any questions?

Of course…

- This likelihood is not right
- We are measuring *variance* of data. This is not χ^2.
- Real thing MCMC is working on is relative likelihood. Accept with probability like(new_state)/like(old_state).
- What is likelihood from power spectrum?

Back to Gaussian

- Gaussian PDF is $\exp(-(x-\mu)^2/2\sigma^2)/\mathrm{sqrt}(2\pi\sigma^2)$
- For χ^2 we could assume that σ is constant.
- This is not the case when we are estimating variances. The denominator matters.

Likelihood from Many Variables

- As usual, for independent gaussians, likelihood is just the product of individual likelihoods
- Also as usual, log likelihood is usually more convenient. For CMB, we can also set expectation $\mu=0$.
- $\log(L)=-1/2 \sum x^2/\sigma^2-1/2\sum\log(2\pi\sigma^2)$
- we can safely ditch 2π (unless you vary over non-Euclidean geometries…)
- Leaves $\log(L)=-1/2\ \chi 2-1/2\sum\log(\sigma^2)$

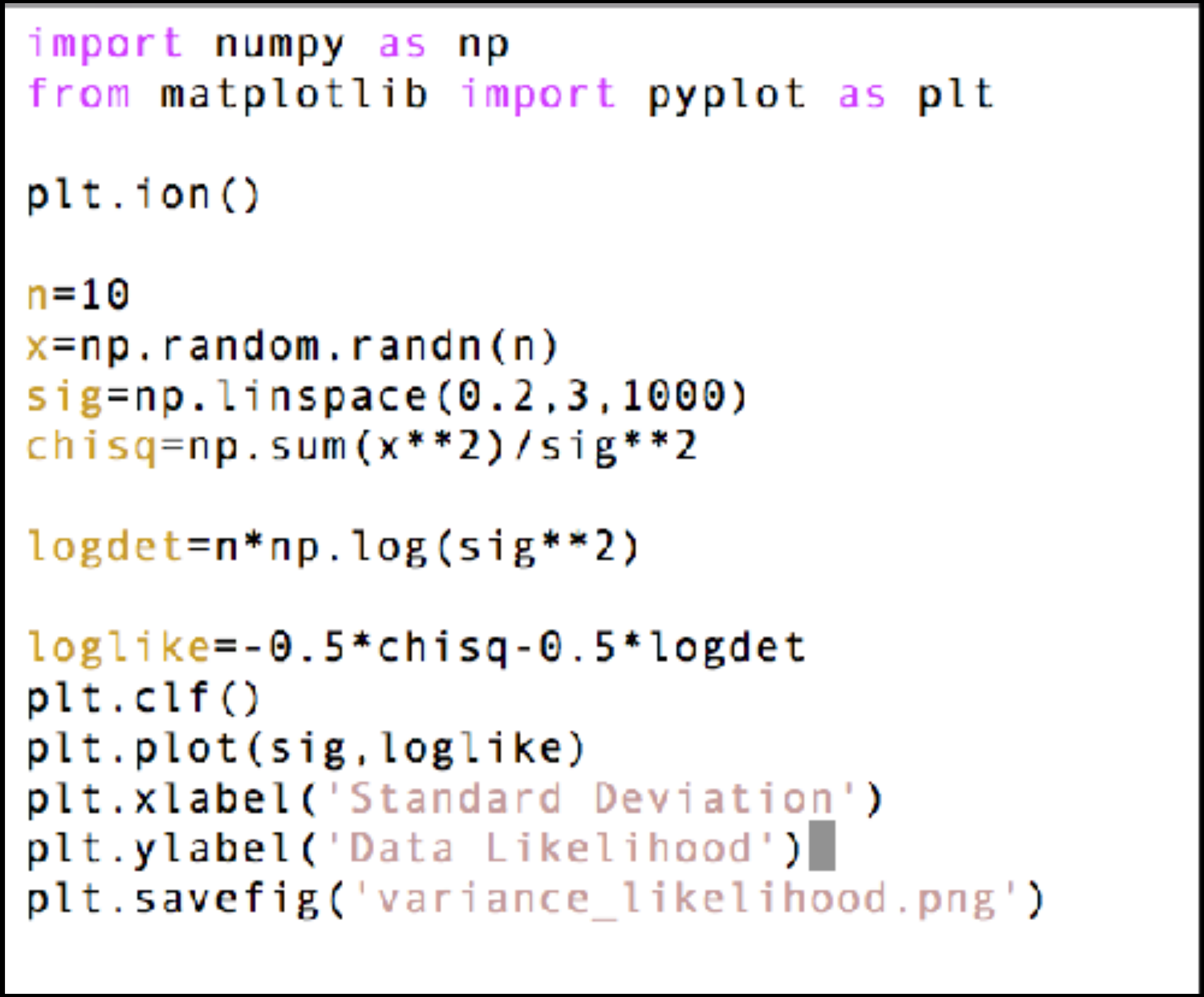
To Matrix Expression

- For uncorrelated data, $N_{ii}=\sigma_i^2$., $N_{ij}=0$ if $i\neq j$.
- Determinant of a diagonal matrix is product of diagonal elements. So, $\sum\log(\sigma_i^2)=\log(\text{product}(\sigma_i^2))=\log(|N|)$
- So, our final result is $-2\log(L)=d^TN^{-1}d+\log(|N|)$
- As you might expect, once we have a proper linear algebra expression, this also works for correlated noise.

