Computational Physics
Lecture 11

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture11_2015.git

Admin

- next week is last week of lectures.
- Nominally, test should be on thursday.
- If you would like, thursday can be a review session, and test can be in the regular slot the next tuesday.
- This only works if *all* of you are free. If you want, please send me *in writing* that you would like that!

Random Numbers

- Let's say we want to make (uncorrelated) Gaussian noise.
- Each point can have its own σ. Then $x_i=\sigma_i g_i$. g_i is a realization of a Gaussian random number (numpy.random.randn) with σ=1.
- As a matrix: $x=\Sigma g$ where Σ is a diagonal matrix. Identical operations to above.
- $\langle xx^T\rangle=\Sigma gg^T\Sigma^T$. $(gg^T)_{ij}=g_ig_j=\delta_{ij}$. So, gg^T=Identity matrix
- $\langle xx^T\rangle=\Sigma gg^T\Sigma^T=\Sigma I\Sigma^T$. I goes away, and Σ is diagonal, so left with $\langle xx^T\rangle=\Sigma^2=\Lambda$ where Λ_{ii}=variance(x_i).

Correlated Random Numbers

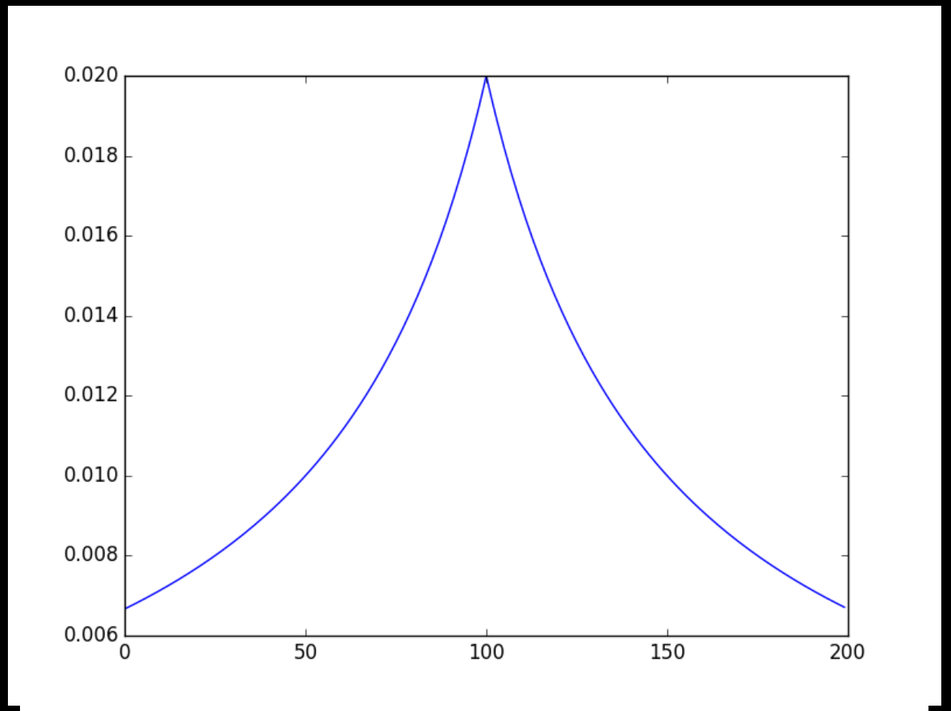
- Now let's take $y=Vx$ for some orthogonal matrix V.
- $\langle yy^T\rangle=\langle(Vx)(Vx)^T\rangle=\langle Vxx^TV^T\rangle$. But $\langle xx^T\rangle=\Lambda$, so $\langle yy^T\rangle=V\Lambda V^T$.
- $y_i=\text{sum}(V_{ik}x_k)=\text{sum}(V_{ik}\sigma_k g_k)$. $y_iy_j=\text{sum}(V_{ik}\sigma_k g_k)\text{sum}(V_{jk}\sigma_k g_k)$.
- The g_k are uncorrelated, cross terms go away. Then $\langle y_iy_j\rangle=\text{sum}(V_{ik}V_{jk}\Lambda_k)$.
- But, that's the ij^{th} element of $V\Lambda V^T$!

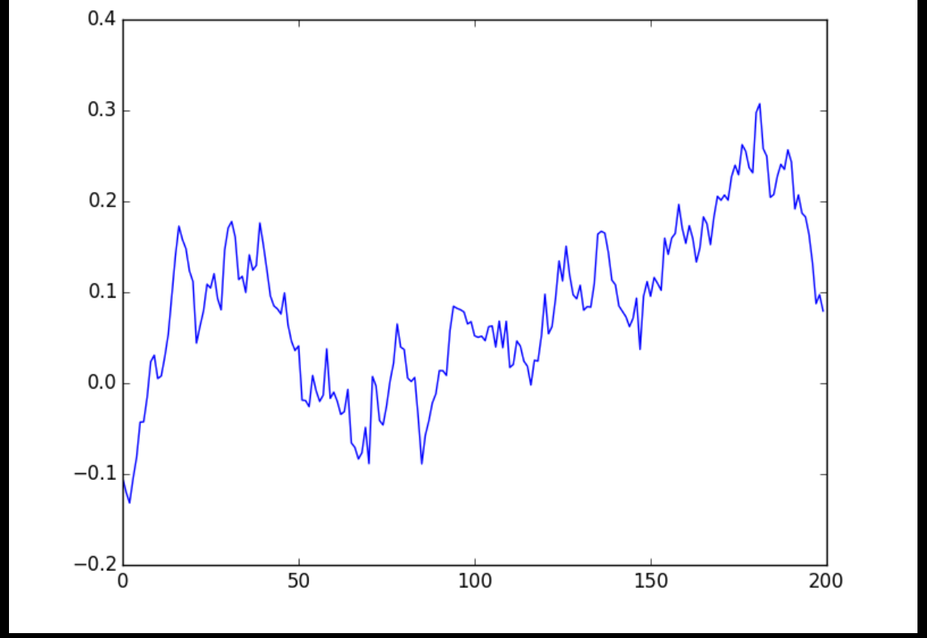
Correlated Random Numbers 2

- Let $N=V\Lambda V^T$. Then $\langle y_i y_j \rangle = N_{ij}$, and is complete description of data.
- To make random correlated data, take eigenvalues/eigenvectors of noise covariance. Make random uncorrelated data x with variance Λ.
- Then multiply x by eigenvectors: $y=Vx$
- In practice, can use most any matrix factorization - Cholesky common.

In Code

```
#correlated_random.py
import numpy
from matplotlib import pyplot as plt

nn=200; # # of data points
cov=numpy.zeros([nn,nn])
for i in range(0,nn):
    for j in range(0,nn):
        #make up a covariance function
        cov[i,j]=1./(50+numpy.abs(i-j))
plt.ion()
plt.clf()
plt.plot(cov[nn/2,:])
plt.savefig('cov_slice.png')
#Take eigenvalues/eigenvectors
lam,v=numpy.linalg.eig(cov)
#make random data with noise from eigenvalues.
#rotate back with eigenvectors
x=numpy.sqrt(lam)*numpy.random.randn(nn)
y=v.dot(x)
plt.clf()
plt.plot(y)
plt.savefig('corrdata_matrix.png')
```





What if we want long data?

- Had to do eigenvalue problem - scales like n^3.
- Often want to generate longer series of data.
- If statistics of y_i and y_j depend only on (i-j), then data are "stationary" and we can use Fourier transforms.

