Computational Physics
Lecture 9

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture9_2015.git

Tutorial Problems 2

- Complete the complex definition to support -,*, and / (__sub__, __mul__, and __div__). Recall that a/b = a*conj(b)/(b*conj(b)). Show from a few sample cases that your functions work. (10)

- Next lecture we will look at n-body simulations. In preparation, write a class that contains masses and x and y positions for a collection of particles. The class should also contain a dictionary that can contain options. Two entries in the dictionary should be the # of particles and G (gravitational constant). The class should also contain a method that calculates the potential energy of every particle, sum(G $m_1 m_2 / r_{12}$) . (10)

Euler Equations with EoS

- We can now write down Euler equations in conservation form with EoS
- $E=1/2u^2+e$, $\rho e=P/(\gamma-1)$. So $P=\rho(\gamma-1)(E-1/2u^2)$
- $\partial Q/\partial t+\partial(f(Q))/\partial x=0$
- $Q=[\rho,\rho u,\rho E]$, $f(Q)=[\rho u, \rho u^2+P, \rho uE+uP]$
- using momentum $p=\rho u$: $Q=[\rho,p,\rho E]$, $f(Q)=[p,pu+P,pE+uP]$

Model Fitting

- Sometime we have data. Sometimes we'd like to model it.
- Variety of techniques used, depending on what noise, model look like.
- Nearly all of these techniques require linear algebra, so we will start with numpy linear algebra.

Matrices vs. Arrays

- Unlike say Matlab, arrays and matrices are different in numpy.
- However, you can treat arrays like matrices and matrices like arrays. This can be confusing.
- * overloaded for matrices to be matrix multiply, but not for arrays.

In Practice

```
import numpy

#make an array of ones
x=numpy.ones([5,5])
print type(x)
#x*x does an element-by-element multiply for arrays
xx=x*x
print xx
#but we can get a matrix-type product with numpy.dot
x2=numpy.dot(x,x)
print x2



#alternatively, we can recast our array as a matrix
y=numpy.matrix(x)
print type(y)
#now multiply does a matrix multiply
yy=y*y
print yy
#but we can still do an element-wise multiply
y2=numpy.multiply(y,y)
print y2
```

```
>>> execfile('matrix_example.py')
<type 'numpy.ndarray'>
[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]
[[ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]]
<class 'numpy.matrixlib.defmatrix.matrix'>
[[ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]
 [ 5.  5.  5.  5.  5.]]
[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]
>>>
```

Vectors

- Be careful mixing vector arrays and matrices. Numpy will sometimes try to guess what you want. If it guesses wrongly, bugs ensue.
- Numpy will interpret a 1-d array as a row vector when multiplying a matrix.
- Numpy will interpret a 1-d array times a 2-d array as a per-column multiply.

```
>>> z=numpy.arange(1,5);print z
[1 2 3 4]
>>> x=numpy.ones([4,4])
>>> print x*z
[[ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]]
>>> print z*x
[[ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]]
>>> x=numpy.matrix(x)
>>> x*z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/System/Library/Frameworks/Python.fra
x.py", line 330, in __mul__
    return N.dot(self, asmatrix(other))
ValueError: matrices are not aligned
>>> z*x
matrix([[ 10.,  10.,  10.,  10.]])
>>>
```

1-d Arrays Ctd.

- Remember when I said a 1-d array is a row vector? Numpy is serious about that.
- Example to the right would otherwise be nonsensical.
- Bottom right - once we cast 1-d array into a matrix, transpose() does something.
- NB - Matlab has no concept of an array distinct from matrix. All 1-d things are either row or column matrices.

```
>>> z=numpy.arange(0,5);print z.shape
(5,)
>>> x=numpy.ones([5,5])
>>> print z*x
[[ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]]
>>> z=z.transpose()
>>> print z*x
[[ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]]
>>> print z.shape
(5,)
>>>
```

```
>>> z=numpy.matrix(z)
>>> print z.shape
(1, 5)
>>> z=z.transpose()
>>> print z.shape
(5, 1)
>>>
```