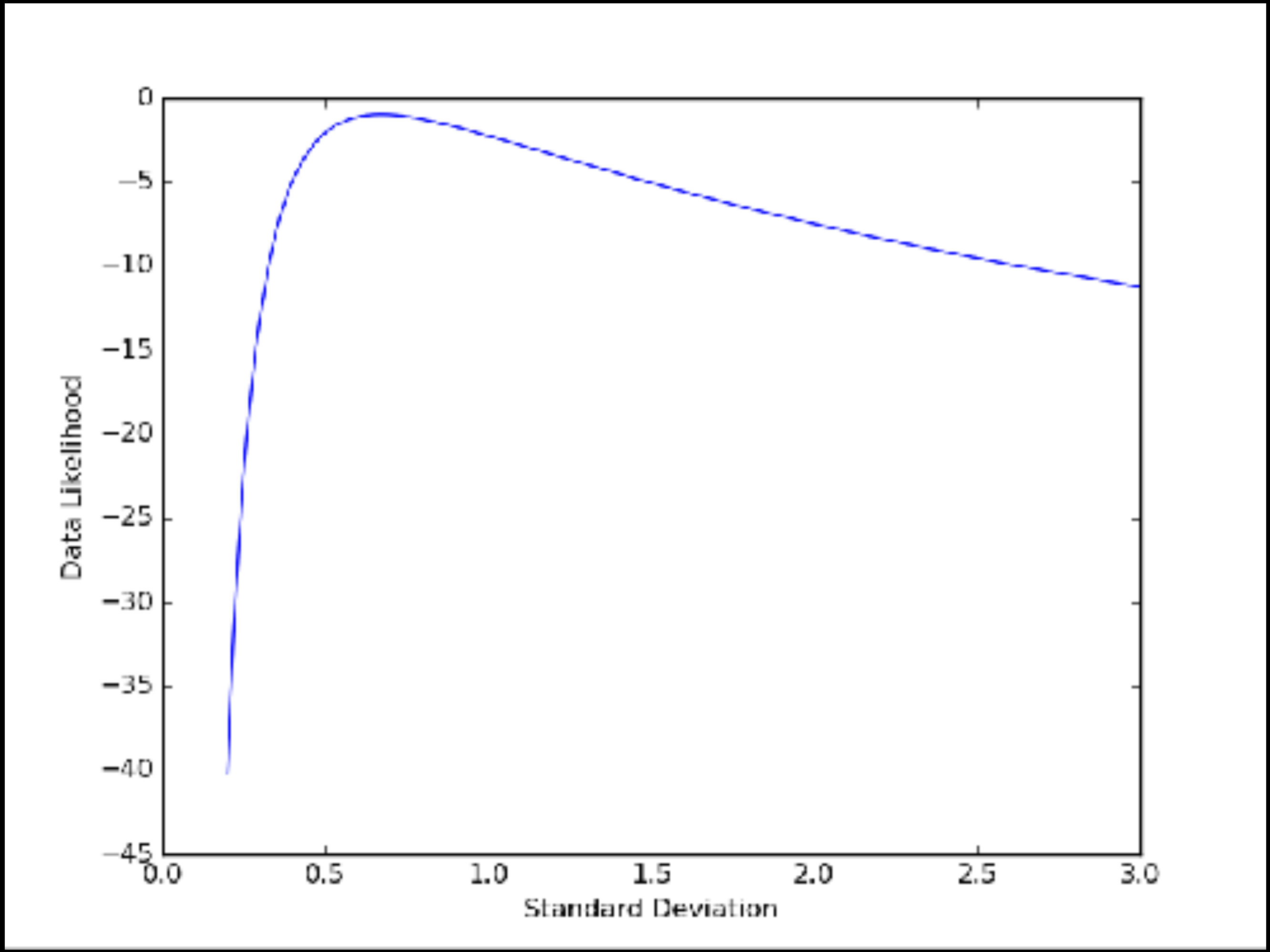
Likelihood Shape

- Once we have correlated data, we come up with a model for the noise, then calculate the likelihood of the observed data given this model.
- This model is *nonlinear* - we can't write down a global solution like we could with χ^2.
- The probability we roll the dice and get very large variance is exponentially small.
- The probability we roll the dice and get very small variance is polynomial small. Shape is skewed.

Likelihood Shape



```
import numpy as np
from matplotlib import pyplot as plt

plt.ion()

n=10
x=np.random.randn(n)
sig=np.linspace(0.2,3,1000)
chisq=np.sum(x**2)/sig**2

logdet=n*np.log(sig**2)

loglike=-0.5*chisq-0.5*logdet
plt.clf()
plt.plot(sig,loglike)
plt.xlabel('Standard Deviation')
plt.ylabel('Data Likelihood')
plt.savefig('variance_likelihood.png')
```



Fitting a Power Spectrum

- This shape is important for low-ell, e.g. measurements of the optical depth to reionization.
- How would we estimate the power spectrum?
- $<d_i d_j>=N_{ij,\mathrm{instrumental}}+\mathrm{Cov}_{ij}(\mathrm{sky})$
- $\mathrm{Cov}_{ij}(\mathrm{sky})=\sum Y_{lm}(\theta_i \phi_j) Y_{lm}(\theta_j,\phi_j) C_l$.
- If we know the instrumental noise, we can tweak the C_l to maximize the likelihood.
- Looking at the likelihood surface gives us the correlations between the C_l. This gives an estimate of error bars.

Likelihood Codes

- Rather than the simplistic χ^2, in real life we'll have to use a full likelihood. Important for optical depth, n_s, tensor-to-scalar ratio…
- Experiments these days usually put out likelihood code that does this properly.
- WMAP has done so, to get it right you actually need maps and noise covariance matrices.
- Do this if you want to do parameters in real life!

Conjugate Gradient

- Problem with steepest descent is new directions tend to lie along old directions.
- If I had a way to stop that from happening, maybe this would work?
- Conjugate gradient is a way to solve Ax=b (where in this case, this A equals our usual $A^TN^{-1}A$) where new steps are A-orthogonal, i.e. $m^\dagger_i A m^\dagger_j=0$ for $i \neq j$.
- Magically, conjugate gradient can stay A-orthogonal to all previous directions without remembering what those directions were!

Conjugate Gradient Recipe

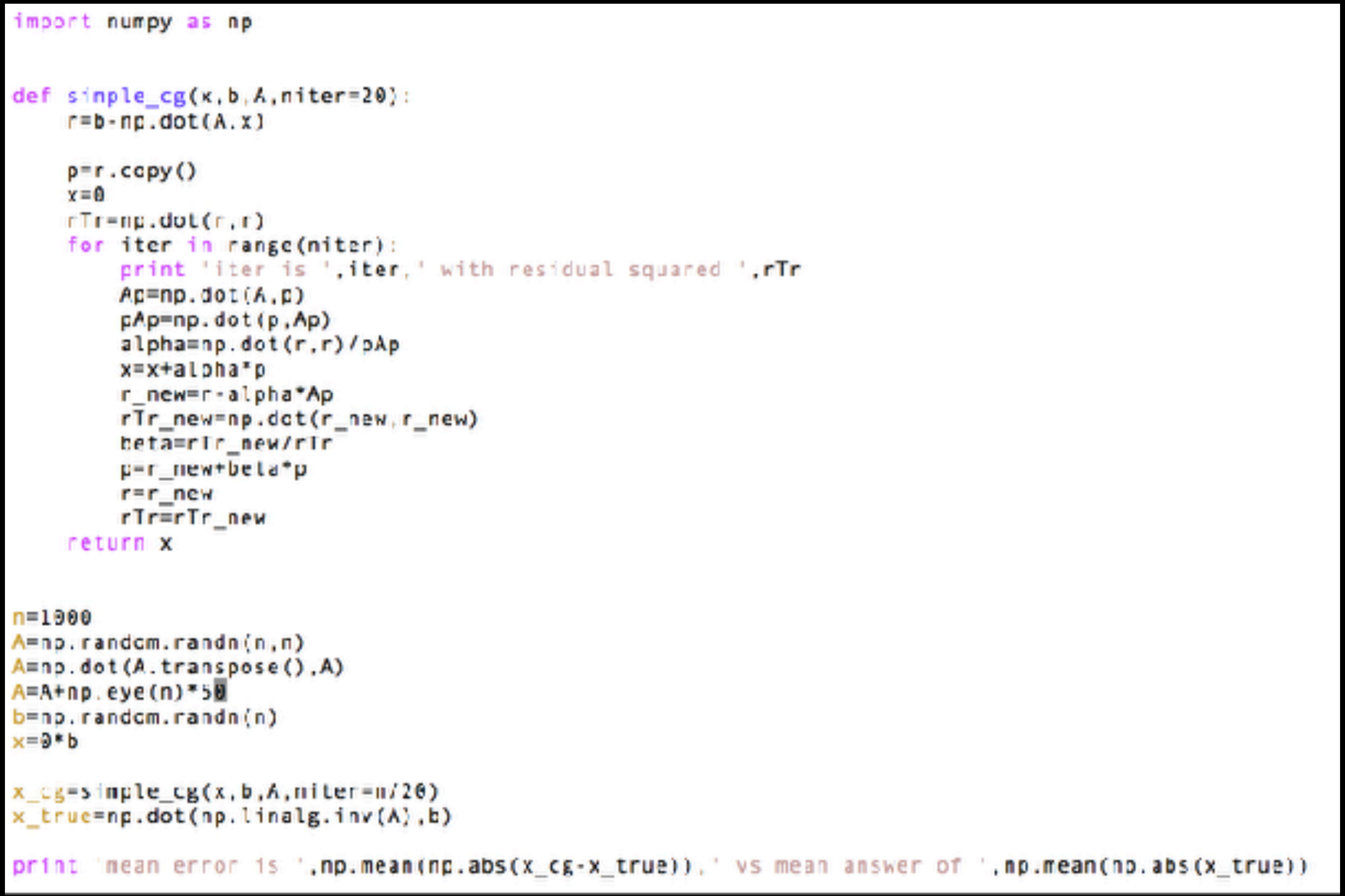
- $r_0 = b - Ax_0$ ($b = A_P^T N^{-1} d$, where A_p is our usual pointing matrix)
- $p_0 = r_0$.
- repeat over k:
- $\alpha_k = r_k^T r_k / P_k^T A p_k$
- $x_{k+1} = x_{k+} \alpha_k p_k$
- $r_{k+1} = r_k - \alpha_k p_k$. (if r_{k+1} “small enough”, quit)
- $\beta_k = r_{k+1}^T r_{k+1} / r_k^T r_k$.
- $p_{k+1} = r_{k+1} + \beta_k p_k$.

