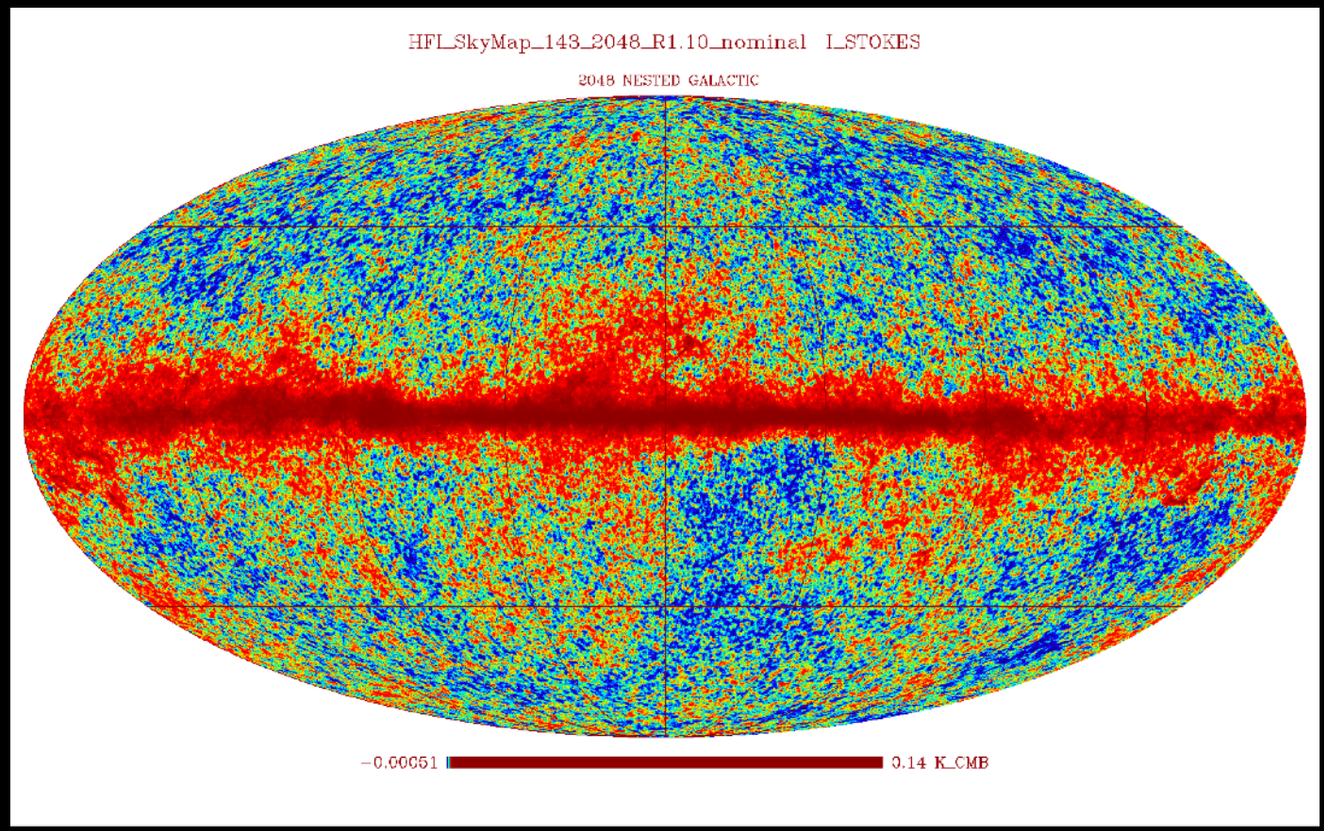
Giant Matrix Problems

- Solving Ax=b is a huge field.
- Least-squares fitting is one case:
  $(A^T N^{-1} A)x = A^T d$. $A^T d \rightarrow b$, $A^T N^{-1} A \rightarrow A$
- Solving Poisson equation ($\nabla^2 V = \rho(x)$) is another.
- One example - making maps of noisy, correlated data.

Mapmaking

- We have noisy timestreams scanning a patch of sky.
- Best map will come from solving $A^T N^{-1} A m = A^T N^{-1} d$
- Just like least-squared we have already done, but problem is now much, much larger.
- Can we do this directly? Let's look at numbers from the Planck Satellite (like WMAP, but more expensive) to see.

