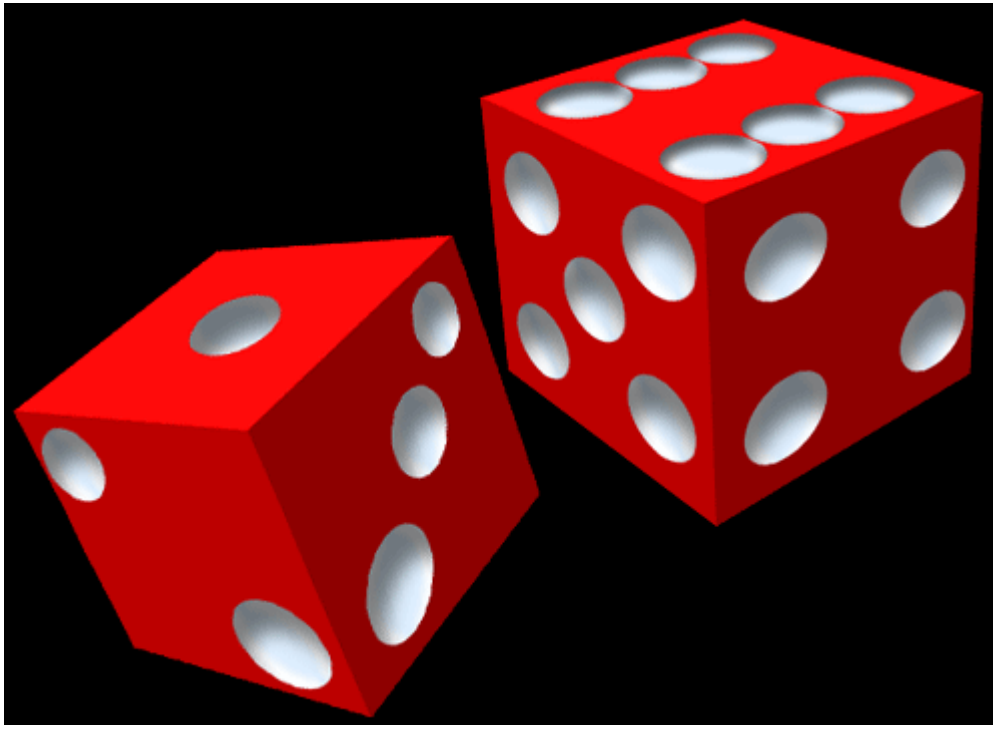
Monte Carlo and Numerical Methods

Scott Oser
Lecture #4



Outline

Last time: we studied Poisson, exponential, and χ^2 distributions, and learned how to generate new PDFs from other PDFs by marginalizing, projecting, etc.

Today:

- Role of simulation in data analysis
- Random number generation
- Generating arbitrary distributions
- Numerical techniques for minimizing functions
- Libraries of numerical routines
- Helpful hints for good computing

The Simulation Approach

Since the data sets we want to analyze are often random, why not use randomness as a tool in our analysis?

Simulation is the procedure of numerically generating “random” data sets, usually on a computer, that can be analyzed identically to the real data. Because you know what went into making the simulation, you can control its parameters, and test the effects of changes on your analysis.

Today we'll look at the why and how of simulation.

Why simulate?

Your data and/or physical model are too complicated to understand analytically.

Suppose you have some very complicated data set from which you want to extract a set of model parameters. (For example, you're fitting the luminosity vs. redshifts of a set of supernova to determine the value of the cosmological constant.) You cannot analytically figure out how to determine the statistical error bars on the fitted values---the errors on individual points aren't Gaussian, and the fitted values are complicated functions of the data.

Solution: generate 1000 fake data sets having the same general properties of your real data set. Fit each of the 1000 data sets, and then look at the width of the distribution of 1000 fitted values. If your simulated data correctly models your real data set, then you get the error without having to mess with error propagation equations.

What do you need to do a simulation?

If you want to write a simulation, you must have:

- An accurate model of how your data points relate to the underlying physics. It should include any relevant physical effects, including justifiable assumptions about the distributions of any "random" quantities. There is no magic here---this model is needed to do any data analysis, not just simulation.
- A means of generating "random numbers". A random number is an "unpredictable" value with known distribution that we will use to model a random variable.

Many "off the shelf" tools exist for generating random numbers, but it's important for you to understand what they do and what their limitations are.

