

Computational Statistics with Application to Bioinformatics

Prof. William H. Press
Spring Term, 2008
The University of Texas at Austin

Unit 5: More on Random Deviate Generation

Unit 5: More on Random Deviate Generation (Summary)

- Derive the theory behind Xorshift random generators
 - matrix representation
 - large matrix powers by successive squaring
- Look at two good empirical tests of randomness
 - GCD test
 - Gorilla test
- Learn several methods for generating random deviates from arbitrary distributions
 - transformation method
 - rejection method
 - ratio-of-uniforms method (“teardrops”)
 - especially when combined with squeezes, e.g. Leva’s Normal algorithm

The biggest period we can hope for is 2^N-1 , since the zero vector is its own cycle of length 1. Test for this “full period” by

$$\mathbf{M}^{2^N - 1} = \mathbf{1}$$
$$\mathbf{M}^{(2^N - 1)/f} \neq \mathbf{1}$$

for every prime factor f of 2^N-1 (Why just these?)

These huge powers of M are actually easily done by successive squaring

$$\mathbf{M}^2, \mathbf{M}^4, \mathbf{M}^8, \dots$$

and then assembling from the binary representation of the desired power.

(Show an example.)

```
FactorInteger[2^32 - 1]
```

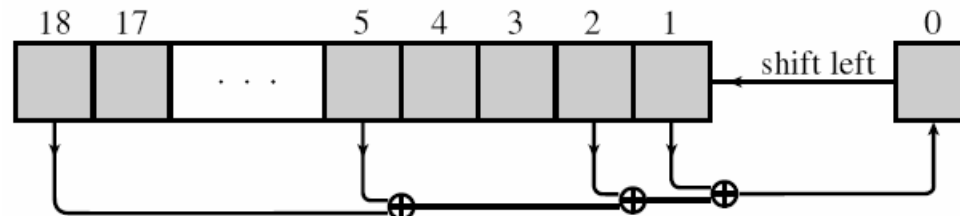
```
{{3, 1}, {5, 1}, {17, 1}, {257, 1}, {65537, 1}}
```

```
FactorInteger[2^64 - 1]
```

```
{{3, 1}, {5, 1}, {17, 1}, {257, 1},  
 {641, 1}, {65537, 1}, {6700417, 1}}
```

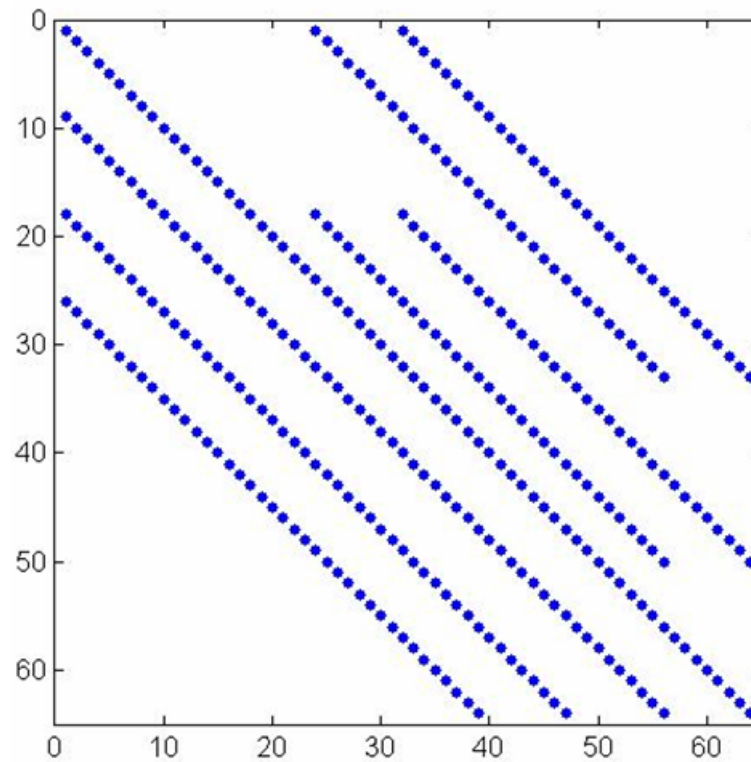
Now, finally, just a brute-force loop over all the possible values k_1, k_2, k_3 . Select full period ones. Do empirical tests.

