Linear Algebra II

Computing for Astro Grad Students

Brief Outline

- Revisit Ax=b, and ways to improve accuracy.
- Expand to try to solve Ax=b when we can't factor/invert A directly. Apply to mapmaking.
- Non-linear least-squares fitting (Levenberg-Marquardt).
- Use Markov-Chain Monte Carlo to fit parameters, estimate errors for complex likelihood surfaces.

Revisiting Ax=b

- However we solve Ax=b, roundoff error might be large. Solution: iterative improvement.
- Say we have a guess at A^{-1}, then let $x^{\dagger}=A^{-1}b$. Ideally, $Ax^{\dagger}$ would equal b, but this will only be approximately true.
- Let $\delta=Ax^{\dagger}-b$, the error in our solution. Then we can "correct" $x^{\dagger}$ by subtracting $A^{-1}\delta$.
- Works because Ax should have less error than $A^{-1}b$. Fast to do since inverting/factoring A was expensive, but further matrix-vector cheap.
- We encourage you to make a habit of this.

Iterative Improvement, contd.

```
-->n=2e3;a=rand(n,n,'normal');b=rand(n,1);ainv=inv(a);

-->x=ainv*b;delta=a*x-b;x2=x-ainv*delta;

-->err1=sqrt(sum((a*x-b).^2));err2=sqrt(sum((a*x2-b).^2));

-->printf("Naive error is %14.4e, improved is %14.4e\n',err1,err2)
Naive error is     1.6468e-10, improved is     2.4513e-12
```

For random matrix order thousand, can improve accuracy by factor of ~100.

Does it Always Work?

$$x_{n+1} = x_n - A^\dagger(Ax_n - b) \quad \text{For approximate inverse } A^\dagger.$$
$$= (I - A^\dagger A)x_n + A^\dagger b$$

If we have a guess for the inverse (not necessarily very good), can show process will converge if absolute values of eigenvalues of $(I - A^\dagger A)$ are all < 1.

Try, say, $A^\dagger = aA^T$. Then if $a < \lambda_0^2$ for largest eigenvalue λ_0, eigenvalues of $aA^\dagger A$ will be between 0 and 1, and can iterate to convergence.

What to set for scale factor a? Optimal is 1/ mean(largest,smallest) eigenvalues of $A^\dagger A$. Some more choices listed in NR.

Why Bother With Crappy $A^\dagger$?

- Well, real A^{-1} can be very hard. Might not even have space to store A, let alone A^{-1}.
- We can get solution as long as we can calculate Ax, and $A^\dagger x$.
- Can use to solve extremely large linear least-squares problem.
- Also, given a direction for correction, can work out length for maximal reduction in χ^2. If $v=A^\dagger(Ax-b)$, then minimum of length of $(A(x-\alpha v)-b)$ is when $\alpha=y^T(Ax-b)/(y^Ty)$ where $y=Av$.

PCG From Command Line

```
-->n=50;a=rand(n,n,'normal')*0.1+eye(n,n);ai=a';

-->b=rand(n,1,'normal');x=ai*b;d=a*x-b;d'*d
 ans  =

    67.45285

-->v=ai*d;y=a*v;alph=(y'*d)/(y'*y);x=x-alph*v;d=a*x-b;d'*d
 ans  =

    6.5263402

-->v=ai*d;y=a*v;alph=(y'*d)/(y'*y);x=x-alph*v;d=a*x-b;d'*d
 ans  =

    4.0177392

-->v=ai*d;y=a*v;alph=(y'*d)/(y'*y);x=x-alph*v;d=a*x-b;d'*d
 ans  =

    3.2564334

-->v=ai*d;y=a*v;alph=(y'*d)/(y'*y);x=x-alph*v;d=a*x-b;d'*d
 ans  =

    2.784958

-->v=ai*d;y=a*v;alph=(y'*d)/(y'*y);x=x-alph*v;d=a*x-b;d'*d
 ans  =

    2.4347558
```

Scilab also has PCG for positive-definite matrices built in. Even easier! If applicable...

```
-->aa=a'*a;

-->[x,flag,err,iter]=pcg(aa,b);d=(aa*x-b);[err iter]
 ans  =

    4.371D-09    46.
```

Example: Mapmaking

- Common problem in astronomy: have observed a patch of sky, have lots of data. What is best map?
- Best in least-squares sense: $P^T N^{-1} P m = P^T N^{-1} \Delta$ for map m, pointing matrix P, noise N, and data Δ. P defined such that $\langle\Delta\rangle = Pm$.
- Example: ACT. Have 400 pts/sec, 1000 detectors, 6 hours of data per night. 400*1000*6*3600= 8e9 data points going to 5°x90° at 30" resolution = 6.5e6 pixels. Needless to say, only prayer is conjugate gradient.