What’s going on?

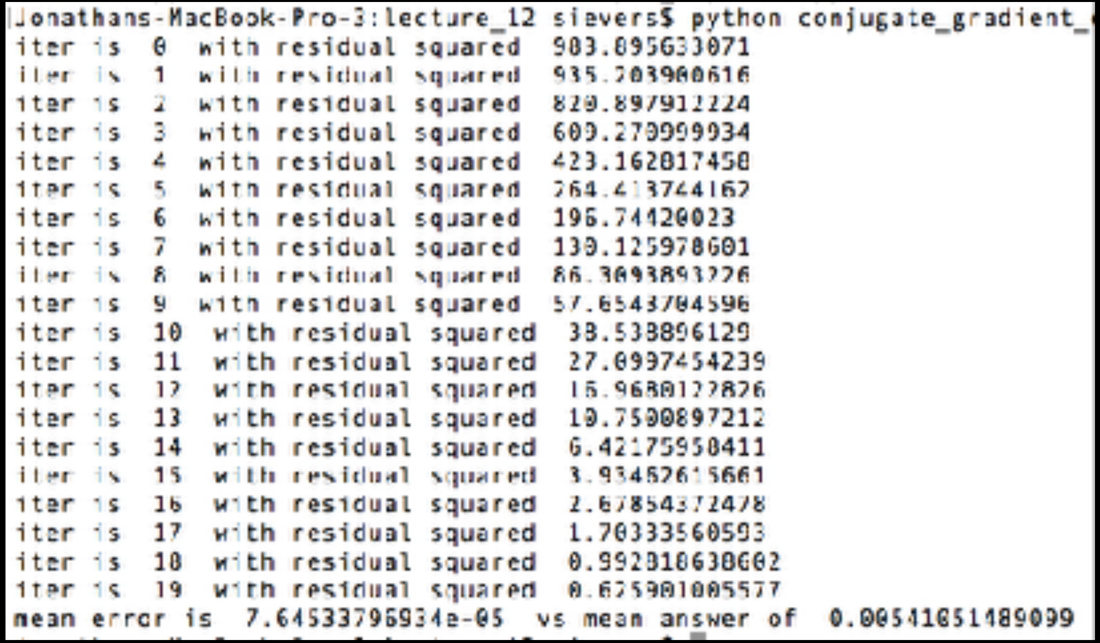
- Not obvious, but CG starts downhill, minimizes, then chooses new direction.
- New direction chosen so that minimizing along it doesn’t mess up previous minimizations.
- In practice, first minimization zeroes out residual along largest eigenvalue.
- Next step zaps next eigenvalue residual, etc.
- Guaranteed to converge after *n* iterations.
- Each step takes n^2 operations, so total work is n^3. Similar to direct.

CG Code

- Turns out, if many eigenvalues are repeated, CG will zap all of them in one step.
- In practice, many problems have limited eigenvalue spectrum. CG can converge in many fewer than n steps then.
- If $A^TN^{-1}A$ is singular, is has zero eigenvalues. Those get left for last, so CG helpfully leaves solution unchanged.



```
import numpy as np


def simple_cg(x,b,A,niter=20):
    r=b-np.dot(A,x)

    p=r.copy()
    x=0
    rTr=np.dot(r,r)
    for iter in range(niter):
        print 'iter is ',iter,' with residual squared ',rTr
        Ap=np.dot(A,p)
        pAp=np.dot(p,Ap)
        alpha=np.dot(r,r)/pAp
        x=x+alpha*p
        r_new=r-alpha*Ap
        rTr_new=np.dot(r_new,r_new)
        beta=rTr_new/rTr
        p=r_new+beta*p
        r=r_new
        rTr=rTr_new
    return x


n=1000
A=np.random.randn(n,n)
A=np.dot(A.transpose(),A)
A=A+np.eye(n)*50
b=np.random.randn(n)
x=0*b

x_cg=simple_cg(x,b,A,niter=n/20)
x_true=np.dot(np.linalg.inv(A),b)

print 'mean error is ',np.mean(np.abs(x_cg-x_true)),' vs mean answer of ',np.mean(np.abs(x_true))
```



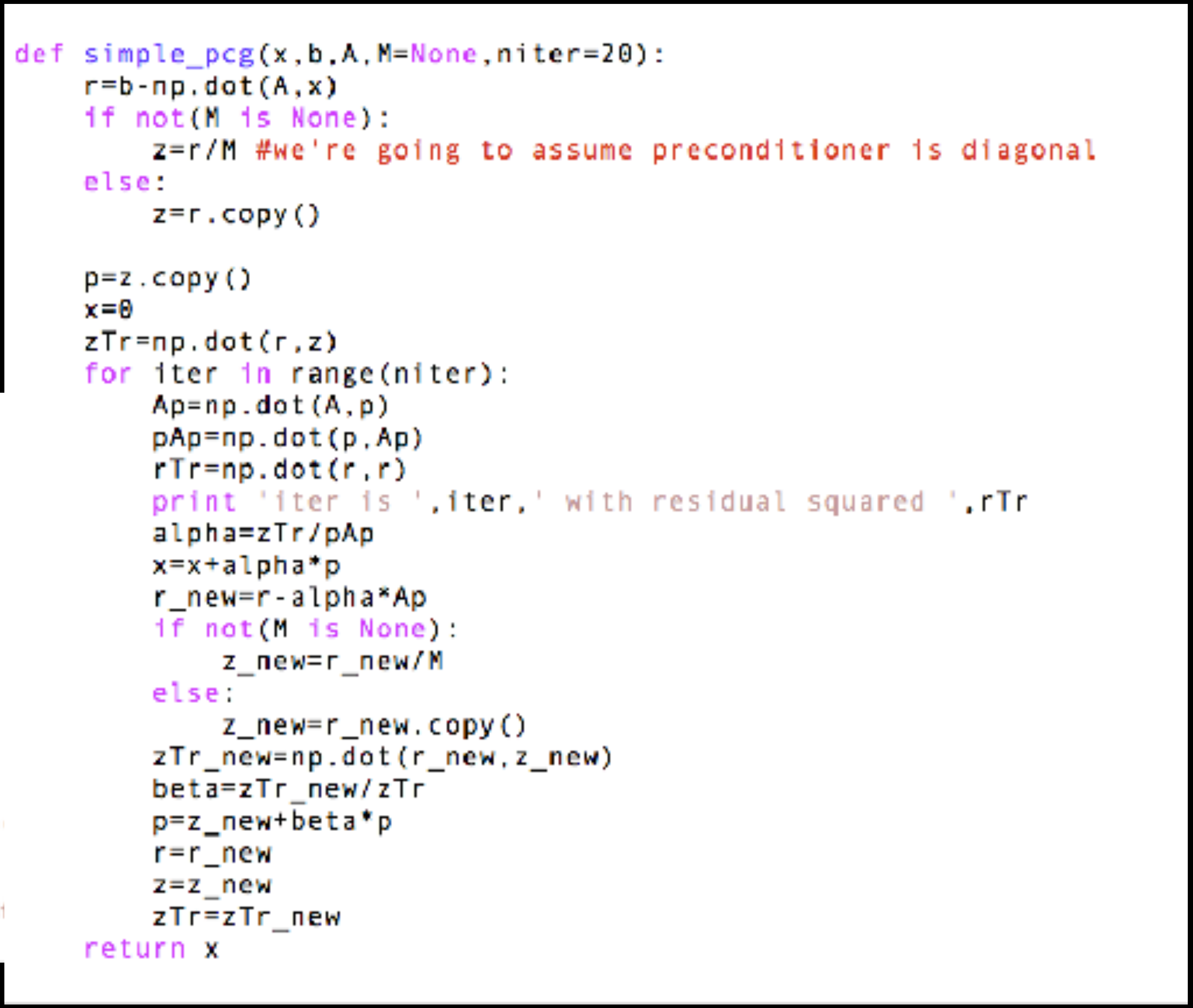
```
Jonathans-MacBook-Pro-3:lecture_12 sievers$ python conjugate_gradient_
iter is  0  with residual squared  983.895633071
iter is  1  with residual squared  935.203900616
iter is  2  with residual squared  820.897912224
iter is  3  with residual squared  609.270999934
iter is  4  with residual squared  423.162017458
iter is  5  with residual squared  264.413744162
iter is  6  with residual squared  196.74420023
iter is  7  with residual squared  130.125978601
iter is  8  with residual squared  86.3093893226
iter is  9  with residual squared  57.6543704596
iter is  10  with residual squared  38.538896129
iter is  11  with residual squared  27.0997454239
iter is  12  with residual squared  16.9680122826
iter is  13  with residual squared  10.7500897212
iter is  14  with residual squared  6.42175950411
iter is  15  with residual squared  3.93462615661
iter is  16  with residual squared  2.67854372478
iter is  17  with residual squared  1.70333560593
iter is  18  with residual squared  0.992818638602
iter is  19  with residual squared  0.625901005577
mean error is  7.64533796934e-05  vs mean answer of  0.00541051489099
```

Adding constant to diagonal shrinks eigenvalue spread, makes CG converge faster.