χ^2

- The PDF of a Gaussian is $\exp(-0.5(x-\mu)^2/\sigma^2)/\sqrt{2\pi\sigma^2}$ with mean μ and standard deviation σ.
- If we have a bunch of data points, which may have different means and standard deviations, then the joint PDF is the product of the PDFs.
- It is often more convenient to work with the log. For many points, $\log(\text{PDF})=\sum -0.5(x_i-\mu_i)^2/\sigma_i^2 - 0.5*\log(2\pi\sigma_i^2)$
- Usually, we know the variance of our data, and want our model to predict the expected value of x_i, which is μ_i. When we compare models, the second part is constant, so we ditch it. log likelihood becomes: $-0.5\sum(x_i-\mu_i)^2/\sigma_i^2$.
- $\sum(x_i-\mu_i)^2/\sigma_i^2$. is χ^2. We can find the maximum likelihood model by minimizing χ^2.

Linear least-squares

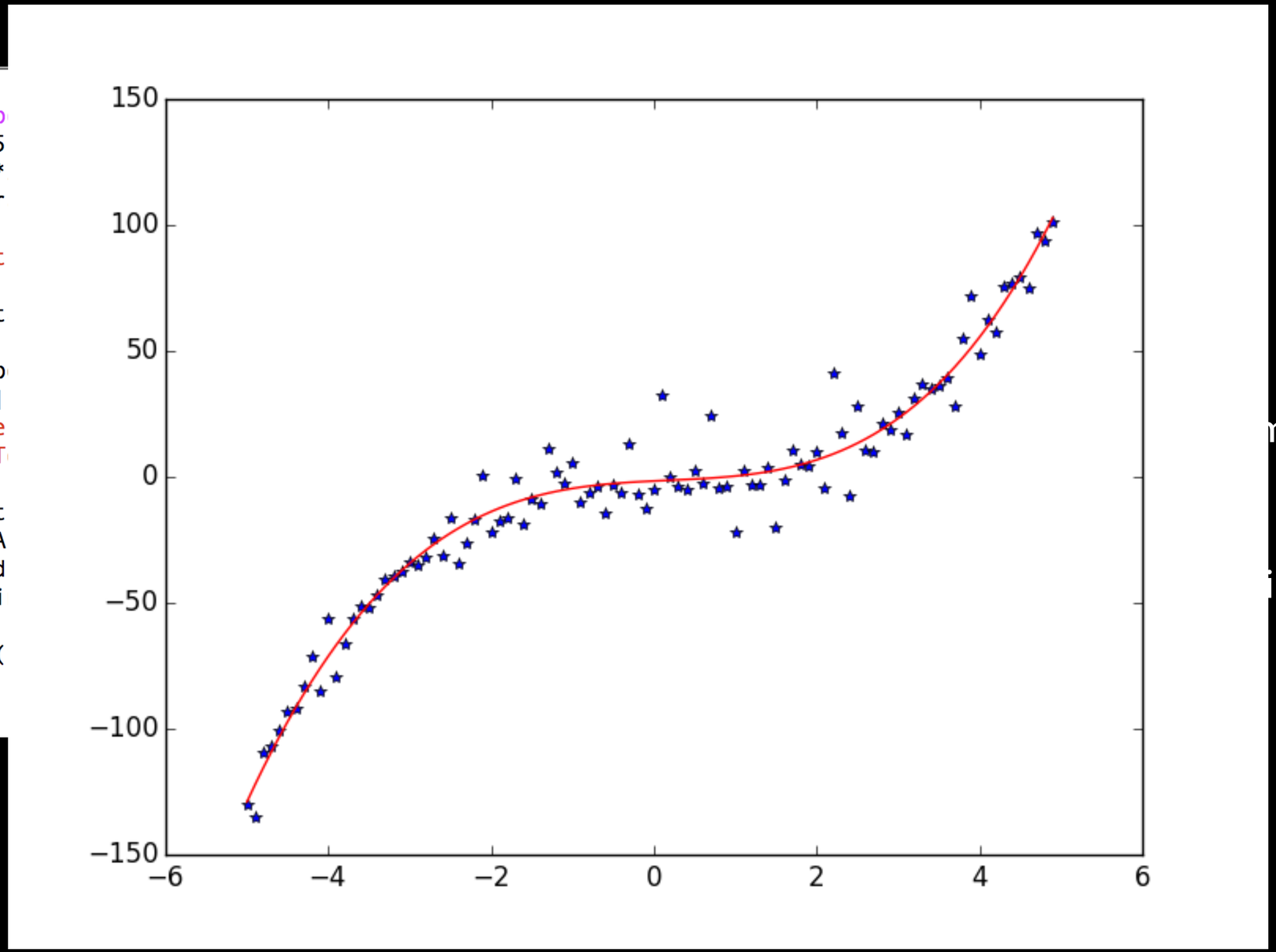
- Rewrite $\chi 2$ with matrices: $(x-\mu)^T N^{-1}(x-\mu)$ for noise covariance matrix N. If N has diagonal elements σ^2, this is identical to previous.
- Let's take simple case that our model depends linearly on a small number of parameters: $\mu_i=\sum A_{ij}m_j$ for model parameters m and matrix A that transforms to predicted values. In matricese: $\mu=Am$
- One example: x(t) is a polynomial in time. Then $\mu_i=\sum t_i^j c_j$.
- With this parameterization, $\chi^2=(x-Am)^T N^{-1}(x-Am)$