Desirable properties of random numbers

A “random number generator” is a a routine that returns a “random” number drawn from a specified probability distribution. Without loss of generality we can consider generating numbers on the interval [0,1). We would like our routine to have the following properties:

- Unpredictability: if I tell you the last N values returned by the routine, you should not (easily) be able to predict what value it will return next.
- Correct coverage: the frequency with which any value *x* comes up is proportional to the underlying PDF *f(x)*---a uniform distribution in this case.
- We don't want our random number to be random.

What do you mean, “We don't want our random number to be random”?

Seems a little counterintuitive, doesn't it, especially since we want our random number generator to be unpredictable.

You could get “true” random numbers by building a circuit that measures the Johnson noise in a resistor, and using the measurement to generate a random value.

But what we really want is a *pseudo-random* number generator. A pseudo-random number is unpredictable, but reproducible. If we want, we can reset the routine and get back the exact same sequence of random numbers.

This is extremely useful for debugging, or separating the effects of random fluctuations from the effects of other elements we may be changing in the code.

Linear Congruent Generators

The most basic random number generator is:

$$I_{j+1} = aI_j + c \pmod{m}$$

Here *m* is the modulus, *a* is called the multiplier, and *c* the increment. All are positive constants. I_j/m will be a number between 0 and 1.

In the best case, the sequence repeats after *m* calls, and all numbers between 0 and *m-1* occur.

In the worst case (which happens too often), the values of *a* and *c* are poorly chosen so that the routine repeats much sooner. Many times the built-in random number generator is such a case. Do not use unknown random number generators unless you trust the source and know that they work.

Weaknesses: sequential numbers are correlated, and least significant bits are less random than higher order bits.

Numerical Recipes: ran0()

Numerical Recipes contains a number of satisfactory random number generators. We'll look at two of them in detail to get an idea of the issues.

ran0:

$$I_{j+1} = a I_j \pmod{m}$$

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2.147 \times 10^9$$

- Period: 2.147 X 10^9
- Obviously must not be seeded with 0
- Correlations evident: if I_j is very small, then I_{j+1} is likely to be small as well. Small numbers tend to be followed by other small numbers.

Numerical Recipes: ran1()

The major problem with ran0 is the correlation between successive entries. The Numerical Recipes routine ran1 tries to fix this by shuffling the output of ran0.

ran0:

$$I_{j+1} = a I_j \pmod m$$

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2.147 \times 10^9$$

- Period: 2.147 X 10^9
- Obviously must not be seeded with 0
- Correlations are much improved by shuffling!
- Recommended for general use, so long as the number of calls to the routine is <10^8---my personal favourite.
- If you require more calls than this, use a better routine---see NumRec.

NumRec Example Code

```
program main

implicit none
integer idum /-982737/
real ran1
integer i

do i=1,10
    write (*,*) ran1(idum)
enddo

end
```

- Forgive the FORTRAN
- Note need to give the seed idum some initial value to start with!
- Compiling this requires a Makefile to compile and link against the NumRec routines. Your version may be different depending on what language you use, and what platform.

How to seed

Be careful with choosing the initial seed! Some common pitfalls:

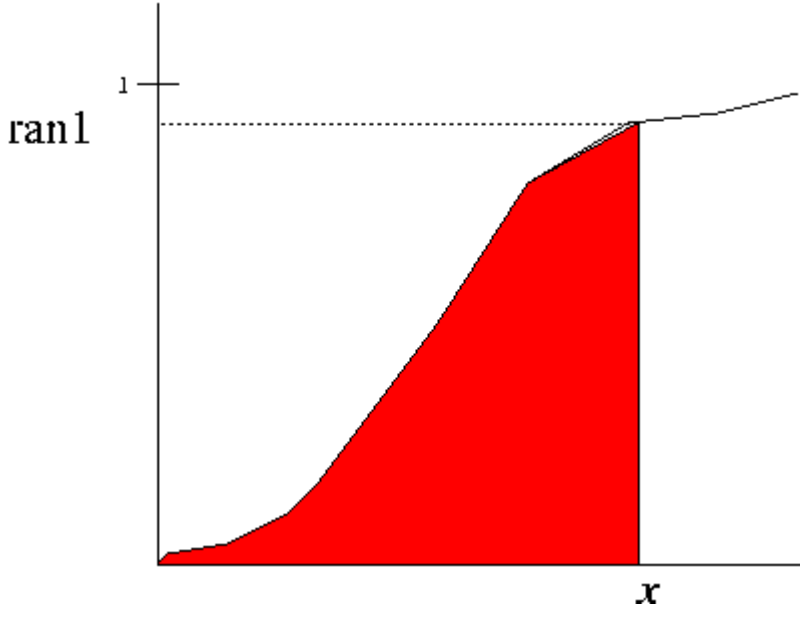
- You run a program 1000 times, intending to create 1000 fake data sets, but all are generated with the same seed. (Solutions: read the seed values from a file or specify as a runtime option.)
- You really do want a “random” sequence. Try seeding using the time on the system clock (esp. lower bits like the number of milliseconds.) But save the seed value with the output in case you want to regenerate the same sequence.
- Read the documentation on the generator. Some want a negative seed value in order to flag it as a re-seed.
- Folklore: seed values that are not divisible by low prime numbers are preferable---don't know if this is true or just superstition.

