1];
  end do:
  time() - tt;
```

3.652 **(3.3)**

```
> tt := time():
  Sample('Uniform'(0, 1), 10^4):
  time() - tt;
```

0. **(3.4)**

On my desktop machine, I sometimes get a result of 0 seconds, sometimes of 0.004. Overall, it looks like this is a speedup by more than a factor 1000; quite substantial indeed!

The Sample help page explains a few more calling sequences for situations where you really want to squeeze the last bit of performance out of your computer: one that can fill a pre-existing Vector, which can help in reducing garbage collection time, and another that returns a procedure, similar to the *makeproc* option to the *Generate* command in the *RandomTools* package. But for most situations, the techniques we've discussed so far are enough.

As a final application, let us set about reproducing the same list of lists we saw in the section about the *RandomTools* package: $10^4$ lists of 2 elements each where the first is a uniformly generated integer from $-10$ to 0, and the second from 0 to 10. The *Sample* command can only generate Vectors, Matrices, and Arrays, not lists; so we will generate a $10^4 \times 2$ Matrix, and convert it to the desired list of lists at the end. Also, we can supply *Sample* with only a single distribution at a time, but that's not a problem: we will generate 20000 numbers from 0 to 10, and then flip the sign of half of them. Another issue is that the data type is float for all values coming out of the *Sample* command; we will fix that at the same time. The final code looks like this:

```
> tt := time():
  m := Sample('DiscreteUniform'(0, 10), [10^4, 2]);
  m[.., 1] := -m[.., 1];
  m := Matrix(m, 'datatype'='integer'[4]);
  result := convert(m, 'listlist'):
  time() - tt;
```

$$m := \begin{bmatrix} \text{10000 x 2 Matrix} \\ \text{Data Type: float}_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

$$m_{(\,)\,..(\,),\,1} := \begin{bmatrix} \text{1 .. 10000 Vector}_{\text{column}} \\ \text{Data Type: float}_8 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

$$m := \begin{bmatrix} \text{10000 x 2 Matrix} \\ \text{Data Type: integer}_4 \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

$$0.040 \qquad \textbf{(3.5)}$$

We see that this method is a bit more work than the one using *RandomTools*, but it is a lot faster.