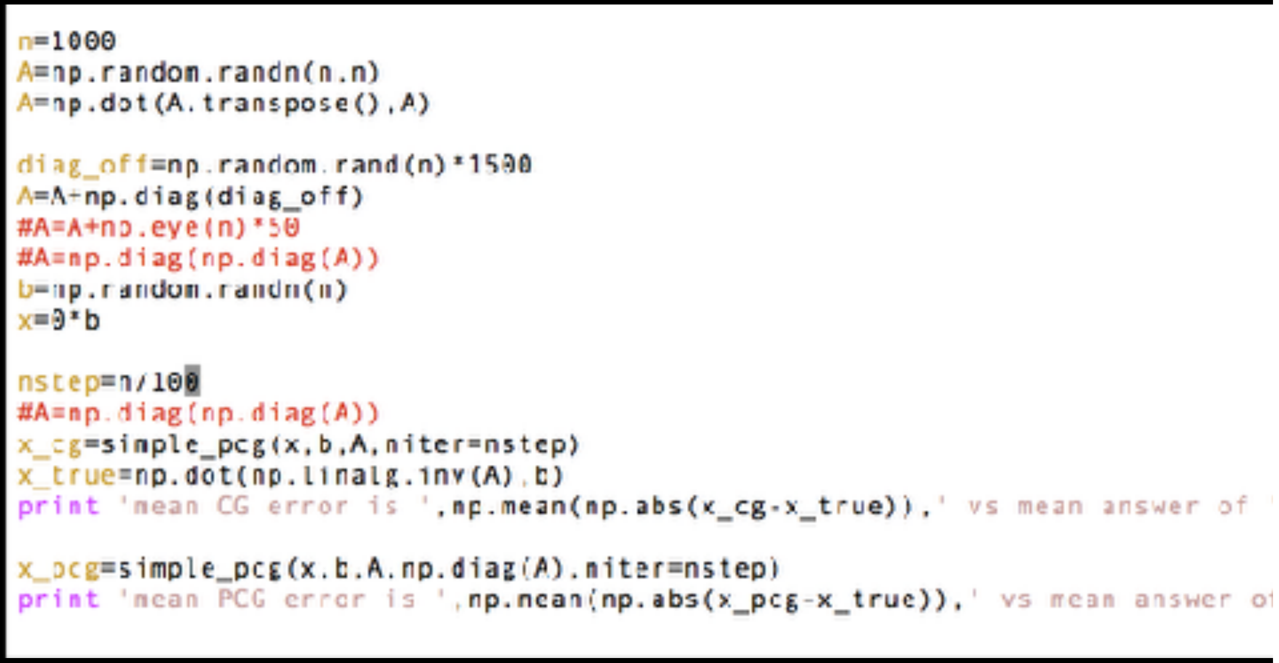
Preconditioned Conjugate Gradient

- If I have some guess as to A^{-1}, call it $A^{\dagger}$, then I could solve $A^{\dagger}Ax=A^{\dagger}b$.
- If spread in eigenvalues of $A^{\dagger}A$ is smaller than in A, then I'll converge faster.
- Preconditioned conjugate gradient is a way of putting in an $A^{\dagger}$. One thing guaranteed to improve matters is 1/diag(A) (called Jacobi preconditioner)

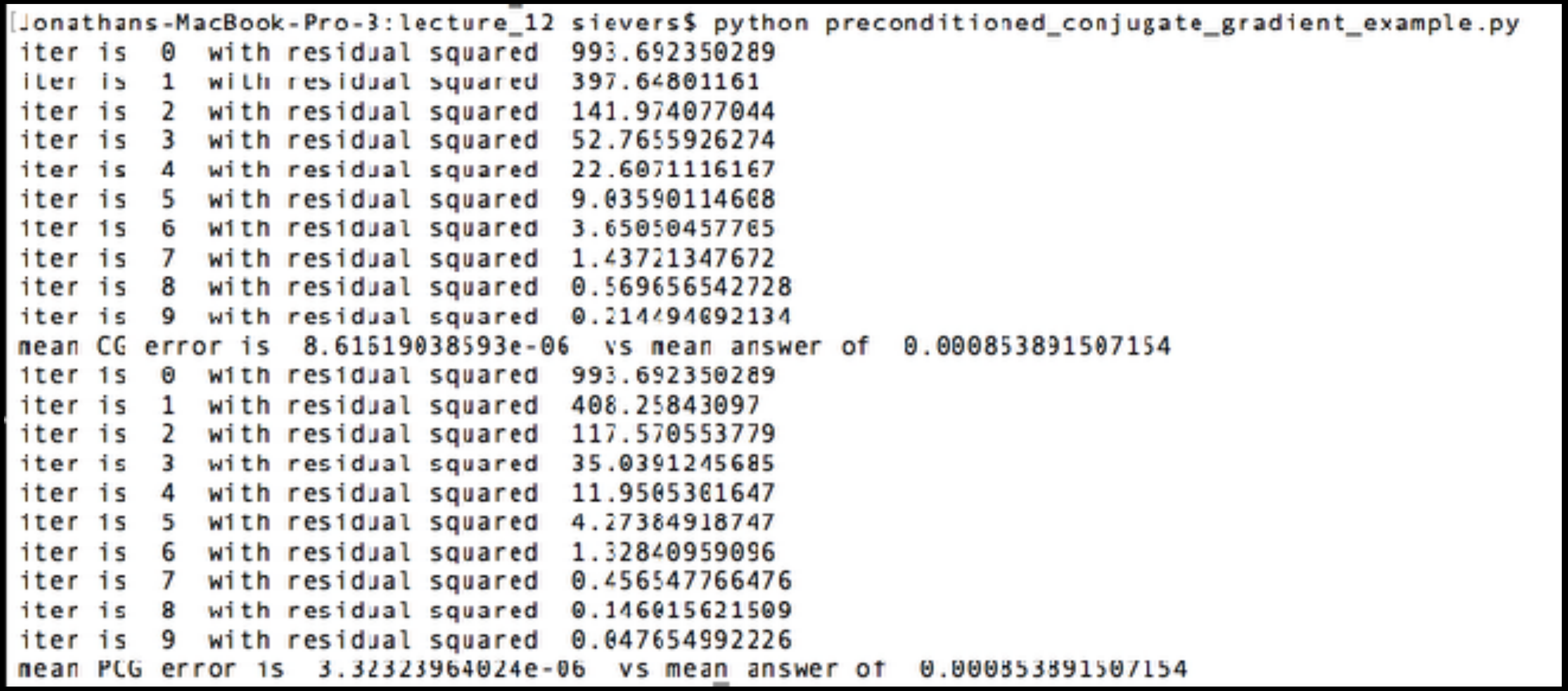
PCG Example



```python
def simple_pcg(x,b,A,M=None,niter=20):
    r=b-np.dot(A,x)
    if not(M is None):
        z=r/M #we're going to assume preconditioner is diagonal
    else:
        z=r.copy()

    p=z.copy()
    x=0
    zTr=np.dot(r,z)
    for iter in range(niter):
        Ap=np.dot(A,p)
        pAp=np.dot(p,Ap)
        rTr=np.dot(r,r)
        print 'iter is ',iter,' with residual squared ',rTr
        alpha=zTr/pAp
        x=x+alpha*p
        r_new=r-alpha*Ap
        if not(M is None):
            z_new=r_new/M
        else:
            z_new=r_new.copy()
        zTr_new=np.dot(r_new,z_new)
        beta=zTr_new/zTr
        p=z_new+beta*p
        r=r_new
        z=z_new
        zTr=zTr_new
    return x
```



```python
n=1000
A=np.random.randn(n,n)
A=np.dot(A.transpose(),A)

diag_off=np.random.rand(n)*1500
A=A+np.diag(diag_off)
#A=A+np.eye(n)*50
#A=np.diag(np.diag(A))
b=np.random.randn(n)
x=0*b

nstep=n/100
#A=np.diag(np.diag(A))
x_cg=simple_pcg(x,b,A,niter=nstep)
x_true=np.dot(np.linalg.inv(A),b)
print 'mean CG error is ',np.mean(np.abs(x_cg-x_true)),' vs mean answer of '

x_pcg=simple_pcg(x,b,A,np.diag(A),niter=nstep)
print 'mean PCG error is ',np.mean(np.abs(x_pcg-x_true)),' vs mean answer of
```



```
Jonathans-MacBook-Pro-3:lecture_12 sievers$ python preconditioned_conjugate_gradient_example.py
 iter is  0  with residual squared  993.692350289
 iter is  1  with residual squared  397.64801161
 iter is  2  with residual squared  141.974077044
 iter is  3  with residual squared  52.7655926274
 iter is  4  with residual squared  22.6071116167
 iter is  5  with residual squared  9.03590114608
 iter is  6  with residual squared  3.65050457765
 iter is  7  with residual squared  1.43721347672
 iter is  8  with residual squared  0.569656542728
 iter is  9  with residual squared  0.214494692134
mean CG error is  8.61519038593e-06  vs mean answer of  0.000853891507154
 iter is  0  with residual squared  993.692350289
 iter is  1  with residual squared  408.25843097
 iter is  2  with residual squared  117.570553779
 iter is  3  with residual squared  35.0391245685
 iter is  4  with residual squared  11.9505301647
 iter is  5  with residual squared  4.27384918747
 iter is  6  with residual squared  1.32840959096
 iter is  7  with residual squared  0.456547766476
 iter is  8  with residual squared  0.146015621509
 iter is  9  with residual squared  0.047654992226
mean PCG error is  3.32323964024e-06  vs mean answer of  0.000853891507154
```