Planck



- Let’s say we have managed to make $A^T N^{-1} A$. How big is it?
- Planck - say maps are n_{side}=1024. have $12*n_{side}^2$ pixels~12e6 total.
- But, we have polarization - so ~40e6 total pixels.
- How long would it take to invert $A^T N^{-1} A$?
- How much storage would it take?

Planck Cost

- Let's say we have managed to make $A^T N^{-1} A$.
- Planck - say maps are n_{side}=1024. have $12*n_{side}^2$ pixels~12e6 total.
- But, we have polarization - so ~40e6 total pixels.
- How long would it take to invert $A^T N^{-1} A$?
  - operation count ~$2n^3$=2*40e6^3=1.28e23 flops. 1 core~2.5e9*4*8~80e9 ops/s. Total of 1.5e12 seconds, or 47,000 core-years.
- How much storage would it take?
  - 40e6^2*8 bytes/pixel=12 million gigabytes of RAM.
- Direct solution not feasible.

Could I verify Solution?

- Say someone gave me the answer. Could I confirm?
- Break A, N^{-1}, A^T into series of operators. Make Am, N^{-1}(Am), and $A^T(N^{-1}Am)$.
- Cost to make $A^TN^{-1}Am$ depends on # of data points.
- Planck 143 - 12 detectors, ~100 Hz, 4 years. Totals ~150e9 samples.
- Am takes n_{data} operations - 150e9.
- FFT-based N^{-1} takes $2*\log_2(\text{chunk\_length})*n_{data}$. 1 day=100*86400, log2~25. Total ops $50*n_{data}$ ~7.5e12.
- $A^T(N^{-1}Am)$ takes n_{data} again, 150e9.

Could I verify ctd.

- Total operation count ~1e13, dominated by noise.
- My 80e9 ops/s core would take 125 seconds. In practice, memory bandwidth is a limiting factor, so A, A^T comparable to N^{-1}, and steps much slower than this.
- Data volume ~150e9*8~1-2e12. Terabytes of RAM is possible, though non-trivial.
- So, if someone gave me a guess at the solution, I could check if it was correct.

Getting to a Solution

- Can't directly solve, so will have to try to solve iteratively by checking how good solutions are.

- One possibility: start at a guess, find gradient, minimize χ^2 along downhill (gradient) direction.

- If m is guess and $m^\dagger$ is new step, have $\chi^2=(d-A(m+m^\dagger))^T N^{-1}(d-A(m+m^\dagger) =(r-Am^\dagger)^T N^{-1}(r-Am^\dagger)$ where r=d-Am

- gradient of $\chi^2=-2A^T N^{-1}(r-Am^\dagger)$. But, we are at our current guess so $m^\dagger=0$, and $\text{grad}(\chi^2)=-2A^T N^{-1} r$.

- Making this gradient takes same operations as checking solution so we can do this.

Minimizing Along Direction

- Let v be my direction (proportional to $A^T N^{-1} r$), and q the distance I step. So, $m^\dagger = vq$. What is q?
- $\chi^2 = (d - A(m+m^\dagger))^T N^{-1} (d - A(m+m^\dagger) = (r - Avq)^T N^{-1} (r - Avq)$
- Minimize w.r.t q, and have $v^T A^T N^{-1} Avq = v^T A^T N^{-1} r$
- q is a scalar, so = $v^T A^T N^{-1} r / v^T A^T N^{-1} Av$
- should look familiar, where my new A is Av