Stationary FT

- $\langle F(k)^{*}F(k')\rangle=\langle \Sigma f(x)\exp(2\pi ikx)\Sigma f(x)\exp(-2\pi ik'x)\rangle$
- Shift x by dx: $\langle \Sigma f(x+dx)\exp(2\pi ik(x+dx))\Sigma f(x+dx)\exp(-2\pi ik'(x+dx))\rangle$
- $=\exp(2\pi i(k-k')dx)\ \langle \Sigma f(x+dx)\exp(2\pi ikx)\Sigma f(x+dx)\exp(-2\pi ik'x)\rangle$
- But, if f(x) is stationary, can shift just f:
- $=\exp(2\pi i(k-k')dx)\langle \Sigma f(x)\exp(2\pi ikx)\Sigma f(x)\exp(-2\pi ik'x)\rangle$
- To be true for all dx, either $\exp()=1$ (i.e. k==k') or $\langle\rangle=0$

Wiener–Khinchin Theorem

- If I have random process with time-invariant statistics, Fourier transform of that will have different wavelengths uncorrelated.
- Showed earlier that $\int f(x)f(x+dx)=\mathrm{IFFT}(F(k)^{*}F(k))$
- Take FFT: $\mathrm{FFT}(\int f(x)f(x+dx))=|F(k)|^2$.
- Wiener–Khinchin theorem: Fourier transform of correlation function is average of Fourier transform of function squared. Called the *power spectrum*.
- To generate long periods of fake data: Fourier transform slice of covariance matrix, take square root, multiply by random amplitudes and phases, Fourier transform back.

FFT-Generated Fake Data

```
#correlated_random_fft.py
import numpy
from matplotlib import pyplot as plt

nn=401; # # of data points
#now just make the first row of covariance matrix
cov=numpy.zeros(nn)
i=0
for j in range(0,nn):
    cov[j]=1./(50+numpy.abs(i-j))
#but since correlation has to be symmetric left-right, need to
#make sure negative indices are same as positive:
cov[1:]=cov[1:]+numpy.flipud(cov[1:])

covft=numpy.real(numpy.fft.fft(cov))
covft[covft<0]=0.0
covft=numpy.sqrt(covft)

xft=numpy.random.randn(nn)+numpy.complex(0,1)*numpy.random.randn(nn)
xft[1:]=xft[1:]+numpy.conj(numpy.flipud(xft[1:]))
xft[0]=numpy.real(xft[0]) #explicitly make offset real
xft=xft*covft


noisy_dat=numpy.fft.ifft(xft)
noisy_dat=numpy.real(noisy_dat)
plt.ion()
plt.clf()
plt.plot(noisy_dat)
plt.show()

#plt.savefig('corrdata_matrix.png')
```

- Create correlation function (not a full matrix!)
- Since FFTs wrap around, make sure that corr[-dx]==corr[dx]
- FFT correlation function, take square root, multiply by random amps/phases
- IFFT, look at results!

Fourier Inverse

- Remember, FFT can be written as a matrix: $F(k)=Gf(x)$, where G is orthogonal. If $f(x)$ is stationary, then $\langle f(x)f(x')\rangle=G^TF(k)^2G$.
- But, I know how to get there by FFTing. The inverse will just be $G^TF(k)^{-2}G$
- $G^TF(k)^2G^*\ G^TF(k)^{-2}G=G^TF(k)^0G=I$
- So, I can invert circulant matrices by FFTing one row, inverting, and FFTing back!
- To check: scipy.linalg has a function toeplitz which will take a row and turn it into a matrix with first element along the diagonal.

Toeplitz Inverse

```
#fft_inverse.py
from scipy.linalg import toeplitz
from  numpy.fft import fft,ifft
from numpy import flipud
import numpy
#make a circulant correlation function
x=1./numpy.arange(1,5);x[1:]=x[1:]+flipud(x[1:])
mat=toeplitz(x)
xinv=numpy.real(ifft(1/fft(x)))
matinv=toeplitz(xinv)
print mat
print mat.dot(matinv)
```

```
>>> execfile('fft_inverse.py')
[[ 1.          0.75        0.66666667  0.75      ]
 [ 0.75        1.          0.75        0.66666667]
 [ 0.66666667  0.75        1.          0.75      ]
 [ 0.75        0.66666667  0.75        1.        ]]
[[  1.00000000e+00  -1.66533454e-16   2.22044605e-16   0.00000000e+00]
 [ -1.11022302e-16   1.00000000e+00  -1.11022302e-16   0.00000000e+00]
 [  2.22044605e-16  -5.55111512e-17   1.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   2.22044605e-16  -2.22044605e-16   1.00000000e+00]]
>>>
```

Kriging

- Let's say we have some data points, and want to estimate what we should get at an unmeasured point. This is called *interpolation*. Several methods exist (linear, polynomial, cubic spline, piecewise cubic Hermite…)

- Let's assume nearby points are Gaussian distributed with known correlation, and have zero mean. Then likelihood is $d^T N^{-1} d$

- If I know all but one point, I can find the most likely value given others by differentiating w.r.t that value.

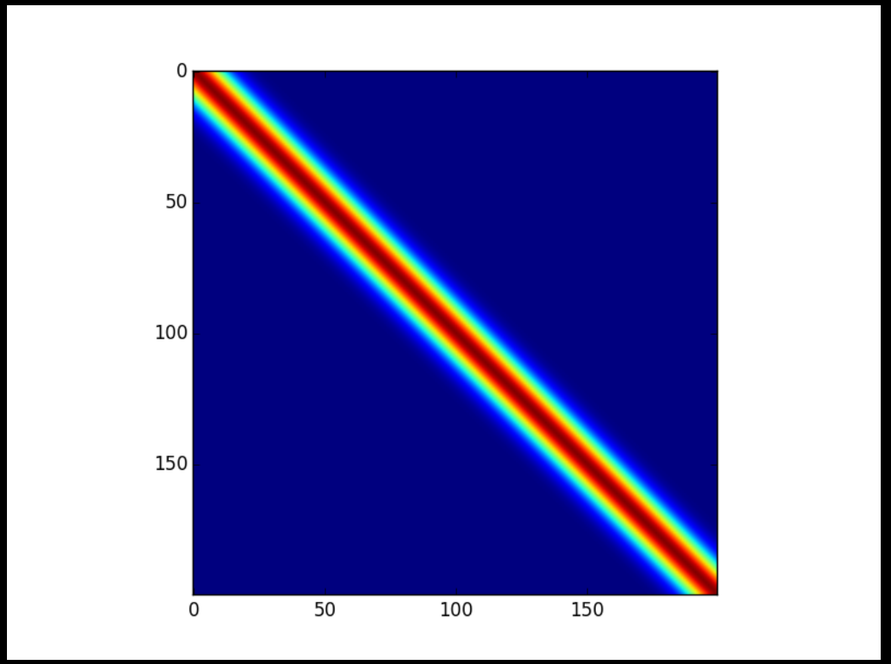
- Method developed in South Africa for searching for gold, called Kriging after originator (a Mr. Krig, who did it for his thesis)

