

Computational Statistics with Application to Bioinformatics

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

Unit 3: Random Number Generators, Tests for
Randomness, and Tail Tests Generally

Unit 3: Random Number Generators, Tests for Randomness, and Tail Tests Generally (Summary)

- Code in C++ a toy multiplicative random number generator (Toyran)
 - how to use NR3 #include file
 - test by binning into (initially) 10 bins
 - how do we know if the results are good enough?
- How to write Matlab mex-functions in C
 - wrote one for Toyran, our toy multiplicative linear congruential generator (MLCG)
- Bin 10^6 deviates by their (integer) values mod 10
 - result should be binomially distributed in each bin
- Discuss p-values and t-values
 - compute for our 10 bins
 - looks OK, but how do we combine them?
- Chi-Square test as optimal way to combine independent t-values
 - or non-independent because of linear constraints (adjust ν)
 - the chi square statistic is Chi Square distributed
 - but only if individual t-values are truly from Normal deviates
- A first serious test of Toyran
 - 10^8 draws, 1000 bins
 - it passes
 - but this is a one-point test only!

continues

- Test the 2-point distribution of our “Toyran” MLCG random generator by a chi-square test
 - 2-D array of bins
 - see it fail!
- Similarly test a Marsaglia Xorshift generator
 - it fails, but only at $\sim 10^8$ draws
- Learn how to combine disparate generators
 - via addition modulo $2N$ or via Xor
 - always a good idea
 - simple combined generator passes our statistical tests
- See what combination of generators is in NR3’s Ran
- Summarize the classic p-value tail test paradigm
 - distinguish α , the critical region, from p , the observed p-value
 - what you are and aren’t allowed to do after seeing the data

We'll do two topics simultaneously, going back and forth between them:

1. Generating random numbers
2. Testing numbers for randomness, as an archetype of p-value (tail) tests generally.

```
#include "nr3.h"           (our first example of C++ using NR3)

struct Toyran1 {
    Uint j;
    Toyran1(Uint jj) : j(jj) {}
    Uint int32() { return (j *= 747796405);}
};

int main() {
    Int i,j, nbins=10;
    VecInt bins(nbins,0);
    Toyran1 ran(11281974);
    for (i=0;i<1000000;i++) ++bins[ran.int32() % nbins];
    for (j=0;j<nbins;j++) cout << j << ": " << bins[j] << "\n";
    exit(0);
}
```

You can download nr3.h (and any of the NR3 C++ routines) from any UT Austin IP address (128.83.x.y) at <http://nrbook.com/institutional>

```
0: 200550
1: 0
2: 199352
3: 0
4: 199838
5: 0
6: 200191
7: 0
8: 200069
9: 0
```

Obviously nonrandom! Generates only even integers! (We wouldn't have noticed this if we looked at the high, not low bits.) Let's try again:

```
uint int32() { return (j = j*32310901+626627237);}
```

```
0: 100048
1: 99833
2: 100093
3: 99874
4: 99801
5: 100272
6: 99809
7: 100248
8: 100249
9: 99773
```

Better (close to 100000 in each bin). But **how close** does it need to be to pass?

We might initially focus on bins 5 and 9, for example.

Want to test the hypothesis H_0 that the generator is random. But against what alternative hypotheses? "Not-random" is not a useful hypothesis to a Bayesian, because can't compute $P(\text{data}|H_1)$. However it's enough for a (frequentist) p-value test.



Matlab detour: This is probably a good time to learn how to interface small pieces of C code to Matlab, both for speed and for easier access to low-level operations.

```
#include "mex.h"

typedef unsigned long Ulong;
Ulong u,m=32310901L,c=626627237L;

Ulong int32() {
    u = u * m + c;
    return u;
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    Ulong *x, *seed;
    int i,j;
    int M, N;

    if (nrhs > 1) {
        seed = (Ulong *)mxGetData(prhs[1]);
        u = *seed;
    }

    M = mxGetM(prhs[0]);
    N = mxGetN(prhs[0]);
    x = (Ulong *)mxGetData(prhs[0]);

    for (j=0;j<N;j++) for(i=0;i<M;i++) {
        *x++ = int32();
    }
    return;
}
```

Modifying the r.h.s. array is really bad Matlab style, since it's mostly a functional language. (And the input array could be a temporary, e.g.) We're doing it here just to keep this example as simple as possible. Later, we'll learn how to create and return l.h.s. arrays.