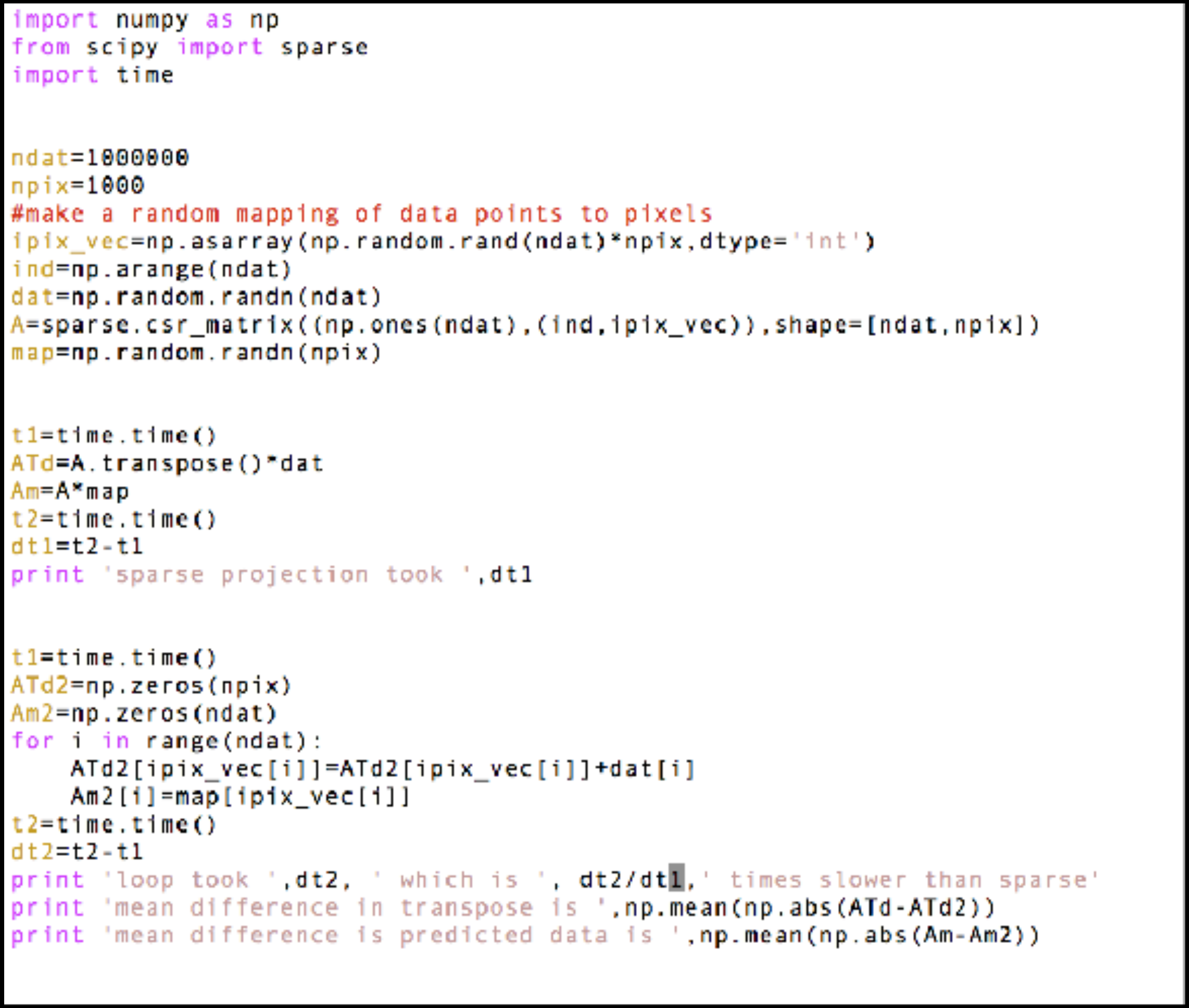
Sparseness

- Pointing matrix is very sparse - one entry per line.
- In python, could do this with (where A is now a vector with entries corresponding to non-zero column):

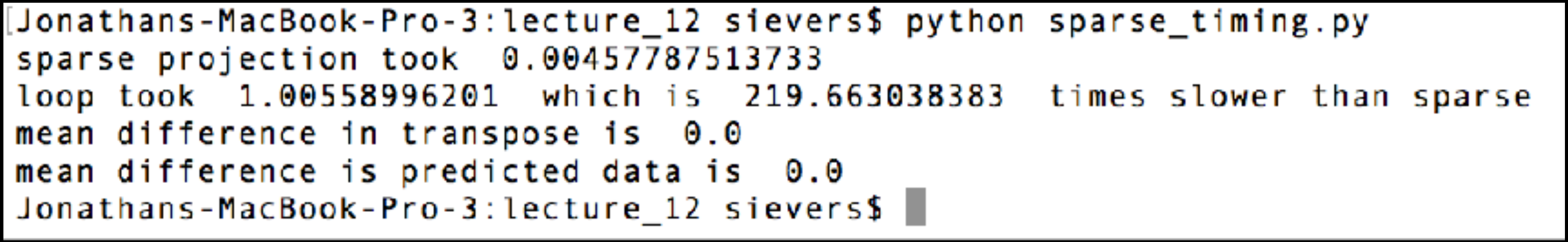
```
for i in range(n):
  Am[i]=m[A[i]]
```

- Extremely slow! However, python has sparse matrix class in scipy.sparse. Essentially, only stores non-zero elements of A.
- A_sparse=sparse.csr_matrix((dat,(range(n),A)) will make a sparse matrix we can use as our usual A. Much faster!