Transforming to generate non-uniform distributions

By now it must have crossed your mind that uniform random numbers between 0 and 1 are not so useful. How do we generate arbitrary distributions? One option is a transform:

$$\int_{-\infty}^{x} f(z)\,dz = F(x) \in [0,1]$$

The basic idea is to generate a number between 0 and 1 and to interpret that as the fraction of the PDF below a given value. Then return that value *x*.



Example: generating an exponential distribution

Exponential distribution has PDF:

$$f(z)=\frac{1}{\tau}\exp(-z/\tau)$$

$$F(x)=\int_0^x \frac{dz}{\tau}\exp(-z/\tau)=\int_0^{x/\tau} dy\, e^{-y}=1-\exp(-x/\tau)$$

So if Y is a random number from 0 to 1, then set Y=*F(x)* and then solve for *x*

$$x=-\tau\ln(1-Y) \qquad \text{or} \qquad x=-\tau\ln Y$$

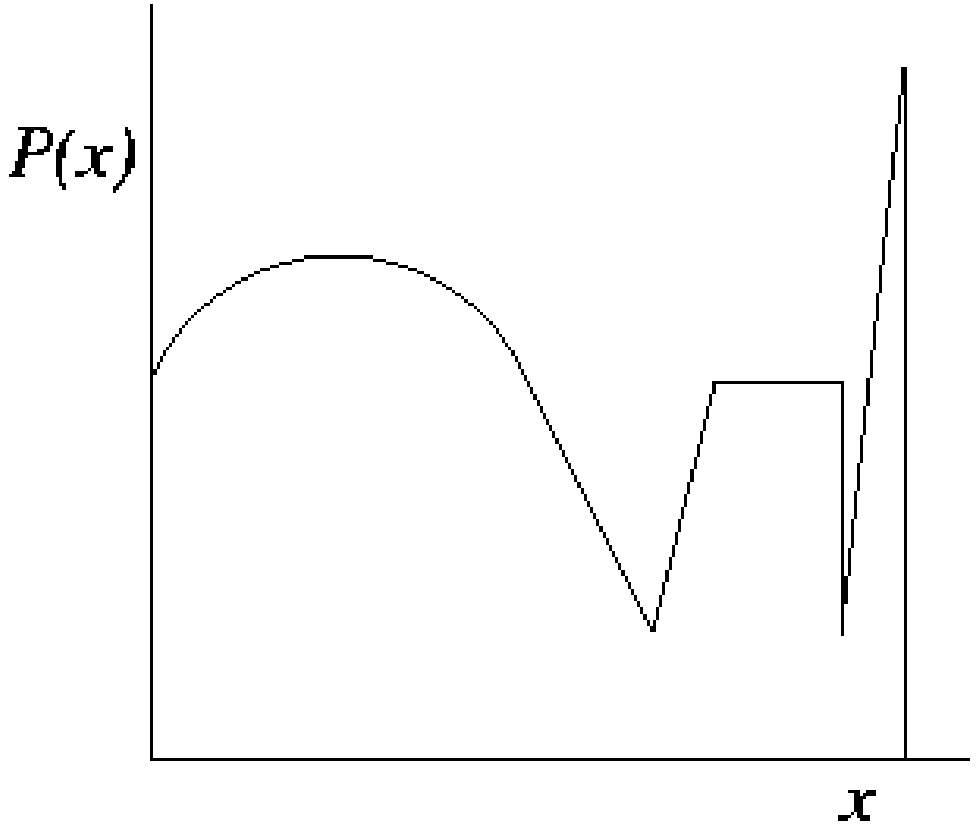
(These are equivalent because both Y and 1-Y will be uniform distributions ranging from 0 to 1.)

Pros and cons of transforming

- Very efficient, provided you can do the integral. In principle you can transform a single random number into any distribution you like.
- This is the ideal (only?) way to generate a random variable with an infinite range.
- Major problem: often you cannot do the integral! For example, even the cumulative integral for a normal (Gaussian) distribution cannot be solved in closed form.