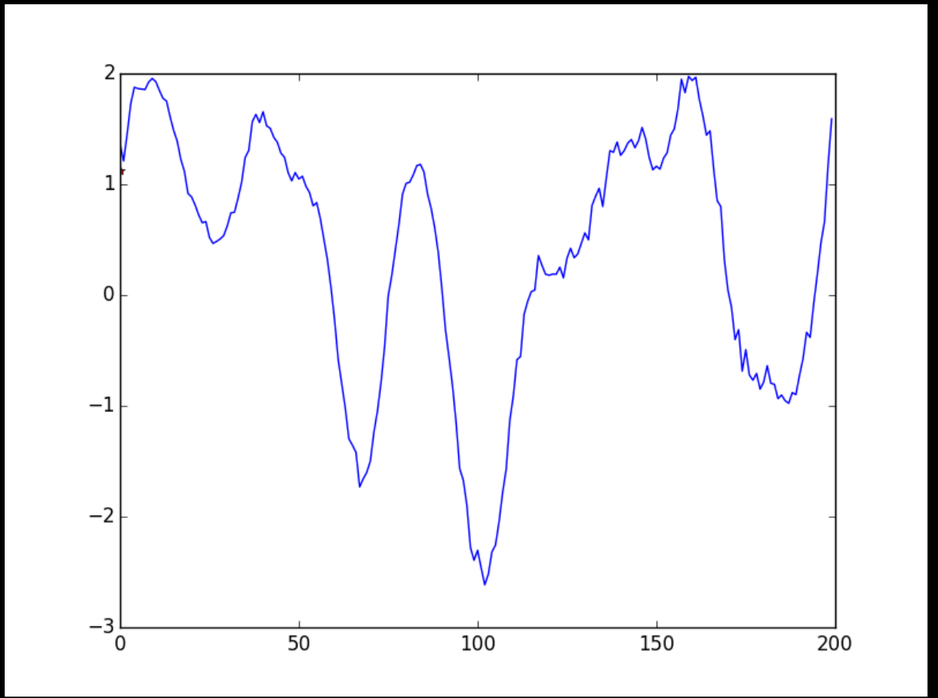
Kriging Algorithm

- Differentiate w.r.t the first data point: d(like)/dx=[1 0 0 0…] N^{-1}d=0
- multiply picks out first row of N^{-1}, so have N^{-1}_{0}d=0
- or, $N^{-1}_{0,0}d_0+N^{-1}_{0,1:}\bullet d(1:)=0$, or $d0=-N^{-1}_{0,1:}\bullet d(1:)/N^{-1}_{0,0}$.
- If data are circulant, can use FFTs for the (fast!) inverse. General matrix inverse works, too.

Kriging Example

```
#fft_inverse.py
from scipy.linalg import toeplitz
from  numpy.fft import fft,ifft
from numpy import flipud,arange,exp
import numpy
import matplotlib.pyplot as plt
x=arange(200);y=exp(-0.5*x**2/8**2)
#make correlation matrix.  add a bit of noise
mat=toeplitz(y)+0.01*numpy.eye(x.size)
plt.ion()
plt.clf()
plt.imshow(mat)
plt.show();plt.savefig('kriging_mat.png')

#take eigenvalues.  Should be real, so force numpy
e,v=numpy.linalg.eig(mat)
e=numpy.real(e);v=numpy.real(v)
x=v.dot(numpy.sqrt(e)*numpy.random.randn(e.size))

#do kriging estimate
mat_inv=numpy.linalg.inv(mat)
x_pred=-numpy.dot(mat_inv[0,1:],x[1:])/mat_inv[0,0]
plt.clf()
plt.plot(x)
plt.plot(0,x_pred,'r*')
plt.show();plt.savefig('kriging_output.png')
print (x[0]-x_pred)/numpy.std(x)
```





Sample Test Questions

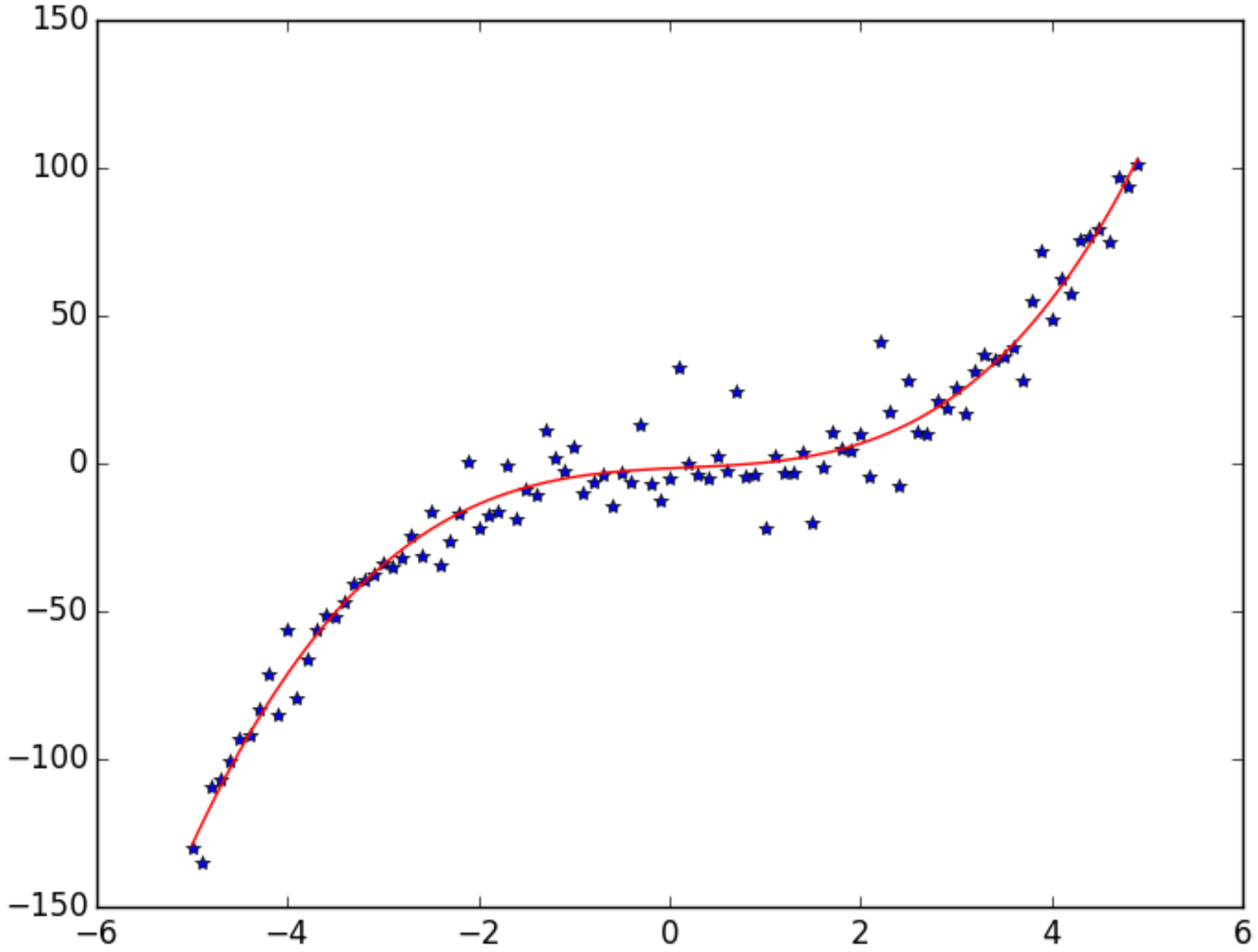
- Will work more on Thursday. Pull from github/practice_test

Least Squares: $\chi^2=(x-Am)^T N^{-1}(x-Am)$

- To find best-fitting model, minimize χ^2. Calculus on matrices works like regular calculus, as long as no orders get swapped.
- $\partial\chi^2/\partial m = -A^T N^{-1}(x-Am)+\ldots=0$ (at minimum)
- We can solve for m: $A^T N^{-1}Am=A^T N^{-1}x$. Or, $m=(A^T N^{-1}A)^{-1}A^T N^{-1}x$