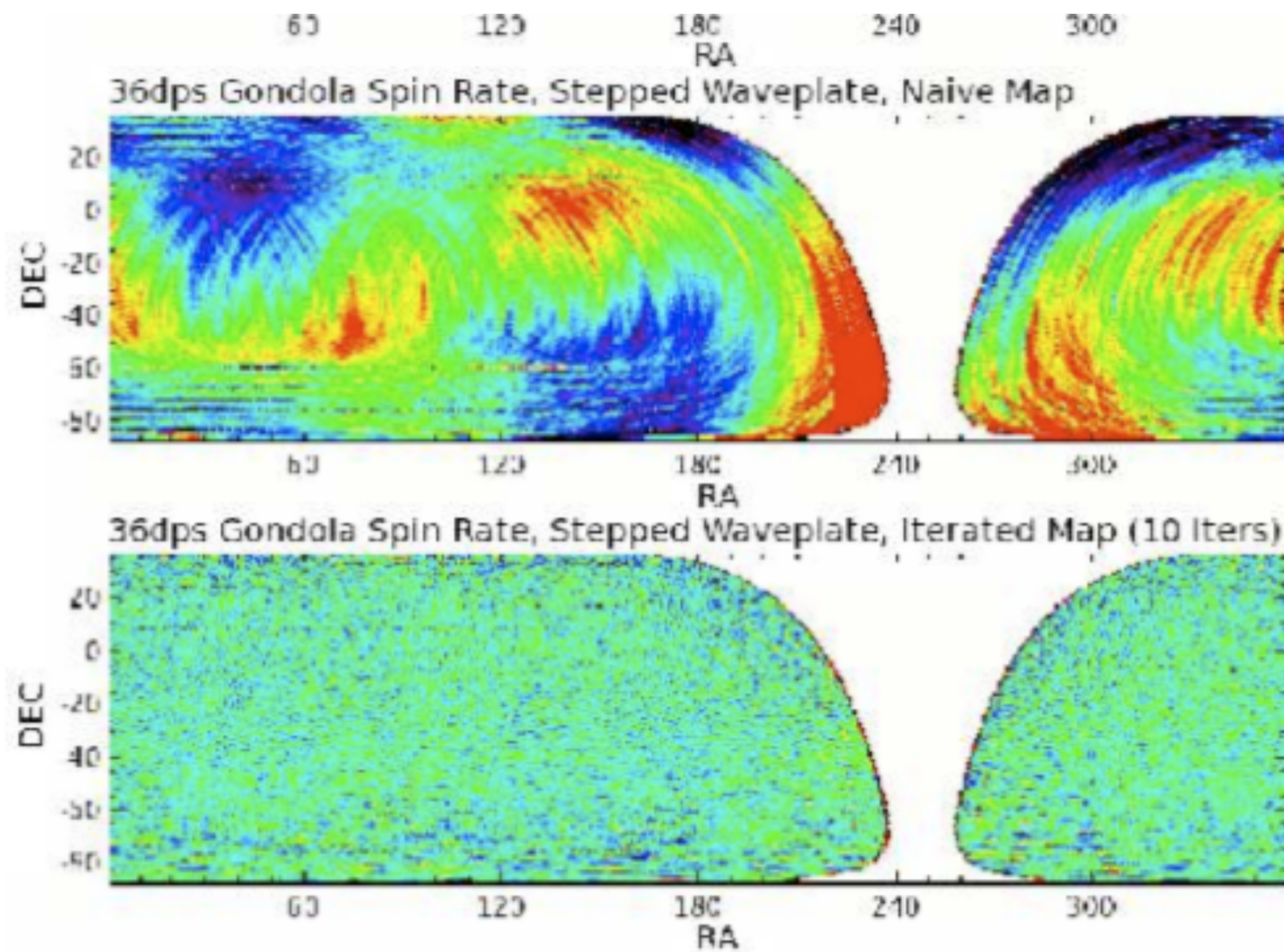
Mapmaking, Cont'd

- What are we doing with $P^T N^{-1} \Delta$? $N^{-1} \Delta$ is called pre-whitening. Makes noise uniform & uncorrelated for all outputs. P^T times that make something that looks like a map (e.g. "dirty map" in radio astronomy).
- How 'bout $P^T N^{-1} P m$? Pm is our guess at what the data should look like. Then we make the whitened dirty map again, do our comparison there.
- So, we make predicted & actual (whitened) dirty maps, iterate until they're the same.
- Works on all sorts of maps (CMB, drizzled HST, others)
- BTW, if P has 1 non-zero entry per row (beam is small), then $P^T P$ is diagonal - helps a bunch if noise diagonal!

A Note on Noise

- Note that we have to whiten the data. We can do this in *any* space. Often the noise is diagonal in Fourier space, so take FFT of data, divide by noise, FFT back to do $N^{-1}\Delta$.
- Common noise is 1/f. If so, then long wavelength modes can be extremely poorly measured, stripes in maps are common result. Two datasets taken in different directions ("cross-linked") can get rid of stripes.
- Often have to estimate noise at same time as map-making, especially if signal in map is comparable to noise.

Pictures!



Spider simulation from Mactavish *et al*. Top panel shows simple mapmaking with 1/f, bottom show mapmaking with cross-linking and 1/f accounted for.

Mactavish et al. 0710.0375

Nonlinear Least-Squares

- For data Δ, parameters *p*, and function F that predicts Δ from *p*, find best parameters *p*.
- General plan of attack for minimizing function: 1) calculate gradient and approximate curvature, and 2) go downhill until hit minimum.
- special form of χ^2 helps us:

$$\chi^2 = [F(p) - y]^T N^{-1} [F(p) - y]$$

Non-linear Least Squares, cont'd

$$\chi^2 = [F(p) - y]^T N^{-1} [F(p) - y]$$