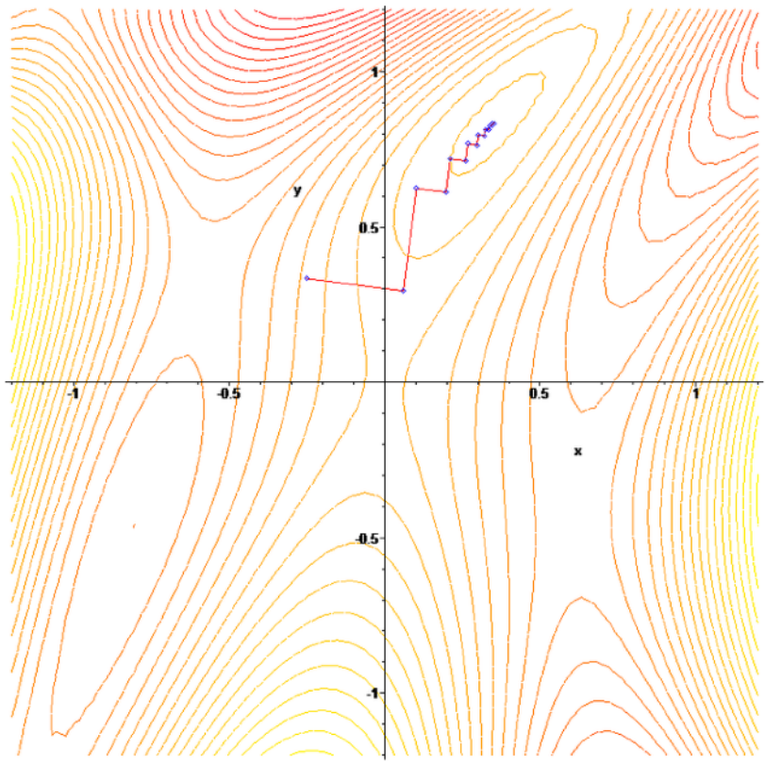
Scheme to Minimize

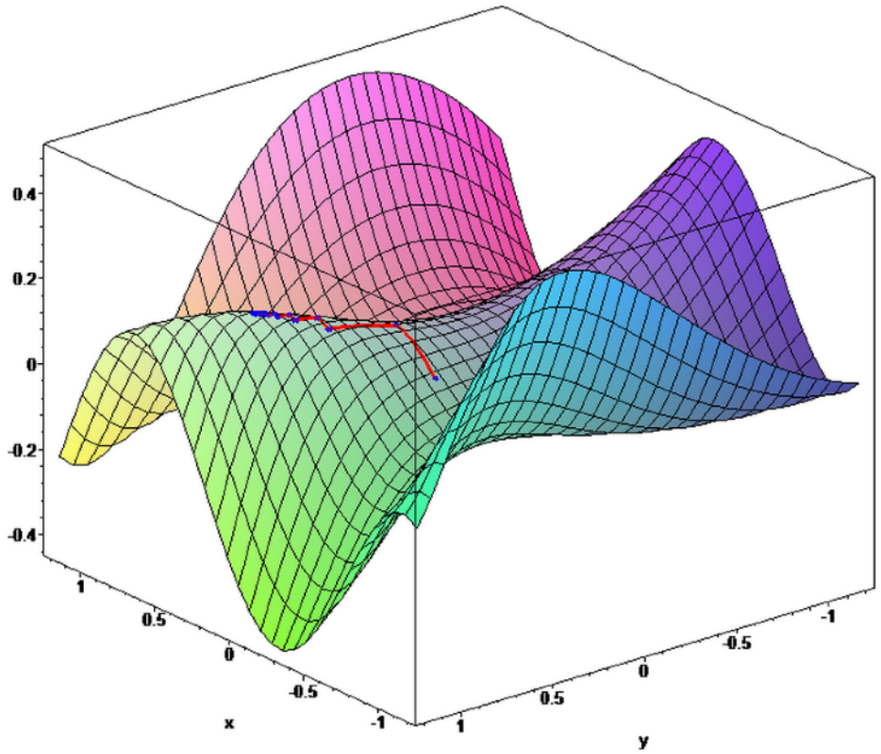
- We have a way to try to solve (enormous!) least-squares now.
- step 0: guess a solution. Could be zero…
- step 1: calculate residual: data-current prediction (Am)
- step 2: calculate downhill direction
- step 3: go downhill until minimum
- step 4: add this step to current guess (update m)
- repeat steps 1 through 4, until we make it downhill.
- This scheme called “steepest descent”. Is it a good scheme?

Steepest Descent

The "Zig-Zagging" nature of the method is also evident below, where the gradient ascent method is applied to $F(x,y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right)\cos(2x + 1 - e^y)$.

e gradient descent algorithm in action. (1: contour)





No! From wikipedia. Key point - new direction always perpendicular to old direction. So, step n+2 will be parallel to step n. Zigagging is inefficient.

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 $A^T N^{-1} A$) 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!