Exa

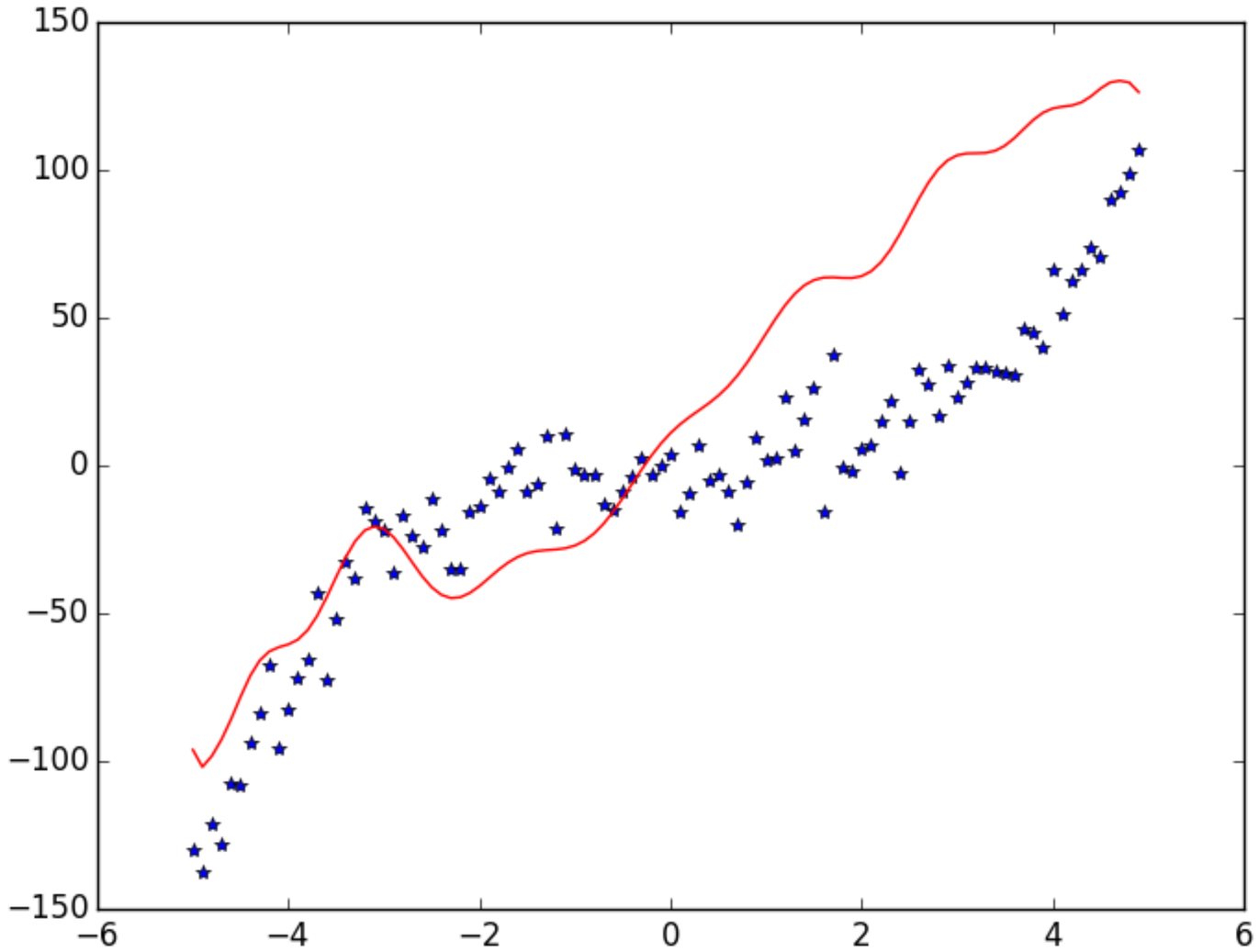
```
import numpy
from matplot
t=numpy.aran
x_true=t**3-
x=x_true+10*

npoly=5  #le
ndata=t.size
A=numpy.zerc
A[:,0]=1.0
for i in ran
    A[:,i]=A
#Let's ignor
#m=(A^TA)^{-
A=numpy.matr
d=numpy.matr
lhs=A.transp
rhs=A.transp
fitp=numpy.l
pred=A*fitp
plt.clf();pl
plt.draw()
```



tc. as matrices rather

s live in numpy.linalg,

Higher Order




```
import numpy
from matplotlib import pyplot as plt
t=numpy.arange(-5,5,0.1)
x_true=t**3-0.5*t**2
x=x_true+10*numpy.random.randn(t.size)

npoly=25 #let's fit 4th order polynomial
ndata=t.size
A=numpy.zeros([ndata,npoly])
A[:,0]=1.0
for i in range(1,npoly):
    A[:,i]=A[:,i-1]*t
#Let's ignore noise for now.  New equations are:
#m=(A^TA)^{-1}*(A^Td)
A=numpy.matrix(A)
d=numpy.matrix(x).transpose()
lhs=A.transpose()*A
rhs=A.transpose()*d
fitp=numpy.linalg.inv(lhs)*rhs
pred=A*fitp
plt.clf();plt.plot(t,x,'*');plt.plot(t,pred,'r');
plt.draw()
plt.savefig('polyfit_example_high.png')
```

Condition # and Roundoff

- Recall that the eigenvalues of a symmetric matrix are real, and the eigenvectors are orthogonal. So, $(A^T N^{-1} A)$ can be re-written $V^T \Lambda V$, where Λ is diagonal and V is orthognal (so $V^{-1}=V^T$).

- $(ABC)^{-1}=C^{-1}B^{-1}A^{-1}$, so inverse$=V^{-1}\Lambda^{-1}(V^T)^{-1}=V^T\Lambda^{-1}V$.

- If a bunch of eigenvalues are really small, they will be huge in the inverse. Double precision numbers are good to ~16 digits, so if spread gets bigger than 10^{16}, we'll lose information in the inverse.

- Ratio of largest to smallest eigenvalue is called the condition number. If it is large, matrices are ill-conditioned, and will present problems.

Condition # of Polynomial Matrices

- Condition # quickly blows up. So, we should have expected problems.

```
import numpy
def get_poly_mat(t,npoly):
    mat=numpy.zeros([t.size,npoly])
    mat[:,0]=1.0
    for i in range(1,npoly):
        mat[:,i]=t*mat[:,i-1]
    mat=numpy.matrix(mat)
    return mat

if __name__=='__main__':
    t=numpy.arange(-5,5,0.1)
    for npoly in numpy.arange(5,30,5):
        mat=get_poly_mat(t,npoly)
        mm=mat.transpose()*mat
        mm=mm+mm.transpose() #bonus symmetrization
        e,v=numpy.linalg.eig(mm)
        eabs=numpy.abs(e)
        cond=eabs.max()/eabs.min()
        print repr(npoly) + ' order poynomial matrix has condition number ' + repr(cond)
```

```
>>> execfile('cond_example.py')
5 order poynomial matrix has condition number 158940.69399024552
10 order poynomial matrix has condition number 2366966250887.5864
15 order poynomial matrix has condition number 2.722363799692467e+19
20 order poynomial matrix has condition number 2.2708595871810382e+25
25 order poynomial matrix has condition number 7.8912167454722334e+31
>>>
```

One Possibility: SVD

- Take noiseless case. Then solving $A^TAm=A^Tx$.
- Singular value decomposition (SVD) factors matrix $A=USV^T$, where S is diagonal, and U and V are orthogonal, and V is square. For symmetric, U=V, S=eigenvalues, but SVD works for any matrix.
- Solutions: $(USV^T)^TUSV^Tm=(USV^T)^Tx$. $VSU^TUSV^Tm=VSU^Tx$
- U^TU=identity, so cancels. $VS^2V^Tm=VSU^Tx$. S2 squares the condition number, so that was bad. We can analytically cancel left-hand V and one copy of S: $SV^Tm=U^Tx$. Then $m=VS^{-1}U^Tx$
- NB - this can be done even faster with QR

SVD Code

- Here's how to take singular value decompositions with numpy.
- This will work better than before, but still won't get us to e.g. 100th order polynomials.
- Main issue is that simple polynomials are ill-conditioned: x^{20} looks a lot like x^{22}.

```
import numpy
from matplotlib import pyplot as plt
t=numpy.arange(-5,5,0.1)
x_true=t**3-0.5*t**2
x=x_true+10*numpy.random.randn(t.size)

npoly=20
ndata=t.size
A=numpy.zeros([ndata,npoly])
A[:,0]=1.0
for i in range(1,npoly):
    A[:,i]=A[:,i-1]*t

A=numpy.matrix(A)
d=numpy.matrix(x).transpose()
#Make the svd decomposition, the extra False
#is to make matrices compact
u,s,vt=numpy.linalg.svd(A,False)
#s comes back as a 1-d array, turn it into a 2-d matrix
sinv=numpy.matrix(numpy.diag(1.0/s))
fitp=vt.transpose()*sinv*(u.transpose()*d)
```