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, $m=(A^T N^{-1}A)^{-1}A^T N^{-1}x$

Example: Polynomial Fitting

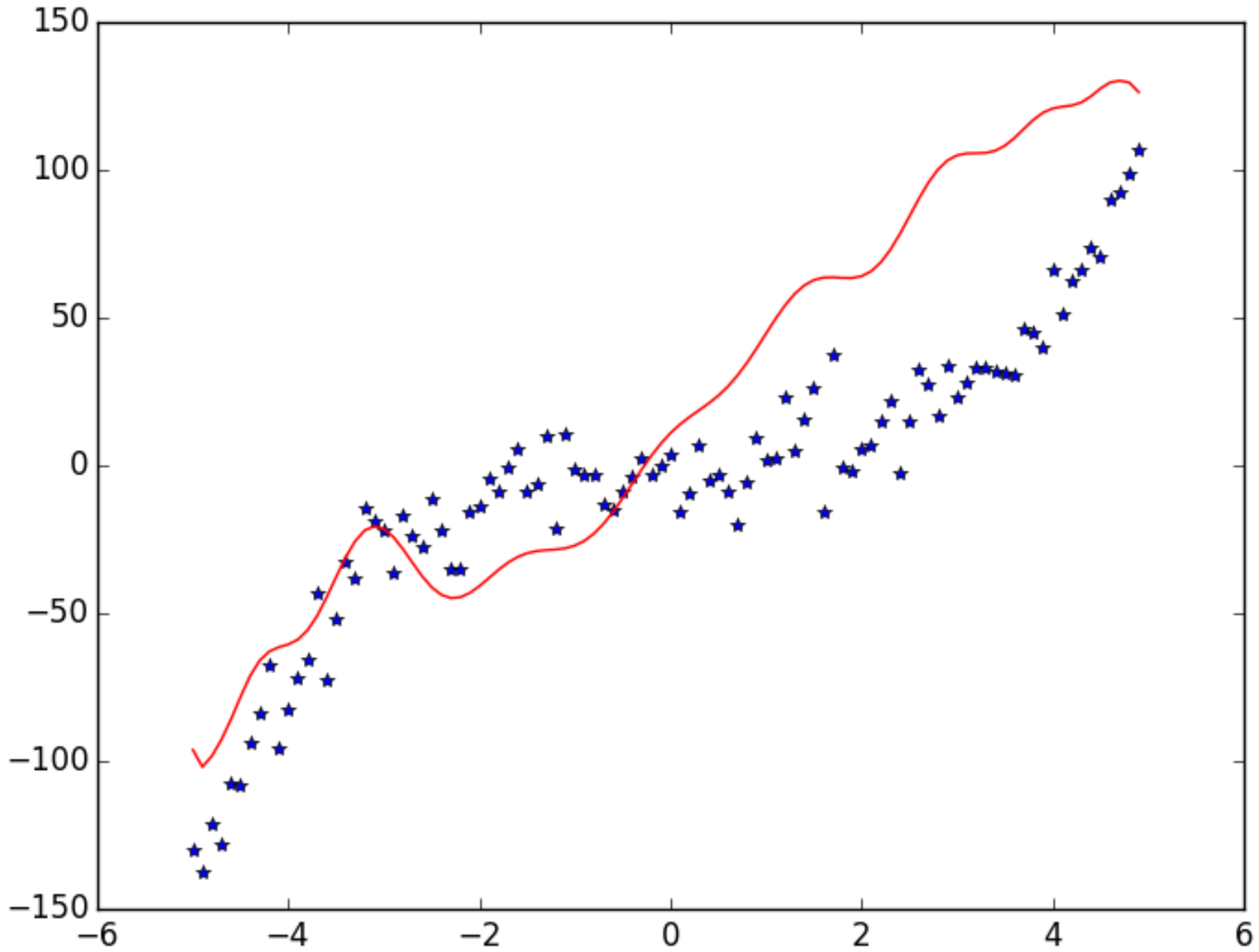
```
import numpy
from matplotlib imp
t=numpy.arange(-5,5
x_true=t**3-0.5*t**
x=x_true+10*numpy.r

npoly=5  #let's fit
ndata=t.size
A=numpy.zeros([ndat
A[:,0]=1.0
for i in range(1,np
    A[:,i]=A[:,i-1]
#Let's ignore noise
#m=(A^TA)^{-1}*(A^T
A=numpy.matrix(A)
d=numpy.matrix(x).t
lhs=A.transpose()*A
rhs=A.transpose()*d
fitp=numpy.linalg.i
pred=A*fitp
plt.clf();plt.plot(
plt.draw()
```