(Can use this same method to find full-period linear shift register taps = primitive polynomials mod 2.)



Bit mixing for an actually good (N=64) Xorshift:

```
n = 64;  
[i , j ] = ndgrid(1: n, 1: n);  
S = @(k) eye(n, n) + (i -j == k)  
spy(S(17)*S(-31)*S(8))
```



Examples of powerful empirical tests of randomness (Marsaglia and Tsang, 2002)

The gcd test: Compute the gcd of successive random pairs

Euclid's algorithm for gcd:

$$\begin{array}{rcl} 366 & = & 1 \cdot 297 + 69 \\ 297 & = & 4 \cdot 69 + 21 \\ 69 & = & 3 \cdot 21 + 6 \\ 21 & = & 3 \cdot 6 + 3 \\ 6 & = & 2 \cdot 3 + 0 \end{array} \left. \begin{array}{l} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \right\} \begin{array}{l} \\ \\ k=5 \\ \\ j=3 \end{array}$$

$$k \sim \text{Normal}(\mu, \sigma^2) \quad \text{empirical result (what you actually do is measure using a known-good generator)}$$
$$\mu \approx 0.842 \ln(n) + 0.653, \quad \sigma^2 \approx 0.515 \ln(n) + 0.166$$

$$P(j) = 6/(\pi^2 j^2) \quad \text{theoretically calculable}$$

count in ~100 bins each for j and k
chi-square test the results

The gorilla test:

From a random sequence of 2^N words, form the sequence of 2^N bits from some particular bit position. The number of times each unique N-bit word should appear somewhere in this sequence is $\sim \text{Poisson}(1)$. So $P(0) = e^{-1}$ and the number of words that don't appear is

$$M \sim \text{Binomial}(2^N, e^{-1})$$

$$\mu = 2^N e^{-1}$$

$$\sigma = \sqrt{2^N e^{-1} (1 - e^{-1})}$$

this is actually not quite right.
Computer scientists can you see why? (Think Boyer-Moore!)

Example: N=4 on a 12-bit generator, testing position 10

```

010010100101
101001010001
011010101110
111010111010
101000010101
010101001010
010100100101
001001000010
101010101010
101010010101
010010001001
000001111100
101010100101
010100101010
010000100101
011110100101
    
```