One time only, you have to tell Matlab what compiler you want to use

```
>> mex -setup
```

```
Please choose your compiler for building external interface (MEX) files:
```

```
Would you like mex to locate installed compilers [y]/n? y
```

```
Select a compiler:
```

```
[1] Intel C++ 9.1 (with Microsoft Visual C++ 2005 linker) in C:\Program Files\Intel\Compiler\C++\9.1
```

```
[2] Lcc-win32 C 2.4.1 in C:\PROGRA~1\MATLAB\R2007b\sys\lcc
```

```
[3] Microsoft Visual C++ 2005 in C:\Program Files\Microsoft Visual Studio 8
```

```
[0] None
```

```
Compiler: 2
```

This one installs with Matlab. It's only a rudimentary C (not C++) compiler, but we'll use it for now. Later we'll see how to interface to C++ in a more convenient way.

```
Please verify your choices:
```

```
Compiler: Lcc-win32 C 2.4.1
```

```
Location: C:\PROGRA~1\MATLAB\R2007b\sys\lcc
```

```
Are these correct?([y]/n): y
```

```
Trying to update options file: C:\Documents and Settings\wpress\Application Data\MathWorks\MATLAB\R2007b\mexopts.bat
```

```
From template: C:\PROGRA~1\MATLAB\R2007b\bin\win32\mexopts\lccopts.bat
```

```
Done . . .
```

```
>>
```

OK, let's try it

```
mex ransi mple.c
a = uint32(zeros(4, 5))
a =
      0      0      0      0      0
      0      0      0      0      0
      0      0      0      0      0
      0      0      0      0      0
ransi mple(a, uint32(12345)); a
a =
  72741554  1338772158  3416074506  3300866454  3848824930
  2233182719  364965755  3949301815  1911973939  1966969711
   17645104  3422742492  3591423432  2879349748   414551648
  2991333013  4140347953  1104062221  3463313705  3767133317
ransi mple(a); a
a =
  333128046  3681685690   200024646  2063106578   335715358
  3156110827  1393660839   568385699  2829283551  4294549595
  2204389644  3732076536  2152402724  1780769424  3413179964
  2687821089  172799741   2992002585  1369046645  1602169873
```

This example is not at all “industrial strength”. In particular, we have to remember to send it only uint32 quantities, or we’ll get meaningless results. We’ll get fancier later; this will do for now.

```
nbins = 10;
b = uint32(zeros(1000000, 1));
ransimple(b, uint32(11281974));
b = mod(b, nbins);
bins = accumarray(b+1, 1, [nbins 1]);
[(0:nbins-1)' bins]
```

Matlab idiom for binning!

ans =

```
0      100048
1      99539
2      99682
3      99810
4      100371
5      100498
6      99834
7      100051
8      100065
9      100102
```

(I guess I used a different seed in the C++ run.)

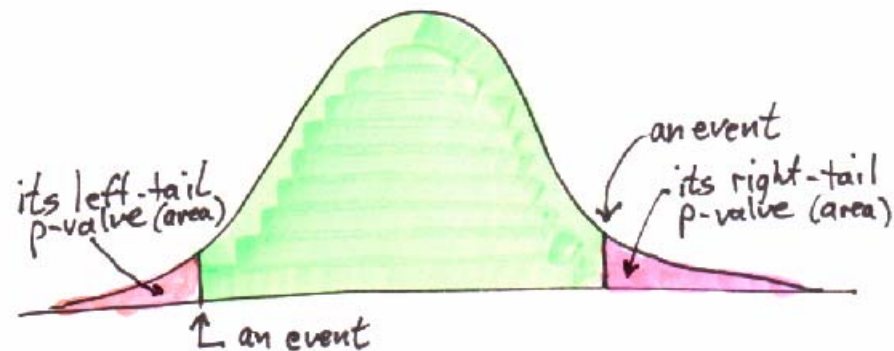
Under the null hypothesis (of perfect random) these should each be binomially distributed with $p=0.1$, $N=1000000$.