matrices rather

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
>>>
```

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.

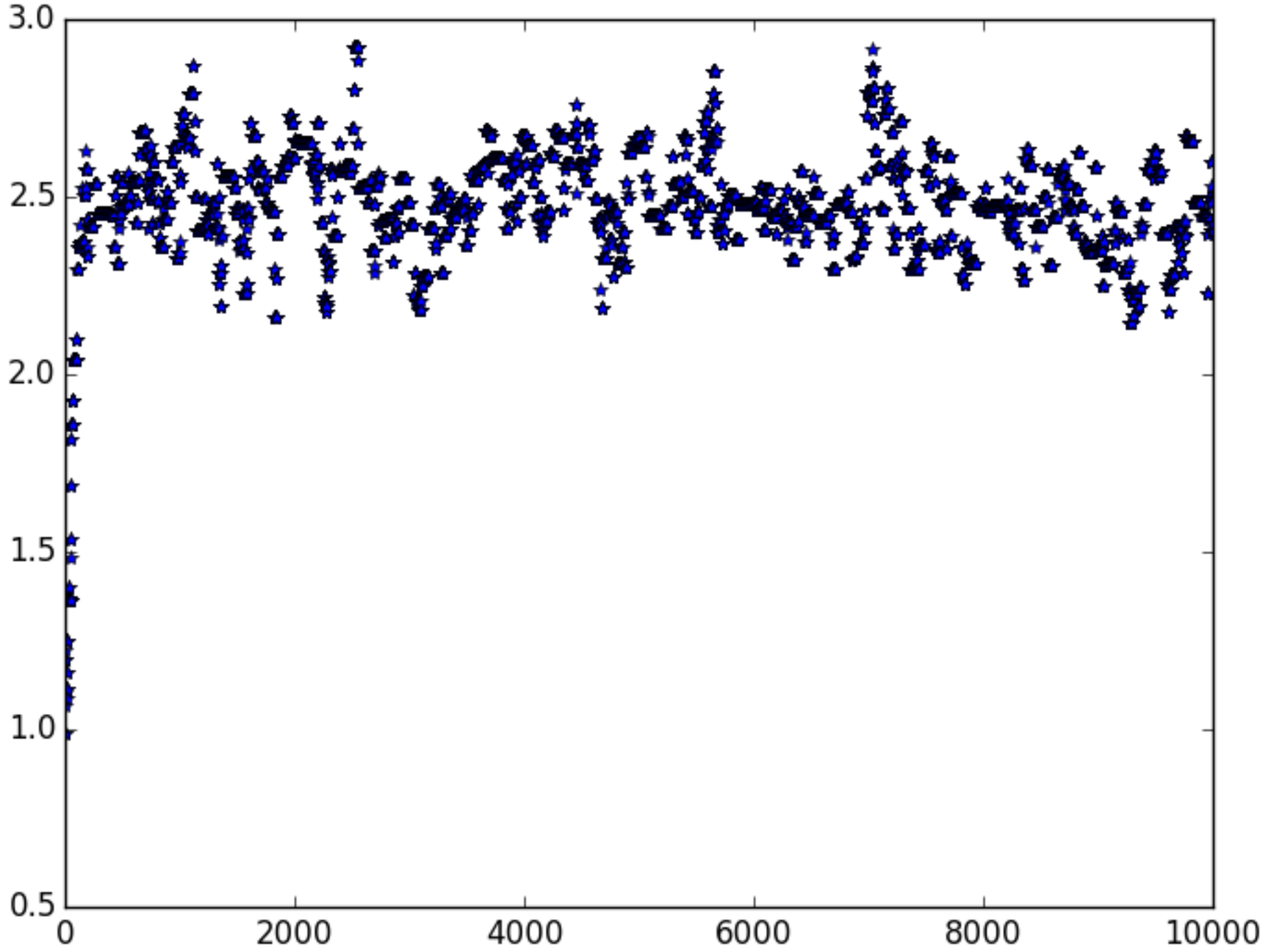
Output

```
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.