Huge Speed Difference



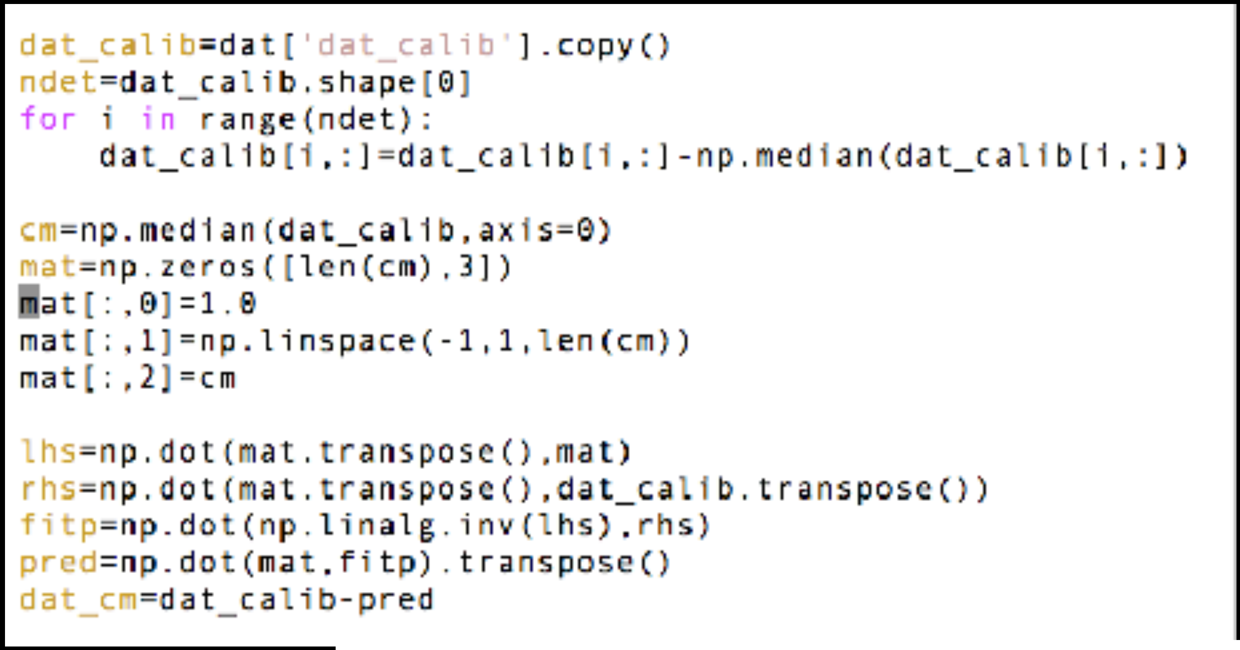
```
import numpy as np
from scipy import sparse
import time


ndat=1000000
npix=1000
#make a random mapping of data points to pixels
ipix_vec=np.asarray(np.random.rand(ndat)*npix,dtype='int')
ind=np.arange(ndat)
dat=np.random.randn(ndat)
A=sparse.csr_matrix((np.ones(ndat),(ind,ipix_vec)),shape=[ndat,npix])
map=np.random.randn(npix)


t1=time.time()
ATd=A.transpose()*dat
Am=A*map
t2=time.time()
dt1=t2-t1
print 'sparse projection took ',dt1


t1=time.time()
ATd2=np.zeros(npix)
Am2=np.zeros(ndat)
for i in range(ndat):
    ATd2[ipix_vec[i]]=ATd2[ipix_vec[i]]+dat[i]
    Am2[i]=map[ipix_vec[i]]
t2=time.time()
dt2=t2-t1
print 'loop took ',dt2, ' which is ', dt2/dt1,' times slower than sparse'
print 'mean difference in transpose is ',np.mean(np.abs(ATd-ATd2))
print 'mean difference is predicted data is ',np.mean(np.abs(Am-Am2))
```



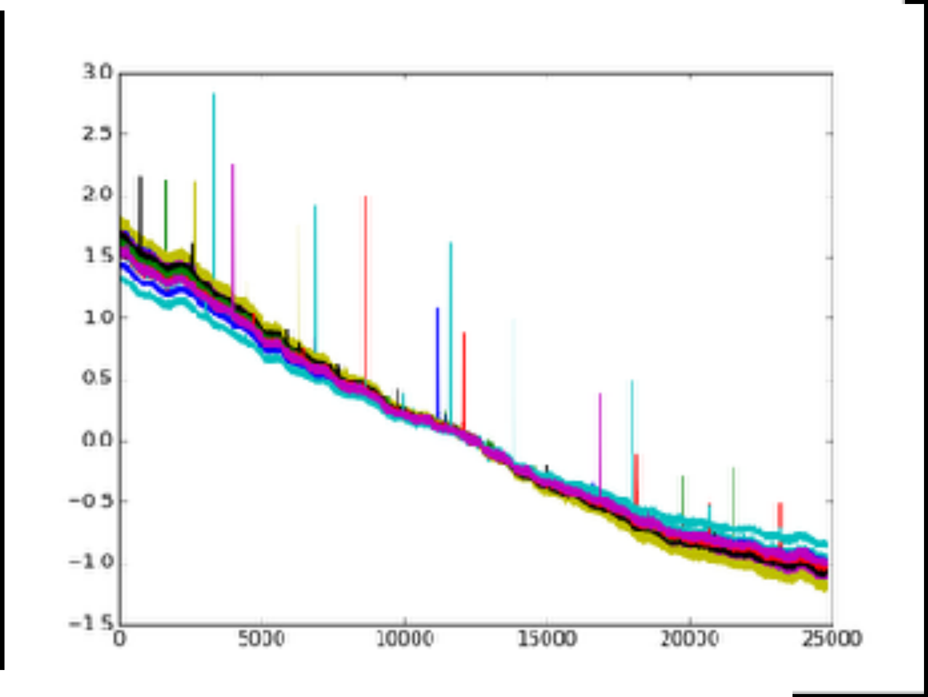
```
[Jonathans-MacBook-Pro-3:lecture_12 sievers$ python sparse_timing.py
sparse projection took  0.00457787513733
loop took  1.00558996201  which is  219.663038383  times slower than sparse
mean difference in transpose is  0.0
mean difference is predicted data is  0.0
Jonathans-MacBook-Pro-3:lecture_12 sievers$
```

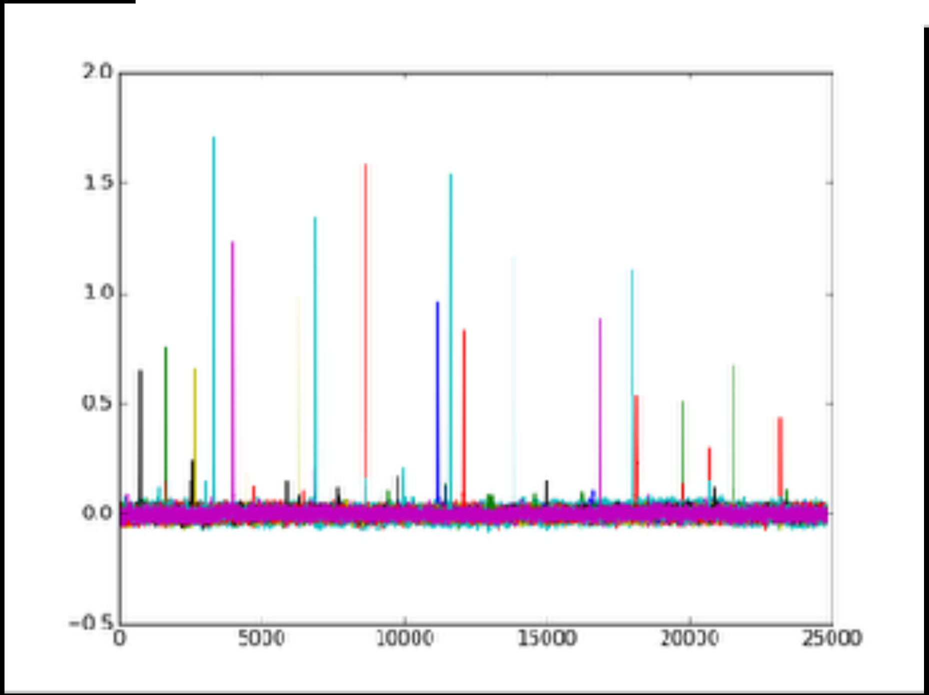
Putting Pieces Together

- Let's look at some actual ground-based data.
- In a very dangerous step, we'll subtract off common mode/detector drifts.
- Middle: raw data.
- Bottom:data with best-fit line+cm amplitude subtracted
- Scan is of a calibrator, so spikes upwards are sky signal.