Now back to work.

The idea of p-value (tail) tests is to see how extreme is the observed data relative to the distribution of hypothetical repeats of the experiment under some “null hypothesis” H_0 .

If the observed data is too extreme, the null hypothesis is disproved. (It can never be proved.)



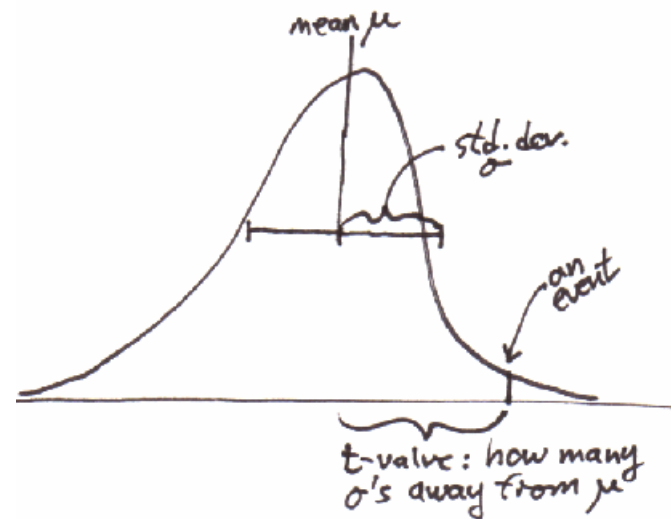
If the null hypothesis is true, then p-values are uniformly distributed in $(0,1)$, in principle exactly so.

There are some fishy aspects of tail tests, which we discuss later, but they have one big advantage over Bayesian methods: You don't have to enumerate all the alternative hypotheses (“the unknown unknowns”).



Don't confuse p-values with t-values

t-value = number of standard deviations from the mean



Intentionally drawn
unsymmetric, not
just sloppy drawing!

It's much easier to compute are scores that depend only on the mean and standard deviation of the expected distribution. But, in general, these are interpretable as “likely” or “unlikely” only relative to a Gaussian (which may or may not be relevant). Often we are in an asymptotic regime where distributions are close to Gaussian. But beware of t-values if not!

The reason that Gaussian's often **are** relevant is, of course, the Central Limit Theorem.

Compute p- and t-values for our random trials

```
p = @(k) betainc(0.1, k, 1000000-k+1);
```

```
p(bi ns)
```

```
ans =
```

```
0.4369
```

```
0.9381
```

```
0.8558
```

```
0.7372
```

```
0.1085
```

```
0.0487
```

```
0.7104
```

```
0.4330
```

```
0.4147
```

```
0.3674
```

Incomplete beta function is the cumulative distribution function of our binomial distribution. (For review, see NR3 6.14.14.)

```
mu = 0.1 * 1000000
```

Np

```
mu =
```

```
100000
```

```
sig = sqrt(0.1*0.9*1000000)
```

$Np(1-p)$

```
sig =
```

```
300
```

compute t-values

compute p-values as if Normal

```
t = @(x) (x-mu)./sig;
```

```
cdf = @(t) 1-normcdf(t, 0, 1);
```

```
[p(bi ns) t(bi ns) cdf(t(bi ns))]
```

CLT applies to binomial because it's sum of Bernoulli r.v.'s: N tries of an r.v. with values 1 (prob p) or 0 (prob $1-p$).

$$\mu = p \times 1 + (1 - p) \times 0 = p$$

$$\sigma^2 = p \times (1 - \mu)^2 + (1 - p) \times (0 - \mu)^2 = p(1 - p)$$

```
ans =
```

```
0.5614 -0.1533 0.5609
```

```
0.6386 -0.3533 0.6381
```

```
0.8465 -1.0200 0.8461
```

```
0.4382 0.1567 0.4378
```

```
0.3008 0.5233 0.3004
```

```
0.1881 0.8867 0.1876
```

```
0.1360 1.1000 0.1357
```

```
0.3826 0.3000 0.3821
```

```
0.9638 -1.7933 0.9635
```

```
0.3624 0.3533 0.3619
```

We've been scoring one bin at a time. But **how to get a single decision based on all bins?** Not just anecdotally stare at all the numbers!