A-Orthogonality

- Let $x = \sum \alpha_k p_k$ where $p_k^T A p_{k'} = \delta_{kk'}$.
- We have $Ax = \sum \alpha_k A p_k = b$
- Multiply on left by one of the p_k. Gives us $\alpha_k p_k^T A p_k$ on LHS, and $p_k^T b$ on RHS.I know $\alpha_k = p_k^T b / p_k^T A p_k$.
- How would my estimate of α_k change if I added some amount of $p_{k'}$ to x? It wouldn't, because of A-orthogonality.
- So, if my search directions are A-orthogonal, solving for the next coefficient doesn't mess up the previous ones.

Iteration

- Say we have p_0. Then $\alpha_0 = p_0^T b / p_0^T A p_0$.
- To find α_1, we can subtract the bit of x we already know, so $\alpha_1 = p_1^T (b - \alpha_0 p_0) / p_1^T A p_1$.
- but, $\alpha_0 p_0$ is just our estimate of x after the first iteration. We call $b - \alpha_0 p_0$ r_1, the first residual.
- So, $\alpha_1 = p_1^T r_1 / p_1^T A p_1$. Similarly, $\alpha_n = p_n^T r_n / p_n^T A p_n$. Usually written $r_n^T r_n / p_n^T A p_n$. since orthogonality means they're the same (modulo roundoff).
- Magic of CG is that we can figure out the next p_k from just our current one plus previous/current residuals.

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:
- $\alpha_k=r_k^T r_k / p_k^T A p_k$. (a bit of magic - r_k's are orthogonal so this actually works, could also be $p_k^T r_k$.)
- $x_{k+1}=x_k+\alpha_k p_k$
- $r_{k+1}=r_k-\alpha_k A p_k$. (if r_{k+1} "small enough", quit)
- $\beta_k=r_{k+1}^T r_{k+1} / r_k^T r_k$.
- $p_{k+1}=r_{k+1}+\beta_k p_k$. (slightly more magic - this keeps next direction A-orthogonal)

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 $A^T N^{-1} A$ is singular, is has zero eigenvalues. Those get left for last, so CG helpfully leaves solution unchanged.

```python
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  423.162817458
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  6.42175958411
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
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 o
```

```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
```

```
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  3.65050457705
iter is  7  with residual squared  1.43721347672
iter is  8  with residual squared  0.569656542728
iter is  9  with residual squared  0.214494092134
mean CG error is  8.61619038593e-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

```python
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$
```

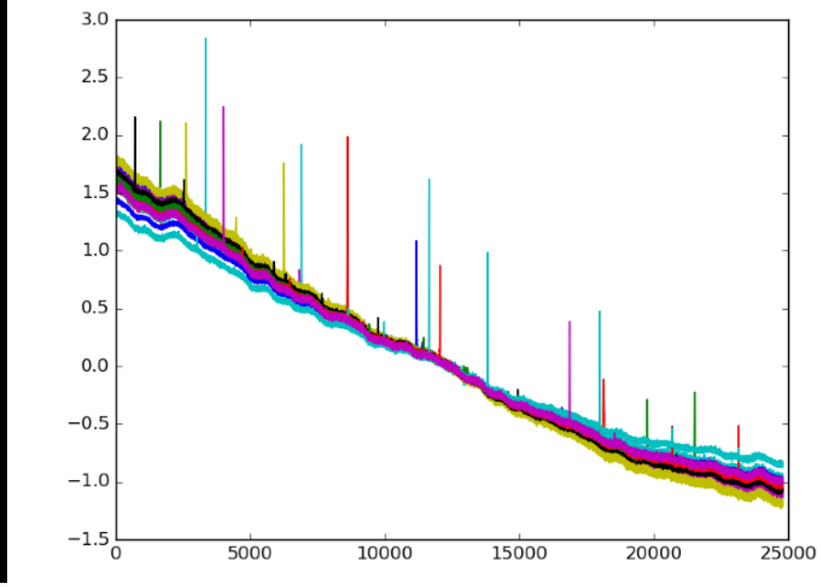
Putting Pieces Together

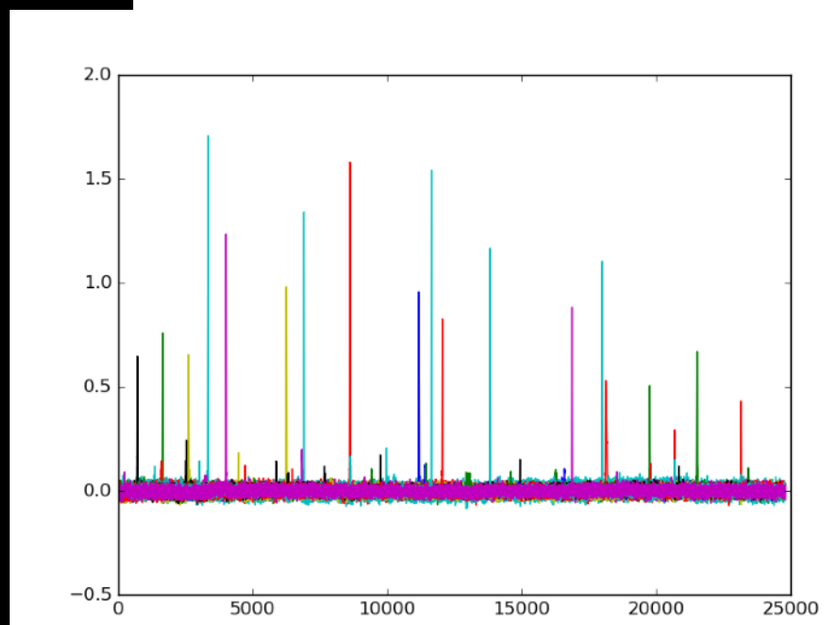
- Let's look at some actual ground-based data.
- In a very dangerous step, we'll subtract off common mode/detector drifts.
- Middle: raw data.
- Bottom:data with best-fit line+cm amplitude subtracted
- Scan is of a calibrator, so spikes upwards are sky signal.