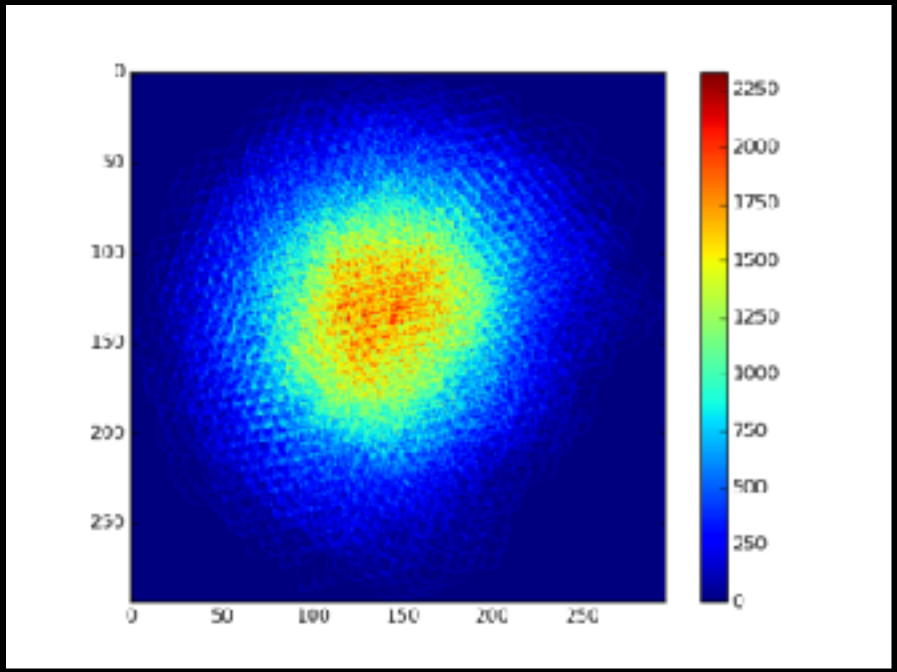
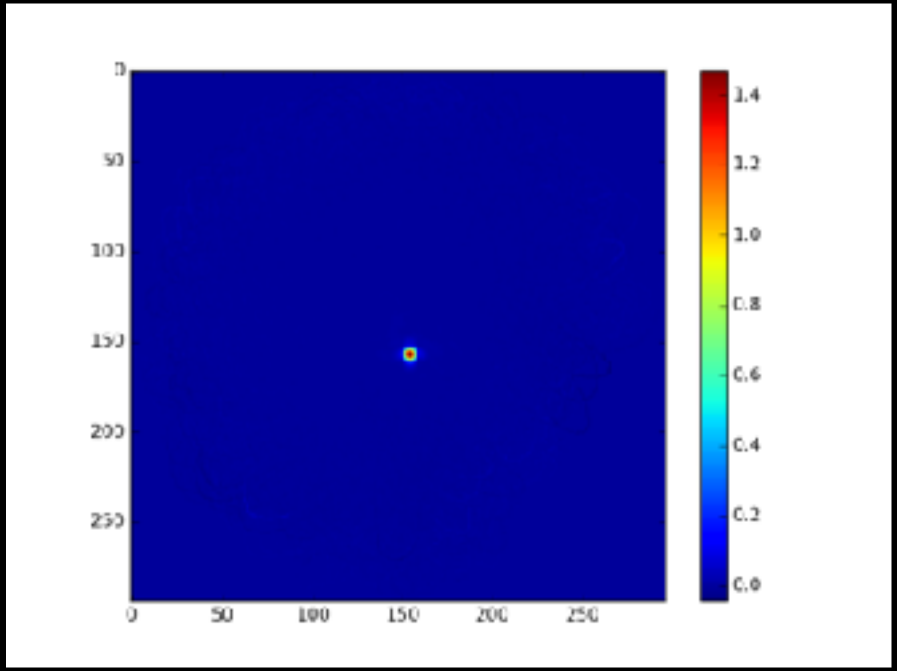
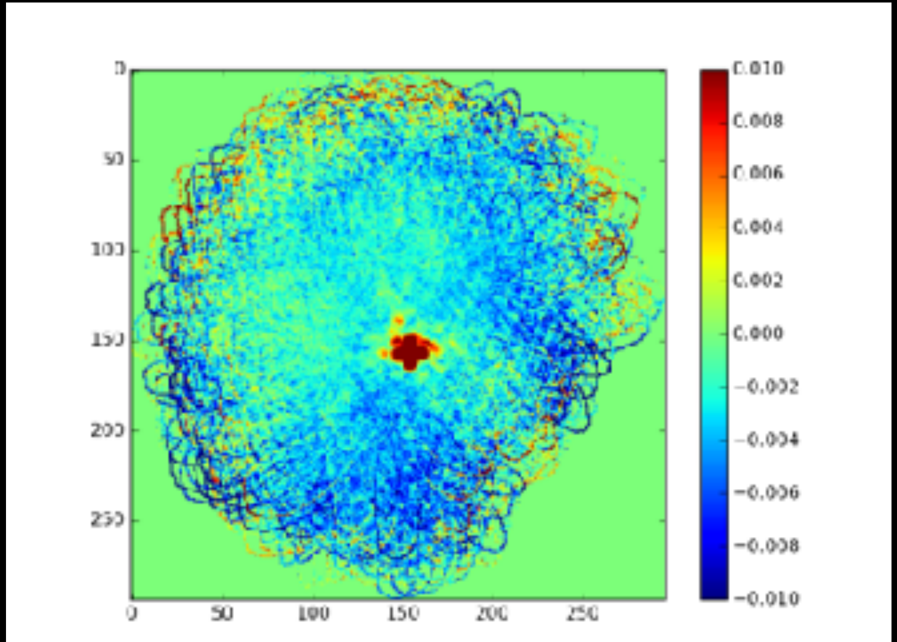
```
dat_calib=dat['dat_calib'].copy()
ndet=dat_calib.shape[0]
for i in range(ndet):
    dat_calib[i,:]=dat_calib[i,:]-np.median(dat_calib[i,:])

cm=np.median(dat_calib,axis=0)
mat=np.zeros([len(cm),3])
mat[:,0]=1.0
mat[:,1]=np.linspace(-1,1,len(cm))
mat[:,2]=cm

lhs=np.dot(mat.transpose(),mat)
rhs=np.dot(mat.transpose(),dat_calib.transpose())
fitp=np.dot(np.linalg.inv(lhs),rhs)
pred=np.dot(mat,fitp).transpose()
dat_cm=dat_calib-pred
```