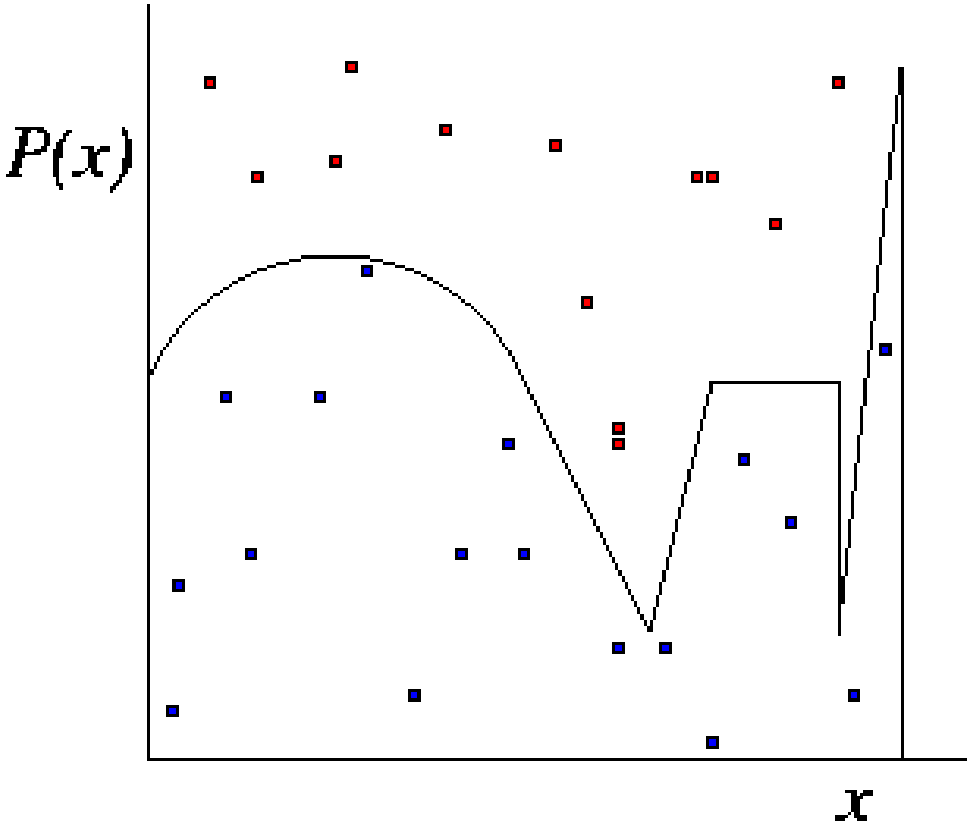
A puzzle

How might you generate random numbers from this complicated-looking distribution?



Acceptance/rejection method

Throw darts!