$$\frac{\partial \chi^2}{\partial p_i} = 2 \frac{\partial F}{\partial p_i} N^{-1} [F(p) - y]$$

$$\frac{\partial^2 \chi^2}{\partial p_i p_j} = 2 \frac{\partial^2 F}{\partial p_i p_j} N^{-1} [F(p) - y] + 2 \frac{\partial F}{\partial p_i} N^{-1} \frac{\partial F}{\partial p_j}$$

If problem is were linear (or even "linear enough"), we'd be done. Move parameters by:

$$\delta p = -\left(\frac{\partial^2 \chi^2}{\partial p_i p_j}\right)^{-1} \frac{\partial \chi^2}{\partial p_i}$$

And we're at the peak. This is n-dimensional Newton's method for finding zeros of derivative. In strictly linear problem, first derivatives of F is our old friend A, and second derivatives of F are 0.

How ‘Bout Those Second Derivatives?

- The first term in the second derivative $F_{pp'} N^{-}1[F - y]$ has second derivatives of F. This could be both a pain to calculate, and expensive (since we need square of # of params).
- Shortcut! If we’re near the peak, then F(p)-y is going to be just a measurement of the noise, and should have on average similar number of positive & negative terms. So, toss it out, and only calculate second term: $F_p N^{-1} F_p$
- Reminder: we’re trying to get gradient of χ^2=0, so as long as we get there, doesn’t matter if we’re not perfect on curvature. There are many paths to salvation...

Notation: $\partial F/\partial p_i$ matrix shorthand is F_p, second deriv $F_{pp'}$, etc.

Non-linear Least Squares, yet again.

- OK - so we have a guess at gradient and curvature. What do we do with them?
- Simplest is to just take a Newton's method step. That ***often*** won't work. In that case, maybe just take a short step downhill. So, what sort of a step might we want to take? In 1-D, heading along -F' will head downhill if our step is small enough.
- Roughly how far should we go? Well, only scale we have is our second derivate. So, diagonal guess is $-\varepsilon F_p[\mathrm{diag}(F_{pp'})]^{-1}$.