- A single, highly extreme bin is probably decisive, but how extreme is extreme?
- There could also be signal hidden in moderate, but consistently low or high, p-values.

We need a way to combine multiple scores: **Chi-Square test**

If you have a bunch of independent (or even linearly dependent) t-values from close-to-Gaussian distributions, then the statistic with the *[technical description here]* optimal tail test is the sum of their squares, which is Chi-Square distributed.

$$\chi^2 = \sum_i t_i^2 = \sum_i \frac{(x_i - \mu_i)^2}{\sigma_i^2}$$

$$\chi^2 \sim \text{Chisquare}(\nu)$$

The number of degrees-of-freedom (DF or ν) is the number of t values, less the number of linear constraints (we'll see why, later)

Here (in C++ NR3 style) is a more serious test of Toyran

```
#include "gamma.h"
#include "incgammabeta.h"

int main() {
    Int i,j, nbins=1000, ntries=100000000, seed=11281975;
    Doub chisq=0., mu,sig,p,ptail;
    VecInt bins(nbins,0);
    Toyran1 ran(seed);
    chisqdist chisqdist(nbins-1.);
    p = 1./nbins;
    mu = ntries*p;
    sig = sqrt(ntries*p*(1.-p));
    for (i=0;i<ntries;i++) ++bins[ran.int32() % nbins];
    for (j=0;j<nbins;j++) chisq += SQR((bins[j]-mu)/sig);
    ptail = chisqdist.cdf(chisq);
    cout << "seed=" << seed << " ntries=" << ntries << " nbins="
         << nbins << " chisq=" << chisq << " ptail=" << ptail << "\n";
    exit(0);
}
```

One linear constraint: ntries is fixed

seed=11281975 ntries=1000000 nbins=10 chisq=4.2202 ptail=0.103669

seed=11281975 ntries=100000000 nbins=1000 chisq=995.806 ptail=0.47743

Here's the same test in Matlab using our ransimple.c mex function

```
function [ptail, chisq] = rantest1D(nbins, ntries, seed)
blocksize = 1e5;
nblocks = ceil(ntries / blocksize);
ntries = nblocks*blocksize;
bins = zeros(nbins, 1);
ransin = uint32(zeros(blocksize, 1));
ransimple(ransin, uint32(seed));
for i = 1:nblocks
    ransimple(ransin);
    bins = bins + accumarray(mod(ransin, nbins)+1, 1, [nbins 1]);
end
p = 1/nbins;
mu = ntries*p;
sig = sqrt(ntries*p*(1-p));
chisq = sum(((bins-mu)/sig).^2);
ptail = 1 - chi2cdf(chisq, nbins-1);
```

```
>> [ptail, chisq] = rantest1D(1000, 100000000, 12345678)
ptail =
    0.6970
chisq =
    975.4680
```

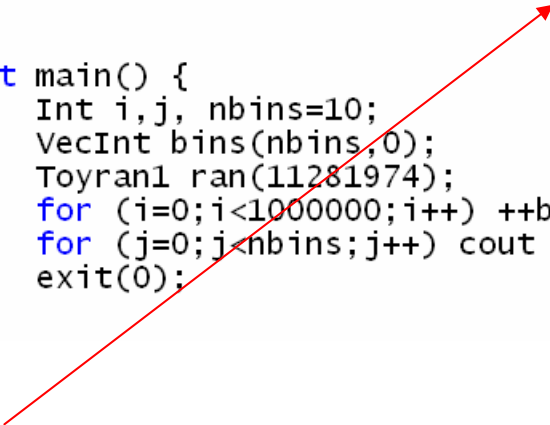
(would be too slow on $N=10^8$ in pure Matlab m-code)

Recall what Toyran looked like:

```
#include "nr3.h"

struct Toyran1 {
    Uint j;
    Toyran1(Uint jj) : j(jj) {}
    Uint int32() { return (j *= 747796405);}
};

int main() {
    Int i,j, nbins=10;
    VecInt bins(nbins,0);
    Toyran1 ran(11281974);
    for (i=0;i<1000000;i++) ++bins[ran.int32() % nbins];
    for (j=0;j<nbins;j++) cout << j << ": " << bins[j] << "\n";
    exit(0);
}
```



This turned out to be bad, so we replaced it with

```
Uint int32() { return (j = j*32310901+626627237);};
```

a multiplicative linear congruential generator (mod 2^{32})

There is mathematics and art in choosing the multiplier (see, e.g., Knuth)

It turns out that the hard part in generating random numbers is not to get their (one-point) distribution to be flat, but to get their n-point distribution to be flat: **sequential correlations**.