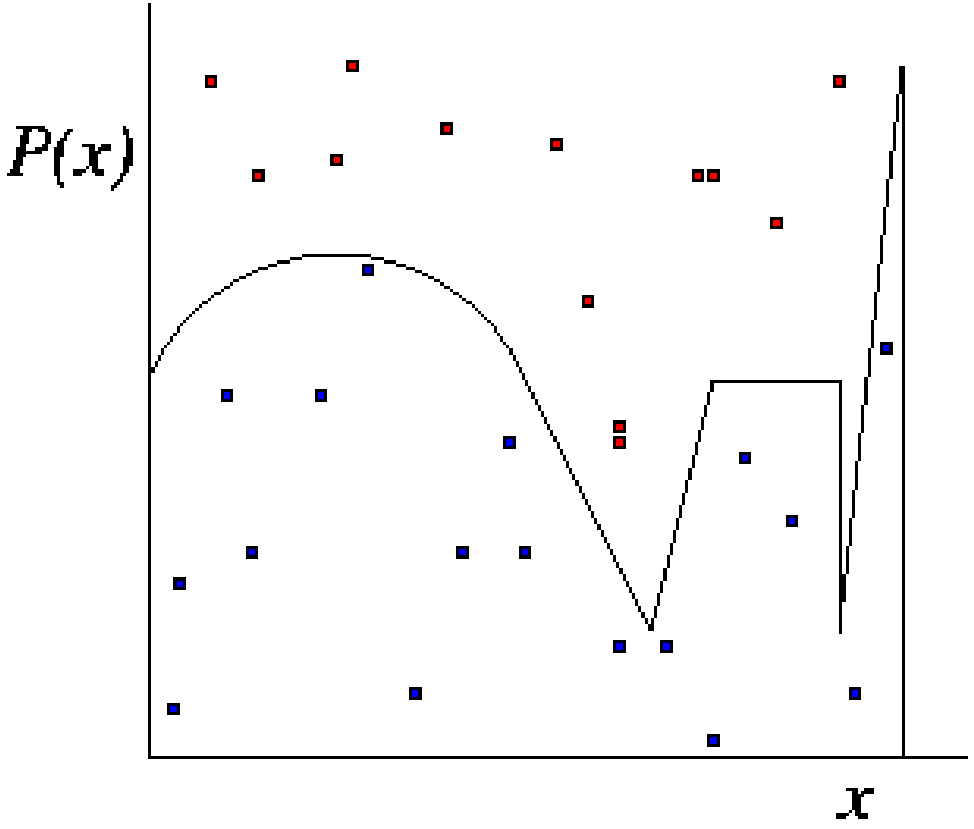


Generate *x* uniformly over the range of validity.

Generate a second random number *y*.

If $y < P(x)$, keep and return this value of *x*, else discard it and try again.

Optimizing acceptance/rejection



Rescale PDF so that its maximum occurs at *P(x)=1*, so you don't waste trials.

If PDF is zero over any range, exclude that range from the region over which you generate initial guesses for *x*.

Can you use acceptance/rejection to generate a perfect Gaussian?

Hybrid Methods: accept/reject over infinite range

How would you generate random numbers from the distribution (defined over the range 0 to 10):

$$f(x) = \frac{1+\cos(x)}{\sqrt{x}}$$

Can't use accept/reject method since f blows up at x=0.

Can't use the transform method because you can't (easily) do the integral.

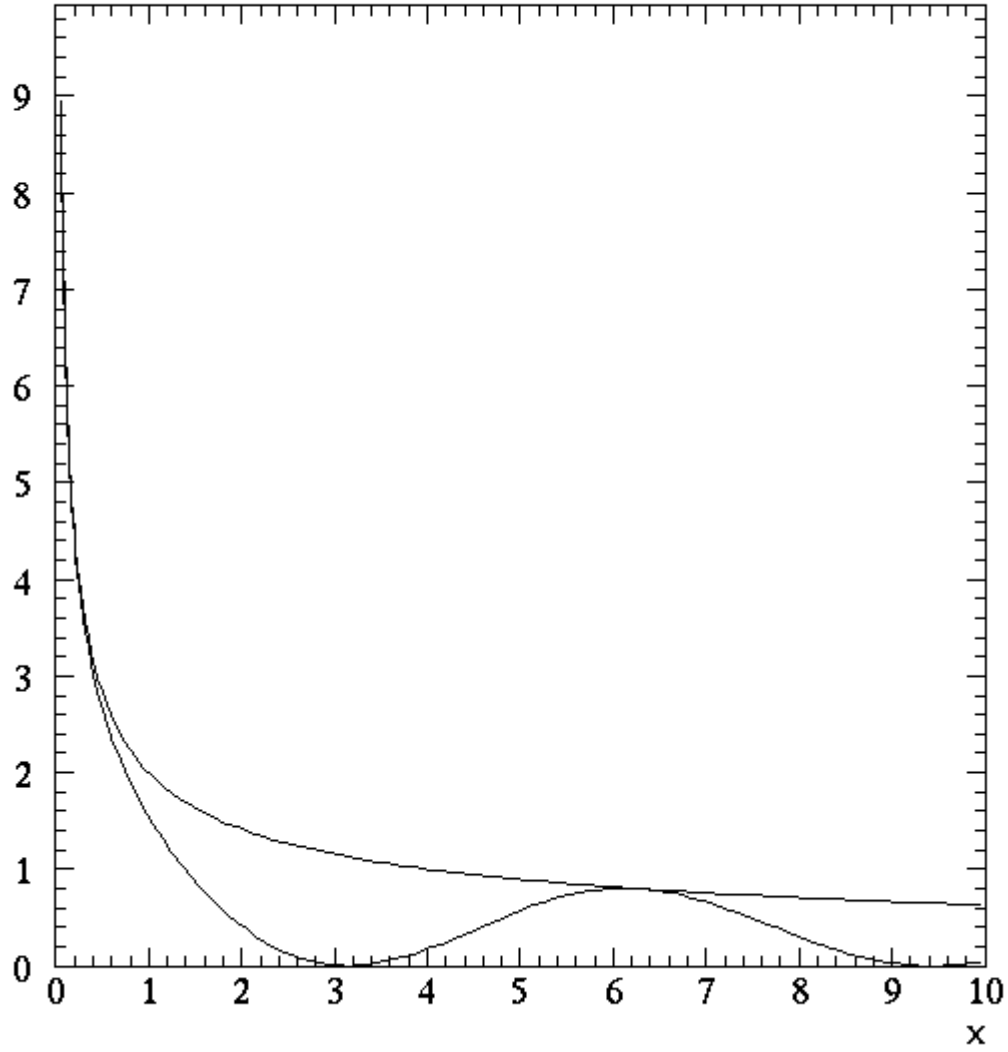
Solution: do a hybrid approach!