Levenberg-Marquardt

- How to smoothly go from diagonal guess when we're bad to Newton's guess when we're good? Solution from Levenberg&Marquardt.
- Introduce control parameter λ, and use as our curvature the same one, but with diagonal scaled by $(1+ \lambda)$. If $\lambda=0$, then we take a Newton's method step. If λ large ($>>1$) then we're mincing gingerly downhill.
- Problem has now reduced to setting λ. If our step worked, then take it and shrink λ. If it failed, stay put and increase λ.
- In practice, we step downhill when we're far from a good solution, then jump to right answer when we're close.

A Note on λ

- ***If*** you have to code this yourself, I have strong feelings about what to do. Standard thing is to start with $\lambda=1$, then multiply/divide by factor of few each step.
- Horrible idea, IMHO. If we fail with $\lambda=10^{-10}$, probably not going to work with $\lambda=10^{-9}$, either. So might as well make λ at least of order unity right away. Also, if $\lambda<<1$, why not just make it zero? Yes, that was rhetorical.
- My modest proposal: start with $\lambda=0$. If all steps work (defined by decrease in χ^2), great! You had a simple problem. If you fail, make λ at least one. Increase as needs be to take a good step, then shrink slowly, since you're probably in a touchy region.

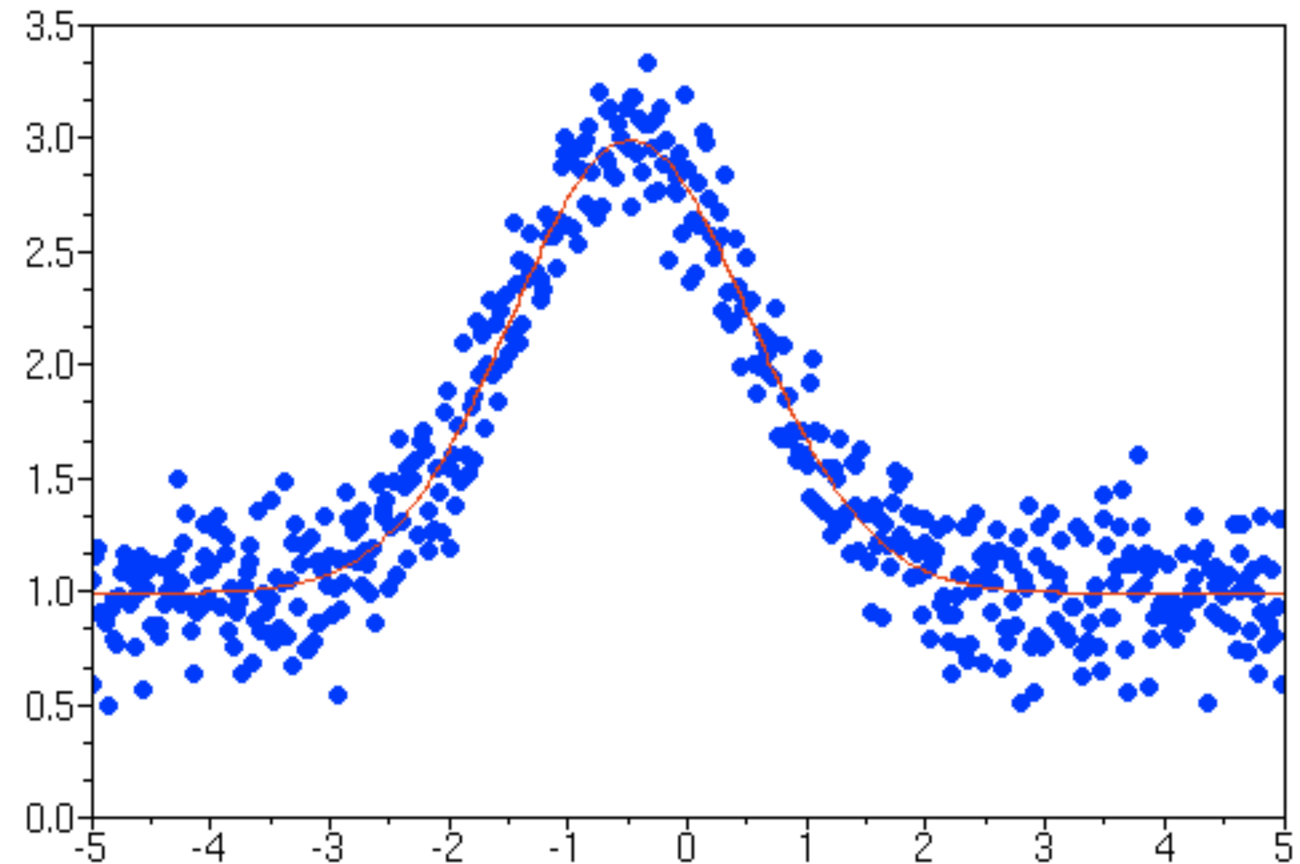
More Tips

- Coding derivatives can be a pain. Many algorithms use numerical derivatives. This can be your friend! If you ever are severely CPU limited, you'll have actually code, though.
- Always know how far to go. Function can flop around near minimum, so stop already! You'll need to tell your code how close to get.
- SCILAB has non-linear fitter: lsqrsolve. It takes a starting parameter vector, a function that return the error in y, the number of data points, optional derivative function, optional stopping criteria (highly recommended) and optional diagonal scalings.

In Practice:

```
-->getf('gausserr.sci')

-->[Y,X,params]=gaussdata();

-->[p2,v]=lsqrsolve(params+0.1,gausserr,length(X));

-->clf;plot(X,Y,'.');plot(X,Y-v','r')
```

```
1 function err=gausserr(params,m)
2   back=params(1)
3   amp=params(2)
4   cent=params(3)
5   width=params(4)