Chi-square test for 2-point distribution:

```
int main() {
    Int i,j,k, n=10, nbins=n*n, ntries=100000000, seed=11281975;
    Doub chisq=0., mu,sig,p,ptail;
    MatInt bins(n,n,0);
    Toyran1 ran(seed);
    Chisqdist chisqdist(nbins-1.);
    p = 1./nbins;
    mu = ntries*p;
    sig = sqrt(ntries*p*(1.-p));
    for (i=0;i<ntries;i++) ++bins[ran.int32() % n][ran.int32() % n];
    for (j=0;j<n;j++) for (k=0;k<n;k++) chisq += SQR((bins[j][k]-mu)/sig);
    ptail = chisqdist.cdf(chisq);
    cout << "seed=" << seed << " ntries=" << ntries << " nbins="
         << nbins << " chisq=" << chisq << " ptail=" << ptail << "\n";
    exit(0);
}
```

```

function [ptail, chisq] = rantest2Dwhp(n, ntries, seed)
nbins = n^2;
blocksize = 1e5;
nblocks = ceil(ntries / blocksize);
ntries = nblocks*blocksize;
bins = zeros(n, n);
ransin = uint32(zeros(2, blocksize));
ransimle(ransin, uint32(seed));
for i = 1:nblocks
    ransimle(ransin);
    bins = bins + accumarray(mod(ransin', n)+1, 1, [n n]);
end
p = 1/nbins;
mu = ntries*p;
sig = sqrt(ntries*p*(1-p));
chisq = sum(((bins(:)-mu)/sig).^2);
ptail = 1 - chi2cdf(chisq, nbins-1);

```

Matlab localizes storage on leftmost subscripts first (opposite of C), so must have 2 first to get consecutive ranns

But then transpose here, since accumarray uses each row as a subscript pair

C++ output:

```
seed=11281975 ntries=100000000 nbins=100 chisq=3.0303e+008 ptail=1
```

or, Matlab output:

```
[ptail, chisq] = rantest2D(10, 100000, 12345678)
```

```
ptail =
```

```
0
```

```
chisq =
```

```
3.0312e+005
```

(I've got my tails reversed between the C++ and the Matlab. Doesn't matter: would reject on either tail.)

Fails spectacularly! Turns out that the random values alternate even and odd, so half the bins are empty! Failure of low order bits is generic to multiplicative congruential generators like this, so we won't go further with this toy generator.

How about a completely different algorithm:

```
uint int32() {  
    j ^= j >> 3;  
    j ^= j << 13;  
    j ^= j >> 7;  
    return j;  
}
```

This is called a “Marsaglia Xorshift generator”.

C++ output:

```
seed=11281975 ntries=1000000 nbins=256 chisq=236.495 ptail=0.208852  
seed=11281975 ntries=100000000 nbins=65536 chisq=64288.1 ptail=0.000265755
```

or, Matlab output:

```
EDU>> mex ransimple.c  
EDU>> [ptail, chisq] = rantest2D(16, 1000000, 12345678)  
ptail =  
    0.5125  
chisq =  
    253.6295  
EDU>> [ptail, chisq] = rantest2D(256, 100000000, 12345678)  
ptail =  
    0.9998  
chisq =  
    6.4277e+004
```

Much better, but still fails at 10^8 draws. We'll come back to the theory of these Xorshift generators a little later.

Notice that if you only tested at 10^6 draws, you'd think all was well.