Hybrid Methods: accept/reject over infinite range

Choose a nice analytic function you can use the transform method on: $g(x)=\frac{2}{\sqrt{x}}$

Generate a random number from this distribution.

Apply accept/reject to result: accept if ran1 < f(x)/g(x), else reject this point.



Generating Gaussian distributions

How might you generate a Gaussian with mean 0 and width 1?

1) Generate the sum of 12 uniform deviates between 0 and 1. By CLT, this is approximately Gaussian with mean of 12 X 0.5 = 6 and variance of 12 X (1/12) = 1. Subtract 6 to get mean of zero.

2) If you have a reliable function that calculates the cumulative integral numerically---e.g. Erf()---use the transform method and invert it.

3) A fiendish trick: generate a pair from a 2D Gaussian:

$$P(x,y) = \exp\left(-\frac{x^2}{2}\right)\exp\left(-\frac{y^2}{2}\right)$$

How does this help???

Generating Gaussian distributions

$$P(x,y)\,dx\,dy = \exp\left(-\frac{x^2}{2}\right)\exp\left(-\frac{y^2}{2}\right)dx\,dy = \exp\left(-\frac{r^2}{2}\right) r\,dr\,d\theta$$

This is a product of an integrable analytic PDF over *r* and one over θ. So use transform method to generate a pair (r,θ).

$$u \equiv r^2/2 = -\ln(\mathrm{ran}_1)$$

$$r = \sqrt{2u}$$

$$\theta = 2\pi\,\mathrm{ran}_2$$

$$x = r\cos\theta$$

$$y = r\sin\theta$$

The result is you get two random variables x and y which independently follow Gaussian distributions. Not super-efficient, but elegant!

My favourite method: call NumRec routine gasdev(idum)!

Three ways of generating Poisson variables

1) Sum N binomial variables with p=λ/N, where N is large. Slow but easy to do. OK if $\lambda \ll N$. Cannot generate values larger than N. Not exact, obviously.

2) Accept/reject over truncated range. Pick a random integer from 0 to N (inclusive), and then use accept/reject. Unlikely to be very efficient! Distribution is obviously truncated.

3) Transform from cumulative (integral) distribution:
   A. Pick a uniform deviate Y from 0 to 1.
   B. Start summing up Poisson distribution from 0 to *m* until sum exceeds Y.

$$S(m) = \sum_{j=0}^{m} e^{-\lambda} \frac{\lambda^j}{j!}$$

   C. Return biggest *m* for which *S(m)* < Y.

My Poisson code:

```
function poisson (mean,idum)

integer poisson
real*8 mean
integer idum
real*8 ran1,x,sum,term

x = ran1(idum)
poisson=0
term = exp(-mean)
sum = term
do while (x .gt. sum)
   poisson=poisson+1
   term = term*mean/poisson
   sum=sum+term
enddo

return
end
```

- In principle, this is *exact*, subject to roundoff error.
- Will be slow for large λ
- If λ gets too big, you might consider using a Gaussian approximation to the Poisson distribution.

Generating spherical coordinates

This works!

```
phi = 2*pi*ran1(idum)

cosz = 2*ran1(idum) - 1.
theta=acos(cosz)

r = 1

x = r*sin(theta)*cos(phi)
y = r*sin(theta)*sin(phi)
z = r*cos(theta)
```

Anecdote: as a young graduate student, I tried to do this by:

- Pick x uniformly from -1 to +1
- Pick y uniformly between $\pm\text{sqrt}(1-x^2)$
- Calculate $z=\text{sqrt}(1-x^2-y^2)$
- Randomly choose the sign for z.

Does this work as well?

Generating correlated random variables

Suppose you need to generate two or more correlated random variables. Calling ran1 twice of course gives you two uncorrelated variables. How can you generate correlated variables?

One way is to use conditional PDFs. Rewrite the joint PDF P(x,y) as f(x)g(y|x). Given distribution f(x), choose a random value of x. Then given g(y|x) (with x known), choose a random value for y from that distribution.