6   ypred=back+amp*exp(-0.5*(X-cent).^2/width^2);
7   err=Y-ypred;
8   //printf("error is %8.3f\n',sum(err.^2));
9 endfunction
10
11 function[Y,X,params]=gaussdata(params,X,noise)
12   if (~exists('noise'))
13     noise=0.2;
14   end
15   if ~exists('X')
16     X=-5:0.02:5;
17   end
18   if ~exists('params')
19     params=[1 2 -0.5 1];
20   end
21   Y=params(1)+params(2)*exp(-0.5*(X-params(3)).^2/params(4)^2);
22   Y=Y+noise*rand(size(Y,1),size(Y,2),'normal');
23 endfunction
```



Sample Gaussian fitter. Make a 1-D noisy Gaussian, fit to it.

Note: unfortunate use of hidden global variables. Makes me sad :-(...

And Now, Drop Standards Even More

- We often want to find not just best-fit parameters but error bars as well. Involves integrating *n*-dimensional function for *n* parameters. Not feasible for lots (i.e. >5-6) of params.
- If we had random samples from distribution, we could look at their statistics. On average, same as whole distribution.
- Markov-chain Monte Carlo (MCMC): simple way of generating random samples from distribution given χ^2 function.
- Same technique can also be used to find peaks of (otherwise) very annoying functions, under name of *simulated annealing*.

MCMC Cont'd

- Simple rule: if we're at position p_n, then try p_{n+1}. Accept if $\chi^2(p_{n+1}) < \chi^2(p)$ (*i.e.* we went downhill). Also accept sometimes even if we went uphill with probability $\exp(-(\chi^2(p_{n+1}) - \chi^2(p)))$.
- If we do this long enough, the accepted steps will sample the probability distribution.
- Note: I said nothing about how to guess p_{n+1} from p_n. The magic of MCMC: *it doesn't matter.* Any uniform way of guessing p_{n+1} from p_n will converge. Better guesses of p_{n+1} will converge *faster*, but won't converge to something different.

MCMC Sampling

- How big a step to take? Too large, we never accept. To small, will accept lots of steps, but they won't have much new information.
- Good solution: pull random samples from covariance matrix, perhaps with scaling.
- So, run chains for a while, then calculate covariances of chain elements to get covariance. Then continue, but sampling from new covariance guess.
- Not strictly speaking legal, but honestly, doesn't matter...