```
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
```





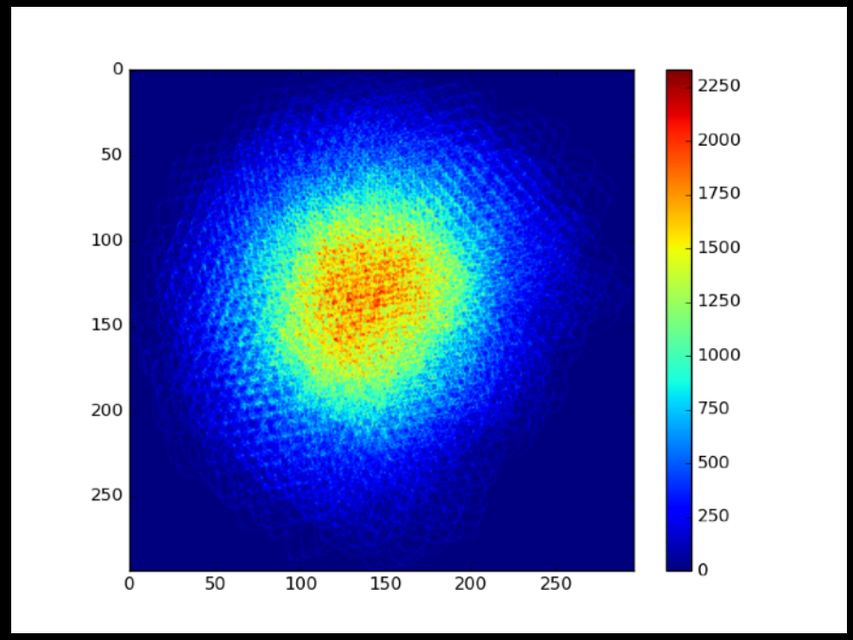
Naive Map

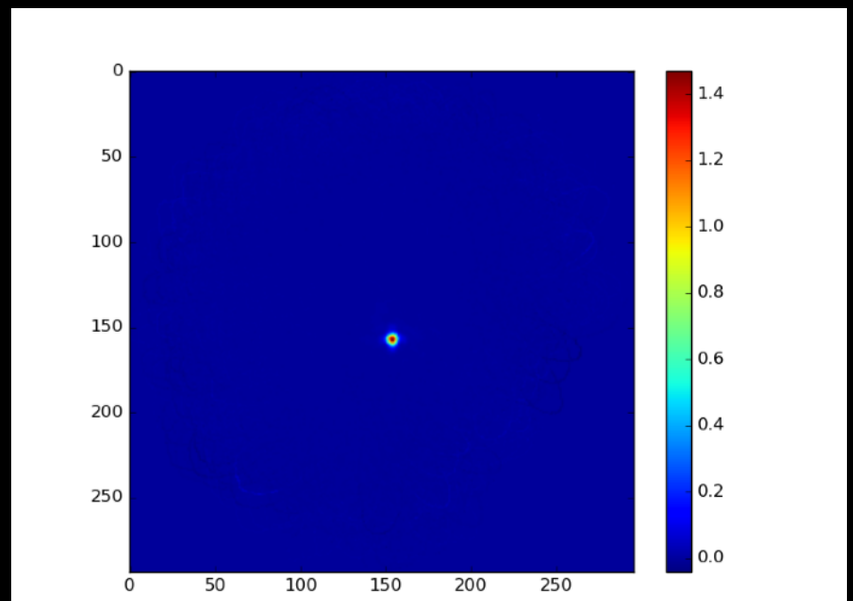
- In fact, naive map of cm-subtracted data is not so awful, at least visually.

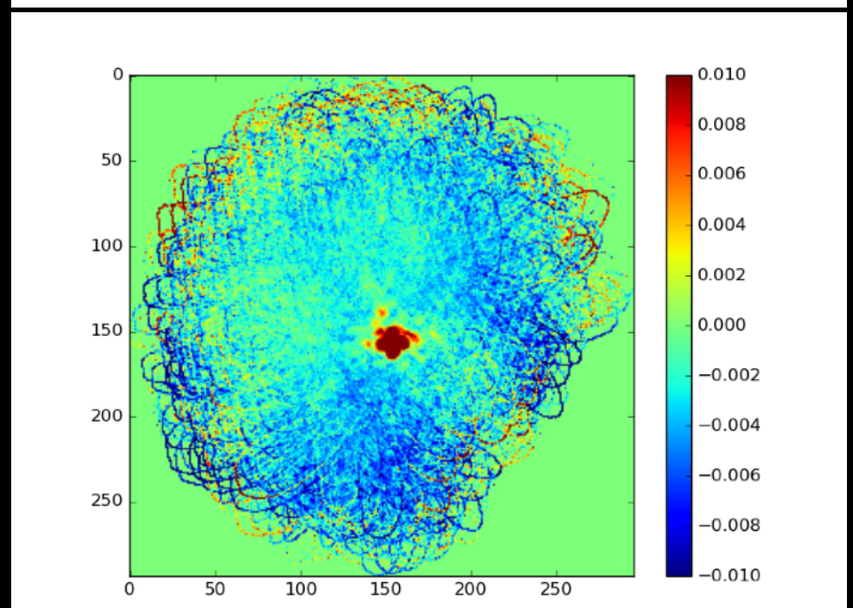
- Top: hits. Middle: naive map of CM-subtracted data. Bottom: stretched colorscale.

- Why would this naive map not be good?

- What could we do better?