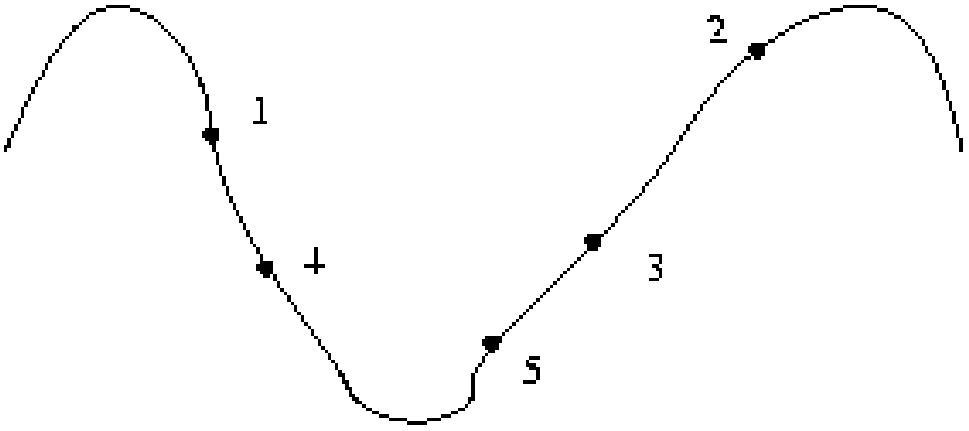
Another is to generate two independent random variables, then to form correlated combinations of them. For example, if you generate two variables X and Y, then what is the correlation between these two quantities:

$$A = X \qquad B = X + Y$$

Function Minimization

Having to minimize a function (usually with respect to fit parameters) is a common problem. Concerns are to find the global mininum, avoid local minima, and compute curvature at the minimum.

Many routines exist: see Numerical Recipes, for example



Works on any function, no derivatives needed.

"Foolproof" method in 1D: bracketing

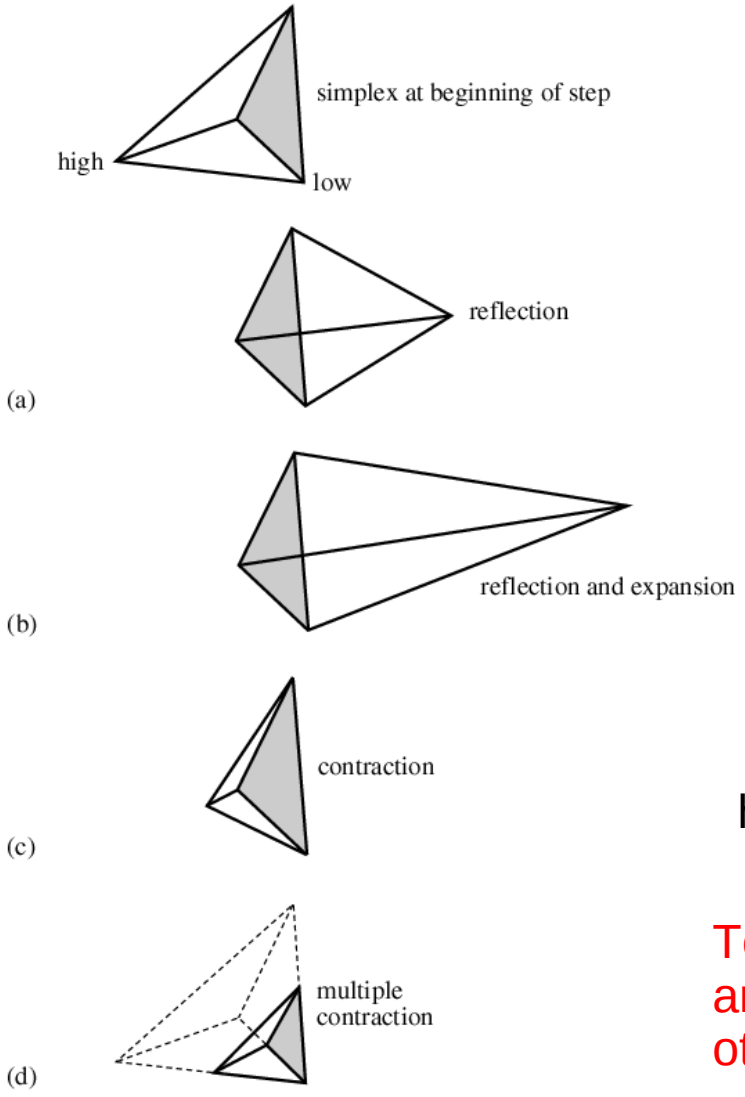
Suppose $f(x_3) < f(x_1)$ and $f(x_3) < f(x_2)$. We then say that the minimum is bracketed between x_1 and x_2. Now start trying points in between to narrow in the bracket.