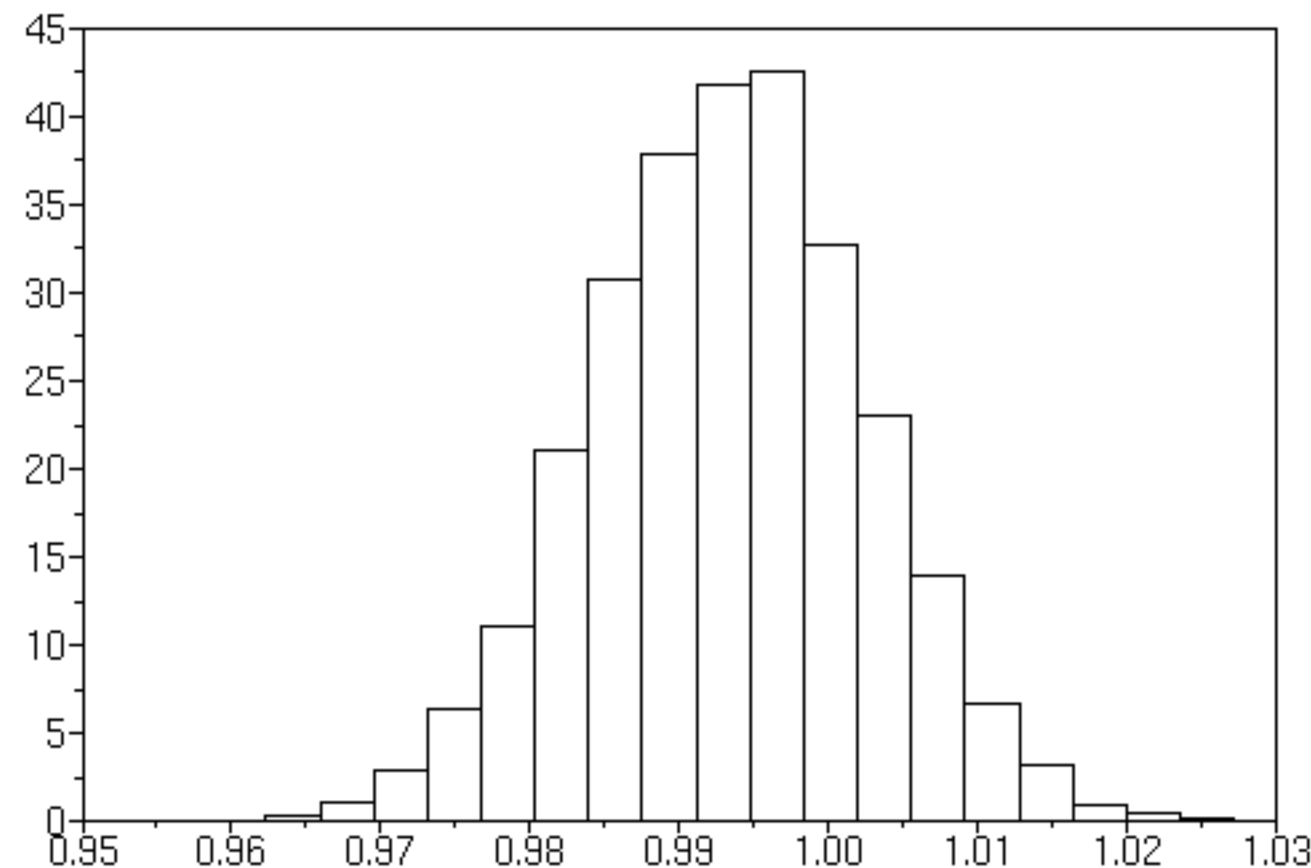
Aside: Making Correlated Gaussians

- Want set of y such that $<y_i y_j>=C_{ij}$.
- If we have $C=V\Lambda V^T$, then let $y=V\Lambda^{1/2}e$ for uncorrelated unit-variance Gaussian randoms e. Then $yy^T=V^T\Lambda^{1/2}ee^T\Lambda^{1/2}V$. On average, $ee^T=I$, so $<yyT>=V^T\Lambda^{1/2}I\Lambda^{1/2}V=V^T\Lambda V=C$. Done.
- Faster: Cholesky. $C=R^TR$, then $y=R^Te$, $yy^T=R^Tee^TR$. In expectation, eeT goes away again, leaving $<yy^T>=R^TR=C$.
- My experience: if close to singular, eigenvalues more stable, but not generally the case for MCMC parameters. Keep in mind if generating realizations of big matrices, though.