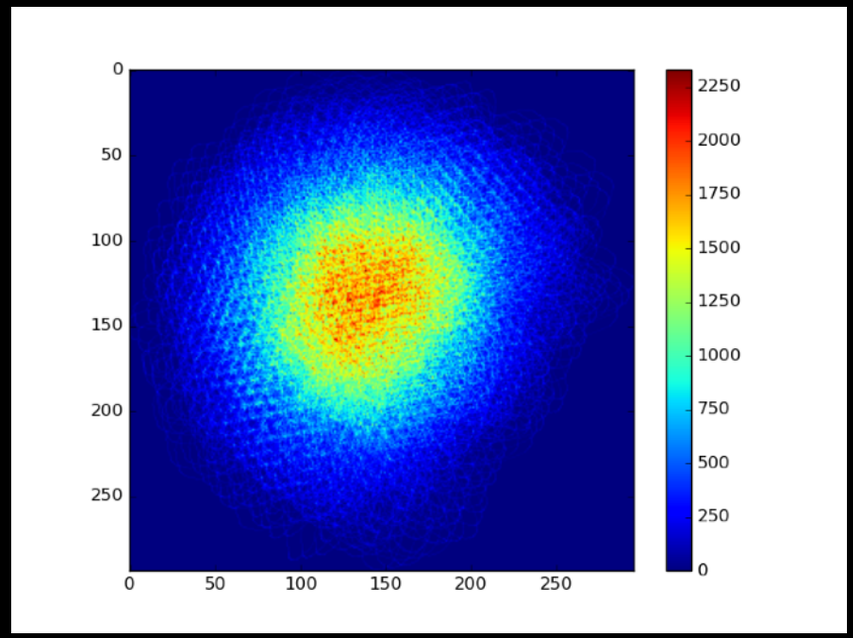


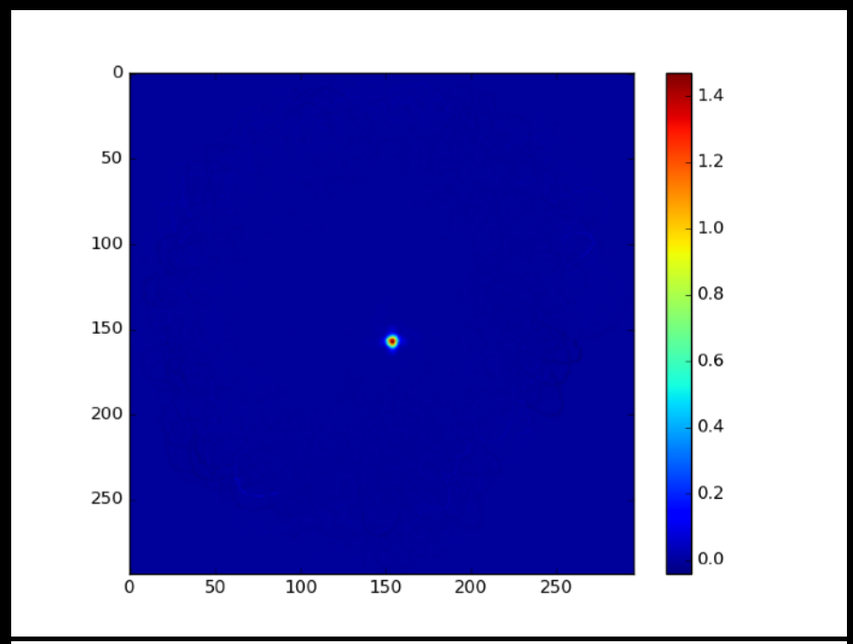


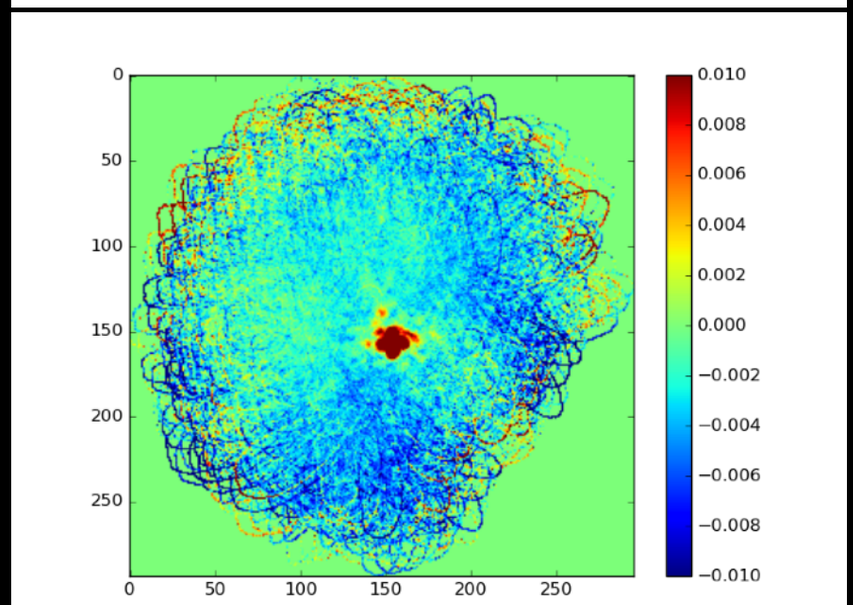


Naive Map

- 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.

CG for Laplace

- For Laplace's equation, we know each cell is the average of its neighbors.

- Key point in CG is we never use elements of A - rather, we multiply vectors by A. If we can do that w/out actually forming a matrix, fine.

- Laplace with fixed boundary conditions: V=ave(neighbors). At an interior point, we have V-ave(neighb)=0. At a point that touches boundary, V-sum(interior)/2n = sum(boundary)/2n.

- We can do this efficiently with a mask and *stencil*. Stencil takes V and subtracts average (each point sees 5/7 points in 2/3 dim). Ax sets boundaries to zero and applies stencil. b sets interior to zero and applies stencil.