Start with (1,2), knowing 3 is lower than either. Try new point 4.

Now bracket is (1,3), with 4 lower. Try point 5. It's lower still.

New bracket is (4,3). Keep going until converges.

The downhill simplex method: “amoeba”



- Start with a simplex of N+1 points.
- Try to move the vertex at which the function is largest to a new point by reflecting across the opposite plane.
- If a move results in lower value, keep going in that direction. Else, try contracting the simplex. Continue until convergence.

The simplex moves and stretches like an amoeba---hence it's NumRec name.

See the animations on the WikiPedia article!

http://en.wikipedia.org/wiki/Nelder-Mead_method

To ensure convergence, try restarting at initial point, and also try with different starting points to check for other minima.

Function Minimization: gradient methods

If you can calculate the gradient of your function, you at least know “which way is down”.

Lots of algorithms:

- Steepest descent: always move your best estimate of the minimum straight downhill---believe it or not, this is not very efficient!
- Fletcher-Reeves conjugate gradient algorithm. Proceeds as a succession of line minimizations. The sequence of search directions is used to build up an approximation to the function's curvature around the minimum.
- Broyden-Fletcher-Goldfarb-Shannon (BFGS) algorithm. Builds up an approximation to the second derivatives using the difference between successive gradient vectors. May be faster for smooth, quadratic-ish functions.

Pick one. Personally I like MINUIT's implementation MIGRAD, or simplex method as a backup.

Function Minimization: grid search

In multiple dimensions, there is only ONE foolproof minimizer.

It's called the “grid search”.

Just scan the value of the function across an N-dimensional grid throughout the plausible range of values.

Yes, this sounds stupid, and it is. But unlike other routines, it is guaranteed to converge, and to avoid local minima.

Plus, you get a multi-dimensional map of the function as a byproduct, which is useful.

If you're struggling to get your minimizer to converge or even run, a grid search may be the fastest solution---not in terms of CPU time, but in terms of YOUR time!

Available software packages

Here are the relevant software routines for three minimization packages. It's up to you to learn how to run them. Whatever you do, don't waste your time writing your own minimizer for this course!

	NumRec	GSL	MINUIT
Simplex	amoeba	gsl_multimin_fminimizer_nmsimplex	SIMPLEX
Conjugate Gradient	frprmn	gsl_multimin_fdfminimizer_conjugate_fr	<none>
BFGS algorithm	dfpmin	gsl_multimin_fdfminimizer_vector_bfgs	MIGRAD

Manuals:

http://www.gnu.org/software/gsl/manual/
http://www.numerical-recipes.com/nronline_switcher.php
http://hep.fi.infn.it/minuit.pdf
http://root.cern.ch/root/html/examples/lfit.C.html

Computing – a few pointers (courtesy of J. Wall)

- **believe nothing**.
- **believe nothing**, especially if program compiles successfully.
- **believe nothing**, except that it's your fault, not the prog/hardware/bugs etc.

- always **declare all variables**, i.e. ‘implicit none’, at outset of program or subroutines. This keeps 'variable discipline', and minimizes mistyped variables.

- write at most **5 lines of code** at a time, eg a single simple read-in do-loop.
- check that every bit does the right thing, using dummy data if necessary.

-write logical constructions with **complete syntax** before filling in what you want it to do, e.g. do-loops.

- **comment** everything.

- have **program skeletons** lying around.

- write in **modular bits**, preferably subroutines.
- this gives you a **library** of useful routines.

Computing – a few more pointers

- use **traps** in read statements to catch end of file, errors

- don't **write long statements**! Break long expressions up into bits. Pretend you're writing for a three-yr-old - in simple steps that you can understand when you come to look at it later.

- don't use **computed goto**

- if you want speed, avoid **if statements**. With a bit of thought, you can do it

- apparently impossible results/failures - totally inexplicable: commonest cause is hidden memory problems. **You've overwritten an array or a variable**.

- **don't make it look beautiful**; use lots of (commented-out) write statements, or ! statements. You think you will never forget how it works/what it does?

- don't get **programmitis.** ***This is not a programming course. Do what you need to do to solve the problem and little more.***

Next time

We'll try to finish a little early, in order to give us time to discuss HW1, which is due today.