Scilab MCMC Example

```
34 function[chains]=simple_mcmc_gauss(params,nelem)
35   n1=round(0.1*nelem); //length of short initial chain
36   p_cur=params;
37   short_chains(n1,length(params))=0;
38   errs_start=0.02*params;
39   np=length(params);
40   chi_cur=simple_gauss_chisq(p_cur);
41   nunique=1;
42   short_chains(1,:)=p_cur;
43   for j=2:n1, //calculate initial chain to estimate covariance
44     p_try=p_cur+errs_start.*rand(1,np,'normal');
45     chi_try=simple_gauss_chisq(p_try);
46     if exp(chi_cur-chi_try)>rand(1,1,'uniform')
47       p_cur=p_try;
48       chi_cur=chi_try;
49       nunique=nunique+1;
50     end
51     short_chains(j,:)=p_cur;
52   end
53   printf("have %d unique out of %d elements in startup.\n',nunique,n1);
54   for j=1:np,
55     short_chains(:,j)=short_chains(:,j)-mean(short_chains(:,j));
56   end
57   nunique=1;myvar=short_chains'*short_chains/n1;
58   L=chol(myvar);
59   chains(nelem,np)=0;
60   chains(1,:)=short_chains(n1,:);
61   for j=2:nelem, //now make another chain, but using our calculated cov
62     p_try=p_cur+rand(1,np,'normal')*L;
63     chi_try=simple_gauss_chisq(p_try);
64     if exp(chi_cur-chi_try)>rand(1,1,'uniform')
65       nunique=nunique+1;
66       p_cur=p_try;
67       chi_cur=chi_try;
68     end
69     chains(j,:)=p_cur;
70   end
71   printf("have %d unique out of %d elements in main chain.\n',nunique,n
72 endfunction
```

```
-->getf('gausserr.sci')

-->[Y,X,params]=gaussdata();

-->chain=simple_mcmc_gauss(params,3e4);
have 327 unique out of 3000 elements in startup.
have 12339 unique out of 30000 elements in main chain.

-->[params;mean(chain,1);std(chain)']
 ans  =

    1.           2.          - 0.5          1.
    0.9933254    1.9789143   - 0.5089526    1.0364581
    0.0108959    0.0219050     0.0111258    0.0143981

-->histplot(20,chain(:,1))
```



Also uses routines for setting up data we used before.