Solution: Different Poly Basis

- There are several families of polynomials that have better properties (Legendre, Chebyshev…). Usually defined on (-1,1) through recursion relations.
- Legendre polynomials are constructed to be orthogonal on (-1,1), so condition number should be good. If our t range is different from (-1,1), rescale so that it is.
- Key relation: $(n+1)P_{n+1}(t)=(2n+1)tP_n(t)-nP_{n-1}(t)$ with $P_0=1$ and $P_1=t$.
- I pick up a power of *t* each time, so these are also polynomials, just written in linear combinations that have better condition number.
- Strongly encourage you to *never* fit regular polynomials. Always use Legendre, Chebyshev…

Legendre Code

```
import numpy
def get_legendre_mat(t,npoly):
    #key relation:  (n+1)P_(n+1)=(2n+1)tP_n - nP_(n-1)
    mat=numpy.zeros([t.size,npoly])
    mat[:,0]=1.0
    if npoly>1:
        mat[:,1]=t
    for i in range(1,npoly-1):
        mat[:,i+1]=((2.0*i+1)*t*mat[:,i]-i*mat[:,i-1])/(i+1.0)
    mat=numpy.matrix(mat)
    return mat

if __name__=='__main__':
    dt=0.001
    t=numpy.arange(-5+dt/2.0,5,dt)
    for npoly in numpy.arange(5,100,5):
        mat=get_legendre_mat(t/5,npoly)
        mm=mat.transpose()*mat
        mm=mm+mm.transpose() #bonus symmetrization
        e,v=numpy.linalg.eig(mm)
        eabs=numpy.abs(e)
        cond=eabs.max()/eabs.min()
        print repr(npoly) + ' Legendre matrix has co
```

```
>>> execfile('cond_example_legendre.py')
5 order poynomial matrix has condition number 9.0000026999767648
10 order poynomial matrix has condition number 19.00005415034467
15 order poynomial matrix has condition number 29.000294368595334
20 order poynomial matrix has condition number 39.000963550102306
25 order poynomial matrix has condition number 49.002402810934953
30 order poynomial matrix has condition number 59.00505642599736
35 order poynomial matrix has condition number 69.009477057966521
40 order poynomial matrix has condition number 79.016336167849929
45 order poynomial matrix has condition number 89.026442681092632
50 order poynomial matrix has condition number 99.040774215288522
55 order poynomial matrix has condition number 109.06052705286851
60 order poynomial matrix has condition number 119.08719407465288
65 order poynomial matrix has condition number 129.12268493401126
70 order poynomial matrix has condition number 139.16951135267718
75 order poynomial matrix has condition number 149.23107516419981
80 order poynomial matrix has condition number 159.31212210407367
85 order poynomial matrix has condition number 169.41946763316335
90 order poynomial matrix has condition number 179.56317279103277
95 order poynomial matrix has condition number 189.75845697330035
>>>
```

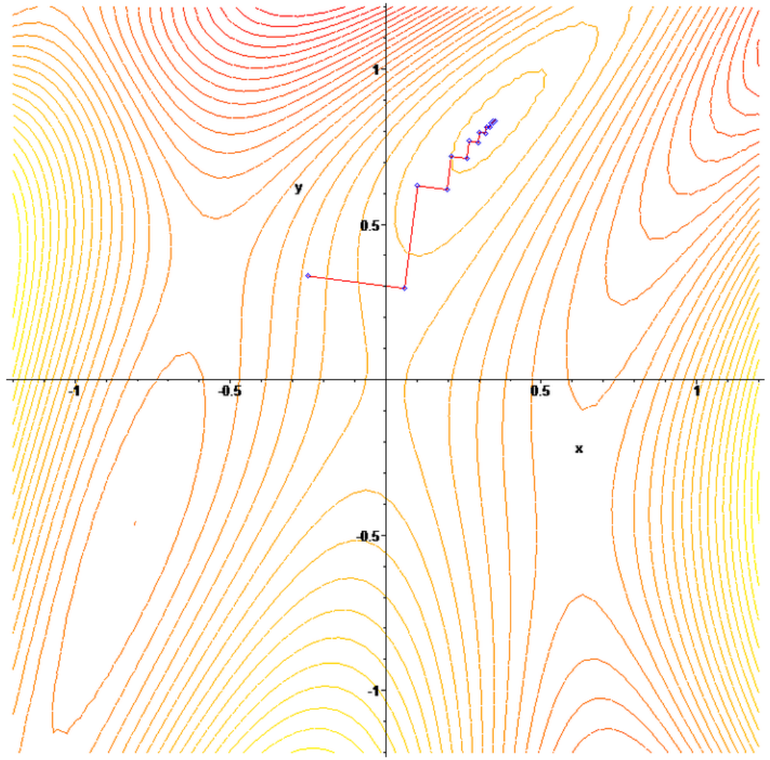
Nonlinear Fitting

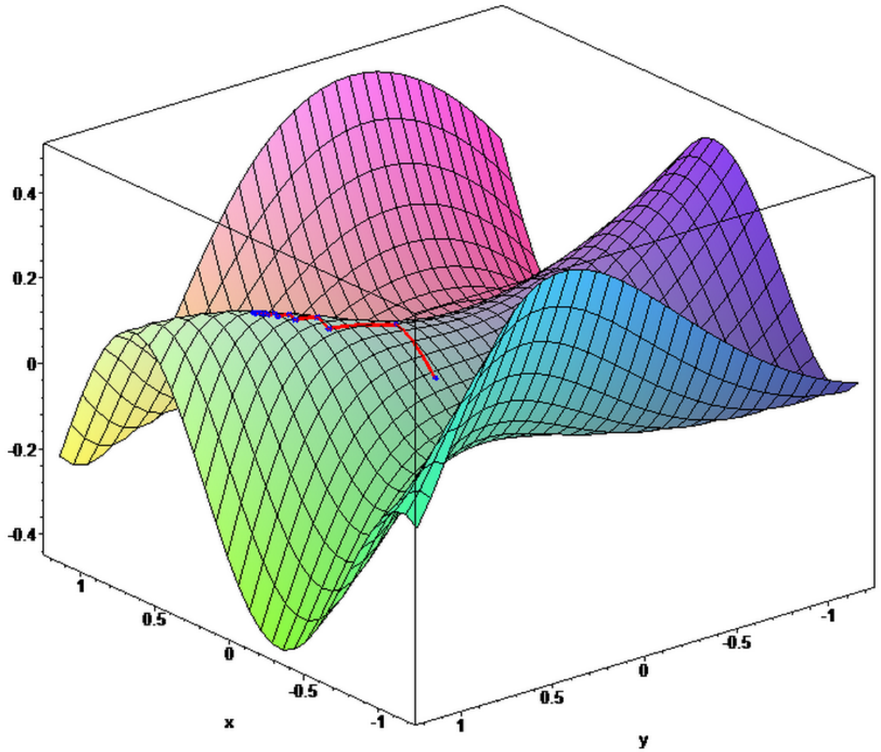
- Sometimes data depend non-linearly on model parameters
- Examples are Gaussian and Lorentzian $(a/(b+(x-c)^2)$
- Often significantly more complicated - cannot reason about global behaviour from local properties. May be multiple local minima
- Many methods reduce to how to efficiently find the "nearest" minimum.
- One possibility - find steepest downhill direction, move to the bottom, repeat until we're happy. Called "steepest descent."
- How might this end badly?

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)





From wikipedia. Zigagging is inefficient.

Better: Newton's Method

- linear: $\langle d \rangle = Am$. Nonlinear: $\langle d \rangle = A(m)$ $\chi^2 = (d-A(m))^T N^{-1} (d-A(m))$
- If we're "close" to minimum, can linearize. $A(m) = A(m_0) + \partial A/\partial m * \delta m$
- Now have $\chi^2 = (d-A(m_0)-\partial A/\partial m\ \delta m\ )^T N^{-1} (d-A(m_0)-\partial A/\partial m\ \delta m\ )$
- What is the gradient?