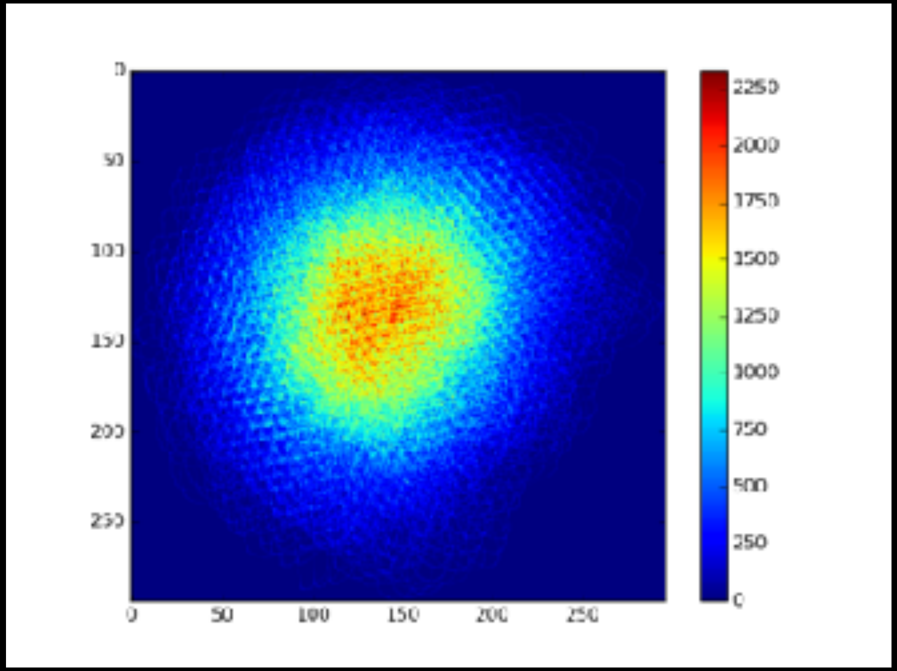
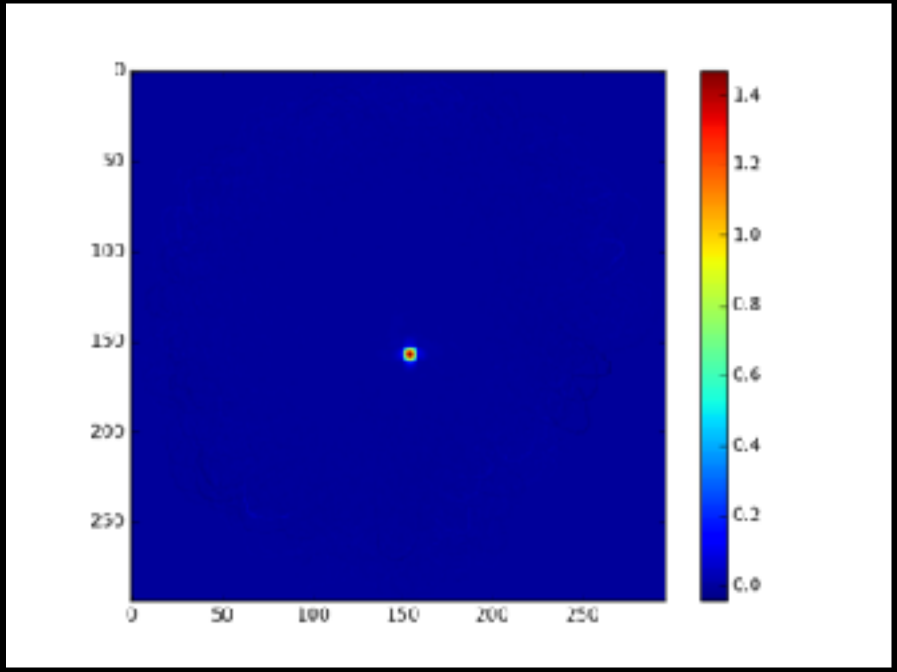
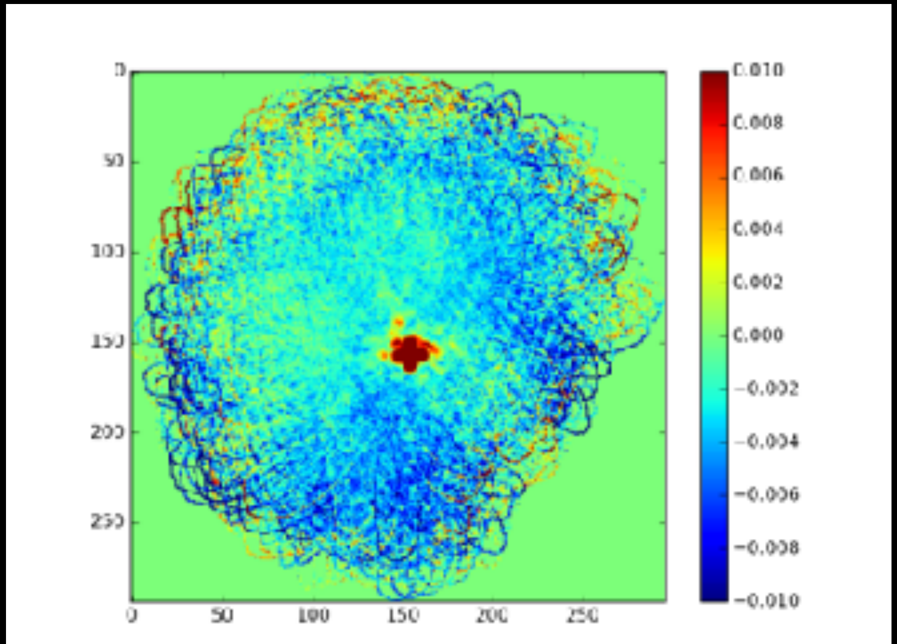
Naive Map

- In fact, naive map of cm-subtracted data is not so awful, at least visually.
- Top: hits. Middle: naive map of CM-subtracted data. Bottom: stretched colorscale.
- Why would this naive map not be good?
- What could we do better?

Naive Map

- In fact, naive map of cm-subtracted data is not so awful, at least visually.
- Top: hits. Middle: naive map of CM-subtracted data. Bottom: stretched colorscale.
- Why would this naive map not be good?
    - CM subtraction removes signal. usually a function of angular scale - introduces a *transfer function*, where map is biased.
- What could we do better?
    - convert CM into a noise, and use a real guess for the noise.

Common Mode as Noise

- A not-crazy model for the noise is each detector has independent noise, plus one noise term constant across all detectors.
- Noise will be a function of frequency, so subtract CM to estimate non-CM part of noise. Do this in Fourier space, as usual.
- What would this noise matrix look like?

Common Mode as Noise

- A not-crazy model for the noise is each detector has independent noise, plus one noise term constant across all detectors.
- Noise will be a function of frequency, so subtract CM to estimate non-CM part of noise. Do this in Fourier space, as usual.
- What would this noise matrix look like?
  - Diagonal part from detector noises, plus a common part. $\langle n_i n_j \rangle$=CM, constant across all detectors. Very much like the noise matrix from HW 2.

Sherman Morrison/Woodbury

- Turns out, we can invert this sort of matrix very quickly.
- $(A+vv^T)^{-1} = A^{-1}-A^{-1}v(1+v^TA^{-1}v)^{-1}v^TA^{-1}$. Woodbury Identity
- Looks long, but if A is diagonal, all these operations are of order *n*. Much faster than n^3.
- So, can put in a common-mode as noise rather than subtract.
- If we had many correlated modes, the same fast inverse trick still works. Extension is called the Sherman-Morrison formula. Useful if, say, readout gives correlated noise as well, or atmosphere has structure smaller than array.

Radio Astronomy

- Unlike (ground-based) optical, radio telescopes are usually diffraction-limited
- What’s the shape of ideal dish?
- Distance from infinity, bouncing off dish, to point should be constant across dish.
- Gives a parabola.

What does the beam look like?

- phase delay across disk leads to imperfect summing of phases
- Integrating the phase gradient across an aperture gives summed electric field
- This is just the Fourier transform of the aperture(!)
- Electric field intensity is then just |FT(aperture)|
- And power is intensity squared.
- Circular aperture works out to be $I_0(2J_1(x)/x)^2$ where $x=k\textit{a}\sin(\theta)$, $k=2\pi/\lambda$, a= dish radius.