Combined generators

```
struct Toyran1 {
    Uint j,k;
    Toyran1(Uint jj) : j(jj), k(j*747796405) {}
    Uint int32() {
        j ^= j >> 3;
        j ^= j << 13;
        j ^= j >> 7;
        k = k*32310901+626627237;
        return j+k;
    }
};
```

```
seed=11281975 ntries=1000000000 nbins=65536 chisq=65429.7 ptail=0.386234
```

about equally good is: `return j^k;`

```
seed=11281975 ntries=1000000000 nbins=65536 chisq=64976.1 ptail=0.060999
seed=11281976 ntries=1000000000 nbins=65536 chisq=65274.9 ptail=0.236526
seed=11381977 ntries=1000000000 nbins=65536 chisq=65343.6 ptail=0.298941
seed=11481981 ntries=1000000000 nbins=65536 chisq=66058.6 ptail=0.925653
```

These are actually not bad for 32 bit generators. Would satisfy 99% of users 99% of the time.

Knowing the algorithm, can you design a test to show residual non-randomness?

A \$10 prize is offered for the first correct example, and another \$10 for the best correct example.

Here's what a good generator looks like:

```
ran.h  struct Ran {
      Implementation of the highest quality recommended generator. The constructor is called with
      an integer seed and creates an instance of the generator. The member functions int64, doub,
      and int32 return the next values in the random sequence, as a variable type indicated by their
      names. The period of the generator is  $\approx 3.138 \times 10^{57}$ .
      Ullong u,v,w;
      Ran(Ullong j) : v(4101842887655102017LL), w(1) {
      Constructor. Call with any integer seed (except value of v above).
          u = j ^ v; int64();
          v = u; int64();
          w = v; int64();
      }
      inline Ullong int64() {
      Return 64-bit random integer. See text for explanation of method.
          u = u * 2862933555777941757LL + 7046029254386353087LL;
          v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
          w = 4294957665U*(w & 0xffffffff) + (w >> 32);
          Ullong x = u ^ (u << 21); x ^= x >> 35; x ^= x << 4;
          return (x + v) ^ w;
      }
      inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
      Return random double-precision floating value in the range 0. to 1.
      inline Uint int32() { return (Uint)int64(); }
      Return 32-bit random integer.
  };
```

Design features of NR3's Ran

- It has 3 generators, each with 64 bits of state
 - LCG
 - Xorshift
 - MWC (not previously discussed)
- It uses 64 bit unsigned arithmetic
- The generators are combined on output, but don't affect each other
- The generators separately pass (or almost pass) a battery of standard tests
- The output is available in various formats
- Faster generators about ~2x faster, but “riskier”
- Some generators in the literature are much slower
 - “Mersenne twistor”
 - “provably random” generators
 - cryptographic generators

Lesson on generators:

Combined generator can both improve the generator and increase the period, here $(2^{32}-1)2^{32}$. **Always** use a combined generator. (Actually, I think the failure of the toy Xorshift generator was period exhaustion, not sequential correlation per se; but it's still a failure.)

Lesson on tail tests:

Don't sweat a value like 0.06. If you really need to know, the only real test is to get (here, simulate) significantly more data. Rejection of the null hypothesis is exponential in the amount of data.

In principle, p-values from repeated tests s.b. exactly uniform in $(0,1)$. In practice, this is rarely true, because some "asymptotic" assumption will have crept in when you were not looking. All that really matters is that (true) extreme tail values are being computed with moderate fractional accuracy. You can go crazy trying to track down not-exact-uniformity in p-values. (I have!)

Summary of the classic p-value test paradigm:

- “null hypothesis”
- “the statistic” (e.g., χ^2)
 - calculable for the null hypothesis
 - intuitively should be “deviation from” in some way
- “the critical region” α
 - biologists use 0.05
 - physicists use 0.0026 (3σ)
- one-sided or two?
 - somewhat subjective
 - use one-sided only when the other side has an understood and innocuous interpretation
- if the data is in the critical region, the null hypothesis is ruled out at the α significance level
- after seeing the data you
 - may adjust the significance level α
 - **may not try a different statistic**, because any statistic can rule out at the α level in $1/\alpha$ tries (shopping for a significant result!)
- if you decided **in advance** to try N tests, then the critical region for α significance is α/N (Bonferroni correction).



they got this wrong: null hypothesis is most likely, unless disproved by the data!