Newton's Method ctd

- Gradient trickier - $\partial A/\partial m$ depends in general on m, so there's a second derivative
- Two terms: $\nabla\chi^2=(-\partial A/\partial m)^T N^{-1}(d-A(m_0)-\partial A/\partial m\ \delta m\ )-(\partial^2 A/\partial m_i \partial m_j\ \delta m)^T N^{-1}(d-A(m_0)-\partial A/\partial m\ \delta m)$
- If we are near solution $d\approx A(m_0)$ and δm is small, so first term has one small quantity, second has two. Second term in general will be smaller, so usual thing is to drop it.
- Call $\partial A/\partial m$ A_m. Call $d-A(m_0)$ r. Then $\nabla\chi^2\approx -A_m^T N^{-1}(r-A_m\delta m)$
- We know how to solve this! $A_m^T N^{-1} A_m\ \delta m = A_m^T N^{-1} r$

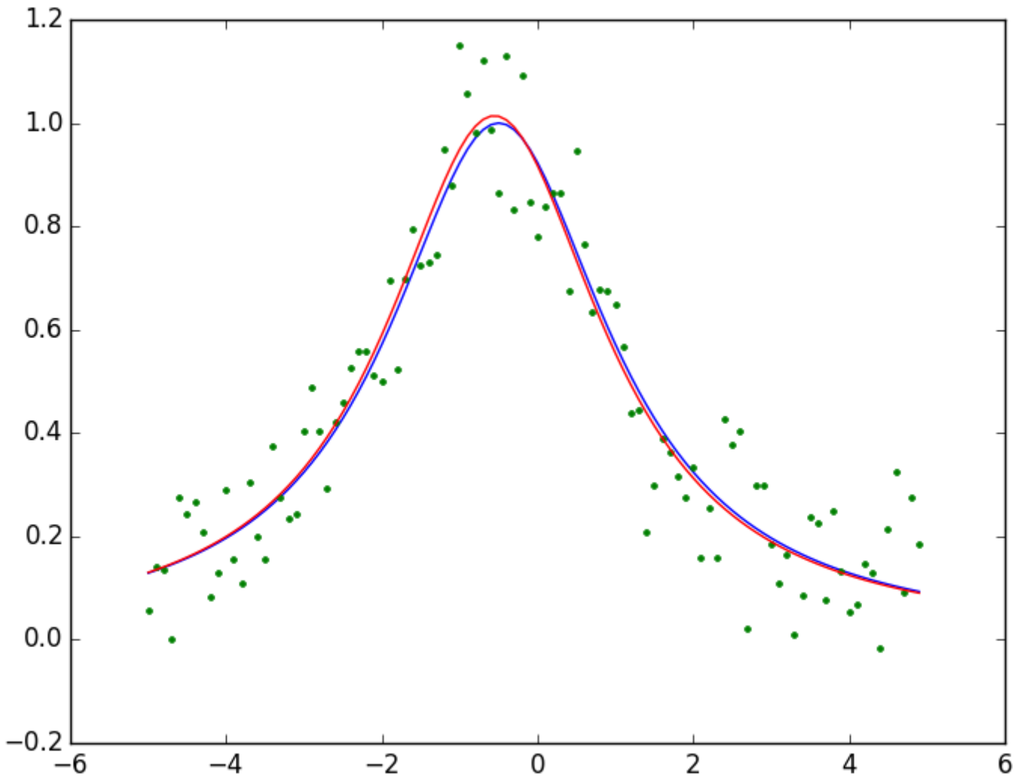
How to Implement

- Start with a guess for the parameters: m_0.
- Calculate model $A(m_0)$ and local gradient A_m.
- Solve linear system $A_m^T N^{-1} A_m \, \delta m = A_m^T N^{-1} r$
- Set $m_0 \rightarrow m_0 + \delta m$.
- Repeat until δm is "small". For χ^2, change should be << 1.

Newton's Method in Action

```
def calc_lorentz(p,t):
    y=p[0]/(p[1]+(t-p[2])**2)
    grad=numpy.zeros([t.size,p.size])
    #now differentiate w.r.t. all the parameters
    grad[:,0]=1.0/(p[1]+(t-p[2])**2)
    grad[:,1]=-p[0]/(p[1]+(t-p[2])**2)**2
    grad[:,2]=p[0]*2*(t-p[2])/(p[1]+(t-p[2])**2)**2
    return y,grad
```

```
for j in range(5):
    pred,grad=calc_lorentz(p,t)
    r=x-pred
    err=(r**2).sum()
    r=numpy.matrix(r).transpose()
    grad=numpy.matrix(grad)

    lhs=grad.transpose()*grad
    rhs=grad.transpose()*r
    dp=numpy.linalg.inv(lhs)*(rhs)
    for jj in range(p.size):
        p[jj]=p[jj]+dp[jj]
    print p,err
```



MCMC

- Nonlinear problems can be very tricky. Big problem - there can be many local minima, how do I find global minimum? Linear problem easier since there's only one minimum.

- One technique: Markov-Chain Monte Carlo (MCMC). Picture a particle bouncing around in a potential. It normally goes downhill, but sometimes goes up.

- Solution: simulate a thermal particle bouncing around, keep track of where it spends its time.

- Key theorem: such a particle traces the PDF of the model parameters, and distribution of the full likelihood is the same as particle path.

- Using this, we find not only best-fit, but confidence intervals for model parameters.

MCMC, ctd.

- Detailed balance: in steady state, probability of state going from a to b is equal to going from b to a ("detailed balance").
- Algorithm. Start a particle at a random position. Take a trial step. If trial step improves χ^2, take the step. If not, *sometimes* accept the step, with probability $\exp(-0.5\delta\chi^2)$.
- After waiting a sufficiently long time, take statistics of where particle has been. This traces out the likelihood surface.

MCMC Driver

```
def run_mcmc(data,start_pos,nstep,scale=None):
    nparam=start_pos.size
    params=numpy.zeros([nstep,nparam+1])
    params[0,0:-1]=start_pos
    cur_chisq=data.get_chisq(start_pos)
    cur_pos=start_pos.copy()
    if scale==None:
        scale=numpy.ones(nparam)
    for i in range(1,nstep):
        new_pos=cur_pos+get_trial_offset(scale)
        new_chisq=data.get_chisq(new_pos)
        if new_chisq<cur_chisq:
            accept=True
        else:
            delt=new_chisq-cur_chisq
            prob=numpy.exp(-0.5*delt)
            if numpy.random.rand()<prob:
                accept=True
            else:
                accept=False
        if accept:
            cur_pos=new_pos
            cur_chisq=new_chisq
        params[i,0:-1]=cur_pos
        params[i,-1]=cur_chisq
    return params
```

- Here's a routine to make a fixed-length chain.
- As long as our data class has a get_chisq routine associated with it, it will work.
- Big loop: take a trial step, decide if we accept or not. Add current location to chain.