} these bits
add 1 to bin
4

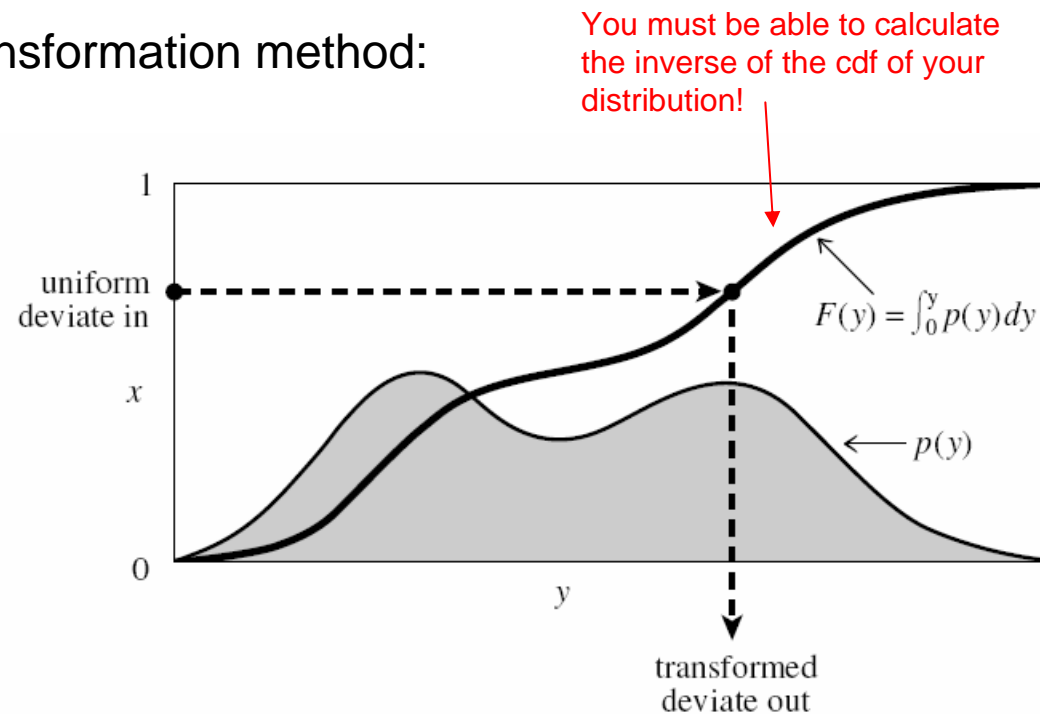
(a realistic N might be 20 to 30, 10^6 to 10^9 bins)

Editorial: In the “old days” it was good enough to convert the N bit integer to a float value between 0 and 1, then test randomness of the resulting “real” values (in various ways). Nowadays, people depend on genuine bitwise randomness, so modern tests look much more number-theoretical, even when their basis is still empirical.

Random deviates from other univariate distributions

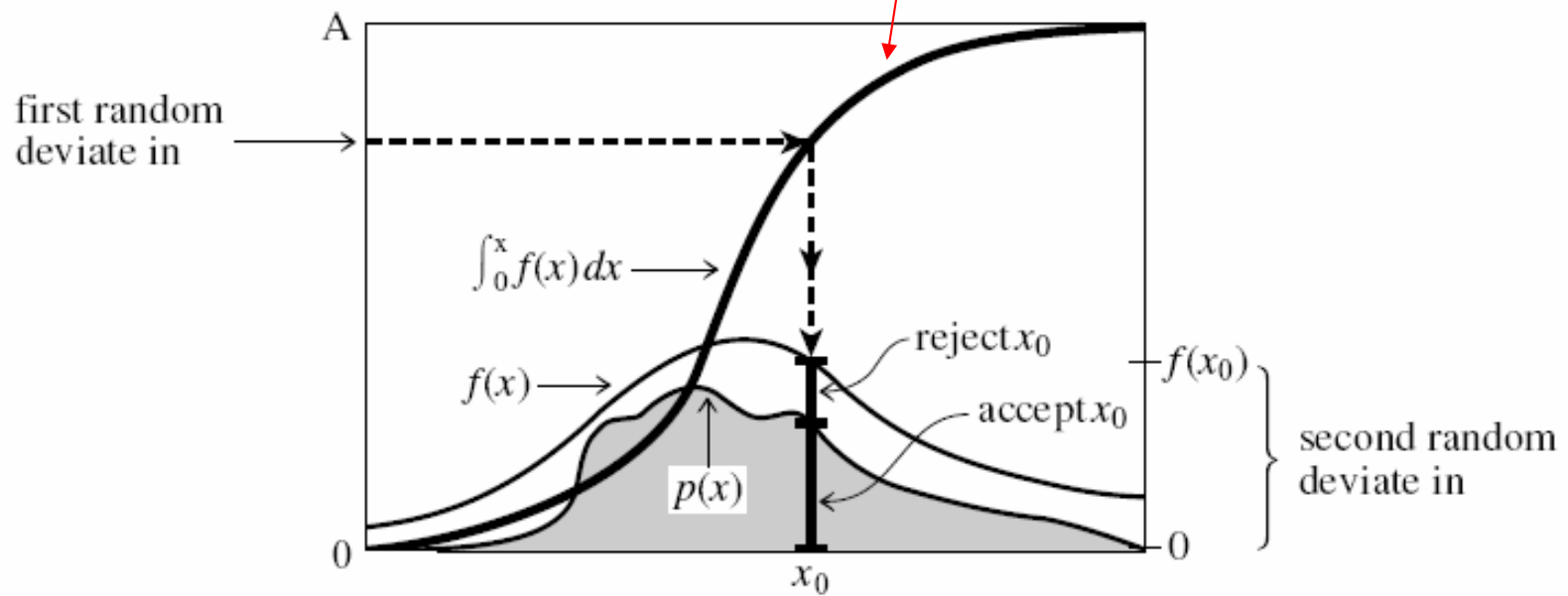
There are lots of nice algorithms specific to common distributions, but there are also some fairly general methods:

Transformation method:



Rejection method:

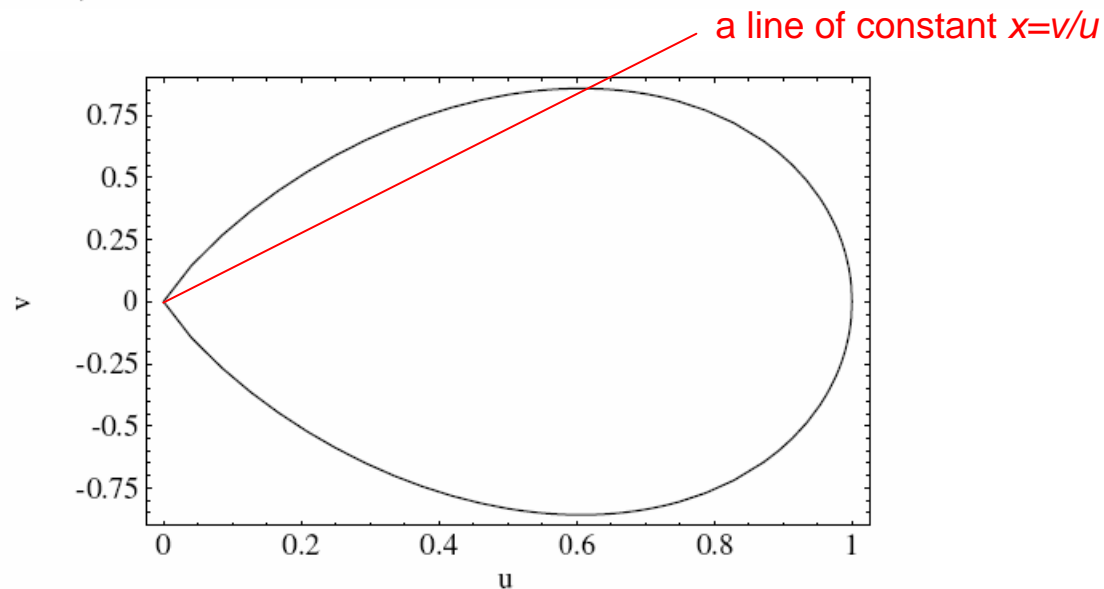
You must have a bounding function for which you are able to calculate the inverse of the cdf!



Ratio of Uniforms Method

(some of the best features of both xformation and rejection)

- Construct the region in the (u, v) plane bounded by $0 \leq u \leq [p(v/u)]^{1/2}$.
- Choose two deviates, u and v , that lie uniformly in this region.
- Return v/u as the deviate.