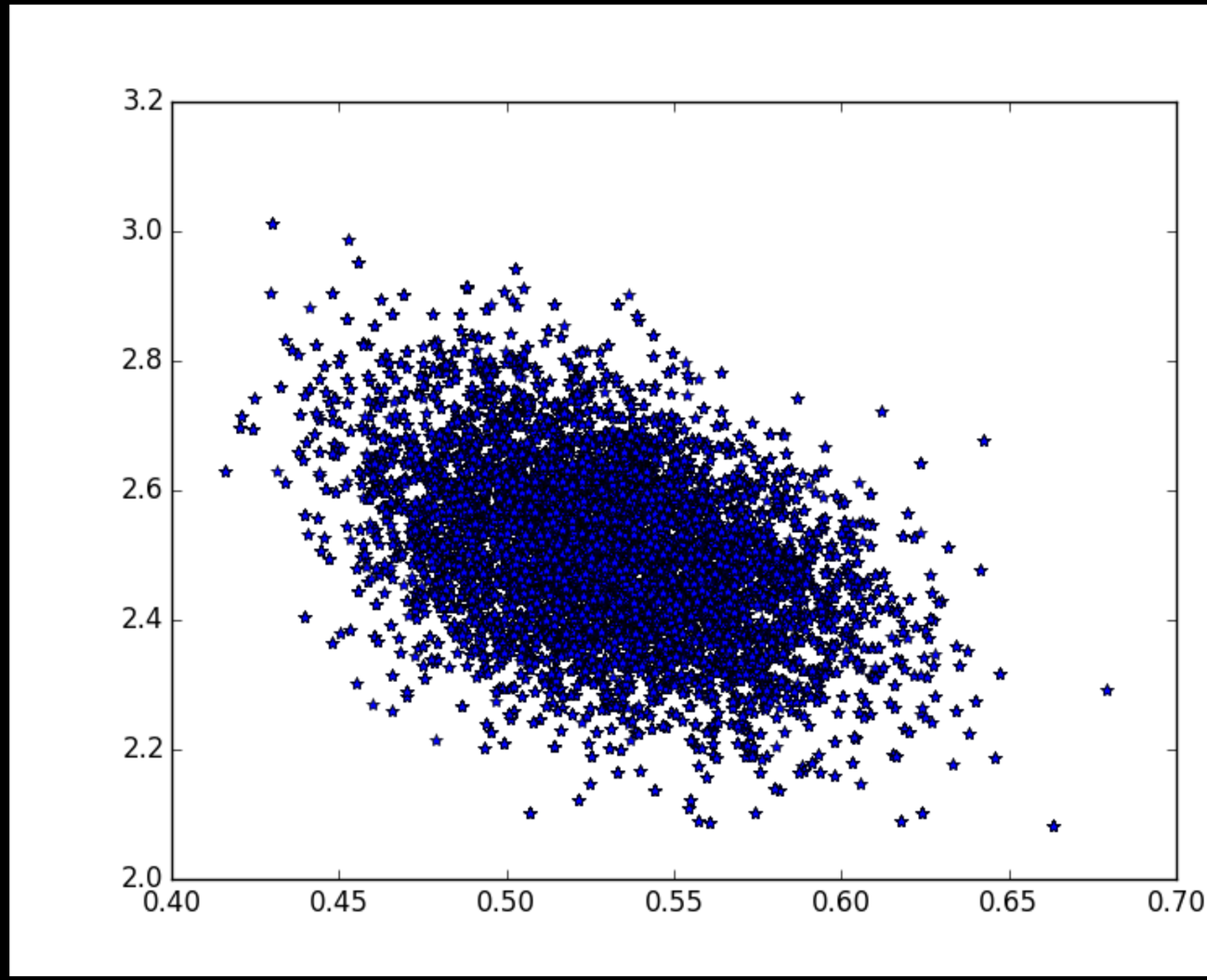
What Chain Looks Like



- 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.