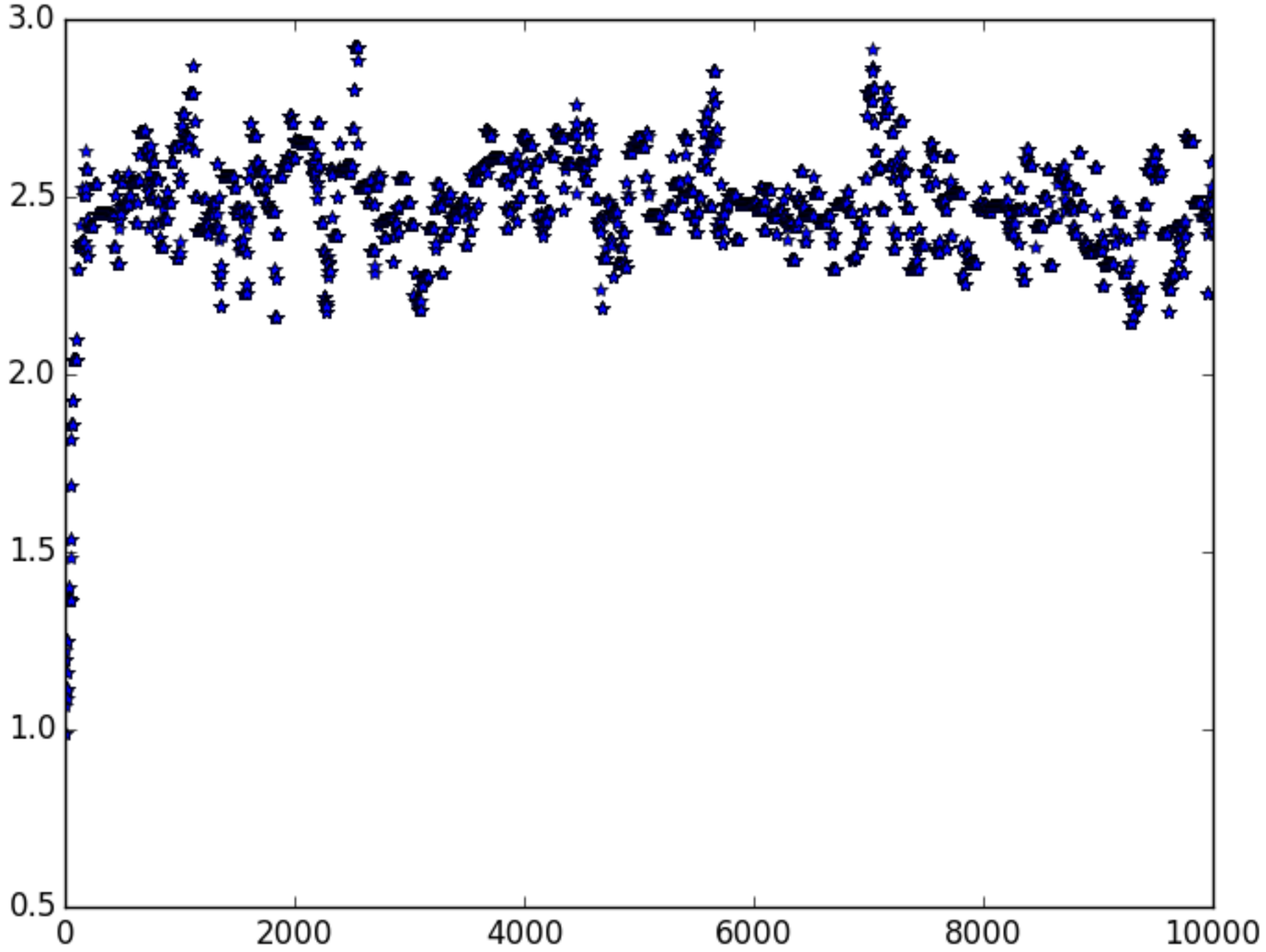
Output

```
if __name__=='__main__':

    #get a realization of a gaussian, with noise added
    t=numpy.arange(-5,5,0.01)
    dat=Gaussian(t,amp=2.5)

    #pick a random starting position, and guess some errors
    guess=numpy.array([0.3,1.2,0.3,-0.2])
    scale=numpy.array([0.1,0.1,0.1,0.1])
    nstep=10000
    chain=run_mcmc(dat,guess,nstep,scale)
    #nn=numpy.round(0.2*nstep)
    #chain=chain[nn:,:]

    #pull true values out, compare to what we got
    param_true=numpy.array([dat.sig,dat.amp,dat.cent,dat.offset])
    for i in range(0,param_true.size):
        val=numpy.mean(chain[:,i])
        scat=numpy.std(chain[:,i])
        print [param_true[i],val,scat]
```

```
>>> execfile('fit_gaussian_mcmc.py')
[0.5, 0.48547765442013036, 0.031379203158769478]
[2.5, 2.5972175915216877, 0.16347041731916298]
[0.0, 0.039131754036757782, 0.030226015774759099]
[0.0, 0.0031281155414288856, 0.03983540490701154]
```

- Main: set up data first. Then call the chain function. Finally, compare output fit to true values.
- Parameter estimates are just the mean of the chain. Parameter errors are just the standard deviation of the chain.

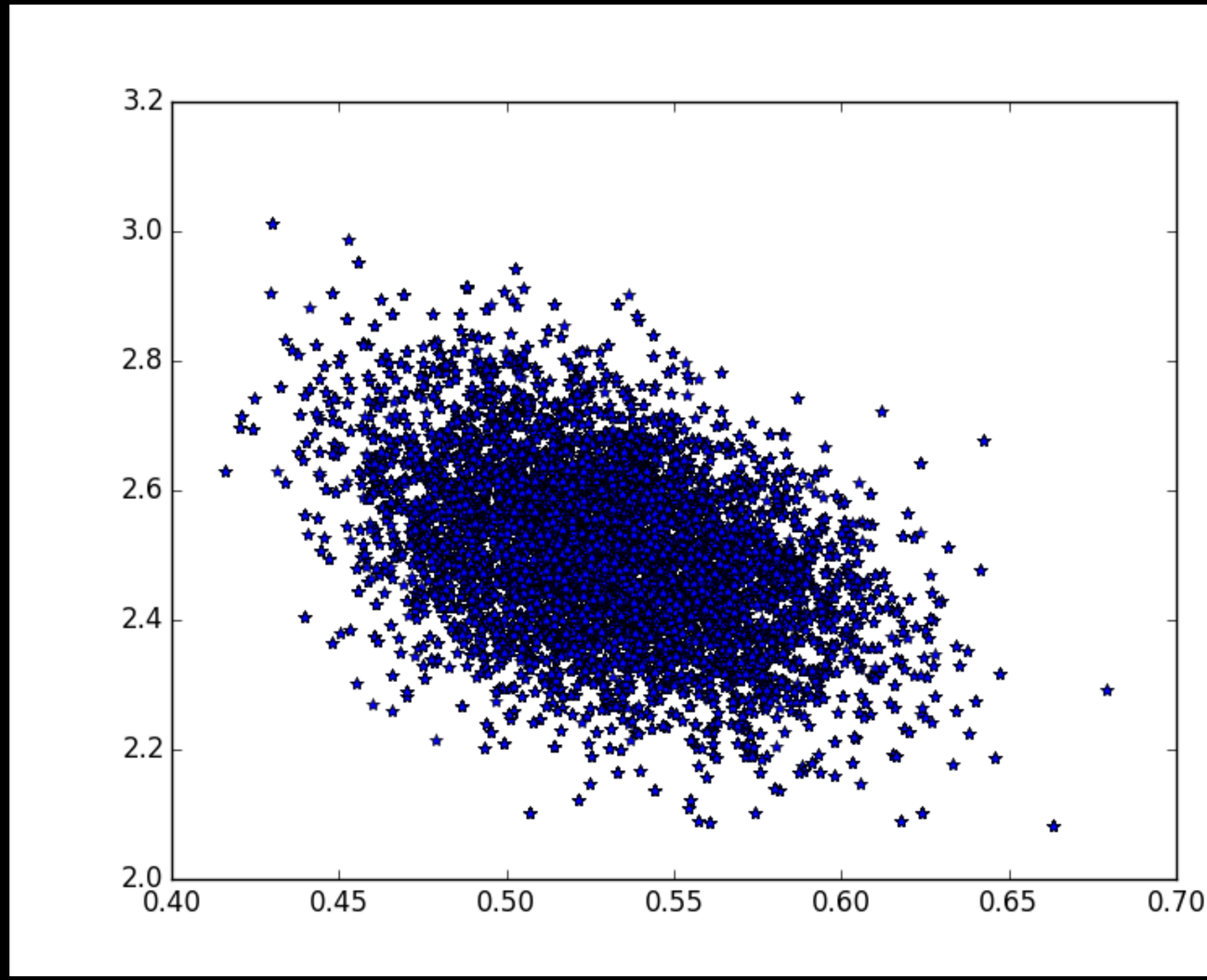
What Chain Looks Like



- Here's the samples for one parameter. Note big shift at beginning: we started at a wrong position, but chain quickly moved to correct value.
- Initial part is called "burn-in", and should be removed from chain.