Proof: We can represent the ordinary rejection method by the equation in the (x, p) plane,

$$p(x)dx = \int_{p'=0}^{p'=p(x)} dp' dx \quad (7.3.18)$$

Since the integrand is 1, we are justified in sampling uniformly in (x, p') as long as p' is within the limits of the integral (that is, $0 < p' < p(x)$). Now make the change of variable

$$\begin{aligned} \frac{v}{u} &= x \\ u^2 &= p \end{aligned} \quad (7.3.19)$$

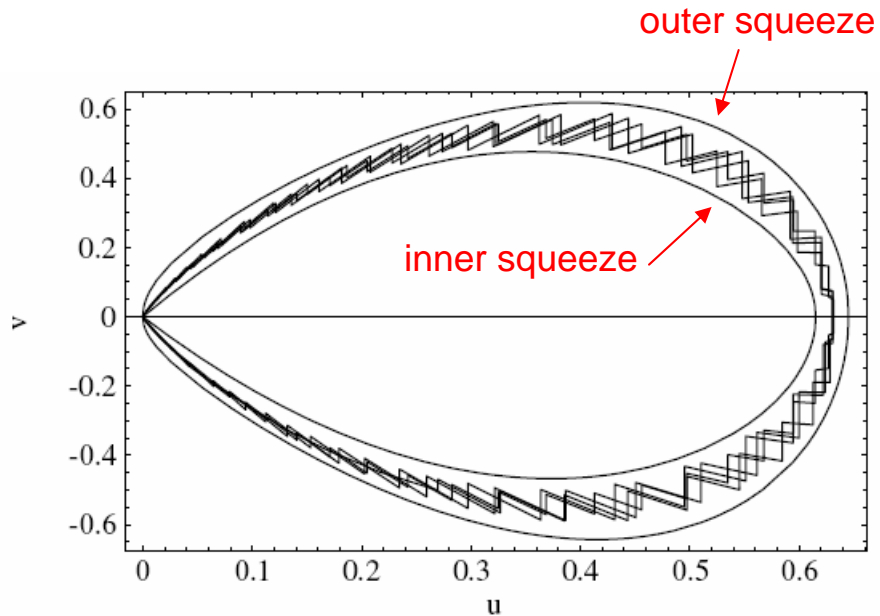
Then equation (7.3.18) becomes

$$p(x)dx = \int_{p'=0}^{p'=p(x)} dp' dx = \int_{u=0}^{u=\sqrt{p(x)}} \frac{\partial(p, x)}{\partial(u, v)} du dv = 2 \int_{u=0}^{u=\sqrt{p(v/u)}} du dv \quad (7.3.20)$$

because (as you can work out) the Jacobian determinant is the constant 2. Since the new integrand is constant, uniform sampling in (u, v) with the limits indicated for u is equivalent to the rejection method in (x, p) .

```
syms u v;
jac = jacobian([v/u u^2], [u v])
jac =
[ -v/u^2, 1/u]
[ 2*u, 0]
abs(det(jac))
ans =
2
```

Ratio of Uniforms is particularly powerful when combined with squeezes

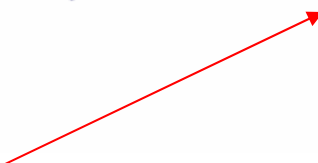


For this particular example the desired distribution is integer valued (binomial deviates), hence the staircases. For a continuous distribution, there would just be a smooth curve between the squeezes (which would typically be too close together to see clearly).

you only compute $p(v/u)$ when you are between the squeezes!

e.g., Leva's algorithm for normal deviates:

```
struct Normaldev : Ran {
  Structure for normal deviates.
  Doub mu,sig;
  Normaldev(Doub mmu, Doub ssig, Ullong i)
  : Ran(i), mu(mmu), sig(ssig){}
  Constructor arguments are  $\mu$ ,  $\sigma$ , and a random sequence seed.
  Doub dev() {
  Return a normal deviate.
    Doub u,v,x,y,q;
    do {
      u = doub();
      v = 1.7156*(doub()-0.5);
      x = u - 0.449871;
      y = abs(v) + 0.386595;
      q = SQR(x) + y*(0.19600*y-0.25472*x);
    } while (q > 0.27597
      && (q > 0.27846 || SQR(v) > -4.*log(u)*SQR(u)));
    return mu + sig*v/u;
  }
};
```



here, only ~1% of the (u,v) area is between the squeezes, requiring the calculation of the log