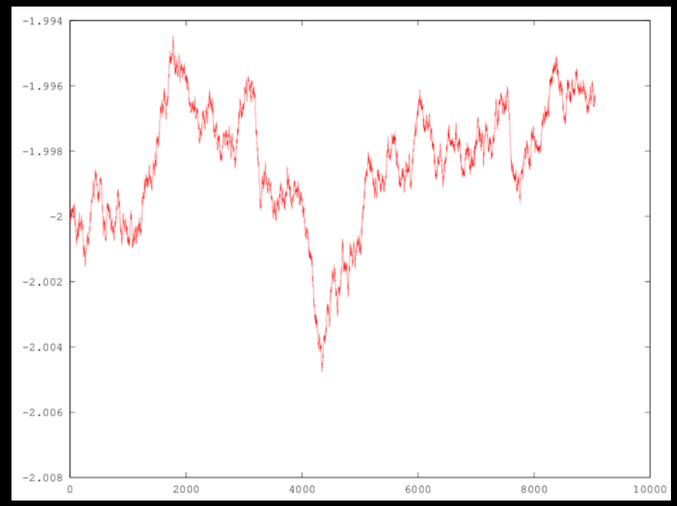


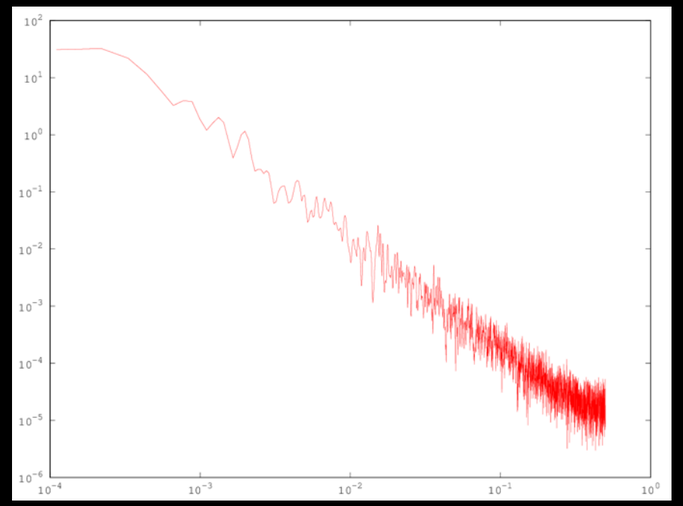
You Gotta Know When to Fold ‘em

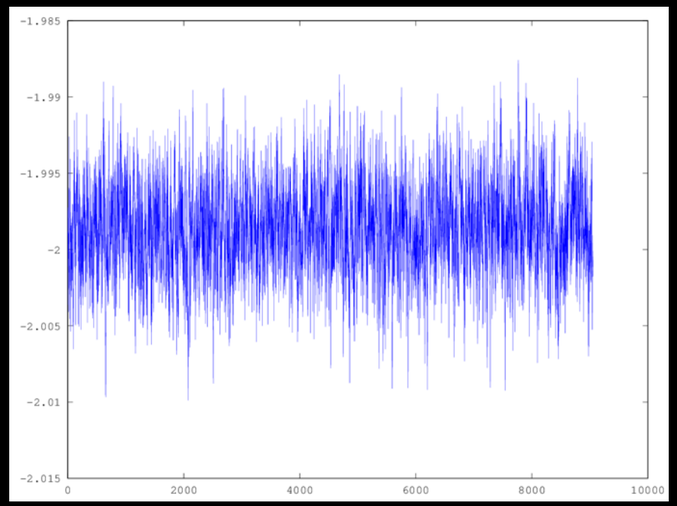
- Trick in doing MCMC is knowing when to stop.
- One standard technique is to run many chains, then look at scatter between them vs. expected scatter.
- Chains *work* independent of step size. However, they work *faster* with a good trial step size. Too large steps, we spend all our time sampling crazy 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.

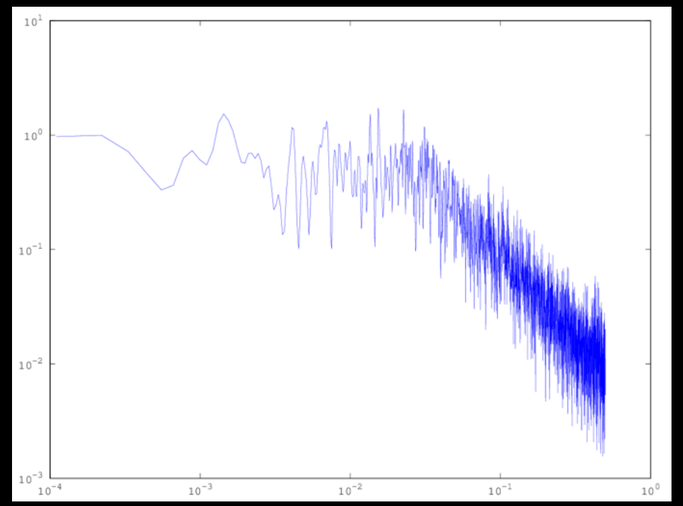
Single-Chain Convergence

- 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)