Covariances

- Naturally get parameter covariances out of chains. Just look at covariance of samples!
- Very powerful way of tracing out complicated multi-dimensional likelihoods.

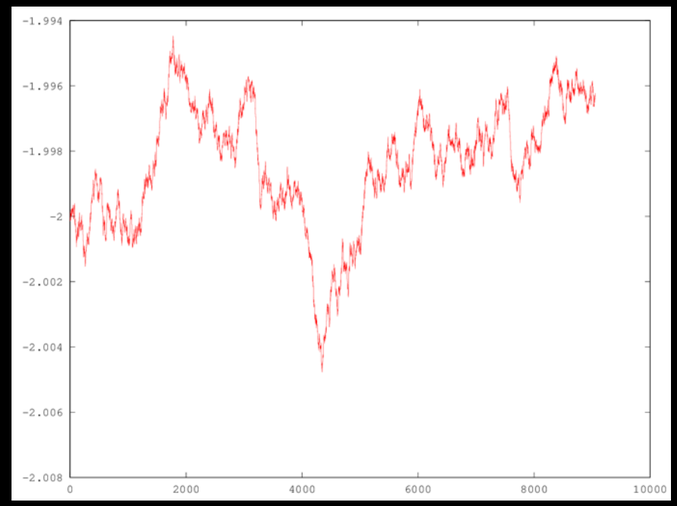


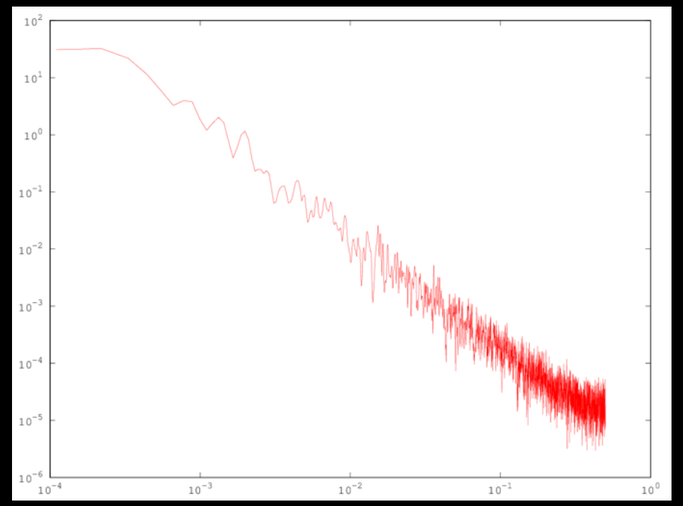
You Gotta Know When to Fold ‘em

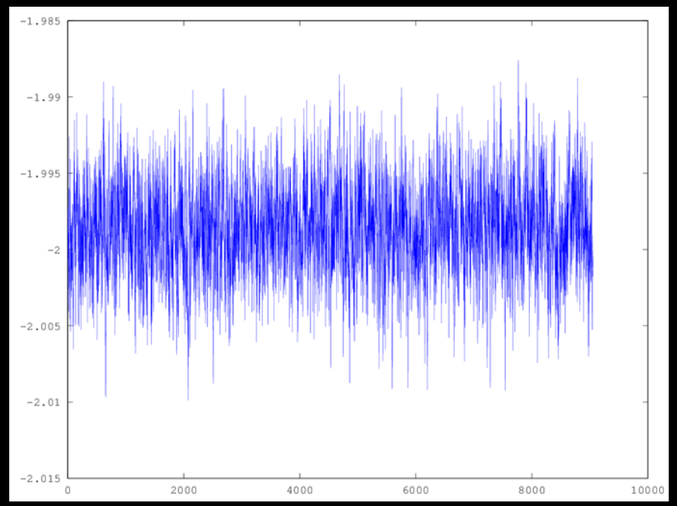
- Trick in doing MCMC is knowing when to stop.
- One standard technique is to run many chains, then look at scatter between them vs. expected scatter.
- Chains *work* independent of step size. However, they work *faster* with a good trial step size. Too large steps, we spend all our time sampling crazy land. Too small and we only move around slowly, so takes many samples to get to a new place.
- Good rule of thumb is you want to accept ~25% of your samples. Run for a bit, then adjust step size and start new chain.

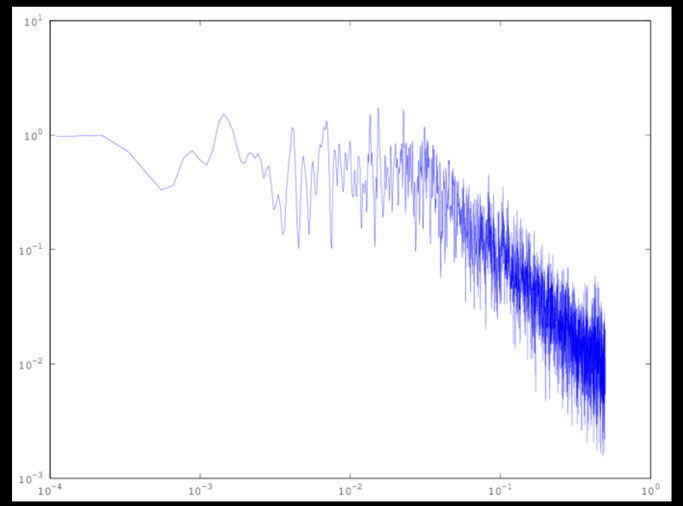
Single-Chain Convergence

- Chains eventually forget their past.
- If you plot chain samples, then eventually they should look like white noise
- FT of converged chain should be flat for large scales (low k)
- top: unconverged chain.
  bottom:converged chain.









Tutorial due dates

- Apologies for confusion. Tutorials for lectures 6 and 7 (nbody, advection) were nominally due today (not 7 and 8 as I mistakenly said on Thursday). Since the lack of clarity was my problem, you have through the end of the week to submit them for full marks.

- Lectures 8 and 9 (hydro, model fitting) are due next Tuesday, May 12.

Hydro Tutorial Problems

- Look at hydro1d.py. you'll see an assert guaranteed to fail at the end of get_bc, the boundary condition routine. Why did I do this? (5)

- Further on in hydro1d, where the derivatives are getting calculated, there's a factor of 1/2 in the pressure gradient. Why? (5)

- Finally, look at the time step calculator. Right now it doesn't implement the CFL condition. Put in a proper timestep calculator. This should find the globally smallest stable timestep and return the input times this value. So, if global CFL limit is 0.3 and we pass in 0.1, the return value should be 0.03 (10).

Model Fitting Tutorial

- Write linear least-squares code to fit sines and cosines to evenly sampled data. Pick the sines and cosines to have integer numbers of periods, so you pick 100 numbers, should have sin/cos(2*pi*n*(0:99)/100). Compare your fit parameters to the FFT of the data. (10)

- Take the mcmc sample code. Add a Lorentzian class ($f(x)=a/(b+(x-c)^2$). Run the fit, and show you get correct answers. (10)

- Modify the mcmc sample code to run a short chain, use that to estimate the Gaussian parameter errors, and then run a longer chain with using the error estimates. What is your accept fraction? (10)

- (Bonus) - we see that the parameters have covariances. You can do even better if you include those covariances in your step size. Write a stepper that uses a parameter covariance matrix - to generate fake data, take the eigenvalues/eigenvectors of the covariance matrix, then multiply sqrt(eigenvalues) by gaussians, and then multiply by transpose of eigenvectors. (5)

More Bonus

- Write a Newton's method routine to fit a Gaussian to data and run it on the same data as an MCMC run. (10) Which takes longer to write? Which takes longer to run?