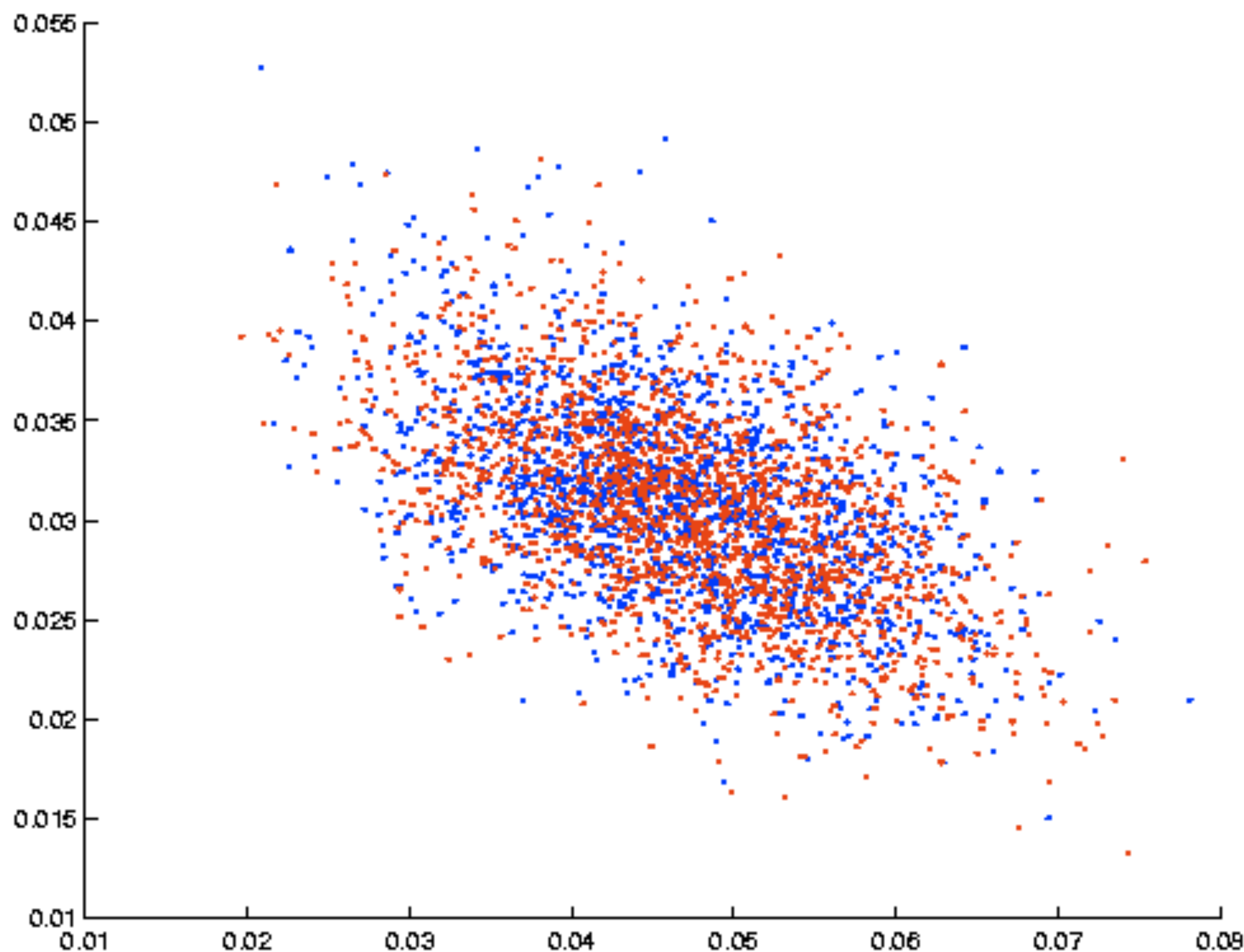
MCMC In Practice

- MCMC is like drinking: art is knowing how much is enough.
- Standard is to start multiple chains from several different locations. Quit when the things you care about from the different chains agree at a level you care about.
- 2,3-σ errors can be poorly determined without really long chains. If you want them, raise temperature of chain - accept step with probability $\exp(-\Delta\chi^2/T)$ for T~a few.
- Have to take account of this when calculating results.
- Also sometimes want bounds on parameters, correlations thereof. MCMC will give this to you as well.

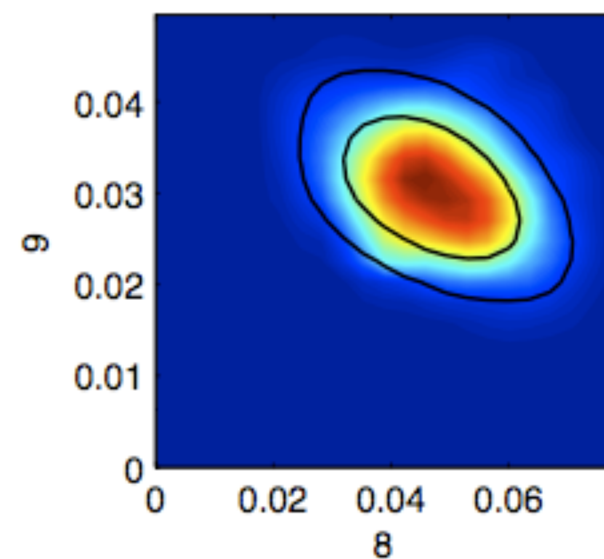
CosmoMC

- Cosmological parameters standard use of MCMC. Program CosmoMC (Lewis & Bridle) standard package for doing cosmological chains, program getdist to analyze them.
- getdist will work on any chain. Can put in limits in variables, account for temperature, limits on derived parameters, 2D plots, etc. Of course, may require hacking source code...
- If ask Antony nicely, can also use CosmoMC on generic problems, just have to write a χ^2 routine.

CosmoMC Fitting Radio Source Properties



Raw Chain Elements, 2 Bins
Red,blue are samples from two different chains.



CosmoMC output,
Same 2 bins.

Metropolis-Hastings (Simulated Annealing)

- Say want to minimize function. A major problem is where to start - there can be may local minima, but only one global. If near local minimum, then Levenberg-Marquardt will happily find it, but it's not what you wanted.
- MCMC good way of sampling large regions of parameter space. Start off with very high temperature, then run for a while, dropping temp. Does a good job of picking out global minima from lots of locals.
- Can often polish up with LM or some other method. Try lots of starting points, make sure end up in same spot.
- Can also be used on discrete problems that don't even have gradients - effectively solves traveling salesman.